JEITA

EDAアニュアルレポート　2010

Annual Report on Electronic Design Automation

－　システム・オン・チップ時代の向こうに　－
Beyond the ″System on a Chip″ era

2011年6月発行

作　　成
EDA技術専門委員会
EDA Technical Committee

発　　行
一般社団法人　電子情報技術産業協会
Japan Electronics and Information Technology Industries Association

【巻頭言】

「高度情報化社会を支える先端LSIのEDA技術発展に向けて」

ＥＤＡ技術専門委員会　2010年度　委員長　　吉田 正昭

半導体製品は、家電製品をはじめとする電子機器だけでなく、自動車、医療機器などにもその適用範囲を広げ、今や社会基盤となった高度に発達した情報システムも半導体製品なしには実現できない時代となっている。そして、この半導体製品を生み出す半導体産業は様々な産業と高度情報化社会を支える基盤産業としてますます大きな役割を担うようになってきている。その中で、半導体自動設計（EDA: Electronic Design Automation）に関する技術は、絶え間なく進化する LSI の高集積化、高性能化、低消費電力化実現の鍵を握る技術として、より一層の技術発展が期待されている。

EDA 技術専門委員会は、電子情報技術産業協会（JEITA）における業界活動組織の一つとして EDA に関わる様々な活動を行なっているが、特に、電子機器の機能・性能を決定するシステム LSI 設計技術に関わる活動をその中心におき、次の三つのテーマを中心に活動している。

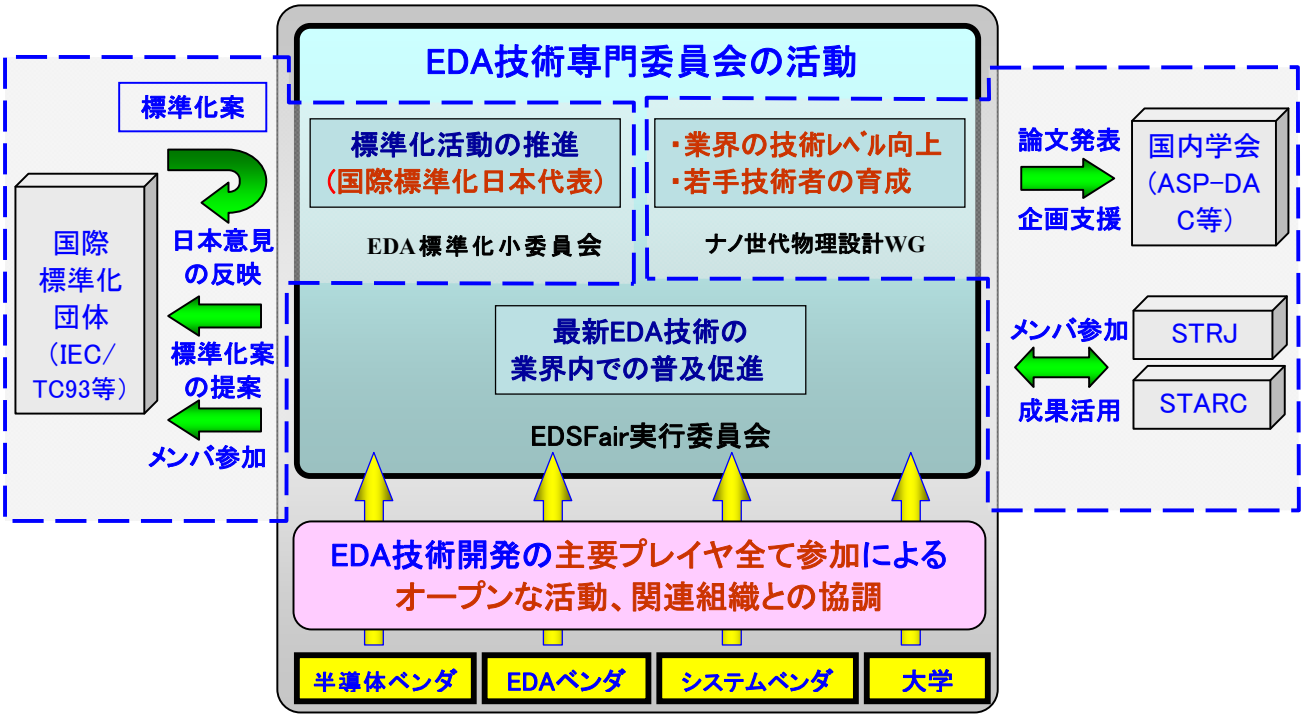
① システム LSI 設計技術に関する動向、関連情報についての調査・検討、及び課題解決への提案
② EDA 技術に関する標準化活動と関連機関、団体への協力と貢献
③ EDA 技術及び標準化の普及、推進のためのイベント開催、支援

①のテーマでは、ナノ世代物理設計ワーキンググループの活動を通して、次世代テクノロジーにおける設計技術・EDA 技術の課題抽出、LSI 設計手法に関するガイドライン作成、各種ライブラリの標準フォーマットの拡張の提案などに取り組んでいる。本年度は、22nm 世代で顕在化するばらつき要素と課題の抽出および検討を行なった。

②のテーマでは、EDA 標準化小委員会の活動を通して、IEC(International Electrotechnical Commission)、IEEE 等の国際的な標準化活動への貢献（提案と検証）を行っている。本小委員会は、IEC TC93/WG2 の国内委員会としての役割も果たしており、日本国の意見を代表する立場にある。近年では、傘下のワーキンググループの活動を通して、SystemC(IEEE 1666)、SystemVerilog(IEEE 1800)、Unified Power Format: UPF(IEEE 1801）の標準化に貢献してきた。さらに新しい活動として今年度 LSI-Package-Board 相互設計環境整備ワーキンググループを立ち上げ、課題の明確化、その解決方法の検討を行なった。また、IEEE での SystemVerilog 標準の改定の動きに対応するため、2008 年度末に一旦活動を停止した SystemVerilog-WG を再開させることとし、1 月より活動を開始した。

③の EDA 技術および標準化の普及、推進活動として、EDSFair（Electronic Design and Solution Fair）2011 を本年 1 月にパシフィコ横浜で開催した。EDSFair は、電子機器の設計技術に関する国内随一の総合的な展示会であり、今年度も特設ステージでの企画イベントを含め、積極的に企画、出展社誘致、来場者誘致活動を行なった。その結果、ＥＤＡ技術と関連技術に関する最新情報に出会える場として 8000 名を超える方々に参加いただいた。

次の図でこれらの委員会活動と関係する団体の関係を示す。



IEC/TC93 ： 国際電気標準会議/設計自動化

ASP-DAC：Asia South Pacific-Design Automation Conference

STRJ ： 半導体技術ロードマップ委員会

STARC ： 半導体理工学研究センター

図- 1　EDA 技術専門委員会と関連組織との関係

EDA 技術専門委員会は、上図の関連組織・団体との密な連携のもと、技術検討、標準化、そしてそれらの普及促進という 3 つの活動領域の活動を通じ、高度情報化社会を支える先端 LSI、高付加価値 LSI の設計を支える EDA 技術の発展、さらには日本の電子情報機器業界の発展に寄与すべく、本年度 18 社約 65 名の業界各社有志の自主的な参画により、運営実行してきた。

半導体および電子機器が切り拓く素晴らしい未来が今後も広がることを確信しつつ、引き続き2011年度も積極的な活動を展開する予定である。

本冊子「ＥＤＡアニュアルレポート2010」は、EDA技術専門委員会の2010年度年次報告として、上記三つの活動領域について、活動成果をまとめたものである。
また、Webにも各種報告を掲載しているので、ご覧いただきたい。
（http://www.jeita-edatc.com/index-jp.html）

2010 年度 JEITA/EDA 技術専門委員会 委員一覧

委員長	吉田　正昭	ルネサスエレクトロニクス㈱	技術開発本部 プラットフォーム インテグレーション統括部 シニアプロフェッショナル
副委員長	今井　浩史	㈱東芝	セミコンダクター社 アナログ・イメージングＩＣ事業部 設計技術開発部 設計メソドロジー開発担当　参事
副委員長	河村　薫	富士通セミコンダクター㈱	開発・製造本部 共通テクノロジ開発統括部 主席部長
監事	大芝　克幸	ソニー㈱	半導体事業本部　設計基盤技術部門 ミックスシグナルデザイン ソリューション部 シニアデザインエンジニア
幹事	太田　光保	パナソニック㈱	セミコンダクター社 システムＬＳＩ事業本部 商品開発センター 設計第三開発グループ チームリーダー
委員	林　靖二	キヤノン㈱	生産技術研究所　実装技術研究部 主任研究員
委員	小野　信任	㈱ジーダット	事業戦略室　室長
委員	野坂　啓介	図研エルミック㈱	リブウェア事業部　販売戦略部 マネージャー
委員	市川　浩司	㈱デンソー	研究開発３部　ＥＭＣ開発室 室長

委員	金子　俊之	㈱トッパンＮＥＣサーキットソリューションズ	設計部 電気特性シミュレーショングループ スーパーバイザー
委員	飯島　一彦	日本シノプシス(合)	技術本部　本部長 バイスプレジデント
委員	楠本　学	日本電気㈱	システム実装研究所 主任
委員	三橋　明城男	メンター・グラフィックス・ジャパン㈱	マーケティング部 ディレクター
委員	前野　酉治	㈱リコー	電子デバイスカンパニー 設計プロセス改革室 アナログ設計技術課 シニアスペシャリスト
特別委員	小島　智	ＮＥＣシステムテクノロジー㈱	ＰＦ統括本部 統括マネージャー
特別委員	田中　正和	パナソニック㈱	セミコンダクター社 システムＬＳＩ事業本部 基板技術開発センター 要素第一開発グループ 主任技師
特別委員	福場　義憲	㈱東芝	セミコンダクター社 アナログ・イメージングＩＣ事業部 設計技術開発部 設計インフラ技術　参事
客員	若林　一敏	日本電気㈱	中央研究所 ビジネスイノベーションセンター

			ＥＤＡ開発センター　研究部長
客員	今井　正治	大阪大学	大学院　情報科学研究科 情報システム工学専攻　教授
客員	岡村　芳雄	㈱半導体理工学研究センター	執行役員・開発第２部長
客員	神戸　尚志	近畿大学	理工学部　電気電子工学科 教授

＜２０１１年3月３１日現在＞

略 語 一 覧

[1]団体・組織の名称

Accellera
VIとOVIを統合した、設計記述言語の標準化に関連する活動機関

ANSI　American National Standards Institute
米国の標準化国家機関

ASP-DAC　Asia and South Pacific Design Automation Conference
アジア・太平洋地域でのEDA関連の国際学会（1995年に始まる）

CENELEC　European Committee for Electrotechnical Standardization
EC（欧州委員会）の電気電子分野に関する標準化機関

DAC　Design Automation Conference
米国で行われるEDA関連の国際学会

DASC　Design Automation Standardization Committee
IEEEの下部組織で設計自動化に関する標準化委員会

ECSI　European Electronic Chips & Systems design Initiative
欧州の設計自動化に関する標準化機関

EDIF Div.　Electronic Design Interchange Format Division
EIAの下部組織で電子系の惜報データ交換規格の検討機関

EIA　Electronic Industries Alliance
米国の電子業界団体(AssociationをAllianceに改称)

JEITA　Japan Electronics and Information Technology Industries Association
社団法人電子情報技術産業協会（電子業界団体）

ICCAD　International Conference on Computer Aided Design
CADに関する国際学会

IEC　International Electrotechnical Commission
電気電子分野に関する国際標準化機関

IEEE　Institute of Electrical and Electronics Engineers,Inc.
米国の電気電子分野の国際的な学会組織

IPC　Institute for Interconnecting and
Packaging Electronic Circuits Industry Association
米国のプリント回路に関する業界組織

ISO　International Organization for Standardization
国際標準化機関

IVC　International Verilog Conference

OVIが主催するVerilog HDL国際学会

JPCA Japan Printed Circuit Association
社団法人日本プリント回路工業会

OSCI Open SystemC Initiative
SystemC の標準化団体

OVI Open Verilog International
Verilog-HDLに関連する技術の標準化と普及推進組織

SEMATECH Semiconductor Manufacturing Technology Initiative （Consortium）
半導体技術を向上するために始まった米国の官民プロジェクト

Si2 Silicon Integration Initiative
設計環境の整備促進を支援する米国の非営利法人（旧CFI）

VASG VHDL Analysis and Standards Group
DASC傘下のVHDL標準化に関するワーキンググループ

VITAL VHDL Initiative Toward ASIC Libraries
VHDLライブラリ標準化団体

VSIA Virtual Socket Interface Alliance
LSIの機能ブロックのI/F標準化を目指している業界団体

[2]標準化・規格に関する技術用語

ALF Advanced Library Format
OVIで検討されたIPをも含むASICライブラリのフォーマット

ALR ASIC Library Representation
ASICライブラリ表現

CALS Computer Aided Logistics Support / Commerce at Light Speed
製品の技術情報の流通に関するすべての標準化活動の総称

CHDS Chip Hierarchical Design System
SEMATECHが要求仕様を作成した0.25-0.18um世代設計システム

CHDStd Chip Hierarchical Design System technical data
CHDSで使用するデータモデルの標準化

DCL Delay Calculation Language
遅延計算のための記述言語

DPCS Delay and Power Calculation System

IEEE1481として標準化推進されている遅延と消費電力の計算機構仕様

ECALS　Electronic Commerce at Light Speed

電子機器、半導体、電子部品等の複合電子部品技術情報の流通ならびにECALS 辞書等の維持管理等、すべての標準化活動の総称

EDI　Electronic Data Interchange

電子データ交換

EDIF　Electronic Design Interchange Format

EIAの下部組織で検討されている電子系の情報データ交換規格

ESPUT　European Strategic Program for Research and Development in Information Technology

欧州情報技術研究開発戦略計画

HDL　Hardware Description Language

ハードウェア記述言語

IP　Intellectual Property

流通/再利用可能なLSI設計資産（本来は知的財産権の意）

JIS　Japanese Industrial Standard

日本工業規格

SDF　Standard Delay Format

遅延時間を表記するフォーマット

SLDL　System Level Design Language

システム仕様記述言語

STEP　Standard for the Exchange of Product Model Data

CADの製品データ交換のための国際規格

VHDL　VHSIC（Very High Speed Integrated Circuit）Hardware Description Language

IEEE1076仕様に基づくハードウェア記述言語

VHDL-AMS　VHDL-Analog and Mixed-Signal (Extensions)

DASCの中で進められているVHDLのアナログ及びミックストシグナルシステムへの拡張

１．ＥＤＡ技術専門委員会の活動

1.1　2010 年度　JEITA/EDA 技術専門委員会　概要

委員会の名称　EDA 技術専門委員会（Electronic Design Automation Technical Committee）

委員会の目的　EDA に関連する技術およびその標準化の動向を調査し、その発展、推進を図り、もって国内外の関係業界の発展に寄与する

委員会の構成

会員会社/委員	14 社/14 名	
特別委員	3 名	
客員	4 名	

委員会の役員

委員長:	ルネサスエレクトロニクス	吉田　正昭
副委員長(正):	東芝	今井　浩史
副委員長(代行):	富士通セミコンダクター	河村　薫
監事:	ソニー	大芝　克幸

下部組織の役員

EDA 標準化小委員会	主査	東芝	今井　浩史
SystemC　WG	主査	東芝	今井　浩史
LPB 相互設計 WG	主査	東芝	福場　義憲
SystemVerilog　WG	主査	パナソニック	浜口　加寿美
ナノ世代物理設計 WG	主査	パナソニック	田中　正和
EDSFair 実行委員会	委員長	パナソニック	太田　光保
EDSFair 企画検討会	主査	ルネサスエレクトロニクス	吉田　正昭
SDF　WG	主査	ソニー	大芝　克幸

<方針>
1.委員会活動の継続的発展
新体制でのスムーズな運営と継続可能な体制の確立
EDSF およびそれに関わる活動の運営見直し
収支均衡策の検討（収入増、支出減）
委員会の効率的な運営
予算の最適運用
報告の簡略化（議事録確認、予算状況確認など）
議論に重点
2.標準化の推進と展開
IEEE-SA に Advanced メンバーとして積極参加
IEEE/DASC の加入継続、強化
国際会議(TC93, DASC)への積極的参加
標準化団体との定期的な会議開催

EDA 技術専門委員会メンバと担当　（敬称略）

委員長:	ルネサスエレクトロニクス	吉田　正昭	EDSFair/ASP-DAC 小委員会　主査、ASP-DAC OC EDSFair 検討会主査、EDA 標準化小委員会委員 EDSFair 実行委員会委員、EDSFair 企画検討会主査
副委員長:	東芝	今井　浩史	EDA 標準化小委員会　主査（IEC/TC93 WG2 国内主査） 国際標準化対応支援委員会　委員（IEC/TC93 担当）
副委員長:	富士通セミコンダクター	河村　薫	EDA標準化小委員会　副主査、内規改訂、広報パンフレット アニュアルレポート、システム・デザイン・フォーラム WG 委員 EDSFair 実行委員会委員
監事:	ソニー	大芝　克幸	システム・デザイン・フォーラム WG 主査、EDA 標準化小委員会委員 EDSFair 実行委員、HP/ML/予算、ASP-DAC リエゾン
幹事:	パナソニック	太田　光保	EDSFair 実行委員長、EDA 標準化小委員会委員 ASP-DAC OC、システム・デザイン・フォーラム WG 委員
委員:	ジーダット	小野　信任	
委員:	メンター・グラフィックス・ジャパン	三橋　明城男	
委員:	日本シノプシス	飯島　一彦	
委員:	リコー	前野　酉治	
委員:	図研エルミック	野坂　啓介	
委員:	キヤノン	林　　靖二	
委員:	デンソー	市川　浩司	
委員:	トッパン NEC サーキットソリューションズ	金子　俊之	
委員:	日本電気	楠本　学	
特別委員:	NEC システムテクノロジー	小島　智	IEC/TC93/WG2 コンベナ、 EDA 標準化小委員会副主査(国際標準化担当)

特別委員:	パナソニック	田中　正和	ナノ世代物理設計 WG　主査
特別委員:	東芝	福場　義憲	LPB WG 主査
客員:	近畿大学	神戸　尚志	IEC/TC93 国内委員長、元委員長、
客員:	大阪大学	今井　正治	上流設計識者、ASP-DAC リエゾン
客員:	STARC	岡村　芳雄	元委員長
客員:	日本電気	若林　一敏	ASP-DAC リエゾン
事務局:	JEITA	進藤　淳二	
事務局:	JEITA	細川　照彦	

活動計画の概要　＜別紙-1 参照＞

委員会の予算　　会費　240,000 円 ＊ 17 社 =4,080,000 円
（会費 14 社＋LPB-WG 特別委員負担金 3 社）

委員会の開催　　年6回程度（予定日: 別紙-2 参照）
必要に応じて幹事会を開催する

担当事務局　　JEITA/電子デバイス部

＜別紙-1＞

活動計画の概要

1. EDA 技術の動向＆関連情報の調査検討、課題解決への提案

(1) 小委員会及び WG による技術動向とニーズ調査
　・最先端テクノロジ ： ナノ世代物理設計 WG
　・設計言語 ： EDA 標準化小委員会および傘下の WG
(2) 関連機関、団体、キーパーソン等との合同会議、意見交換、交流
　・STARC, STRJ 等
(3) 国内外の学会,研究会,イベントへの参加と連携
　・ASP-DAC2011, DAC2010, IEICE など

2. EDA 技術および標準化の普及推進のためのイベント実施、支援

(1) 「EDSFair2011」(横浜)
　・経済情勢、出展者・参加者のニーズ変化、EDA-TC 幹事会社数の減少を踏まえ、“身の丈にあった EDSF 開催”を継続
　・特設ステージの運営体制および運営方法制の確立(幹事会社＋委員会社有志＋JESA)
(2) 各種ワークショップ、講演会の開催
　・システム・デザイン・フォーラムの実施を検討
(3) ASP-DAC2011との連携

3. EDA に関する標準化活動への貢献と関連機関、団体への対応

(1) EDA 設計言語およびモデル標準化のための技術的検討と提案
SystemC-WG を継続し、SystemC に関する提案活動を実施
　・LPB－WG を正式発足させ、LPB 統合設計環境標準化に向けた活動を開始
　・SystemVerilog,Verilog-HDL,VHDL,A-HDL、PowerFormat などは、EDA 標準化小委員会で必要に応じて対応
(2) 国際的な関連機関、団体への参画・連携と標準化活動への協力
　・EDA 標準化小委員会で対応
　・IEC/TC93 国際会議(10 月@シアトル)、WG2(@DAC or @DVCon のいずれか)への参加
　・JIS-CENELEC 情報交換会への協力
　・IEEE/DASC,IEEE-SA, OSCI, Accellera, Si2 等との連携強
　　1 月 DASC 会議 Host@EDSFair
　　IEEE－SA の参加資格を Advanced に変更し、活動強化

4. 委員会活動の広報

(1) 広報パンフレットの配布@EDSFair2011
(2) アニュアルレポートの発行(下記 HP でも公開)
(3) WWW ホームページの公開
(4) 活動成果の発表
　・システムデザインフォーラム ： 標準化活動
　・学術論文 ： ナノ世代

＜別紙-2＞

2010 年度　JEITA/EDA 技術専門委員会 会合予定

年/月	技術専門委員会	懇親会	関連イベント
2010/4	4/23(金)（東京地区） 議事録 ジーダット ・10 年度事業計画説明、承認 ・10 年度小委員会/WG 計画説明、承認 ・09 年度会計収支と 10 年度会計予算説明、承認 ・09 年度版アニュアルレポート作成状況報告		・軽井沢 WS (4/19-20)　@軽井沢
2010/5			
2010/6	6/25(金)（東京地区） 議事録 リコー ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・委員名簿更新内容確認		・DAC2010 (6/13-6/18)@Anaheim, CA
2010/7			
2010/8			
2010/9	9/17(金) （関西地区）議事録 メンター ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・予算消費状況		・DA シンポジウム 2010 (9/2-3) @豊橋
2010/10			・TC93 国際会議（2010/10/8～11）@シアトル
2010/11	11/19(金)（東京地区） 議事録 リコー ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・EDSFair 用パンフレット作成手順説明		・ICCAD2010 (11/7-11) @San Jose, CA ・デザインガイア（2010/11/29-12/1）@九大
2010/12			
2011/1	1/14(金)（関西地区） 議事録 図研 ・小委員会/WG 進捗報告 ・半導体部会/半導体技術委員会報告内容説明 ・10 年度体制協議・EDSFair 用パンフレット内容確認 ・アニュアルレポート作成分担 ・予算消費状況報告		・ASP-DAC2011 (1/25-28) @横浜 ・EDSFair2011(1/27-28)　@横浜
2011/2			・ISSCC2011(2/20-24) @SF, CA ・DVCon2011(2/28-3/3)@San Jose, CA
2011/3	3/18(金) （関東地区） 議事録 キヤノン ・半導体部会/半導体技術委員会報告内容説明 ・10 年度小委員会/WG の年間活動報告 ・EDSFair2011 報告・実行委員会活動報告 ・10 年度予算消費状況報告 ・11年度事業計画説明 ・11年度予算案	○	

1.2　JEITA/EDA 技術専門委員会ホームページ

1.2.1　開設の目的

電子情報技術産業協会（JEITA）の EDA 技術専門委員会の活動状況を公開し、EDA 技術に関する標準化や技術調査に関するご理解とご協力を頂くことを目的とする。

1.2.2　ホームページの詳細

2006 年度にホームページを一新し、よりわかりやすく、また、欲しい情報に容易にアクセスできるような構成にした。ホームページは日本語版の他、英語版も用意し海外からの利用者の利便性を考慮している。日本語版、英語版の切り替えは簡単にできるように構成されている。本委員会の成果は、ドキュメントとして閲覧できると共に、ダウンロードすることもできる。

（1）ホームページの URL

http://www.jeita-edatc.com/

2009 年度までは、大阪大学のご協力で、大阪大学のサーバーにホームページを設置させて頂き、またデータの更新など、メンテナンスについてもご協力を頂いた。2009 年度に、jeita-edatc.com のドメインを取得すると共に、サーバーを外部業者のレンタルサーバーへ移行した。データ更新やメンテナンスは、担当委員が行っている。

（2）エントリーページの構成

日本語版、英語版はそれぞれ次のエントリーで構成されている。

日本語版：	英語版：
委員会の紹介	Introduction of a committee
委員会活動	Committee activity
公開資料ライブラリ	Open data library
イベント・関連機関	Event・A related organization
お問い合わせ	Inquiry
サイトマップ	Site map

（3）委員会の紹介/Introduction of a committee

委員長挨拶、活動と成果、メンバーをサブエントリとする。本委員会の概要、前年度の活動内容・成果、本年度の活動計画、委員会メンバーを紹介している。

（4）委員会活動/Committee activity

下記の委員会の活動を紹介している。

- EDA 標準化小委員会（EDA 標準化小委員会および傘下の SystemC WG、LSI･パッケージ･ボード相互設計 WG、SystemVerilog WG、PowerFormat WG）
- ナノ世代物理設計 WG
- EDSFair 実行委員会

（5）公開資料ライブラリ/Open data library

「公開資料ライブラリ」のページでは、EDA 技術専門委員会内の各委員会・WG の活動報告や各委員からの発表資料等を適宜掲載している。主な掲載資料を以下に示す。なお、英語版では、日本語版と同一の日本語資料を掲載している。

- EDA 技術専門委員会（過去のアニュアルレポート）
- EDA 標準化小委員会（SystemC 推奨設計メソドロジ、Power Format 比較表など）
- ナノ世代物理設計 WG（過去の資料）
- EDSFair 実行委員会（システムデザインフォーラムの紹介）
- システムレベル設計研究会（旧サイトへのリンク）
- その他（過去の委員会活動報告）

（６）イベント・関連機関/Event・A related organization

「関連の会議」では、次の関係の深い EDA 関連技術委員会を紹介している。

- IEEE/DASC（電気電子学会/設計自動化標準化委員会）
- IEC/TC93（国際電気標準会議/デザインオートメーション標準化技術委員会）

また、「関連機関」では、本委員会に関連のある 17 機関を紹介し、委員会名称はそれぞれのホームページへリンクされている。

1.3　2010 年度 JEITA/EDA 技術専門委員会 年間実績・予定表

月	EDA 技術専門委員会	
	幹事会	委員会
2010 年 4 月		4/23（金）　14:00-17:00　JEITA　403 会議室
5 月		
6 月	6/25（金）　11:00-13:00	6/25（金）　14:00-17:00　JEITA　412 会議室
7 月		
8 月		
9 月	9/17（金）　11:00-13:00	9/17（金）　14:00-17:00　JEITA 関西支部　第 1 会議室
10 月		
11 月	11/19（金）　11:00-13:00	11/19（金）　14:00-17:00　JEITA　412 会議室
12 月		
2011 年 1 月	1/14（金）　11:00-13:00 1/28（金）　15:00-17:00 EDS Fair 会場	1/14（金）　14:00-17:00　電子会館　４階会議室
2 月		
3 月		3/18（金）　13:00-17:00　JEITA　412 会議室 （3/11 発生の東日本大震災のため 4 月に延期）

月	EDA 標準化小委員会関連
2010 年 4 月	4/21(水) 13:00-17:00 第 1 回 LPB 相互設計 WG　JEITA　412 会議室 4/26(月) 14:00-17:00 第 1 回 SystemC-WG　JEITA　417 会議室
5 月	5/14(金) 11:00-13:00 第 1 回 EDA 標準化小委員会　機会振興会館　69 号会議室 5/26(水)　9:00-12:00 第 2 回 LPB 相互設計 WG　JEITA　412 会議室
6 月	6/11(金) 15:15-17:15 第 2 回 SystemC-WG　JEITA　417 会議室
7 月	7/ 2(金) 11:00-13:00 第 2 回 EDA 標準化小委員会　機会振興会館　69 号会議室 7/ 6(火) 13:00-17:00 第 3 回 LPB 相互設計 WG　JEITA　412 会議室
8 月	
9 月	9/ 3(金) 11:00-13:00 第 3 回 EDA 標準化小委員会　機会振興会館　69 号会議室 9/ 3(金) 13:00-17:00 第 4 回 LPB 相互設計 WG　JEITA　412 会議室 9/10(金) 15:15-17:15 第 3 回 SystemC-WG　JEITA　417 会議室
10 月	10/12(火) 13:00-17:00 第 5 回 LPB 相互設計 WG　JEITA　412 会議室
11 月	11/ 5(金) 11:00-13:00 第 4 回 EDA 標準化小委員会　機会振興会館　69 号会議室 11/30(火) 13:00-17:00 第 6 回 LPB 相互設計 WG　JEITA　412 会議室
12 月	12/ 3(金) 10:00-17:00 集中審議 SystemC-WG　富士通セミコンダクター　新横浜 12/27(月) 13:00-17:00 臨時 SystemC-WG　東芝マイクロエレクトロニクスセンター
2011 年 1 月	1/11(火) 13:00-17:00 第 7 回 LPB 相互設計 WG　JEITA　412 会議室
2 月	2/ 4(金) 11:00-13:00 第 5 回 EDA 標準化小委員会　機会振興会館　63 号会議室
3 月	3/ 8(火) 13:00-17:00 第 8 回 LPB 相互設計 WG　JEITA　412 会議室 3/11(金) 14:00-17:00 第 4 回 SystemC-WG　JEITA　417 会議室

月	ナノ世代物理設計 WG 関連
2010 年 4 月	
5 月	5/14(金) 10:00-17:00　第 1 回ナノ世代物理設計 WG　JEITA　417 会議室
6 月	6/25(金) 10:00-17:00　第 2 回ナノ世代物理設計 WG　JEITA　417 会議室
7 月	
8 月	8/20(金) 10:00-17:00　第 3 回ナノ世代物理設計 WG　JEITA 関西支部　第 2 会議室
9 月	
10 月	10/28(木) 10:00-17:00　第 4 回ナノ世代物理設計 WG　JEITA 関西支部　第 2 会議室
11 月	

12 月	12/14(火) 10:00-17:00　第 5 回ナノ世代物理設計 WG　JEITA　417 会議室
2011 年 1 月	
2 月	2/22(火) 10:00-17:00　第 6 回ナノ世代物理設計 WG　JEITA 関西支部　第 1 会議室
3 月	3/25(金) 10:00-17:00　第 7 回ナノ世代物理設計 WG　JEITA　406 会議室 (3/11 発生の東日本大震災のため 4 月に延期)

月	EDS Fair 実行委員会関係
2010 年 4 月	4/22(木) 14:00-17:00　第 1 回 EDSFair2010 実行委員会　TKP 大手町ｶﾝﾌｧﾚﾝｽｾﾝﾀｰ　18 階会議室
5 月	
6 月	6/ 4(金) 14:00-17:00　第 2 回 EDSFair2010 実行委員会　JEITA　401 会議室
7 月	
8 月	
9 月	9/ 2(木) 14:00-17:00　第 3 回 EDSFair2010 実行委員会　JEITA　414 会議室
10 月	10/14(木) 14:00-17:00　第 4 回 EDSFair2010 実行委員会　日本ｴﾚｸﾄﾛﾆｸｽｼｮｰ協会　会議室
11 月	11/ 9(火) 13:00-16:00　臨時　EDSFair2010 実行委員会　日本ｴﾚｸﾄﾛﾆｸｽｼｮｰ協会　会議室
12 月	12/13(月) 14:00-17:00　第 5 回 EDSFair2010 実行委員会　日本ｴﾚｸﾄﾛﾆｸｽｼｮｰ協会　会議室
2011 年 1 月	1/27(木) 17:30-19:00　臨時　EDSFair2010 実行委員会　パシフィコ横浜　CM4 会議室
2 月	2/24(木) 14:00-17:00　第 6 回 EDSFair2010 実行委員会　日本ｴﾚｸﾄﾛﾆｸｽｼｮｰ協会　会議室
3 月	3/ 8(火) 14:00-17:00　第 1 回 EDSFair2011 実行委員会　ｱｯﾄﾋﾞｼﾞﾈｽｾﾝﾀｰ大手町 CR-2D 号室

月	システム・デザイン・フォーラム 2011 WG 関係
2010 年 4 月	
5 月	5/21(金) 14:00-16:00　第 1 回ｼｽﾃﾑ･ﾃﾞｻﾞｲﾝ･ﾌｫｰﾗﾑ 2011WG　日本ｴﾚｸﾄﾛﾆｸｽｼｮｰ協会　会議室
6 月	6/24(木) 14:00-17:00　第 1 回ｼｽﾃﾑ･ﾃﾞｻﾞｲﾝ･ﾌｫｰﾗﾑ 2011WG　日本ｴﾚｸﾄﾛﾆｸｽｼｮｰ協会　会議室
7 月	
8 月	
9 月	
10 月	
11 月	
12 月	
2011 年 1 月	
2 月	
3 月	

月	関連行事
2010 年 4 月	
5 月	
6 月	
7 月	
8 月	
9 月	
10 月	
11 月	
12 月	
2011 年 1 月	
2 月	
3 月	

２．各技術委員会の活動報告

2.1　ナノ世代物理設計ワーキンググループ(Nano Scale Physical Design Working Group)

2.1.1　目的

半導体デバイス・配線テクノロジの進化に伴い、新たな設計上の課題があらわれてきている。また、これらの課題に対処するため各社が開発した設計手法や設計情報が、標準化されず、設計環境の開発・サポートコストが削減できない事例や、EDA ツール間あるいは半導体ベンダと顧客との間の情報授受がうまく行えないといった事例が増えてきている。

上記課題を背景として、本ワーキンググループでは、次のような調査、課題検討、及び標準化を実施することにより、より効率的な設計環境の実現に貢献することを目的として活動を行っている。

- 次世代(22 ナノメータ)以降のテクノロジ･ノードにおける、LSI の物理設計・検証に関する課題の抽出、および解決手法の提案
- 半導体ベンダとその顧客との間でやり取りするライブラリや設計情報等を規定する、設計ルール・ガイドラインの作成
- LSI の物理設計、検証手法の精度、互換性や効率を向上できるライブラリ・設計情報フォーマットの標準化
- 各種ライブラリを用いて行う検証が十分な精度で行えるかを判定するための標準ベンチマークデータの作成

2.1.2　活動内容

2007 年 5 月から活動を開始し、今年度は、ばらつきに関わる下記のテーマを取り上げて調査や検討を行った。

- 配線ばらつき感度/SSPEF フォーマットの適用範囲調査
- NBTI ばらつきの論文調査、影響検討配線ばらつきの表現手段の調査、検討

今年度の活動の成果として、以下の２項目について調査を行い、有用な知見を得た。

(1)配線ばらつき感度/SSPEF フォーマットの適用範囲調査

IEEE1481 の Sensitivity-SPEF フォーマット（2010/3/11 制定　IEEE 1481-2009）は、配線形状に関して従来概ね 4 コーナーの検証を行っていた配線グローバルばらつきに対し、Statistical Static Timing Analysis(SSTA)を用いた統一的な扱いを可能としている。ただし、配線寸法に対する RC の感度を線形にモデル化しており、非対称分布が扱えない、寸法ばらつきの大きい領域で誤差が大きい、という制約がある。そのため、我々はこのフォーマットの精度を検証し、実用可能な精度となるためのモデル化手法、フォーマット改善検討、あるいは、寸法ばらつきの範囲を明確化し配線プロセス工程に対してフィードバックを行うべく、ITRS ロードマップ値を用いた検証ワーク

を実施した。また、SPEF 感度式の誤差を最小限となるモデルを提案し、22nm プロセスを想定した精度検証を行った。検証の結果、提案モデルによれば、単純配線において容量精度 3%、遅延精度 1%でのモデル化が可能であることを示した。

(2) プロセスばらつきを考慮した集積回路の NBTI 故障率解析

NBTI(Negative Bias Temperature Instability)による MOSFET のしきい値電圧変動と、その現象に伴うしきい値電圧ばらつきの増大は、微細 CMOS 回路の信頼性劣化に対する懸念となる可能性がある。特に NBTI によるしきい値電圧のばらつき増大に関して、いくつかの論文が発表され、その影響が議論されている。これまで報告された NBTI によるしきい値電圧劣化のしきい値電圧変動ばらつきを分布特性の観点で詳細に調査し、22nm プロセスでの分布パラメータを予測し、CMOS 回路を例題に、信号遅延の NBTI 劣化分布をモンテカルロ解析することにより、NBTI 劣化特性とそのばらつき分布を正確に推定し、遅延特性分布のワーストケース特性に与える影響を解析した。NBTI のしきい値電圧劣化の分布形状が対数正規分布である可能性を指摘し、その影響が 1 FIT レベルの遅延故障率にどの程度影響するかを定量的に見積もった。更にこの解析を実用的な時間で実施するためには、新たなモンテカルロ解析手法の開発が必要であり、Important Sampling の手法を取り入れている。本年度成果として、CMOS の微細化に対する NBTI 劣化特性分布、回路特性ばらつき分布、回路故障率を予測するメソドロジーを提供した。

これらの活動で得られた成果は、次のような形態により無償で一般に公開する。

- アニュアルレポート
- JEITA のホームページ
- 関連学会の研究会・学会における発表や論文誌への投稿

成果の詳細は本アニュアルレポートの付録に掲載した。また、前年度から今年度にかけての成果の一部を以下の学会にて発表した。

[1] 「RTN を考慮した回路特性ばらつき解析方法の検討」, 情報処理学会 DA シンポジウム, 2010 年 9 月.

また、本年度は EDS Fair 2011/特設ステージにおいて、プロセス微細化におけるシステム LSI の設計課題をテーマとした、「システム LSI 設計の今後　～22nm 時代に向けて～」と題したパネル・ディスカッションを開催した。さらに、EDS Fair 会期中に開催された IEEE DASC 会議に参加、配線ばらつきを表現する Sensitivity SPEF フォーマットの策定に対応する DPC-WG との連絡を継続している。

2.1.3　関連機関の動向

米国に本部を置き、世界各国に多数の会員を持つ国際的な学会であり、電気および電子関連の標準化活動を長年にわたり実施している IEEE。その下部組織として、エレクトロニクス産業における設計自動化関連の標準化活動を行っている DASC (Design Automation Standards Committee があ

る。その中のワーキンググループのひとつに、P1481 Circuit Delay and Power Calculation (DPC) Working Group があり、本ワーキングのテーマのひとつである配線ばらつきの表現フォーマット(Sensitivity SPEF)策定を行っている。

一方日本では、半導体 MIRAI(Millennium Research for Advanced Information Technology)プロジェクトが、ASRC（産総研次世代半導体研究センター)、ASET（技術研究組合 超先端電子技術開発機構)、Selete（株式会社 半導体先端テクノロジーズ）を中心とする産官学で共同し、半導体最先端技術の「壁」を克服する共通基盤としての役割を担っている。具体的には、半導体の微細加工において、32nm の技術世代以降の LSI の消費電力や処理速度といった基本的な性能を格段に向上させる技術を開発している。また、半導体理工学研究センターSTARC（http://www.starc.or.jp/）では、SoC 設計技術等の先導開発(あすか 2)が行われている。

2.1.4 参加メンバー

主査	田中 正和	パナソニック㈱
副主査	佐方 剛	富士通セミコンダクター㈱
委員	古川 且洋	㈱ジーダット
同	成木 保文	メンター・グラフィックス・ジャパン㈱
同	山中 俊輝	㈱リコー
同	金本 俊幾	ルネサスエレクトロニクス㈱
同	増田 弘生	ルネサスエレクトロニクス㈱
特別委員	奥村 隆昌	㈱半導体理工学研究センター
客員	佐藤 高史	京都大学
客員	橋本 昌宜	大阪大学
オブザーバ	袖 美樹子	ルネサスエレクトロニクス㈱

２．２　EDA 標準化小委員会

２．２．１　EDA 標準化小委員会

（１）発足の背景とミッション

JEITA/EDA 技術専門委員会の標準化活動は、1990 年の EIAJ/EDIF 研究委員会設立に始まり、当初は EDA に関するグローバルな重要課題に対して日本の業界を代表する唯一の機関として、特に設計記述言語の仕様標準化とその啓蒙等に多大な貢献を果たしてきた。

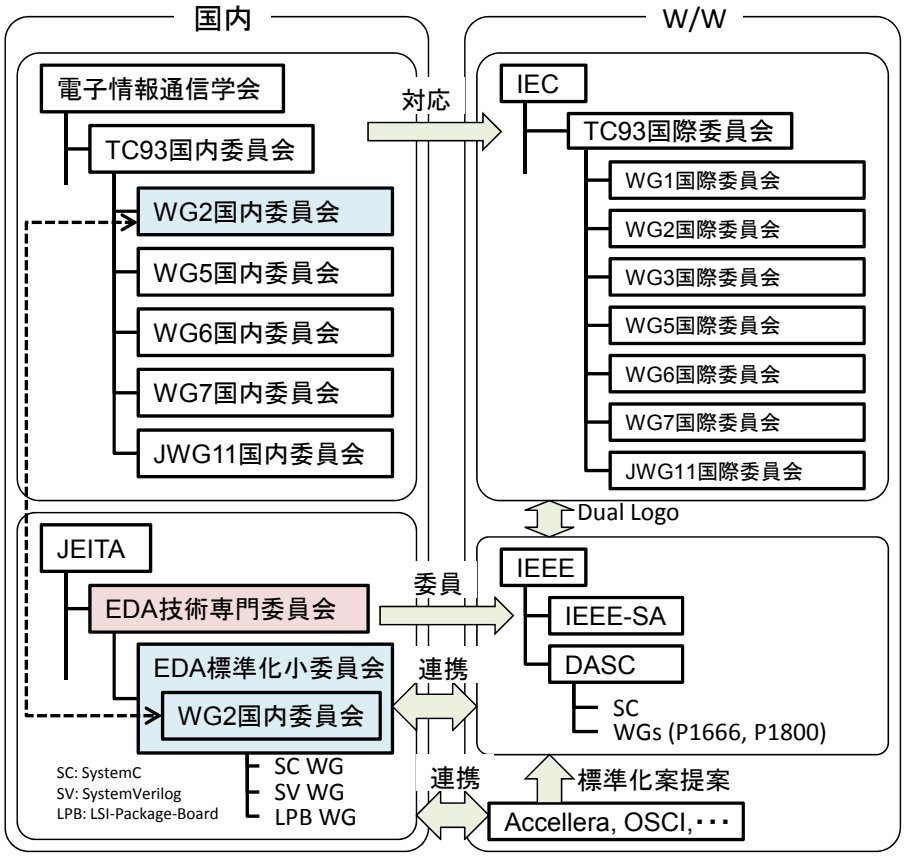
近年、設計記述言語は高度化し、普及が進んだ。しかし、設計生産性の更なる向上および、それを支える EDA ツールの効率的な開発・利用を進めるためには、設計技術言語の国際標準化は依然として重要なテーマである。そこで、標準化関連の活動をより明確に位置づけるため、2000 年 11 月に本小委員会が設立された。

世界的に見れば EDA 関連の標準は IEC（International Electrotechnical Commission）と IEEE（The Institute of Electrical and Electronics Engineers）で議論、制定されてきた。IEC ではデザインオートメーションを議論する TC（Technical Committee）93、IEEE はコンピュータソサイエティの DASC（Design Automation Standards Committee）、および SA（Standards Association）である。これまでは IEEE で定められた標準を IEC でも追認するものも多かった。2003 年より議論は IEEE の DASC/SA のワーキンググループでも、標準の制定は IEC と IEEE で同時にできるようになった（Dual Logo）。

国内では IEC の対応機関は、日本工業標準調査会（JISC：Japanese Industrial Standards Committee）である。また、TC 毎に国内委員会があり、電子情報通信学会や JEITA 内に組織化されている。TC93 とハードウェア設計記述言語関連のワーキンググループ（WG2）の国内委員会は電子情報通信学会にある。

本小委員会は IEC/TC93/WG2 国内委員会を兼ねて活動するという協調体制を 2002 年度に確立した（図-1 参照）。その結果、EDA 標準化小委員会の委員が IEC/TC93/WG2 の各種標準化提案を直接審議することができるようになった。

2003 年度には、SystemC および SystemVerilog の標準化を業界として検討・推進する目的で、それぞれワーキンググループを発足させた。2007 年度には、CPF（Common Power Format）と UPF（Unified Power Format）の二つの Power Format の標準化案の議論と統一を目的に、検討ワーキンググループを発足させた。SystemC は、ますます重要性が認識されているシステムレベルの設計言語のひとつであり、SystemVerilog は IEEE1364（Verilog HDL）の後継・検証技術の拡張である。CPF/UPF の Power Format は、主にシステム LSI の低消費電力化設計の効率化を目的とした設計言語である。これらワーキンググループは、日本の標準化組織として、海外の関連団体と連携し、言語仕様の専門的な技術検討と改善提案を通じて、標準化へ貢献すること目指して活動を行っている。Power Format 検討ワーキンググループは目的を達成したため、2010 年 3 月に解散した。



図－１　EDA 標準化小委員会と他の標準化組織との関係

以上のような発足の背景の中で、2002 年度に本小委員会は、そのミッションを以下のように規定し、活動に取り組んでいる。

目的：

EDA 標準化小委員会（以下本小委員会）は EDA（Electronic Design Automation）関連技術の標準化動向の調査、標準化の推進、標準の策定、標準案の調査、標準の保守・改定、などを推進し、もって国内外の関連業界の発展に寄与することを目的とする。

活動内容：

本小委員会は EDA および関連技術の標準化に関して、

・内外の動向を調査、検討し、

・技術および関連業界の発展に資する提案の必要性を模索し、

・必要かつ可能な場合には、関係機関に対して提案を行い、

・内外の標準化関連機関との連携・協調・協力を推進し、

・特に、デザインオートメーション/設計記述言語（TC93/WG2）WG の活動を支援し、

・また広報活動を行う。

（2）２００９年度 EDA 標準化小委員会メンバー（2011 年 3 月現在 敬称略）

主査	今井 浩史	（株）東芝
副主査	河村 薫	富士通セミコンダクター（株）
副主査	小島　智	NEC システムテクノロジー（株） （TC93/WG2 ココンベナ、IEEE-SA/DASC 委員）
委員	大芝 克幸	ソニー（株）
委員	太田 光保	パナソニック（株）
委員	吉田 正昭	ルネサスエレクトロニクス(株)
特別委員	相京　隆	（株）半導体理工学研究センター （DFT エクスパート）
特別委員	田中 正和	パナソニック（株） （ナノ世代物理設計 WG 主査）
特別委員	中森 勉	富士通セミコンダクター（株） （Power Format エクスパート）
特別委員	福場 義憲	（株）東芝 （LPB 相互設計 WG 主査）
特別委員	浜口 加寿美	パナソニック（株） （SystemVerilog WG 主査）
特別委員	星野　民夫	（株）アプリスター （HDL エクスパート）
特別委員	石河久美子	富士通マイクロソリューションズ（株） （DCL エクスパート）
客員	今井　正治	大阪大学
客員	神戸　尚志	近畿大学 （TC93　国内委員長）

（3）２０１０年度活動 国内活動

EDA標準化小委員会としては、年5回の会合を行い、傘下のSystemCワーキンググループ、LPB相互設計ワーキンググループの活動状況の確認、標準化関連課題の議論を行った。

今年度は以下の活動を行った。

①LSI、パッケージ、PCB相互設計ワーキンググループ発足

LSI、パッケージ、PCBを相互的に扱う設計環境（以下、LPB相互設計環境）は重要であるが課題が多く、その課題は各社で共通する部分が多いという認識のもと、2009年度に準備ワーキンググループで準備を進め，正式にLPB相互設計ワーキンググループを立ち上げた。

②SystemVerilogワーキンググループ再開

言語標準化に関しては、IEEEにおいてSystemVerilog標準の改訂が開始されたため、1月にSystemVerilogワーキンググループを再開した。

③設計言語俯瞰図メンテナンス

2009年度に設計言語標準間の関係把握を用意とするために、記述対象要素に対する設計フローと言語標準の関係を示す設計言語俯瞰図を作成し、IEC/TC93国際会議で紹介した。その場で何件かのフィードバックを頂き、本年度はこのフィードバックを参考に、設計言語俯瞰図のメンテナンスを行った。また、IEC/TC92/WG2国際会議で設計言語俯瞰図をIEC Technical Reportとして発行することを提案し、承認された。本アニュアルレポートに設計言語俯瞰図のTechnical Reportの素案を報告する。

（4）２０１０年度活動 国際活動

関連する標準化関連の組織・団体との連携として、IEC/TC93国際会議への参加、JEITA-IEEE/DASC情報交換会の開催などを通じて、活発な交流を行った。

①IEEE DASC 情報交換

2010年度はIEE DASC会議に4回参加。JEITA EDA技術専門委員会の状況報告も実施。2回は米国開催で、小島副主査を派遣。2回は日本で開催して頂いた。7月開催の場ではP1666 WG、1月開催の場ではP1800 WGの詳細な状況について情報交換を実施した。

DAC Meeting　6月16日　アナハイム
SystemC Japan　7月1日　新横浜
EDSFair　1月28日　横浜
DVCon Meeting　3月2日　サンホセ

②IEC/TC93/WG2 国際連携

小島副主査（WG2 コ・コンビナ）、下記 2 回の IEC/TC93/WG2 会議に派遣し、国際標準化活動を推進するとともに、EDA 技術専門委員会（日本側）の意見反映を実施した。

IEC/TC93 国際会議　10 月 8 日　シアトル
DVCon　3 月 2 日　サンホセ

２．２．２ IEEE/DASC（電気電子学会/設計自動化標準委員会）・IEEE-SA

（1） 活動の概要

IEEE は米国に本部を置く電気、電子、情報、などの国際的な学会である。また、この分野の標準化活動を長年にわたり、しかも広範囲に実施している。DASC、SA は IEEE の下部組織として、エレクトロニクス産業における設計自動化関連の標準化活動を行っている。

活動の中心は、標準設計記述言語（HDL：Hardware Description Language）の VHDL と Verilog HDL に関連する設計と検証であり、タイミング情報、論理合成、算術関数とテストの標準化に注力している。これら設計言語に関連して、システムレベルまで適用範囲を拡大して、Analog Mixed Signal、ソフトウェアとハードウェア協調設計等の拡張の標準化を検討してい

る。2005 年には SystemC と SystemVerilog という高位設計技術言語、設計と検証を統合した記述言語の標準化作業が完了した。

（２）JEITA/EDA 技術専門委員会との関連

これまでは EDA 技術専門委員会は IEEE/DASC のメンバーとして関連する WG に参加し、標準化案に日本の意見を反映してきた。2004 年 12 月には IEEE-SA の正式メンバーにもなり、IEEE の標準化活動に、ドラフトレビュー・標準化案の改善の提案・投票を通じて積極的に参加している。

特に、今年度は、SystemC Japan に合わせて 7 月 1 日に、EDSFair2011 に合わせて 2011 年 1 月 28 日に IEEE/DASC との情報交換会を 2 回、日本で開催した。7 月 1 日開催の会議には、米国からは Stan Krolikoski 氏（DASC 委員長、SystemC WG 委員長)、本小委員会からは、今井主査（SystemC WG 主査兼任)、小島副主査、SystemC WG メンバー出席した。1 月 28 日開催の会議には、Stan Krolikoski 氏（DASC 委員長、SystemC WG 委員長)、Dennis Brophy 氏（SystemC WG セクレタリ)、本小委員会関係からは、今井主査（SystemC WG 主査兼任)、小島副主査、神戸客員、浜口 SystemVerilog WG 主査、田中ナノ世代物理設計 WG 主査が出席した。また、国外からの電話会議出席者も多数あり、盛会であった。会議において、EDA 技術専門委員会からは、活動状況の紹介と、SystemC 及び SystemVerilog の状況確認を行った。

２．２．３ IEC/TC93 （国際電気標準会議/デザインオートメーション）

（１）活動の概要

IEC は 1906 年に設立された国際標準化機関であり、本年が 104 年目あたる。設計自動化を取り扱う IEC/TC93 は 1992 年に設立された。TC93 の全体会議は毎年開催されており、スイス、英、仏、米、デンマーク、日、英、米、独、伊と開催されてきた。最近は、2003 年 11 月・2004 年 10 月 米国・Piscataway、2005 年 9 月 日本・京都（奈良)、2006 年 9 月 ドイツ・ベルリン、2007 年 9 月米国・ガイザーズバーグ、2008 年 10 月シンガポール、2009 年 9 月京都での開催があり、本年度は、10 月に米国シアトルで開催された。

（２）TC93 の組織と参加国

IEC のメンバー資格には、P（Participating） と O（Observer） の二種類があるが、P メンバーが 4 カ国、O メンバーが 20 カ国である。P メンバーとしては、日本、中国、米国、韓国が登録されており、O メンバーとして、オーストラリア、ベルギー、チェコ共和国、デンマーク、エジプト、フィンランド、フランス、ドイツ、ハンガリ、インド、アイルランド、イタリア、オランダ、ロシア、セルビア、シンガポール、スペイン、スウェーデン、ウクライナ、イギリスが登録されている。会議国は日本（国際議長：唐津氏)、幹事国は米国（国際幹事：Victor Berman 氏）が担当している。

（３）TC93 の組織とワーキンググループ（WG）

TC93 は７つの WG/JWG から構成されている。　特に、WG2、WG3、WG6 及び、WG7 は日本から提案も含め積極的な貢献をしてきた。今までの各 WG の主な活動を示す。

・WG1：モデルのハーモナイゼーション：

（a）STEP Electrical（ISO 規格）と EDA 標準の整合性の検討、(b）EDIF と AP-210 との整合性の検討。(c）言語間の Interoperability の検討

・WG2：部品・回路・システム記述言語：

(a) 設計検証記述言語 SystemVerilog とシステム記述言語 SystemC の言語仕様拡張と利用普及のためのモデル化を検討。(b）国際規格のメンテナンス（VHDL、VHDL-AMS、PSL、SDF、DCL/SPEF、IBIS 等）

・WG5：規格適合性（コンフォーマンス）テストの具体的事案の議論。

・WG6：再利用可能部品ライブラリ

日･米･欧の各プロジェクト間の仕様整合と連携の検討、日本からは JEITA/ECALS プロジェクトの成果を提案している。IBIS も話題に取り上げられている。最近は電子カタログの流通に関する規格案が議論の中心となっている。

・WG7：システムテスト記述言語、ATML（Automatic Test Markup Language）の検討。

・JWG11：記述の XML 化の流れへの取り組み方の議論。

（４）TC93 国内委員会と主要メンバー（2011 年 3 月現在。敬称略）

・TC93 国際会議

議長：唐津 治夢（SRI インターナショナル）

国内専門委員会

委員長：神戸 尚志（近畿大学） *

幹事： 古井 芳春（シルバコ・ジャパン）

委員：今井 浩史（東芝）、小島 智（NEC システムテクノロジー）、
星野 民夫（アプリスター）、高橋 満、山下 寛巳（E-SML）、
小泉 徹（JPCA）、柴田 明一（JPCA）

・WG1：(設計データハーモナイゼーション)

主査：古井 芳春（シルバコ・ジャパン、国際ココンビナ）

・WG2：(ハードウェア設計記述言語)

主査：今井 浩史（東芝） *

国際ココンベナ： 小島 智（NEC システムテクノロジー） *

委員：標準化小委員会と同じ

・WG5：(テスト、コンフォーマンス技術)

主査： 神戸 尚志（近畿大学、国際ココンビナ）

・WG6：（再利用可能部品ライブラリ）

主査： 高橋 満（国際ココンビナ）

・WG7：(システムテスト記述言語)

主査： 山下 寛巳（E-SML）

委員：唐津 治夢（SRI インターナショナル、国際ココンビナ）

・JWG11：（電子基板設計）小泉 徹（JPCA)、柴田 明一（JPCA）

*印は EDA 技術専門委員会からの参加者

（５）TC93 シアトルの報告

2010 年の国際会議は、10 月に米国シアトルで開催された。会議には、日本、米国、韓国、中国の計 4 カ国から参加があり、TC93 プレナリー会議、WG1、WG2、WG3、 WG6、WG7、JWG11 の 7 会合が開催された。

WG2 では、小島副主査から国際ココンベナとして、Dual logo である IEEE 規格と IEC 規格の関係を整理体系化のうえ、2011 年以降に予定されているデュアルロゴ案件を確認し、今後のメンテナンス計画を策定した。

また小島副主査から EDA 技術標準化小委員会で作成した設計言語俯瞰図を IEC Technical Report として発行することを提案し、承認された。

（６） IEC 規格投票について

本年は、次の 2 件が IEEE 標準からのデュアルロゴとして、FDIS（Final Draft International Standard）投票が行われた。両標準に対して、標準化小委員会内で協議のうえ、賛成票を投じた。結果として、両標準ともに IEC 規格として採択された。

１）IEC 61691-1-2 Ed 1.0: VHDL Language Reference Manual

２）IEC 62530-1 Ed 1.0: System verilog - Unified hardware design, specification, and verification language

２．２．４ SystemC ワーキンググループ報告

(1) 背景

ハードウェア記述言語によるシステム LSI の設計は、VHDL（IEEE 1076）や Verilog-HDL（IEEE 1364）の標準化への JEITA の貢献とともに広く普及して、産業界で活用されている。一方、半導体の微細化技術は開発がさらに加速され、既に 1000 万ゲート規模の LSI が開発されるに至り、さらに抽象度の高いレベルからの設計が必須となってきている。1990 年代半ばより複数のシステムレベル設計言語の提案が行われ、標準化推進団体が結成されたものもあった。この中で、C++言語の上に構築された SystemC は広く半導体メーカ、システムメーカ、EDA ベンダーの賛同を得て、Open SystemC Initiative（OSCI） が結成され、標準化のための言語仕様の策定と整備が進められてきた。

システムレベル設計言語としての要件を備えた SystemC 2.0 のリファレンスシミュレータがまず 2001 年 10 月にリリースされ、その後 2003 年 5 月に言語参照マニュアル（Language Reference Manual、 以下 LRM）が一般公開された。この LRM が 2004 年 11 月に OSCI より IEEE に移管され、IEEE P1666 として正式な標準化プロセスが開始された。並行して OSCI にて開発されていた SystemC 2.1 の言語拡張仕様も IEEE P1666 標準の一部として追加移管され、2005 年 12 月に IEEE Std. 1666-2005 として SystemC のコア言語部分が標準化された。

2009 年 9 月に IEEE DASC に SystemC 標準のメインテナンスと TLM（Transaction Level Modeling）の標準化を審議する P1666 SystemC WG が組織され、2011 年 1 月に WG 内の投票が終了している。2011 年内には IEEE Std. 1666-2010 として標準化が計画されている。

(2) 目的

上記のように標準化が進められた SystemC は、SoC（System on Chip）の開発のためのシステムレベル記述言語として既に設計や検証に幅広く使われるようになり、欠くことのできない言語となってきている。設計言語は設計の基本となるもので、この標準化策定に早くから関わることは、産業界にとって次世代の設計手法を構築する上で非常に重要なことである。

本ワーキンググループは 2003 年 10 月に設置され、日本国内における唯一の SystemC の標準化関連組織として、IEEE P1666 で進められる SystemC 標準作業に対して日本の産業界として意見を述べ、国内事情・要求事項を取り込んだ形で国際標準化に貢献していく。また、SystemC に関連した調査結果をアニュアルレポートやユーザ・フォーラム等で積極的に情報発信を行うことで、SystemC を利用した設計手法の国内普及を図り、ひいては日本の産業界の国際競争力を高めることを目指す。

(3) これまでの成果

2003年10月に発足した後、これまでに次のような成果をあげた。

① SystemC 標準化活動

- 2003 年度には OSCI より 2003 年 5 月に一般公開された LRM についてレビューを行い、問題点を 62 件抽出し（うち 46 件については 2003 年度アニュアルレポートに一覧を記載）、IEEE 並びに OSCI に報告した。

- 2004 年度には IEEE P1666 のメンバーとして活動を行い、IEEE 版の LRM(Draft)をレビューし、43 件の問題点を IEEE に報告した。
- 2005 年度には SystemC 2.1 が追加された IEEE P1666 版 LRM についてレビューを行い、19件の問題点を抽出し IEEE に報告した。2005 年 12 月 5 日に IEEE Std. 1666-2005 として SystemC の基本言語部分の標準化が完了した。プレスリリースも発行され、EDA-TC/JEITA としてもコメントを掲載した。
- 2006 年度には OSCI よりリリースされた TLM（トランザクションレベルモデリング）2.0 ドラフト 1 、及び合成サブセットドラフトについてレビューを行い、問題点や要望事項をそれぞれ 4 件、57 件 OSCI に報告した。
- 2007 年度には OSCI よりリリースされた TLM 要求仕様、用語集について分析を行い、また 11 月にリリースされた TLM 2.0 ドラフト 2 についてレビューを行い、問題点や要望事項を 10 件 OSCI に報告した。
- 2008 年度には OSCI よりリリースされた TLM 2.0 正式版のユーザーズマニュアルについてレビューを行い、問題点や要望事項を 7 件 OSCI に報告した。また、ユーザーズマニュアルの抄訳を作成し、2008 年度アニュアルレポートにて公開した。
- 2009 年度には TLM 2.0 LRM のレビューを実施し、問題点や要望事項を IEEE P1666 WG へフィードバックした。
- 2009 年度には OSCI より公開された合成サブセット Draft1.3 のレビューを実施し、OSCI へフィードバックした。
- 2010 年度は、IEEE P1666 WG に投票権のあるメンバーとして言語仕様の更新と TLM-2.0 の標準化に参加した。LRM のレビューを行い、問題点や要望事項を 15 件報告した。2011 年 1 月には P1666 WG 内で賛成投票を行った。

② SystemC 技術調査

- 2004 年度には、過去5年間に一般に公開されている SystemC 関連の論文や発表資料等 50 件の調査を行い、報告書を作成した。
- 2005 年度には、SystemC 2.1、TLM(トランザクションレベルモデリング)、合成サブセットのテーマを定め、それぞれ分科会形式で掘り下げた調査を行った。
- 2006年度にはTLMに関する動向調査を実施し、結果をSystemCユーザ・フォーラム2007にて公表。欧州ユーザと比較し、国内ユーザは低抽象度のモデルを利用する傾向が高いことを掴んだ。TLMの標準化の遅れにより再利用性があまり高くないことから、RTL設計に近いレベルでの利用に留まっていると予想された。欧州では標準化を待たずに社内でTLM標準化を行い、一丸となって利用を進めているようであった。
- SystemCを用いた高位合成を効果的に行うためには記述スタイルの標準化が望ましいため、スタイルガイドの骨子を検討し、「合成スタイルガイド構成要件」として公開した。
- 2007年度にはSystemCを用いた推奨設計メソドロジについて検討し、合成編につい

て審議を完了し、2007年度アニュアルレポートにて公開した。

- 2008年度には引き続きSystemCを用いた推奨設計メソドロジについて検討し、2007年度に作成した合成編以外の部分について審議を完了し、2008年度アニュアルレポートにて公開した。

③ SystemC 普及活動

- 2004 年度より、それまでの OSCI から引き継いで JEITA EDA 技術専門委員会の主催で SystemC ユーザ・フォーラムを開催してきた。
- 2005 年 1 月 27 日に SystemC ユーザ・フォーラム 2005 を開催。受講料は無料。定員 200 名のところ 250 名弱の聴講者が訪れ、立ち見が出るほどの盛況であった。
- 2006 年 1 月 27 日に SystemC ユーザ・フォーラム 2006 を開催。この回より受講料を徴収することにした。定員 200 名のところ、事前予約では満員であったが、実際に会場に訪れた聴講者は 172 名で前年比 30%減となった。 また、アンケートは 134 名の方に記入いただけた。
- 2007 年 1 月 26 日に SystemC ユーザ・フォーラム 2007 を開催。前回より値上げした影響か、聴講者は前年比 14%減の 148 名と減少した。また、アンケートは 132 名の方に記入いただけた。
- 2008 年 1 月 25 日に SystemC ユーザ・フォーラム 2008 を開催。この回は TLM に関するトピックを多くしたせいか、聴講者は前年比 12%増の 166 名と増加した。 また、アンケートは 144 名の方に記入いただけた。
- 2009 年 1 月 23 日に SystemC ユーザ・フォーラム 2009 を開催。この回は聴講者が前年比 26%減の 123 名と大幅に減少した。 TLM 2.0 に関しては 2008 年夏に山場を越えており、SystemC そのものにおける大きな変化がなかった点と、景気の悪化もひとつの要因と考えられる。また、アンケートは 113 名の方に記入いただけた。
- 2009 年度は、SystemC ユーザ・フォーラムは開催しなかったが、7 月に開催された SystemC Japan の場をお借りしてアンケート調査を実施した。
- 2010 年 7 月に開催された SystemC Japan 2010 において SystemC ワーキンググループの活動を紹介した。また、この場をお借りしてアンケート調査を実施した。調査結果については本アニュアルレポートで報告する。

(4) 参加メンバー

主査	今井 浩史	東芝
副主査	中西 早苗	ルネサスエレクトロニクス
委員	西園寺 修	日本シノプシス
	竹村 和祥	パナソニック
	立岡 真人	富士通セミコンダクター
	旦木 秀和	ソニー
	牧野 潔	メンター
	村上 和隆	リコー
特別委員	吉永 和弘	半導体理工学研究センター
客員	今井 正治	大阪大学

（計 10 名）

2.2.5 SystemVerilog ワーキンググループ

(1) 背景

Verilog HDL(ハードウエア記述言語)は、1995 年に初めて IEEE1364 として標準化がなされ、その 5 年後にディープサブミクロン対応の機能を追加し、IEEE1364-2001 として改訂された。
半導体の微細化技術がさらに進化するなかで、業界では、システム LSI の大規模・複雑化に対応するため、機能検証の質と効率を飛躍的に向上することを目的としたテストベンチやアサーション／プロパティ記述のための検証用言語が実用化された。

SystemVerilog(ハードウエア記述・検証言語)は、①Verilog HDL に比べ「記述量の削減」「曖昧性の排除」を実現する設計用記述構文　②機能検証の質と効率を向上させる検証用言語　の２つの言語体系で構成されており、2005 年に IEEE1800 として標準化がなされた。

その後 2008 年に、２つの言語 Verilog HDL(IEEE1364)と SystemVerilog(IEEE1800)は IEEE1800-2009 として、ひとつの言語に統合された。

旧 SystemVerilog-WG では、2003 年 10 月から 2009 年 3 月末までの 5 年半の間、この 2 回の標準化作業を支援し、ワーキンググループの活動を終了した。

2010 年に改めて、米国 IEEEp1800-WG では、2012 年に IEEE1800-2012 としての標準化をゴールとして SystemVerilog の改訂作業を開始したことを踏まえ、EDA 技術専門委員会は、日本の標準化窓口として、2011 年 1 月に SystemVerilog-WG を再結成した。

(2) 目的

SystemVerilog の言語仕様改訂において、日本の半導体産業界の活用で技術的な課題がないように確認・要請をしていくことで、適用容易性を高め、設計品質の向上、国際的な競争力確保といった結果につなげる。

本ワーキンググループは、業界各社から参加したエキスパートにより、SystemVerilog 言語仕様の技術的検討をおこない、IEEE1800-2012 承認まで国際標準化に貢献すること、SystemVerilog に関する最新情報の収集、日本国内での SystemVerilog の普及推進を図ることを目的としている。

(3) 活動方針とゴール

2012年の改訂は、旧ワーキンググループで詳細な検討・規格化したSystemVerilog(IEEE1800-2009)から大きく変更するものでないため、主要なポイントのみ確認する。適用上の課題がないことを確認したうえで、IEEE-SAの最終投票に参加し、標準化承認を行うことをゴールとする。

(4) 関連機関の動向

IEEEp1800-WG では、2012 年度の SystemVerilog 改訂に向けて、以下の５つのサブワーキングが活動をしている。

① SV-AC: Assertions　② SV-EC: Test benches

③ SV-CC: Foreign APIs　④ SV-DC: Analog Connection

⑤ SV-BC: Design Subset

・今後の主な活動スケジュール

2011年10月　改訂内容の決定

2012年01月　ドラフト策定　＆　投票

2012年07月　投票コメントへの対応版ドラフト策定　＆　投票

2012年12月　IEEE Std.1800 -2012 リリース

・各サブワーキングで議論している主な改訂ポイント

①SV-AC

Real type support in assertions

Output arguments in checkers

Interface formals in checkers

Assertion system functions

Clock inference in sequences

②SV-BC

Add parameterized tasks and functions

Improve macro specification

Improve interface ports and virtual interfaces

Improve design specification and elaboration

Enumeration extension

③SV-CC

Allow C++ classes and SystemVerilog classes to be shared via DPI

Representation of super and this in VPI

Support for Unified Coverage Interoperability Standard

Improve interaction between DPI and VPI

Create VPI data model extensions for Checkers

④SV-EC

Polymorphic behaviour of instantiation

Constraint composition

Legality to assign an interface containing a class declaration to a virtual interface

Aspect Oriented Programming (AOP) features

Dotted names within inlined constraints

⑤SV-DC

Development of a roadmap for enhancements in this area

(5) 2010年度の主な活動内容

2011年1月にSystemVerilog-WGを結成し、第一回会議を1月28日に開催。同日、IEEEp1800-WGと合同Web会議を開催し、今後の活動においてお互いに連携する方針を確認した。また、IEEEp1800-WGの５つのサブワーキンググループの各代表から、現在議論されている主要改訂ポイントのトップ２５項目について説明を受け内容を理解した。

(6) 今後の進め方について

2010年度に続き、2011年度もSystemVerilog-WGの活動を継続する。IEEEp1800-WGの改訂ポイントの議論状況がすべて掲載されているデータベースを定期的に参照し、課題のないことを確認する。

2011年11月のプレ投票に向けて、最終的に改訂内容を確認・議論し、投票に参画する。この投票で出てきたコメントがどのように反映されるかも確認した後に、2012年に最終投票を行う。

(7) 参加メンバ

主　　査	浜　口　加寿美	パナソニック(株)
委　　員	高　嶺　美　夫	ルネサスエレクトロニクス(株)
同	土　屋　丈　彦	㈱東芝
同	竹田津　弘　州	パナソニック(株)
同	下　坂　仁　志	富士通セミコンダクター(株)
同	李　　建　道	メンター・グラフィックス・ジャパン㈱
同	戸　田　　亮	日本シノプシス(合)

以　上

2.2.6 LPB (LSI・パッケージ・ボード)相互設計ワーキンググループ

2.2.6.1 LPB 相互設計ワーキンググループ（本委員会）

2.2.6.1.1 背景と目的

システムの高速化と電源・インターフェースの低電圧化でタイミングやノイズに対するマージンが少なくなってきている。またコスト競争の激化によりコストと性能のバランス設計は益々重要となっている。これまで設計はＬＳＩ・パッケージ・ボード(以下 LPB)それぞれ別々に設計ガイドに従って行われてきた。しかしながら設計マージンの減少に伴いあらかじめ LPB 各所個別の設計ガイドを定めることが難しくなり、LPB 全体で協調して設計ターゲットを決める必要が出てきた。すなわちシステムの設計途上でシミュレーションによって設計指針を決める手法への変革が必要であり、その為には迅速でかつ正確なシミュレーションを行える環境が必要となった。

LPB が相互に協調して設計すべき部分の一例として電源網の設計がある。LPB 上の電源網：Power Delivery Network（以下 PDN）（図 1）はノイズを小さくする為にインピーダンスを下げるように設計をする。LPB 各部に存在するインダクタンス成分とキャパシタンス成分により共振が起こる。共振が発生する周波数ではインピーダンスが極端に大きくなる。すなわち LPB 各部の PDN モデルを正確にシミュレーションに反映しないと全体の PDN の設計が正しく出来ない（図 2）。このシミュレーションを実行するためには LSI の PDN 等価モデル、パッケージやボードの形状情報、構成材料の電気特性、構成部品の等価モデルなど様々な情報を揃えなければならない。この際に LPB の各部分の設計情報の書式や仕様の違い、必要パラメータが共有されていない事などが効率的な設計が出来ない要因となっている。

このように、シミュレーションをとりまくモデルやインターフェースの方法・形式は定まったものが無く、セットアップにかかる時間とコストは膨大であり、LPB が協調する障壁となっている。

JEITA では協調設計環境の効率的な構築を可能とするインターフェース方法を模索する為に LPB 相互設計 WG を設置し活動している。

図1. LPBの電源網(PDN)の等価回路

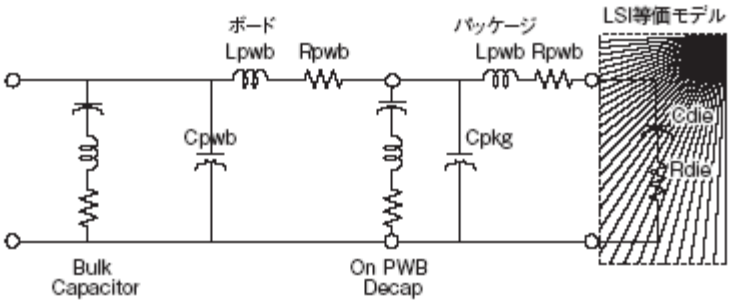
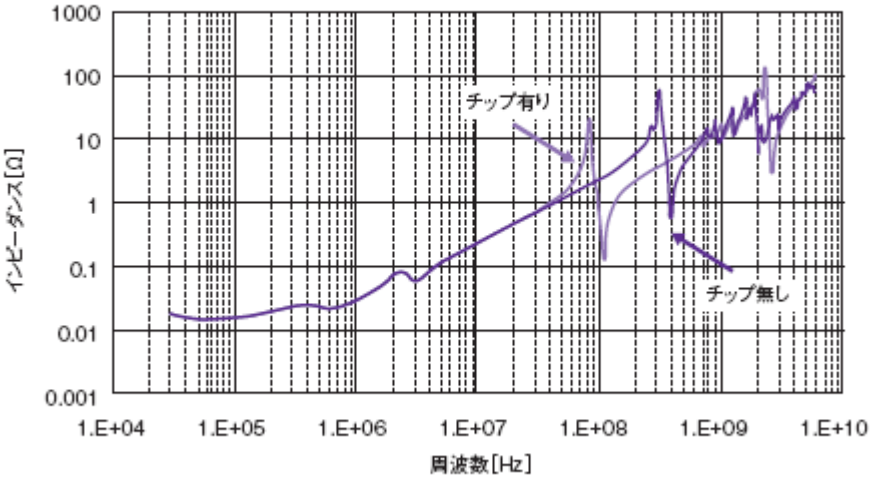


図2. ボード電源インピーダンスにおけるチップ電源モデル有無の影響



2.2.6.1.2 発足・変遷

2009年7月準備WG発足、2010年4月より正式WG

（参加メンバ）１６社／２０名＋特別委員3名　客員１大学／1名

主査：福場義憲／（株）東芝

委員構成は半導体・セット・ＥＤＡ・大学と多様

JEIA-LPB 参加者リスト

お名前	会社・学校	役職
福場義憲	（株）東芝	主査
田中修治	ソニー（株）	副主査
市川浩司	（株）デンソー	委員
大槻隆志	（株）リコー	委員
大坪祐司	（株）図研	委員
金子俊之	（株）トッパンNECサーキットソリューションズ	委員
久島憲司	メンターグラフィックス・ジャパン（株）	委員
楠本学	日本電気（株）	委員
齊藤義行	パナソニック（株）	委員
佐藤厚志	富士通セミコンダクター（株）	委員
永野民雄	ルネサスエレクトロニクス（株）	委員
林靖二	キヤノン（株）	委員
青木考哲	（株）東芝	副委員
岡野資睦	（株）東芝	副委員
小澤要	富士通セミコンダクター（株）	副委員
木本努	ルネサスエレクトロニクス（株）	副委員
古賀一成	（株）図研	副委員
冨島敦史	（株）東芝	副委員
中川裕之	富士通ＶＬＳＩ（株）	副委員
吉田敦史	（株）リコー	副委員
須藤俊夫	芝浦工業大学	客員
板橋裕行	ＡＴＥサービス（株）	特別委員
人見忠明	アパッチデザインソリューションズ（株）	特別委員
渡辺亨	アンシス・ジャパン（株）	特別委員
小島智	ＮＥＣシステムテクノロジー（株）	オブザーバー
吉田正明	ルネサスエレクトロニクス（株）	オブザーバー

2.2.6.1.3 議論概要

設計のフローを模式化すると 3 つの階層にまとめられる。

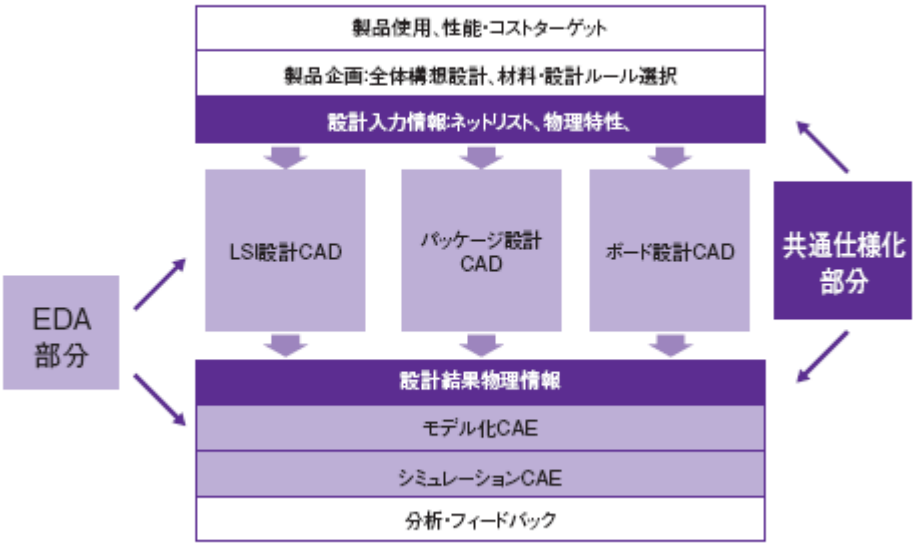
1、製品企画・仕様から LPB 各個の設計ターゲットを決める工程

2、LPB 各個の CAD 設計

3、LPB 設計結果の解析（CAE）・評価・設計へのフィードバック

このうち EDA（CAD／CAE）の入力部分となるところは設計部門・ツール・ベンダーによって仕様が統一されておらず、この部分の共通仕様が模索されている。（図 3）

図3. 相互設計における共通化領域



流通するべき情報としては

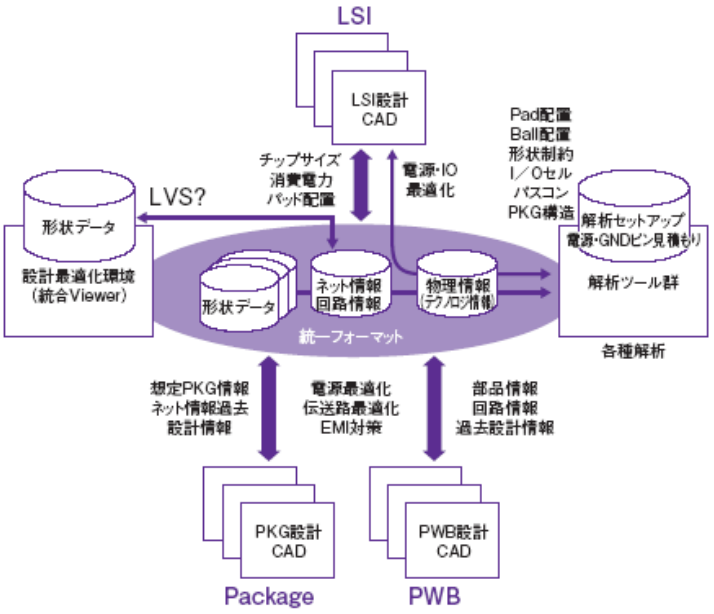
・ネットリスト

・形状データ

・電気・物理・熱特性や制約などのパラメータ

これらがＬＰＢ環境で共通・統合化されて運用される必要がある。（図 4）

図4. 情報・データ共有イメージ



2.2.6.1.4 活動内容

ＷＧ内で３つのサブＷＧを構成して活動をしている。

- インターフェースフォーマット SWG

各設計フェーズ、あるいは解析において必要となる項目と情報の洗い出しと整理、さらに、洗い出した項目や情報が既存フォーマットで表現可能かどうかを調査し、それを元にインターフェースフォーマットとしての共通仕様案を策定することを目的とする。

- 企画・広報 SWG

本 WG の活動内容を公開することを目的として活動しており、SDF や EDS フェア特設ステージの企画などを行う。

- リファレンスフローSWG

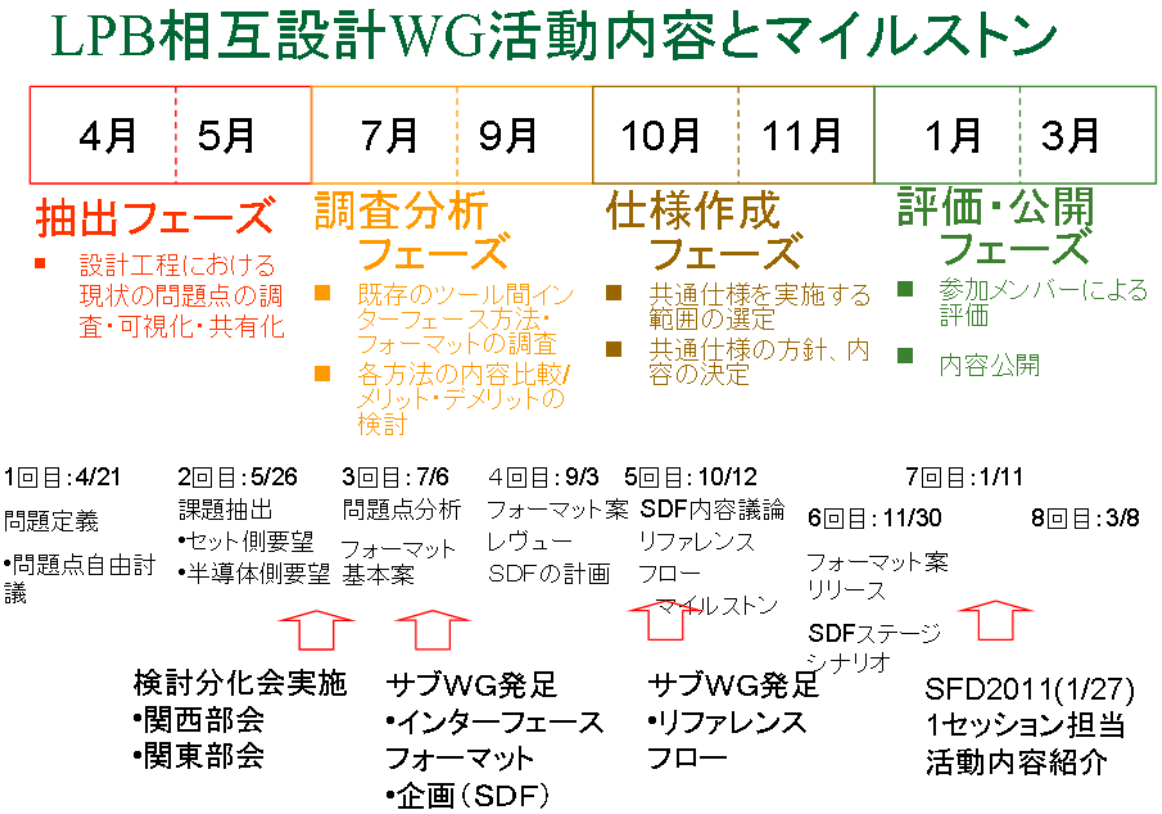
標準的な LPB 相互設計フローを定義し、このフローにおける共通インターフェースフォーマットの活用方法を提案する。

以上、3 つの SWG が協力しあって効率的な LPB 相互設計環境の構築を可能とするインターフェースフォーマット案を検討している。

各ＳＷＧの活動内容と、成果については個々のＳＷＧのアニュアルレポートにまとめる。

2.2.6.1.5 活動記録

2010 年度では計 8 回のＷＧを開催した。



議論のフェーズを次の 4 段階で行った。

１、抽出フェーズ

２、調査分析フェーズ

３、仕様作成フェーズ

４、評価・公開フェーズ

8 回の本会の間に分科会、サブWGの活動を多数行い仕様をまとめ、本会議にて審議決定を行った。

- 2011 年 12 月 27 日にＥＤＳフェア特設ステージを使い、ＳＦＤ２０１１の企画としてパネルディスカッションを行った。また、ＡＳＰ－ＤＡＣとのコラボレーション企画として同日に特設ステージで座談会も行いＬＰＢメンバーが参加した。（広報ＳＷＧアニュアルレポート参照）
- ＥＤＳフェアでは活動内容の紹介として配布パンフレットに投稿した。（別添資料参照）

- 2011 年 2 月 24 日　ＪＥＩＴＡ標準化戦略WG主催の標準化戦略セミナーで活動内容と成果を説明。

2.2.6.1.6 成果

- ＬＰＢインターフェースフォーマットＶ１．０を作成した。詳細はインターフェースフォーマットＳＷＧのアニュアルレポートを参照。

- EDS フェア特設スタジオにてＬＰＢ相互設計WGの理念と活動内容を紹介した。

以降サブワーキンググループごとの活動報告

2.2.6.2 ＬＰＢ相互設計ＷＧインターフェースフォーマットサブワーキンググループ

2.2.6.2.1 目的

LPB(LSI-Package-Board)相互設計ワーキンググループで、LPB を統合的に扱う設計環境の実現に向け、その課題の分析をする中で、LSI、Package、Board の設計・解析で扱うツールが異なり、データフォーマットが異なること、更に、Package、Board の設計・解析環境では、EDA ベンダ間でデータフォーマットが統一されていないため、必要情報が揃わない、設計・解析のセットアップに膨大な時間とリソースがかかる、結果が共用できない、といった課題が明らかになった。

そこで、ツール間のデータフォーマットの統一を検討するサブワーキンググループを立上げ、LPB 全体での接続記述、形状情報、解析のベースとなる物性や設定条件に着目、それらの定義や書式等の検討を行い、提案を作成することにより、より効率的な設計環境の実現に貢献することを目的として活動を行っている。

- 検討するフォーマットの種類、構成、定義範囲の明確化
- 既存フォーマット調査、一覧（メリット・デメリット）の作成
- 用語集（WG での名称、内容定義、LSI/Package/Board での名称例）の作成

2.2.6.2.2 活動内容

2010 年 7 月から活動を開始し、実際に流通可能なフォーマットの選定を目標に、対象とするフォーマットの範囲・定義すべき項目の検討、既存の代表的なフォーマットの調査を行った。必要な定義項目を検討する中で、対象フォーマットを目的別に以下の「ネットリスト」「階層間定義」「設計制約」「形状データ」の４つにまとめた。

- ネットリスト：LSI-PKG-PWB 全体の論理接続仕様を定義する。
- 階層間定義：搭載する部品の配置仕様を定義する。
- 設計制約：LSI-PKG-PWB 全体のテクノロジ、製造性のルールを定義する。
- 図形データ：電気的特性などの解析のための形状データを定義する。

また、LSI、Package、Board 設計では、設計文化が大きく異なり、用いる用語、表現方法が統一されていないため、用語集の作成を行った。

今年度の活動の成果として、以下ついて調査・検討を行い、有用な知見を得た。

ネットリスト

LSI-PKG-PWB 全体の論理接続チェック(LVS)、接続仕様を定義する。将来的には機能検証や形式検証への適用も考慮し、検討した。既存フォーマットを調査した結果、Verilog-HDL（IEEE1364）の書式が最も要求を満たす結果となった。ただし、SPICE 記述でのネットリストへの変換も考慮し記述方法には制限を付ける。また、不足している電源系の記述については、信号として表現し、Verilog-HDL として矛盾のない方法として、それが電源であることが分かる属性をコメントとして追記することとする。

- ポートは「inout」、ネットは「wire」
- コメントのキーワードは「PG_NET」

- 記述例：inout VSS, VSS1; * PG_NET VSS */

階層間定義

PKG、PWB に搭載する部品の配置仕様（部品の端子の物理定義と上位階層での配置情報）を定義する。またネットリストでブラックボックスとして扱う部品のモデル（Spara など）の割付けも行う。既存フォーマットの結果、該当する記述がないため、独自フォーマット(XLM 形式)として作成した。

設計制約

LSI-PKG-PWB 全体のテクノロジ（層定義、L/S などのデザインルール）、製造性のルールを定義する。また、検討当初、解析条件として別フォーマットとして検討していた電気的特性などの解析のための情報も本フォーマットに含めることとした。既存フォーマットの結果、該当する記述がないため、独自フォーマット（XLM 形式）として作成した。

具体的な電的特性などの制約項目は今後の課題とする。

図形データ

PKG、PWB の電気的特性などの解析のための形状データを定義する。解析ツールへの共通・中間ファイルとして用いることを目標に、既存フォーマットを調査した結果、Apache 社の解析用ファイル（PCB/Package Data Exchange File（XFL）Format）が最も要求仕様を満たす結果となった。

成果の詳細は本アニュアルレポートの付録に掲載した。

2.2.6.2.3　　参加メンバー

リーダー	永野　民雄	ルネサスエレクトロニクス㈱
サブリーダー	金子　俊之	㈱トッパン NEC サーキットソリューションズ
メンバー	佐藤　厚志	富士通セミコンダクター㈱
メンバー	青木　孝哲	㈱東芝
メンバー	吉田　敦史	㈱リコー
メンバー	渡辺　亨	アンシス・ジャパン㈱
メンバー	板橋　裕行	ATE サービス㈱
メンバー	本田　裕	ATE サービス㈱
メンバー	人見　忠明	アパッチデザインソリューションズ㈱
メンバー	久島　憲司	メンター・グラフィックス・ジャパン㈱
メンバー	大坪　祐司	㈱図研
メンバー	古賀　一成	㈱図研

2.2.6.3ＬＰＢ相互設計ＷＧ企画・広報サブワーキンググループ

2.2.6.3.1 目的

LPB 協調設計環境に関わる活動内容の広報

2.2.6.3.2 活動内容

「協調設計が日本の製品を変える！」というスローガンのもと、これまでにない規模の JEITA 委員 20 人余りの活動内容を、日々、協調設計における課題にチャレンジしている第一線関係者に向けて、いかにその必要性を認識してもらい、日本の進むべき方向性が１つに向かっていくことが出来るよう、さまざまな方法でその普及に努めることを活動の中心とする。

具体的な活動内容

JEITA の企画運営する EDSFair2011 への出展企画
上記と関連し、併設される ASP-DAC2011(Asia and South Pacific-Design Automation Conference) との座談会企画

2.2.6.3.3 活動実績

- EDSFair2011 に LPB 協調設計の特設ステージ（上記 2-1）を設け、以下のようなパネル議題を設け、90 分に及ぶ熱いディスカッションが行われた。結果として、聴講者 285 名と盛況なステージとなった。

  関連 URL1:
  http://techon.nikkeibp.co.jp/article/NEWS/20110208/189432/?ST=print （注）
  関連 URL2： http://www.edsfair.com/special/stage.html （セッション２、セッション３部分）
  （注）登録制サイトの為、別途 ID,パスワードの設定が必要な場合があります
  上記 URL1 の PDF 化： EDSF2011LPB 広報_part1.pdf

- EDSFair で併設される ASP-DAC(Asia and South Pacific-Design Automation Conference) 2011 とのコラボレーションを座談会（上記 2-2）という形で実施され、夕方 6 時スタートという時間帯にも関わらず、多くの来場者の関心を集めることが出来た

  関連 URL3：
  http://techon.nikkeibp.co.jp/article/NEWS/20110208/189433/?ST=print （注）
  （注）登録制サイトの為、別途 ID,パスワードの設定が必要な場合があります
  上記 URL3 の PDF 化： EDSF2011LPB 広報_part2.pdf

2.2.6.3.4 広報メンバー

リーダー　大坪　祐治　　（株）図研
サブリーダー　久島　憲司　メンター・グラフィックス・ジャパン（株）
メンバー　板橋　裕行　　ATE サービス（株）
メンバー　人見　忠明　　アパッチデザインソリューションズ（株）

最後にこの広報の企画にあたり、司会の日経 BP の小島郁太郎様との折衝など大いなる尽力を頂きました広報リーダー大坪　祐治様に感謝致します。

2.2.6.4　ＬＰＢ相互設計ＷＧリファレンスフローサブワーキンググループ

2.2.6.4.1 目的

リファレンス協調設計フローを定義する。

2.2.6.4.2 活動内容

LPBWG では、LPB 相互設計時の接続記述、形状情報、解析設定条件に着目し、それらの定義や書式等の検討を行い、提案を作成する。

リファレンスフローSWG では、インターフェースフォーマット SWG が策定したフォーマットで設計フローが流れるかの検証を行う。ただし、モデル化やシミュレーションの共通化については、競争領域であるので言及しない。

活動内容は、

LSI ベンダ側、システム側各社のフロー、必要なインターフェースファイルを確認する

LSI 側視点、システム側視点のフローをマージし、リファレンスフローを想定する

インターフェースファイルフォーマットとの整合性を確認し、フィードバックを行う

標準化されたインターフェースファイルフォーマットを使った解析事例や使い方事例については、次年度以降の課題とした。

2.2.6.4.3 活動実績

SWG を 3 回開催し、各社のフローから共通部分を抜き出してリファレンスフローを想定した。

想定フローは、

- 構想設計、実設計、検証の 3 段階に分ける
- 構想設計段階では、LSI-PKG-PCB 間でイタレーションが繰り返される
- 実設計、検証段階は、各社の競争領域になるので、詳細はフロー化せず、割愛する
- 構想設計段階の終了状態は
  - PWB 上の部品配置が決定されている状態(配線は未)
  - LSI のピン配置が決定している状態(Swap は可)

とした。

上記想定フローにおいて、ネットリスト、階層間定義、形状データ、設計制約の 4 つのインターフェースファイルが問題なく使用できるかを確認した。また、インターフェースファイルの各項目が、フローのどの段階の Input/Output なのかについても規定した。なお、フォーマットへのフィードバックは適宜実施している。詳細は添付資料参照。

2.2.6.4.4 参加メンバー

リーダー	中川 祐之	富士通 VLSI（株）
サブリーダー	市川 浩司	(株)デンソー
サブリーダー	林 靖二	キヤノン（株）
メンバー	大槻 隆志	（株）リコー
メンバー	岡野 資睦	（株）東芝

メンバー	楠本 学	日本電気（株）
メンバー	冨島 敦史	（株）東芝

３．各種イベント(主催／協賛)報告

3.関連イベント（主催／協賛）報告

3.1 Electronic Design and Solution Fair 2011 (EDSFair2011)

社団法人 電子情報技術産業協会（JEITA） 半導体部会主催により、2011 年 1 月 27 日（木）～28 日（金）の 2 日間、パシフィコ横浜にて、半導体に関連する設計、製造技術の専門性の高い情報を一堂に集めた展示会「Electronic Design and Solution Fair 2011 (略称:EDSFair2011)」を開催した。

出展社数は、118 社／団体、来場者 8,016 人であった。

今回の EDSFair は、「設計維新！」をキャッチフレーズに、世界最先端技術・サービスの展示・セミナーを開催するとともに、毎回好評の特設ステージでの設計者向けのオープンセッション、国内外のベンチャー企業を集めたゾーン設置や新興ベンダのガイド・ツアー、多彩なコンファレンス等を展開し、広く情報発信を行った。また、今回は、過去来場者アンケートで多かった「業務多忙で来場できない」という声にお応えし、1 日目の開催時間を 20:00 までの延長も行っている。

特設ステージにおいては、今年は「設計維新！」として新たなソリューションの創成の一助となるべく、産学間または技術者間の交流の場として、また、来場者に最先端の情報により深く触れていただく場として、同時開催の Asia and South Pacific Design Automation Conference 2011(ASP-DAC 2011) Designers‘ Forum との連携による LSI・Package・Board 相互設計に関わる共同セッションや、第一線の設計者が「成功へのヒント、失敗の教訓」を語るセッション等を開催した。いずれのセッションも 200 名前後の聴衆を集め、立ち見がでるほどであった。

また、エンジニアが注目する最新技術やトピックスに関する展示をまとめた特別ゾーンとして、普段接することが少ない国内外のベンチャー企業のソリューションを集めた「新興ベンダエリア」、産学官の技術交流を深める大学の研究発表の場となる「ユニバーシティ・プラザ」、さらには、EDA 開発に携わる国内のベンチャー企業が一同に集結し、日本企業ならではの「ものづくり力」を活かした技術や製品をアピールする、「JEVeC ビレッジ」を設置した。特に、新興の企業の見学を希望する来場者に向けては、日本の設計技術・EDA 技術の第一人者が、ツアー・ガイドとしてブースへ同行訪問し、各社の技術紹介・質疑応答を日本語でサポートする「新興ベンダ・ガイド・ツアー」を実施した。各ツアーとも約 30～約 50 名の参加者がおり、訪問した新興ベンダからは非常によい企画と好評であり、ツアー参加者からも新興ベンダへアクセスするきっかけができたと大変好評であった。

3.1.1 EDSFair2010 の概要

(1) 開催期間：2011 年 1 月 27 日（木）～1 月 28 日（金）　2 日間

(2) 場所：パシフィコ横浜（展示ホール、アネックスホール）

(3)主催：社団法人電子情報技術産業協会（JEITA）

協力：Electronic Design Automation Consortium（EDAC）

後援：経済産業省、アメリカ合衆国大使館、外国系半導体商社協会（DAFS)、横浜市（順不同）

協賛：社団法人電子情報通信学会（IEICE)、社団法人情報処理学会（IPSJ)、
社団法人日本電子回路工業会（JPCA）（順不同）

運営：一般社団法人日本エレクトロニクスショー協会（JESA）

(4) 開催概況

① 来場者数：8,016 名（前年 9,300 名）

② 出展者数：118 社/199 小間（113 社/231 小間）

③ 出展者セミナー：82 セッション、延べ聴講者数 2,491 名（84 セッション、延べ聴講者数 2,337 名）

④ スイートルーム：3 社（前年 4 社）

⑤ ユニバーシティ・プラザ：8 大学研究室（前年 9 大学研究室）

⑦日別来場者数

1 月 27 日（木）　※晴れ	1 月 28 日（金）　※晴れ	合計
3,856 名（前回 4,288 名）	4,160 名（前回 5,012 名）	8,016 名(前回 9,300 名)

⑧来場者業種内訳

	2011 年	2010 年	偏差
機器メーカ部門	25.90%	28.20%	-2.30%
半導体・電子部品メーカ部門	43.80%	40.40%	3.40%
ツールベンダー	5.40%	6.10%	-0.70%
設計関連サービス	9.60%	11.00%	-1.40%
商社・営業	5.10%	5.80%	-0.70%
出版・マスコミ	1.30%	1.00%	0.30%
その他	8.90%	7.20%	1.70%
無回答	0.00%	0.30%	-0.30%

⑨過去からの来場者数推移

開催年	初日	2 日目	合 計
2011 年	3,856 名	4,160 名	8,016 名
2010 年	4,288 名	5,012 名	9,300 名
2009 年	3,953 名	5,164 名	9,117 名
2008 年	4,604 名	5,827 名	10,431 名
2007 年	4,956 名	6,180 名	11,136 名
2006 年	5,006 名	5,997 名	11,003 名
2005 年	5,066 名	6,087 名	11,153 名
2004 年	4,764 名	6,023 名	10,787 名
2003 年	5,095 名	6,445 名	11,540 名
2002 年	4,324 名	6,227 名	10,551 名
2001 年	5,048 名	6,839 名	11,887 名

初日の来場者数が前回比 10%減、2 日目が前回比 17%減、合計で総来場者数が前回比 13.8%減少した結果となった。今年度の減少の要因としては、FPGA/PLD Design Conference の CEATEC への移動、統合による半導体メーカー数の減少も要因として考えられる。しかし、長期的にも来場者・出展者ともに減少傾向にあるため、最先端設計ソリューションの情報発信・普及・推進による、関連業界の発展に寄与という目的達成に向けての対応策が必要と考える。

3.1.2 出展カテゴリー

ハードウェア・ソリューション

- システム LSI
- ASIC/ASSP
- MPU/MCU/DSP
- FPGA/PLD デバイス
- その他

ハードウェア開発環境(EDA)

- ハードウェア開発環境(EDA)

EDA/LSI 設計関連ツール

- システムレベル設計(RTL より高位)
- 論理設計(RTL～ネットリスト)
- 論理検証
- アナログ設計・検証
- レイアウト
- レイアウト検証・解析
- LSI 信号解析
- テスト設計(DFT/BIST/ATPG/など)
- DFM 関連(OPC/RET/PSM/LRC/TCAD/など)
- ASIC プロトタイピング
- その他

PCB/SiP 設計関連ツール

- 回路図作成
- アナログ設計・検証
- レイアウト
- SI/PI/EMC 解析
- 電磁界解析
- 熱解析
- その他

ソフトウェア・ソリューション

- 組込み OS
- デバイスドライバ
- ファームウェア
- ミドルウェア
- 仮想開発環境・技術
- その他

LSI テスト、計測器

- LSI テスタ
- PCB テスタ
- 計測器
- その他

IP コア、マクロ、セルライブラリ

- IP コア、マクロ、セルライブラリ

組込みプロセッサ開発環境

- リコンフィギャラブルプロセッサ
- ICE
- デバッカ
- マイコン CASE
- コンパイラ/クロスコンパイラ
- シミュレータ
- ハード・ソフト協調設計環境
- その他

設計サービス関連（LSI/PCB）

- デザインセンタ
- 設計サービス
- 設計コンサルティング
- 試作・製造
- IP 流通サービス
- その他

設計インフラ(WS/PC、ネットワーク)

- 設計インフラ(WS/PC、ネットワーク)

設計データ管理ツール

- 設計データ管理ツール
- その他

マスクメーカ、ファウンダリメーカ

- マスクメーカ、ファウンダリメーカ

大学(研究室)、コンソーシアム

- 大学(研究室)、コンソーシアム

PR 関連

- 出版物

その他

- その他

3.1.3 出展者一覧

（株）アイヴィス
アットデザインリンクス（株）
　Accelicon Technologies, Inc.
アトレンタ（株）
アパッチデザインソリューションズ（株）
（株）アプリスター
　The DINI Group
アルデック・ジャパン（株）
アンシス・ジャパン（株）
EDN Japan
イノテック（株）
　Arteris Inc.
　Avalent Technologies, Inc.
　Calypto Design Systems, Inc.
　Duolog Technologies Ltd.
　Jazz Semiconductor, Subsidiary of Tower
　MoDeCH Inc.
　Target Compiler Technologies N.V.
　Interoperable PDK Alliance
　スプリングソフト（株）
　Synopsys, Inc.
（株）エーイーティー
AWR Japan
エートップテック（株）
カーボン・デザイン・システムズ・ジャパン（株）
（株）開門
兼松エレクトロニクス（株）
カリプト・デザイン・システムズ（株）
（株）クレディスト
　デル（株）
　UnivaUD 日本事務所
CyberTec（株）
　Jasper Design Automation
サイバネットシステム（株）
（株）産業タイムズ社
CM エンジニアリング（株）
JEITA EC センター EDA 標準 WG
（株）スピナカー・システムズ
　CAST 社
　Verific Design Automation 社
スプリングソフト（株）
タナーリサーチジャパン（株）
TEKLATECH A/S
日本イヴ（株）
日本オラクル（株）
日本ケイデンス･デザイン･システムズ社
　イノテック（株）
　アーム（株）
　（株）エッチ・ディー・ラボ
　カリプト・デザイン・システムズ（株）
　CM エンジニアリング（株）
　中部東芝エンジニアリング（株）
　東芝情報システム（株）
　日本システムウエア（株）
　Magillem Design Services（マジレム社）
日本シノプシス合同会社
Berkeley Design Automation, Inc.
パルシックジャパンリミテッド
（株）半導体理工学研究センター（STARC）
PHYSWARE INC
フォルテ・デザイン・システムズ（株）
富士通（株）
プロトタイピング・ジャパン（株）
　IRIS Technologies, Inc.
　KaiSemi LTD.
MunEDA GmbH
メンター・グラフィックス・ジャパン（株）
　グローバルファウンドリーズ・ジャパン（株）
　エス・エム・アイ・シージャパン（株）
　アーム（株）
　富士通セミコンダクター（株）
　日本ナショナルインスツルメンツ（株）
　ザイリンクス（株）

新興ベンダエリア

IC サービス（株）
　　CMSC, Inc.
　　IST, Inc.
ICScape
AVERY DESIGN SYSTEMS
AnSem NV
（株）エイアールテック
　　神戸大学 工学部 永田研究室
　　広島大学 先端集積システム工学研究室
シグナル・プロセス・ロジック（株）
（株）シンコム
　　Micrologic Design Automation
　　ClioSoft Inc
　　Amiq EDA
DOCEA POWER
　　HD Lab
Dorado Design Automation, Inc.
（株）トプスシステムズ
NANGATE, INC.
NextOp Software, Inc.
BEEcube Inc.
Blue Pearl Software
Vennsa Technologies, Inc.
POLYTEDA Software Corporation
REAL INTENT, INC.
Lynguent, Inc.

JEVeC ビレッジ

アートグラフィックス
（株）アストロン
ギガヘルツテクノロジー（株）
　　（株）リキッド・デザイン・システムズ
ケイレックス・テクノロジー（株）
　　（株）Trigence Semiconductor
（株）ジーダット
（株）ジェム・デザイン・テクノロジーズ
（株）数理システム
TOOL（株）
日本 EDA ベンチャー連絡会

ユニバーシティ・プラザ

会津大学　齋藤研究室
愛媛大学　樋上・高橋研究室
大阪大学　今井・武内研究室
九州工業大学　梶原・温研究室
九州工業大学　中村研究室
東海大学　清水尚彦研究室
名古屋大学　組込みリアルタイムシステム研究室
広島大学　アルゴリズム論研究室

50 音順　（※字下げは共同出展）

3.1.4 出展傾向

出展者数は 118 社(新規出展企業 12 社)であった。前年比 4%増(共同出展を含む)ではあったが、出展小間数としては14%減で、EDSFair となってから最も少ない小間数であった。

	出展者数	小間数
2011 年	**118 社**	**199 小間**
2010 年	113 社	231 小間
2009 年	143 社	317 小間
2008 年	169 社	339 小間
2007 年	154 社	348 小間
2006 年	148 社	343 小間
2005 年	119 社	336 小間
2004 年	104 社	306 小間
2003 年	99 社	320 小間
2002 年	91 社	370 小間
2001 年	92 社	366 小間

	2011 年		2010 年	
	社数	小間数	社数	小間数
通常出展	73 社	177小間	91 社	211 小間
新興ベンダ(ブース出展)	26 社	22 小間	22 社	20 小間

3.1.5 出展者セミナー

出展者セミナーは、昨年よりカテゴリー別のトラック制とし、また、事前登録を可能とすることで、聴講者の利便を図っている。1 セッション 45 分間で、30～100 名の適正人数のお客様に向けて集中 PR が行える出展者セミナールームを出展者に提供した。2011 年は 6 会場にて 82 セッションを開催した。また、聴講者総数は 2,491 名で昨年度の 2,337 名を、約 7%上回った。

トラック別内訳	セッション数	27 日聴講者数計	28 日聴講者数計
システム設計・検証	20	87 名	314 名
ロジック＆フィジカル設計/検証	9	70 名	307 名
AMS 設計・検証/PCB	13	210 名	106 名
機能検証/テスト設計	18	223 名	463 名
LowPower/IP/DFM	13	177 名	174 名
フリー	5	63 名	72 名
スペシャル	4	128 名	97 名

3.1.6 特別ゾーン

新興ベンダエリア

国内外新興企業 26 社（昨年比 4 社増）の最新ソリューションが集まり、設計開発者向けに最新情報を紹介した。

出展者：

IC サービス（株）
　　CMSC, Inc.
　　IST, Inc.
ICScape
AVERY DESIGN SYSTEMS
AnSem NV
（株）エイアールテック
　　神戸大学 工学部 永田研究室
　　広島大学 先端集積システム工学研究室
シグナル・プロセス・ロジック（株）
（株）シンコム
　　Micrologic Design Automation
　　ClioSoft Inc
　　Amiq EDA
DOCEA POWER
　　HD Lab
Dorado Design Automation, Inc.
（株）トプスシステムズ
NANGATE, INC.
NextOp Software, Inc.
BEEcube Inc.
Blue Pearl Software
Vennsa Technologies, Inc.
POLYTEDA Software Corporation
REAL INTENT, INC.
Lynguent, Inc.



JEVeC ビレッジ

日本の EDA の発展を目指して設立された「日本 EDA ベンチャー連絡会（JEVeC）」との協力による特別企画。EDA 開発に携わる国内のベンチャー企業が一同に集結し、日本企業ならではの「ものづくり力」を活かした技術や製品をアピールした。

出展者：

アートグラフィックス
（株）アストロン
ギガヘルツテクノロジー（株）
　　（株）リキッド・デザイン・システムズ
ケイレックス・テクノロジー（株）
　　（株）Trigence Semiconductor
（株）ジーダット
（株）ジェム・デザイン・テクノロジーズ
（株）数理システム
TOOL（株）
日本 EDA ベンチャー連絡会

ユニバーシティ・プラザ

産学の交流を促進すると共に、大学研究機関による研究成果を発表する場とする企画である。設計技術に関する研究成果を発表実演した。今回は本プラザへの出展にも出展費用負担をお願いした。

8 大学研究室

- **非同期式回路の設計技術**
  会津大学　コンピュータ理工学部　齋藤研究室
- **VLSI における新しい故障モデルに基づく故障検査法の開発**
  愛媛大学　大学院理工学研究科　電子情報工学専攻　樋上・高橋研究室
- **生体情報センシングシステム**
  大阪大学 今井・武内研究室
- **LSI の低消費電力テスト技術と劣化検知技術**
  九州工業大学　電子情報工学科　梶原・温研究室
- **素子劣化を考慮した回路設計技術**
  九州工業大学　マイクロ化総合技術センター 中村研究室
- **モデルベース LSI 設計**
  東海大学 専門職大学院 組み込み技術研究科　清水尚彦研究室
- **システムレベルにおける高速性能見積りを用いた MPSoC の設計空間探索と性能改善支援**
  名古屋大学　大学院情報科学研究科　組込みリアルタイムシステム研究室
- **多層プリント配線基板設計支援システム MULTI-PRIDE**
  広島大学　大学院工学研究科　アルゴリズム論研究室

3.1.7 新興ベンダ・ガイド・ツアー

日本の設計技術・EDA 技術の第一人者がツアー・ガイドとして、特設ステージで新興企業の主要な技術をご紹介後、ブースへ同行訪問し、各社の技術紹介・質疑応答をサポートする「新興ベンダ・ガイド・ツアー」を実施した。昨年までは、対象のベンダを海外企業に限定していたが、日本の新興ベンダーにも是非来場者に紹介したいソリューションがあることなどから、今年から枠を広げ日本の企業も対象とした。

新興ベンダ・ツアー参加企業：１８社

第 1 回　14:45-15:55	第 2 回　13:00-14:10	第 3 回　14:45-15:55
(株)シンコム	Lynguent Inc.	AVERY DESIGN SYSTEMS
POLYTEDA Software	(株)エイアールテック	Vennsa Technologies, Inc.
Corporation	(株)トプスシステムズ	NextOp Software, Inc.
IC サービス(株)	シグナル・プロセス・ロジック(株)	DOCEA POWER
Dorado Design Automation, Inc.	アートグラフィックス	REAL INTENT, INC.
Interoperable PDK Alliance	(株)ジェム・デザイン・	CM エンジニアリング(株)
TEKLATECH A/S	テクノロジーズ	

EDSFair2011 では、ツアー方式のブース訪問を企画。ツアー・ガイドが引率して海外新興ベンダのブースを訪問、コミュニケーションのサポートを日本語で行うことにより、来場者が新興ベンダの技術を理解しやすいように支援を行った。

新興ベンダ・ガイド・ツアーでは、二人の設計技術のエキスパートがツアー・ガイドを努めた。1 月 28 日は、(株)半導体理工学研究センター(STARC)の開発第 1 部部長代理 村方 正美 氏、29 日は独立行政法人科学技術振興機構イノベーション推進本部 産学基礎基盤推進部 事業計画・調整担当 主任調査員　秋山 俊恭 氏である。二人とも、LSI 設計技術や EDA に精通し、講演経験も豊富なベテランである。

新興ベンダ・ガイド・ツアーは、まず特設ステージにおいて、ツアー・ガイドによりツアーで訪問するベンダの企業紹介が行われ、各ベンダの代表者もステージ上で紹介した。ブース訪問に先立って、ベンダがどのような会社か、特徴とする製品や技術は何かを事前に理解してもらった。　企業紹介のスライドも日本語で準備されていたので、参加者にとっては、ブース訪問前にベンダについてある程度の知識を得ることができた。　ステージでの企業紹介が一通り終わると、ガイド・ツアーの旗のもと、ツアー・ガイドを先頭に紹介されたベンダのブースを訪問した。
各ブースでは、最初そのベンダの代表がパネルや、PC を使って製品や技術をさらに紹介。ツアー・ガイドは、必要に応じて海外ベンダの説明を日本語に訳したり、また Q&A では参加者が聞きたいであろう質問を代わって行ったり、日本語でなされた質問を英語で伝えたり、また、回答を日本語で伝えたりと、ツアー参加者の技術の理解のためのコミュニケーションのサポートを行っていた。
各ツアーは、大まかに設計分野ごとでまとめられ各ツアーとも毎回約 30 名から 50 名の参加者があり、参加者総数 122 名（昨年 75 名）で昨年比約 63%増と、これまでで最も盛況であった。

実施回数：1 月 27 日 1 回、1 月 28 日　2 回
・ツアーA：　31 名
・ツアーB：　37 名
・ツアーC：　54 名
※バーコード読み取り件数



3.1.8 キーノートスピーチ

(株)NTT ドコモ 執行役員 研究開発推進部長 尾上 誠蔵 氏を招き、キーノートスピーチをいただいた。

ご講演の概要

ドコモは 2010 年 12 月に LTE のサービスを開始した。LTE は技術仕様上 300Mbps のデータ速度を実現するもので当初は最大 75Mbps が提供された。携帯電話のデータ需要急増の背景もあり、LTE は世界の多くのオペレータからの支持を得て、携帯電話の実質的な唯一の世界標準として、商用化の動きが活発化している。本講演では、携帯電話市場の状況から LTE に対するドコモの取組みや世界の動向、更に、その後の発展シナリオについて紹介する。

3.1.9 特設ステージ

場　所：展示フロア内　参加無料

1 月 27 日(木)には、最新 EDA 技術を紹介する「システム・デザイン・フォーラム」の特別協力によるセッションを実施した。JEITAEDA 技術専門委員会の活動から、ナノ物理設計セッション、LSI・パッケージ・ボード(LPB)セッションで構成し、設計者の知りたい情報の発信を行った。また、今回は、ASP-DAC(第 16 回アジア南太平洋設計自動化会議)の同時開催に合わせて、ASP-DAC デザイナーズフォーラムとの連携した企画を開催した。1 月 28 日(金)には、昨年、好評であった、第一線で活躍する設計者のセッションやローパワー設計セッションを今年も開催した。



1 月 27 日(木)

■13:15-14:30　セッション1

「システム LSI 設計の今後　～22nm 時代に向けて～」

【モデレータ】【オーガナイザ】

田中 正和 氏

JEITA EDA 技術専門委員会 ナノ世代物理設計 WG 主査

パナソニック(株) システム LSI 事業本部 基盤技術開発センター 要素第一開発グループ 主任技師

【パネリスト】

小林 和淑 氏

京都工芸繊維大学 工芸科学研究科電子システム工学専攻 教授

佐藤 高史 氏

京都大学 情報学研究科通信情報システム専攻 教授

橋本 昌宜 氏

大阪大学 情報科学研究科情報システム工学専攻 准教授

聴講者数約 240 名

「22nm プロセス世代における大規模システム LSI 設計において、ばらつき、低電力、信頼性の設計課題は、それぞれ重要であり、どれも避けて通ることはできません。高性能なシステム LSI 設計のためには、さらなる設計技術と EDA 技術の進歩が必要となります。真の設計課題とは何か？第一線で活躍する 3 人の研究者によるパネルディスカッションで、現状の課題と今後進むべき方向性を議論し明らかにします。」

■16:00-17:30　セッション 2

「設計維新:こうすればできる、LSI・パッケージ・ボードの協調設計」

～協調設計が日本の製品を変える！～

【モデレータ】

小島 郁太郎 氏

日経 BP 社 Tech-On！編集 編集委員

【パネリスト】

岡野 資睦 氏

(株)東芝 デジタルプロダクツ＆ネットワーク社 デジタルプロダクツ開発センター

実装開発センター CAD・CAE 推進担当 主務

金子 俊之 氏

(株)トッパン NEC サーキット ソリューションズ 設計部 マネジャー

林　靖二 氏

キヤノン(株) 生産技術本部 生産技術研究所 実装技術第三研究室 主任研究員

田中 修治 氏

ソニー(株) コンスーマー・プロフェッショナル＆デバイスグループ

半導体事業本部 設計基盤技術部門 ミックスシグナルデザインソリューション部 担当部長

佐藤 厚志 氏

富士通セミコンダクター(株) 開発・製造本部 設計共通技術統括部 第一技術部

大槻 隆志 氏

(株)リコー 電子デバイスカンパニー画像 LSI 開発センターCAD 室 シニアスペシャリスト

板橋 裕行 氏

ATE サービス(株) デザインソリューショングループ ゼネラルマネージャ

久島 憲司 氏

メンター・グラフィックス・ジャパン(株) テクニカルセールス本部

Advanced System Platform グループ シニアプロダクトスペシャリスト

人見 忠明 氏

アパッチデザインソリューションズ(株) シニアアプリケーションエンジニア

大坪 祐司 氏

(株)図研 技術本部 EL セクション S&P グループ グループリーダー

永野 民雄 氏

ルネサスエレクトロニクス(株) 技術開発本部 プラットフォームインテグレーション

統括部 アナログ設計技術開発部 主任技師

聴講者数約 290 名

「激化する競争の荒波を日本の工業製品はどう乗り越えていくのか？日本流の機能・性能と品質を追求しつつコストで対応するにはどうすればいいのか？LSI・パッケージ・ボードの協調 設計に注目が集まるのはそういった切実な思いからでしょう。 本セッションでは、各分野の第一線で奮闘中の技術者が集まり現場の生の声で議論を戦わ せます。チップ、セット、EDA、3 分野のパネルディスカッションを一挙に見られるのは滅多に ない機会、製品の最適化と競争力 UP に日々取り組んでいるマネジメント・技術者

必見です。」

■18:00-18:45　セッション 3

ASP-DAC コラボ企画（ショートスピーチ＆座談会）

LSI・パッケージ・ボード協調設計はどのように行われているか？

～協調設計の事例紹介と普及への課題～

□ショートスピーチ 「パワーインテグリティと協調設計」

須藤 俊夫 氏

芝浦工業大学 機能電子回路研究室 教授

■座談会

【パネリスト】

ASP-DAC 側代表

永田　真 氏

神戸大学大学院 システム情報学研究科 教授

ASP-DAC2011 デザイナーズフォーラム副委員長

松波 敬祐 氏

ソニー(株) 生産本部設計技術センター設計技術戦略課 統括課長 DE(Distinguished Engineer)

福場 義憲 氏(株)

(株) 東芝 セミコンダクター社 アナログ・イメージイング IC 事業部

設計技術開発部 設計インフラ担当 参事

JEITA EDA 技術専門委員会 LPB 相互設計 WG 主査

JEITA・LPB 相互設計 WG 代表

岡野 資睦 氏

(株) 東芝 デジタルプロダクツ＆ネットワーク社デジタルプロダクツ開発センター

実装開発センター CAD・CAE 推進担当 主務

田中 修治 氏

ソニー(株) コンスーマー・プロフェッショナル＆デバイスグループ 半導体事業本部

設計基盤技術部門 ミックスシグナルデザインソリューション部 担当部長

大坪 祐司 氏(株)

図研 技術本部 EL セクション S&P グループ グループリーダー

【オーガナイザ】

永田　真 氏

神戸大学大学院 システム情報学研究科 教授

ASP-DAC2011 デザイナーズフォーラム副委員長

福場 義憲 氏

JEITA EDA 技術専門委員会 LPB 相互設計 WG 主査

(株) 東芝 セミコンダクター社 アナログ・イメージイング IC 事業部

設計技術開発部設計インフラ担当 参事

聴講者数約 170 名

「開催初日、20:00 まで延長された時間を最大限に活かし、同時開催されている第 16 回アジア南太平洋設計自動化会議(ASP-DAC 2011)との LPB/CPB に関するコラボ企画を実施いたします。
LSI・パッケージ・ボードと、チップ・パッケージ・ボード、民間と学会の 2 つの観点から、CPS 協調設計の必要性と普及の問題点、普及させるべき施策などを対談形式で議論いたします。ここでしか聴けない最新動向ですので、ぜひ、ご参加ください。」

1 月 28 日(金)

■10:30-12:00　セッション 4

設計者が語る成功へのヒント、失敗の教訓

【司会】

松澤　昭 氏

東京工業大学 大学院理工学研究科 電子物理工学専攻 教授

【プレゼンテータ】

丸子 健一 氏

ソニー(株) 半導体事業本部 研究開発部門 アナログ回路開発 2 部 1 課

中島 雄二 氏

ルネサスエレクトロニクス(株) 技術開発本部 ミックスドシグナルコア開発統括部
アナログコア開発第一部

吉岡 正人 氏

(株)富士通研究所 デザインソリューション研究部

力野 邦人 氏

エプソントヨコム(株) QI プロジェクト 主事

【オーガナイザ】

松澤　昭 氏

東京工業大学 大学院理工学研究科 電子物理工学専攻 教授

聴講者数約 190 名

「前回、10 周年記念特別企画として実施した、設計者による設計者のための企画。多くの方に聴講いただき、好評をいただきました。今年も、第一線で活躍する設計者のセッションをご用意いたします。
技術開発には成功や失敗がつきものです。成功には、ひらめき、発想の転換やコツなどがあり、失敗もその理由を上手く掴むと成功への足がかりになります。
このセッションでは何が技術開発の成功への鍵だったのか、どのように発想し課題を解決したのか、どのような失敗をし、何を学んだのか、を語っていただき、聴講される皆様に、講演者の経験と発想を伝えます。」

■16:00-17:30　セッション 5

ユーザから見た携帯電話の省エネ技術　～スマートフォンへ向けた取り組み～

【モデレータ】

冨山 宏之 氏

立命館大学 理工学部 教授

【パネリスト】

入田 隆宏 氏

ルネサス モバイル(株) モバイルマルチメディア事業本部 SoC 事業部

モバイル SoC 設計第一部 担当部長

神山　剛 氏

(株)NTTドコモ 先進技術研究所

山下 浩一郎 氏(株)

富士通研究所 プラットフォームテクノロジー研究所 主任研究員

聴講者数約 170 名

「チップベンダから携帯端末メーカ、通信キャリアまでの新進気鋭の若手研究者が携帯電話の省エネルギー技術を解説いたします。設計現場で必要とされる技術から携帯端末ユーザが必要としている技術まで幅広い技術ニーズを紹介いたします。

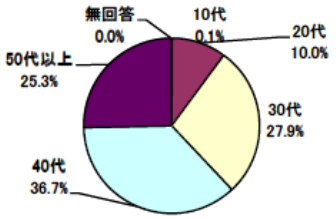
ハードウェアエンジニアだけでなく携帯電話に関わるシステムソフトウェアやアプリケーションエンジニアも必見です。特に、若手エンジニアは同世代のエンジニアの話が聞ける絶好の機会ですので、ぜひ、ご参加ください。」

3.1.10 全来場者入場登録票アンケート回答 集計結果

入場登録票アンケートによる来場者プロファイルを以下に示す。

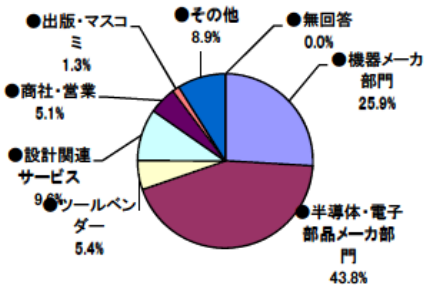
1) ■年代

	2011年	2010年	2009年
10代	0.1%	0.1%	0.1%
20代	10.0%	11.6%	16.2%
30代	27.9%	28.4%	29.2%
40代	36.7%	36.5%	32.7%
50代以上	25.3%	23.2%	18.3%
無回答	0.0%	0.2%	3.5%
合計	100.0%	100.0%	100.0%



2) ■業種

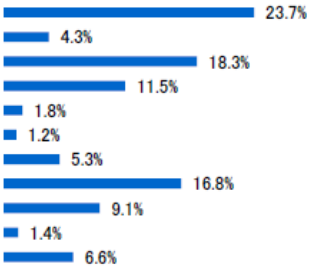
	2011年	2010年	2009年
●機器メーカ部門	25.9%	28.2%	24.7%
●半導体・電子部品メーカ部門	43.8%	40.4%	41.1%
●ツールベンダー	5.4%	6.1%	6.3%
●設計関連サービス	9.6%	11.0%	11.7%
●商社・営業	5.1%	5.8%	4.8%
●出版・マスコミ	1.3%	1.0%	1.5%
●その他	8.9%	7.2%	6.2%
●無回答	0.0%	0.3%	3.7%



※業種詳細

3) ●機器メーカ部門

	2011年	2010年	2009年
●機器メーカ部門	25.9%	28.2%	24.7%
コンピュータ関連機器	23.7%	18.7%	21.9%
ネットワーク関連機器	4.3%	5.5%	5.8%
一般民生機器	18.3%	12.1%	14.7%
画像処理機器	11.5%	11.2%	10.4%
医療機器	1.8%	2.5%	2.2%
アミューズメント	1.2%	0.7%	0.6%
自動車・輸送機器	5.3%	6.1%	3.8%
産業機器(機械・精密機器等)	16.8%	18.4%	17.1%
通信機器	9.1%	10.2%	8.8%
放送機器	1.4%	2.3%	3.2%
その他	6.6%	12.5%	11.5%



4) ●半導体・電子部品メーカ部門

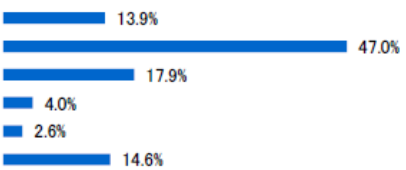
	2011年	2010年	2009年
●半導体・電子部品メーカ部門	43.8%	40.4%	41.1%
システムLSI、ASIC、マイコン、メモリ	84.1%	83.7%	81.9%
FPGA/PLD	2.5%	4.2%	5.2%
ディスプレイ	0.8%	0.8%	1.7%
電子コンポーネント	3.6%	3.8%	3.1%
プリント基板	2.0%	2.1%	2.4%
その他	7.0%	5.5%	5.8%



5) ●ツールベンダー

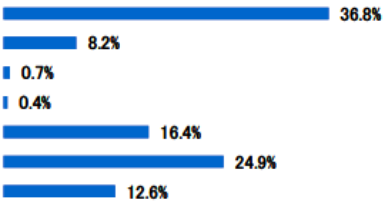
※ツール関連が主要営業品目である商社・代理店も含む

	2011年	2010年	2009年
●ツールベンダー	5.4%	6.1%	6.3%
機能・論理設計ツール	13.9%	16.7%	20.3%
LSI設計ツール	47.0%	51.0%	49.1%
プリント基板設計ツール	17.9%	11.9%	13.4%
マイコンツール	4.0%	3.8%	2.2%
ハードウェア・ボード機器	2.6%	1.9%	3.0%
その他	14.6%	14.8%	12.1%



6) ●設計関連サービス

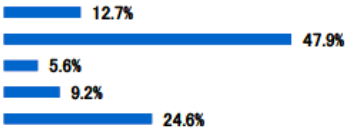
	2011年	2010年	2009年
●設計関連サービス	9.6%	11.0%	11.7%
デザインハウス	36.8%	37.0%	45.6%
IPプロバイダ	8.2%	9.3%	5.3%
IP流通サービス	0.7%	0.0%	0.0%
ネット環境	0.4%	0.3%	0.2%
教育・コンサルタント	16.4%	11.1%	13.0%
ソフト開発	24.9%	29.1%	24.2%
その他	12.6%	13.2%	11.6%



7) ●商社・営業

※ツール関連が主要営業品目である商社・代理店は除く

	2011年	2010年	2009年
●商社・営業	5.1%	5.8%	4.8%
電子機器	12.7%	10.1%	9.1%
半導体	47.9%	52.3%	52.3%
電子部品	5.6%	9.5%	10.2%
ツール	9.2%	12.6%	9.7%
その他	24.6%	15.6%	18.8%



8) ●出版・マスコミ

	2011年	2010年	2009年
●出版・マスコミ	1.3%	1.0%	1.5%

9) ●その他

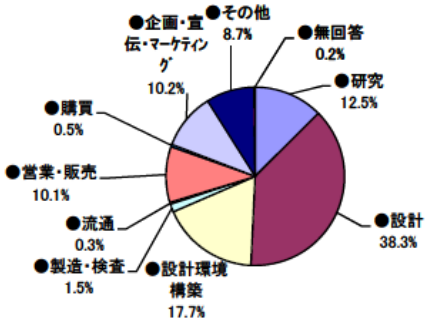
	2011年	2010年	2009年
●その他	8.9%	7.2%	6.2%

10) ●無回答

	2011年	2010年	2009年
●無回答	0.0%	0.3%	3.7%

11) ■職務

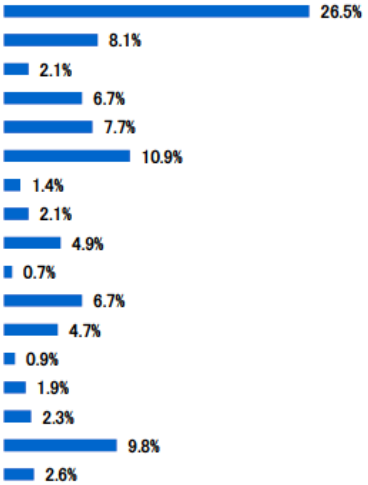
	2011年	2010年	2009年
●研究	11.3%	12.5%	12.1%
●設計	35.8%	38.3%	37.6%
●設計環境構築	23.1%	17.7%	19.6%
●製造・検査	1.8%	1.5%	1.4%
●流通	0.1%	0.3%	0.1%
●営業・販売	8.6%	10.1%	9.5%
●購買	0.3%	0.5%	0.3%
●企画・宣伝・ﾏｰｹﾃｨﾝｸﾞ	10.0%	10.2%	8.3%
●その他	9.0%	8.7%	7.4%
●無回答	0.0%	0.2%	3.7%



※職務詳細

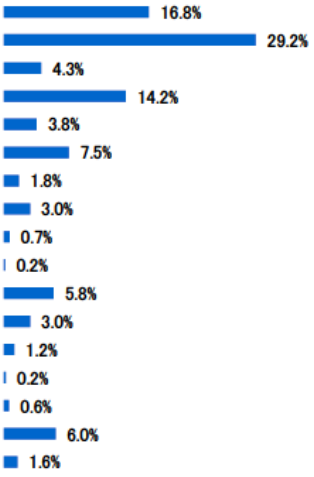
12) ●研究

	2011年	2010年	2009年
●研究	11.3%	12.5%	12.1%
システムレベル	29.4%	26.5%	27.3%
機能(RTL)	7.3%	8.1%	9.7%
論理(ゲートレベル)	3.5%	2.1%	3.4%
レイアウト	6.3%	6.7%	8.1%
テスト	8.2%	7.7%	6.5%
アナログ	11.7%	10.9%	7.2%
カスタム	0.9%	1.4%	1.1%
IPマクロ	0.9%	2.1%	1.8%
リソ/マスク/プロセス/製造	6.3%	4.9%	5.9%
TCAD	0.9%	0.7%	1.4%
FPGA/PLD	3.2%	6.7%	7.0%
PCB	6.0%	4.7%	3.8%
IC Package	2.5%	0.9%	2.7%
SiP	1.9%	1.9%	1.1%
装置実装	0.9%	2.3%	0.7%
ソフトウェア・ファームウェア	10.1%	9.8%	6.1%
無回答	0.0%	2.6%	6.3%



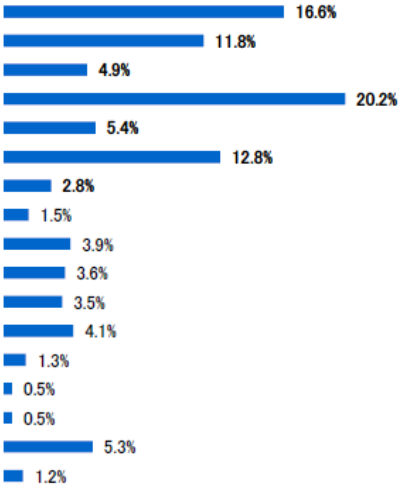
13) ●設計

	2011年	2010年	2009年
●設計	35.8%	38.3%	37.6%
システムレベル	15.1%	16.8%	15.7%
機能(RTL)	30.6%	29.2%	29.7%
論理(ゲートレベル)	5.0%	4.3%	5.8%
レイアウト	13.3%	14.2%	14.4%
テスト	5.1%	3.8%	5.3%
アナログ	11.8%	7.5%	7.6%
カスタム	2.5%	1.8%	1.8%
IPマクロ	3.1%	3.0%	1.7%
リソ/マスク/プロセス/製造	1.1%	0.7%	0.8%
TCAD	0.4%	0.2%	0.4%
FPGA/PLD	3.1%	5.8%	5.3%
PCB	3.1%	3.0%	3.4%
IC Package	1.6%	1.2%	0.7%
SiP	0.6%	0.2%	0.2%
装置実装	0.4%	0.6%	0.8%
ソフトウェア・ファームウェア	3.2%	6.0%	3.8%
無回答	0.0%	1.6%	2.5%



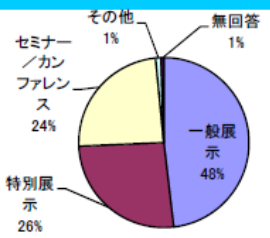
14) ●設計環境構築

	2011年	2010年	2009年
●設計環境構築	23.1%	17.7%	19.6%
システムレベル	13.5%	16.6%	14.6%
機能(RTL)	12.6%	11.8%	12.1%
論理(ゲートレベル)	4.2%	4.9%	6.4%
レイアウト	20.2%	20.2%	21.4%
テスト	9.5%	5.4%	5.4%
アナログ	12.6%	12.8%	10.6%
カスタム	6.4%	2.8%	4.3%
IPマクロ	1.4%	1.5%	1.5%
リソ/マスク/プロセス/製造	3.4%	3.9%	4.3%
TCAD	2.0%	3.6%	1.7%
FPGA/PLD	2.0%	3.5%	2.8%
PCB	5.0%	4.1%	4.9%
IC Package	1.9%	1.3%	1.5%
SiP	0.9%	0.5%	0.6%
装置実装	0.2%	0.5%	0.7%
ソフトウェア・ファームウェア	4.2%	5.3%	4.6%
無回答	0.0%	1.2%	2.6%



15) ■ご来場の目的

	2011年	2010年	2009年
●一般展示	48.6%	48.2%	40.0%
●特別展示	22.7%	26.2%	32.2%
●セミナー／カンファレンス	26.8%	24.1%	24.2%
●その他	1.8%	1.0%	0.5%
●無回答	0.0%	0.6%	3.1%



※ご来場の目的詳細

16) ●展示関連

	2011年	2010年	2009年
●展示関連	48.6%	48.2%	40.0%
展示ブース	86.7%	88.2%	87.1%
スイートデモ	13.3%	11.8%	12.9%



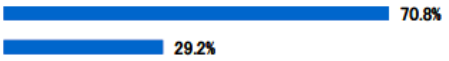
17) ●特別展示

	2011年	2010年	2009年
	22.7%	26.2%	32.2%
電磁界解析・SI/PI テクノロジ・ゾーン			20.8%
FPGAビレッジ		39.8%	26.0%
インドパビリオン			7.9%
新興ベンダ・エリア	30.9%	23.3%	15.1%
JEVeCビレッジ	21.7%	19.2%	16.8%
ユニバーシティ・プラザ	20.5%	17.8%	13.3%
キーノートスピーチ	27.0%		24.9%



18) ●セミナー／カンファレンス

	2011年	2010年	2009年
	26.8%	24.1%	24.2%
出展者セミナー	70.8%	75.8%	52.0%
特設ステージ	29.2%		
FPGA/PLD Design Conference		24.2%	11.3%
システム・デザイン・フォーラム			11.8%

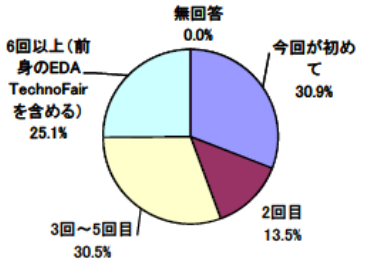


19) ●無回答

	2011年	2010年	2009年
	0.0%	0.6%	3.1%

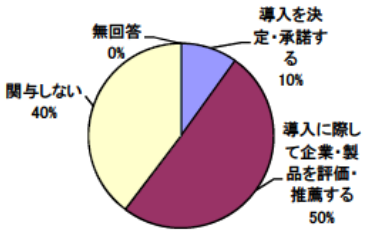
20) ■来場頻度を教えてください

	2011年	2010年	2009年
今回が初めて	30.9%	36.1%	34.3%
2回目	13.5%	13.1%	13.2%
3回～5回目	30.5%	28.7%	28.5%
6回以上（前身のEDA TechnoFairを含める）	25.1%	21.8%	20.1%
無回答	0.0%	0.3%	4.0%
合計	100.0%	100.0%	100.0%



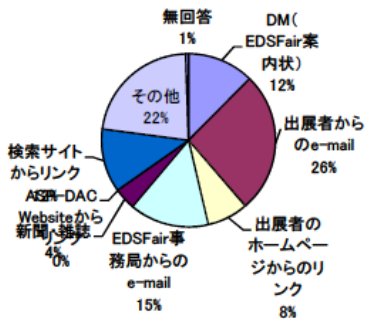
21) ■あなたの製品導入権限について教えてください

	2011年	2010年	2009年
導入を決定・承諾する	9.9%	9.8%	9.4%
導入に際して企業・製品を評価・推薦する	50.4%	47.8%	45.2%
関与しない	39.7%	42.0%	41.1%
無回答	0.0%	0.4%	4.4%
合計	100.0%	100.0%	100.0%



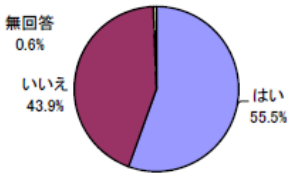
22) ■Electronic Design and Solution Fairをどちらでお知りになりましたか？

	2011年	2010年	2009年
DM（EDSFair案内状）	14.7%	12.4%	11.1%
出展者からのe-mail	26.2%	26.3%	24.4%
出展者のホームページからのリンク	6.9%	7.7%	8.3%
EDSFair事務局からのe-mail	16.5%	14.8%	14.4%
新聞・雑誌	3.6%	4.1%	4.5%
ASP-DAC Websiteからリンク	1.1%		
検索サイトからリンク	12.6%	11.9%	11.3%
その他	18.4%	22.3%	21.6%
無回答	0.0%	0.5%	4.3%
合計	100.0%	100.0%	100.0%



23) ■今後、Electronic Design and Solution Fairからの情報配信を希望しますか？

	2011年	2010年	2009年
はい	59.6%	55.5%	55.6%
いいえ	40.4%	43.9%	39.9%
無回答	0.0%	0.6%	4.6%
合計	100.0%	100.0%	100.0%

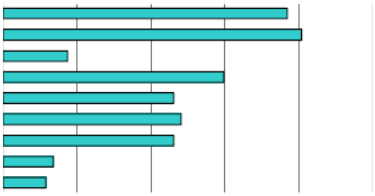


3.1.10 出展者アンケート回答 集計結果

Web 出展社アンケートの回答結果を示す。回答数 48（内新興ブース企業 12）。

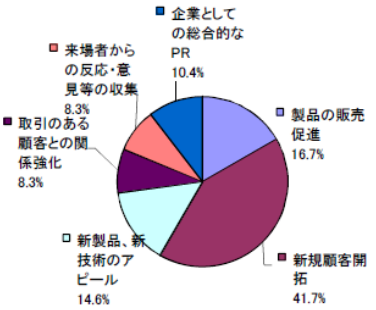
Q1-1. 今回の出展にあたっての目的について該当するものをお選びください。(いくつでも)

	2011年度(内新興ブース企業)		2010年度(参考)
製品の販売促進	19.2%	17.0%	16.3%
新規顧客開拓	20.2%	20.8%	18.7%
商談の成約、契約	4.3%	1.9%	3.6%
新製品、新技術のアピール	14.9%	18.9%	16.3%
取引のある顧客との関係強化	11.5%	7.5%	9.9%
来場者からの反応・意見等の収集	12.0%	15.1%	14.3%
企業としての総合的なPR	11.5%	11.3%	13.5%
競合他社に対し優位性を示す	3.4%	3.8%	4.4%
社員の意識向上、社内活性化	2.9%	3.8%	3.2%
合計	100.0%	100.0%	100.0%



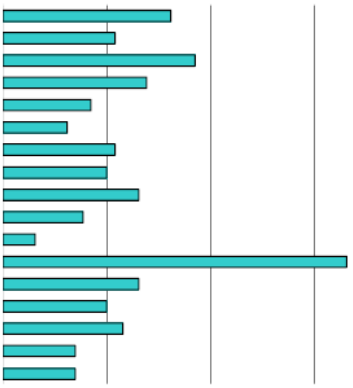
Q1-2. なかでも今回の出展にあたってもっとも重視した目的一箇所をお選びください。

	2011年度(内新興ブース企業)		2010年度(参考)
製品の販売促進	16.7%	8.3%	17.5%
新規顧客開拓	41.7%	41.7%	39.7%
商談の成約、契約	0.0%	0.0%	0.0%
新製品、新技術のアピール	14.6%	16.7%	11.1%
取引のある顧客との関係強化	8.3%	8.3%	6.3%
来場者からの反応・意見等の収集	8.3%	16.7%	9.5%
企業としての総合的なPR	10.4%	8.3%	14.3%
競合他社に対し優位性を示す	0.0%	0.0%	0.0%
社員の意識向上、社内活性化	0.0%	0.0%	0.0%
無回答	0.0%	0.0%	1.6%
合計	100.0%	100.0%	100.0%



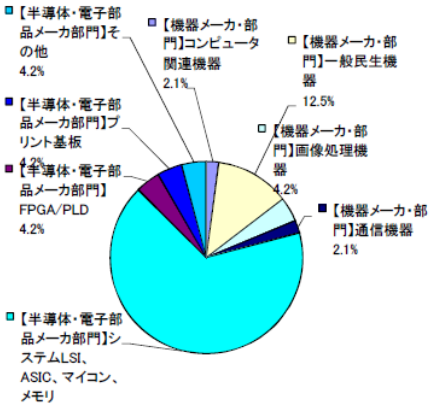
Q2. 出展にあたってターゲットとした来場者の業種について該当するものをお選びください。(いくつでも)

	2011年度(内新興ブース企業)		2010年度(参考)
【機器メーカ・部門】コンピュータ関連機器	8.1%	11.5%	6.7%
【機器メーカ・部門】ネットワーク関連機器	5.4%	5.8%	6.7%
【機器メーカ・部門】一般民生機器	9.2%	13.5%	8.4%
【機器メーカ・部門】画像処理機器	6.9%	9.6%	7.5%
【機器メーカ・部門】医療機器	4.2%	3.8%	5.3%
【機器メーカ・部門】アミューズメント	3.1%	1.9%	4.2%
【機器メーカ・部門】自動車・輸送機器	5.4%	1.9%	5.6%
【機器メーカ・部門】産業機器(機械・精密機器等)	5.0%	5.8%	5.9%
【機器メーカ・部門】通信機器	6.5%	7.7%	7.5%
【機器メーカ・部門】放送機器	3.8%	3.8%	5.0%
【機器メーカ・部門】その他	1.5%	1.9%	2.2%
【半導体・電子部品メーカ部門】システムLSI、ASIC、マイコン、メ	16.5%	21.2%	14.2%
【半導体・電子部品メーカ部門】FPGA/PLD	6.5%	9.6%	7.0%
【半導体・電子部品メーカ部門】ディスプレイ	5.0%	1.9%	3.9%
【半導体・電子部品メーカ部門】電子コンポーネント	5.8%	0.0%	3.6%
【半導体・電子部品メーカ部門】プリント基板	3.5%	0.0%	3.4%
【半導体・電子部品メーカ部門】その他	3.5%	0.0%	2.8%
合計	100.0%	100.0%	100.0%



なかでも出展にあたって最も重視したターゲット一箇所をお選びください。

	2011年度(内新興ブース企業)		2010年度(参考)
【機器メーカ・部門】コンピュータ関連機器	2.1%	0.0%	0.0%
【機器メーカ・部門】ネットワーク関連機器	0.0%	0.0%	0.0%
【機器メーカ・部門】一般民生機器	12.5%	16.7%	7.9%
【機器メーカ・部門】画像処理機器	4.2%	0.0%	3.2%
【機器メーカ・部門】医療機器	0.0%	0.0%	3.2%
【機器メーカ・部門】アミューズメント	0.0%	0.0%	0.0%
【機器メーカ・部門】自動車・輸送機器	0.0%	0.0%	0.0%
【機器メーカ・部門】産業機器(機械・精密機器等)	0.0%	0.0%	3.2%
【機器メーカ・部門】通信機器	2.1%	0.0%	1.6%
【機器メーカ・部門】放送機器	0.0%	0.0%	0.0%
【機器メーカ・部門】その他	0.0%	0.0%	1.6%
【半導体・電子部品メーカ部門】システムLSI、ASIC、マイコン、メ	66.7%	33.3%	54.0%
【半導体・電子部品メーカ部門】FPGA/PLD	4.2%	8.3%	3.2%
【半導体・電子部品メーカ部門】ディスプレイ	0.0%	0.0%	0.0%
【半導体・電子部品メーカ部門】電子コンポーネント	0.0%	0.0%	3.2%
【半導体・電子部品メーカ部門】プリント基板	4.2%	0.0%	4.8%
【半導体・電子部品メーカ部門】その他	4.2%	0.0%	1.6%
無回答	0.0%	41.7%	12.7%
合計	100.0%	100.0%	100.0%



3.1.11 EDSFair2011 実行委員会

実行委員長	パナソニック(株)	太田　光保
副委員長	日本シノプシス合同会社	藤井　浩充
委　　員	イノテック(株)	坂井　仁
委　　員	(株)ジーダット	太田　裕彦
委　　員	ソニー(株)	大芝　克幸
委　　員	TOOL(株)	中根　麻子
委　　員	日本ケイデンス・デザイン・システムズ社	平沢　寿美子
委　　員	富士通セミコンダクター(株)	河村　薫
委　　員	丸紅情報システムズ(株)	半田　信裕
委　　員	メンター・グラフィックス・ジャパン(株)	柴田　多英子
委　　員	ルネサスエレクトロニクス(株)	吉田　正昭
委　　員	(社)電子情報技術産業協会	進藤　淳二

（順不同）

3.1.12 まとめ

今回の EDSFair では、来場者増に向け、開催時間の延長、ASP-DAC デザイナーズフォーラムとの連携や、好評のステージ企画やワインの夕べ等、多くの取り組みを行ってきた。今年から事務局を中心に企画推進を行ったステージ企画であったが、立ち見がでるほどであり概ね好評であった。また、来場者アンケートでの開催時間延長への自由記述コメントや、新興ベンダ・ガイド・ツアーは過去最高の参加者数であったことを見ても、各企画は概ね好評であったと言える。

しかしながら、実績数値としては、出展規模・参加者数ともに昨年比で約 14%の減と、残念な結果になっている。

参加者数・出展者数の減少には、FPGA/PLD Design Conference の併催中止、EDA ベンダーの寡占化による出展規模の減少や、大手半導体メーカーの統合等のような構造的な要因も考えられるが、今後は、開催目的の達成に向けて、新たな施策をとって行く必要があると考える。

3.2　システム・デザイン・フォーラム 2011

3.2.1　概要

最新の EDA 技術の標準化推進、業界内での普及･促進活動の一環として、EDA 技術専門委員会主催による“システム・デザイン・フォーラム”を、EDSFair と同期して継続開催してきた。

しかし、昨年度“システム・デザイン・フォーラム 2010”は、次の２つの理由で開催を見送った。(1)EDA-TC の各活動のフェーズが、有償で開催するシステム・デザイン・フォーラムに合致しなかった。(2)EDSFair 特設ステージでの無償開催も検討したが、基本目標である標準化活動の成果のアピール・展開と、特設ステージのショー的側面から来る期待とを両立させるテーマ設定は難しいと判断した。

今年度は、この結果を踏まえつつ、“システム・デザイン・フォーラム 2011”を開催すべく、委員会活動アピールの重要な場としての認識のもと、セッションテーマと費用の両面から開催に向けて検討した。

その結果、費用面では収支バランスの崩れにより独立したフォーラムとしての開催は難しいが、テーマでは、ナノ世代物理設計 WG および LPB 相互設計 WG から魅力的な講演が可能である回答を得たため、特設ステージ企画検討会(幹事および有志で構成)とも協議し、EDSFair2011 特設ステージに「特別協力」する形で初日(1/27)に３セッションを無償で開催した。各セッションの詳細は、3.1.9 特設ステージ を参照いただきたい。

3.2.2　システム・デザイン・フォーラム 2011WG 活動報告

１年のブランクを克服するべく検討を重ねた。しかし、費用面では、単独開催を実現する妙案は見出せなかった。

システム・デザイン・フォーラムは、独立採算で、聴講料(資料代)と海外関連団体からの協賛金を収入とする。しかし、2009 年まで協賛頂いて来た OSCI (The Open SystemC Initiative)が、“システム・デザイン・フォーラム 2010”が開催されなかったことに失望し、企業共催であった SystemC Japan の主催者となり“SystemC Japan 2010”を 7 月に入場料無料で開催したことで、今後の協賛金の提供を受けることが難しくなった。収入が減る一方で、会場費ならびに有料化するための資料製本費・登録システム代・会場人件費は大きく減らすことが出来ず、収支のバランスが崩れてしまった。

WG 活動紹介の場としてのセッションテーマ検討過程では、３つの WG のうち、SystemC WG が、7 月の“SystemC Japan 2010”の特別講演で活動を紹介するため、半年後の“システム・デザイン・フォーラム 2011”では新規性のある講演ができないとの理由で辞退した。

しかし、残る２つの WG からは、講演可能であるとの回答を得た。LPB 相互設計 WG は、WG 設立初年度であり、特設ステージでのテーマとしても期待が大きかった。EDSFair と同時開催の ASP-DAC Designer's Forum からの引き合いも来たが、WG の有力メンバーによるパネルセッションが可能である。ナノ世代物理設計 WG は、ファブライト化で専門化が減る傾向に

は有るものの招待講演が可能である。いずれも魅力ある講演が期待できる。

以上の検討結果から、「単独での開催は困難だが、特設ステージでのセッション講演を希望する」との結論をまとめ、特設ステージ企画検討会に持ち込み検討を依頼した。

その結果、ナノ世代物理設計 WG で１つ、LPB 相互設計 WG で２つ(単独と ASP-DAC とのコラボレーション)の、計３セッションを担当することが決定した。

セッション枠が決定した後は、両 WG で講演内容の詳細について検討して頂き、また準備面では EDSFair2011 の特設ステージとして EDSFair2011 事務局にサポートして頂いたため、この WG としては 2 回の開催で終了した。

3.2.3 システム・デザイン・フォーラム開催実績

・1990 年から 1994 年　“EDA 標準化フォーラム”を 4 回開催。
　EDA 標準化活動の発表とその一般への普及を図ることを目的。

・1999 年から 2002 年　“EDA フォーラム”を 2 回開催。
　EDA 技術専門委員会の活動に係る内容の発表、討論の場を目的。

・2004 年　“システム・デザイン・セミナー”
　最新の設計技術、課題を設計事例とともに紹介するため 2 日間の日程で開催。

・2005 年　“システム・デザイン・フォーラム 2005”
　1 日目　SystemVerilog ユーザ・フォーラムと SystemC ユーザ・フォーラム。
　2 日目　SoC に関連した設計技術、課題等を含めた設計事例を紹介する 2 セッションと、LSI、パッケージ、基板を含めた統合設計に関するパネル討論のセッションを開催。

・2006 年　“システム・デザイン・フォーラム 2006”
　“SystemVerilog ユーザ・フォーラム”と“SystemC ユーザ・フォーラム”の 2 セッション。両設計言語の標準化動向の紹介、チュートリアル、設計適用事例紹介を実施。

・2007 年　“システム・デザイン・フォーラム 2007”
　“SystemVerilog ユーザ・フォーラム”と“SystemC ユーザ・フォーラム”の 2 セッションに、65nm 以下のプロセスノードで深刻化するプロセスばらつきを打破する最新の設計技術動向を紹介するフィジカル･デザイン･フォーラムを新たに加え、計 2 日間 3 セッションを開催。

・2008 年　“システム・デザイン・フォーラム 2008”
　“SystemC ユーザ・フォーラム”と“Power Format フォーラム”の 2 セッションを開催。“SystemC ユーザ・フォーラム”では、最新の SystemC 標準化動向、TLM2.0 のチュートリアル、JEITA SystemC ワーキンググループの取り組みの報告と、設計適用事例の紹介を実施。“Power Format フォーラム”では、最新の低消費電力設計技術の紹介と、個々に Power Format 標準化を目指す二つの団体 Accellera Organization, Inc.、Si2(Silicon Initiative, Inc.)双方からの標準化活動の最新状況や設計適用事例の紹介と、JEITA Power Format 検討ワーキンググループの Power Format の標準化に対す

る検討状況の報告を行った。

・2009 年　“システム・デザイン・フォーラム 2009”

「SystemC ユーザ・フォーラム 2009」に加えて、新たに、プロセス微細化による製造ばらつきの問題に対して、「最先端統計から見た 32nm ばらつき予測と設計法」をテーマとした、「ナノ世代物理設計フォーラム」を開催。SystemC ユーザ・フォーラム 2009 では、OSCI (Open SystemC Initiative) による SystemC の最新動向の紹介、JEITA SystemC ワーキンググループによる、システム設計から実装、検証を含む SystemC 推奨設計メソドロジの紹介、半導体理工学研究センター(STARC)による TL モデリングガイドの紹介、SystemC を用いた高位合成適用事例、および TLM2.0 を利用した回路設計事例を報告。また、ナノ世代物理設計フォーラムでは、プロセスの微細化により、新たな設計上の課題としてあらわれてきた製造ばらつきによる設計の収束性および製造時の良品率の低下に対処するため、ばらつきの影響を考慮できる統計的な設計手法の現状を報告した。

・2010 年　開催断念。

・2011 年　EDSFair2011 特設ステージに「特別協力」にて３セッション担当。

3.2.4　今後の課題

費用面の検討結果から、今後も有償による独立採算は難しい。しかし、委員会活動アピールの重要な場としての位置づけは変わらず、また過去の聴講者アンケートからも内容の充実と期待が大きいことが確認されており、継続開催していくことが望まれる。

協賛金に頼らない開催、展示会と切り離した開催、他のセミナーとの共催など、開催形態を模索検討し、最適解を求めて行きたい。

3.2.5　システム・デザイン・フォーラム 2011WG 委員（敬称略、順不同）

主査	大芝　克幸	EDA-TC 幹事	ソニー
ｱﾄﾞﾊﾞｲｻﾞ	河村　薫	SDF 前年度主査(相当)	富士通セミコンダクター
委員	太田　光保	EDSFair 実行委員長/SDF2012 主査	パナソニック
委員	小島　智	EDA 標準化小委副主査/IEC TC93WG2	NEC システムテクノロジー
委員	西園寺　修	SystemC WG 委員	シノプシス
委員	田中　正和	ナノ世代物理設計 WG 主査	パナソニック
委員	福場　義憲	LPB 相互設計 WG 主査	東芝
委員	今井　正治	ASP-DAC リエゾン	大阪大学
委員	若林　一敏	ASP-DAC リエゾン	日本電気
事務局	近藤　寛		日本ｴﾚｸﾄﾛﾆｸｽｼｮｰ協会

3.3 ASP-DAC 2011

3.3.1 はじめに

Asia and South Pacific Design Automation Conference (ASP-DAC) は、VLSI およびシステム LSI の設計技術や設計自動化技術をテーマにしたアジア太平洋地域での最大規模の国際会議である。ASP-DAC は米国で開催されるこの分野のトップ・コンファレンスである Design Automation Conference (DAC)、International Conference on Computer Aided Design (ICCAD) や欧州で開催される Design, Automation and Test in Europe (DATE) とはシスター・コンファレンスの関係にあり、お互いにリエゾンを交換して協力関係を持っている。

ASP-DAC は、電子情報通信学会や情報処理学会などの学会だけでなく、電機メーカおよび半導体メーカの業界団体である JEITA（会議開始当時は EIAJ）と EDSF（会議開始当時は EDAT）の支援のもとで 1995 年に開始された。業界団体である JEITA が ASP-DAC のような国際会議の支援を行っているのは、次のような理由による。電機メーカや半導体メーカが国際競争力のある電子製品の開発を行うためには、マーケッティングや製品企画だけでなく、大規模・高機能・低消費電力のシステム LSI の最適設計を短期間で行える設計力を持つ必要がある。そのためには、最新の設計自動化技術についての情報収集と研究開発を行う必要がある。一流の国際会議を国内で開催することにより、わが国からより多くの技術者と研究者が参加して最先端の設計技術および設計自動化技術についての情報収集、情報交換などを行うことが可能になる。

3.3.2 会議の開催経緯

ASP-DAC の第 1 回目の会議は 1995 年 8 月 30 日から 9 月 1 日にかけて幕張メッセの日本コンベンションセンターで、情報処理分野の国際学会である International Federation on Information Processing (IFIP) の TC10 WG10.2 および WG10.5 に属する CHDL および VLSI という名称の 2 つの国際会議と並列開催の形で開催された。第 2 回目は 1997 年 1 月に開催され、それ以降毎年 1 月に開催されてきた。この間、1999 年には香港（中国）で、2002 年にはバンガロール（インド）でそれぞれ開催された。2007 年以降は、日本と国外で交互に開催するというローテーションで運営されている。今回の会議（ASP-DAC 2011）は 16 回目で、横浜市パシフィコ横浜で 1 月 25 日（火）から 28 日（金）の日程で開催された。

3.3.3 ASP-DAC 2011 の概要

ASP-DAC 2011 の概要を表 1 に示す。一般講演としては、26 カ国から投稿された 300 編の論文の中から 104 編が採択され、3 日間にわたって並列の 4 つのトラック、26 のセッションで発表された。表 1 からもわかるように、論文の投稿数については前回台湾で開催された ASP-DAC 2010 より少ない 300 件であったが、論文の採択率は例年並みの 35%を維持し、この分野での他の国際会議（DAC, ICCAD, DATE）とほぼ同じ水準を維持している。これまでどおり、ASP-DAC は名実ともに一流の国際会議であると評価できる。

基調講演のタイトルと講演者を表 2 に、特別セッションのタイトルを表 3 に示す。また、表 4 に、昨年に引き続いて実施されたデザイナーズ・フォーラムのセッション・タイトルを示す。表 5 には、有料チュートリアルのタイトルを示す。デザイナーズ・フォーラムでは、EDSFair との連携企画(5D, 8D)も組まれている。

発表された論文の中から、表 6 に示す論文が選ばれ、Best Paper Award が授与された。また、デザ

イン・コンテストに応募した作品の中から、表 7 に示す Best Design Award 並びに Special Feature Award 各 1 件が選ばれて表彰された。前回と同じく学生フォーラムも実施された。

表 1: ASP-DAC 2009、2010、2011 の比較

開催年	2009 年	2010 年	2011 年
日時	2009 年 1 月 19 日（月） ～22 日（木）	2010 年 1 月 18 日（月） ～21 日（木）	2011 年 1 月 25 日（火） ～28 日（金）
会場	横浜市（日本） パシフィコ横浜	台北（台湾） Taipei International Convention Center	横浜市（日本） パシフィコ横浜
併設展示会	EDSF 2009	---	EDSF 2011
論文投稿数	355	340	300
論文投稿国（地域）数	33	29	26
論文採択数（採択率）	116（33%）	119（35%）	104（35%）
キーノート アドレス	3 件	3 件	3 件 （表 2 参照）
一般講演	24 セッション（116 編）	30 セッション（119 編）	24 セッション（104 編）
特別セッション（招待講演等）	4 セッション	5 セッション	5 セッション （表 3 参照）
デザイン・コンテスト	1 セッション	1 セッション	1 セッション
学生フォーラム	昼休みに実施 （Student Forum）	昼休みに実施 （Student Forum）	昼休みに実施 （Student Forum）
ポスターボード	―	―	―
有料チュートリアル	7 件 （全日 1 件、半日 6 件）	5 件 （全日 3 件、半日 1 件）	6 件　（表 5 参照） （全日 2 件、半日 4 件）
デザイナーズ・フォーラム	4 セッション （招待講演 2、 パネル討論 2）	4 セッション （招待講演 2、 パネル討論 2）	4 セッション （表 4 参照） （招待講演 2、 パネル討論 2）

表 2: 基調講演

講演タイトル	講演者
Non-Volatile Memory and Normally-Off Computing	Takayuki Kawahara (Hitachi Ltd., Japan)
Managing Increasing Complexity through Higher-level of Abstraction: What the past has taught us about the future	Ajoy Bose (Atrena Inc., U.S.A.)
Robust Systems: from Clouds to Nanotubes	Subhasish Mitra (Stanford Univ., U.S.A.)

表 3: 特別セッションのタイトル

種類	セッション・タイトル
招待講演	セッション 2D: Emerging Memory Technologies and Its Implication on Circuit Design and Architectures
	セッション 3D: Recent Advances in Verification and Debug
	セッション 4D: Advanced Patterning and DFM for Nanolithography beyond 22nm
	セッション 3C:Post-Silicon Techniques to Counter Process and Electrical Parameter Variability
	セッション 9D: Virtualization, Programming, and Energy-Efficiency Design Issues of Embedded Systems

表 4: デザイナーズ・フォーラムのタイトル

種類	セッション・タイトル
招待講演	セッション 6D: Emerging technologies for wellness applications
	セッション 8D: State-of-The-Art SoCs and Its Design Methodology (EDSFair との連携企画)
パネル討論	セッション 5D: C-P-B co-design/co-verification technology for DDR3 1.6G in consumer products (EDSFair との連携企画)
	セッション 9D: Advanced packaging and 3D Technologies

表 5： Tutorial のタイトル

トラック	種類	タイトル
1	半日	Advanced CMOS Device Technologies (1)
2	半日	Advanced CMOS Device Technologies (2)
3	半日	3D Integration (1)
4	半日	3D Integration (2)
5	全日	Post Silicon Debug
6	全日	MPSoC: Multiprocessor System on Chip

表 6: Best Paper Award が授与された論文

論文タイトル・著者
3A-1: "Co-design of Cyber-Physical Systems via Controllers with Flexible Delay Constraints", Dip Goswami, Reinhard Schneider, Samarjit Chakraborty (Tech. Univ. of Munich, Germany)

表 7: デザイン・コンテストにて表彰された設計

種類	論文タイトル・著者
Best Design Award	1D-1: "A H.264/MPEG-2 Dual Mode Video Decoder Chip Supporting Temporal/Spatial Scalable Video", Cheng-An Chien, Yao-Chang Yang, Hsiu-Cheng Chang, Jia-Wei Chen, Cheng-Yen Chang, Jiun-In Guo, Jinn-Shyan Wang (National Chung Cheng Univ., Taiwan), Ching-Hwa Cheng (Feng Chia Univ., Taiwan)
Special Feature Award	1D-15: "A 58-63.6GHz Quadrature PLL Frequency Synthesizer Using Dual-Injection Technique", Ahmed Musa, Rui Murakami, Takahiro Sato, Win Chiavipas, Kenichi Okada, Akira Matsuzawa (Tokyo Inst. of Tech., Japan)

3.3.4 論文の投稿状況

1998 年から 2011 年の、ASP-DAC への論文投稿数の地域別の推移を図 1 に示す。ただし、ASP-DAC 2002 については、インドのバンガロールで同時開催された VLSI Design 2002 への投稿論文を含む。図 1 に示すように、ここ数年は、安定的に 300～400 件の投稿がある。名実ともに、ASP-DAC は設計自動化分野の国際会議として定着したと言ってよいであろう。

表 8 に、日本からの論文投稿数の推移と、全世界から投稿された論文に占める割合を示す。日本からの論文投稿数が全体に占める割合は、2000 年をピークにして、10%前後に低下している。論文投稿数が多かったのは、米国の 73 編（前回は 109 編）、中国の 39 編（前回は 41 編）、日本の 31 編（前回は 33 編）、台湾の 44 編（前回は 52 編）、インドの 13 編（前回は 13 編）、韓国の 11 編（前回は 11 編）であった。

図 1　地域別論文投稿数

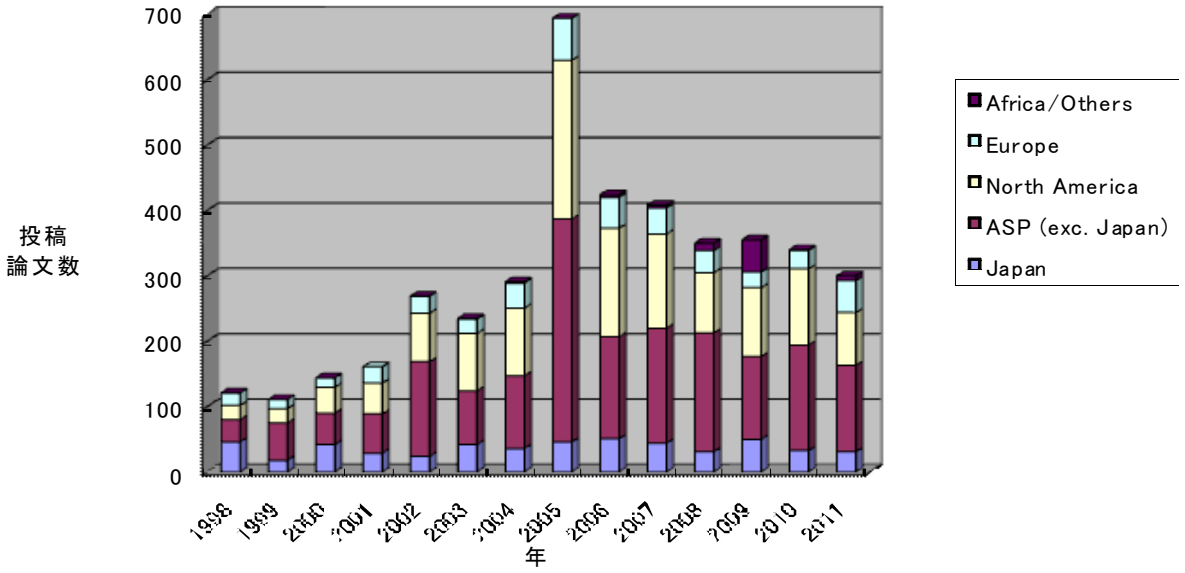


表 8 日本からの論文投稿数と全体に占める割合

年 地域	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011
日本 (割合)	42 (29%)	29 (18%)	24 (9%)	42 (18%)	36 (12%)	46 (7%)	51 (12%)	44 (11%)	31 (9%)	50 (14%)	33 (10%)	31 (10%)
全体	144	161	269	235	291	692	424	408	350	355	340	300

次に、研究分野別の論文投稿数および採択論文数を表 9 に示す。ASP-DAC 2011 では、研究分野を 13 種類に分類して論文の査読と採否の決定を行った。今回論文投稿数が多かった分野は、分野 7 の物理設計のセッション、分野 1 のシステムレベルのモデリングとシミュレーションのセッション、分野 3 のシステムレベルのメモリ・通信並びに Network-on-chip のセッション、分野 10 の製造容易化設計並びに統計的設計手法のセッションであった。

表 9: 分野別の論文投稿数と採択論文数

分野	研究分野	投稿数	採択数	採択率
1	System-Level Modeling and Simulation/Verification	16	6	37.5%
2	System-Level Synthesis and Optimization	20	6	30.0%
3	System-Level Memory/Communication Design and Networks on Chip	28	9	32.1%
4	Embedded and Real-Time Systems	29	8	27.6%
5	High-Level/Behavioral/Logic Synthesis and Optimization	24	10	41.7%
6	Validation and Verification for Behavioral/Logic Design	17	5	29.4%
7	Physical Design	31	11	35.5%
8	Timing, Power, Thermal Analysis and Optimization	32	12	37.5%
9	Signal/power Integrity, Interconnect/Device/Circuit Modeling and Simulation	20	7	35.0%
10	Design for Manufacturability/Yield and Statistical Design	24	8	33.3%
11	Test and Design for Testability	19	8	42.1%
12	Analog, RF and Mixed Signal Design and CAD	12	4	33.3%
13	Emerging Technologies and Applications	28	10	35.7%
	合　　計	300	104	34.7%

3.3.5 参加者の内訳

ASP-DAC への地域別の参加者数の推移を図 2 に示す（チュートリアルのみの参加者を除く）。また、日本からの参加者の推移を表 10 に示す（日本開催時のみ）。今回の全参加者数は 458 名であった。前回（367 名）と比べると、参加者が増加した。日本からの参加者数は全体の 53.9%の 247 名であった。

図 2 地域別参加者の推移

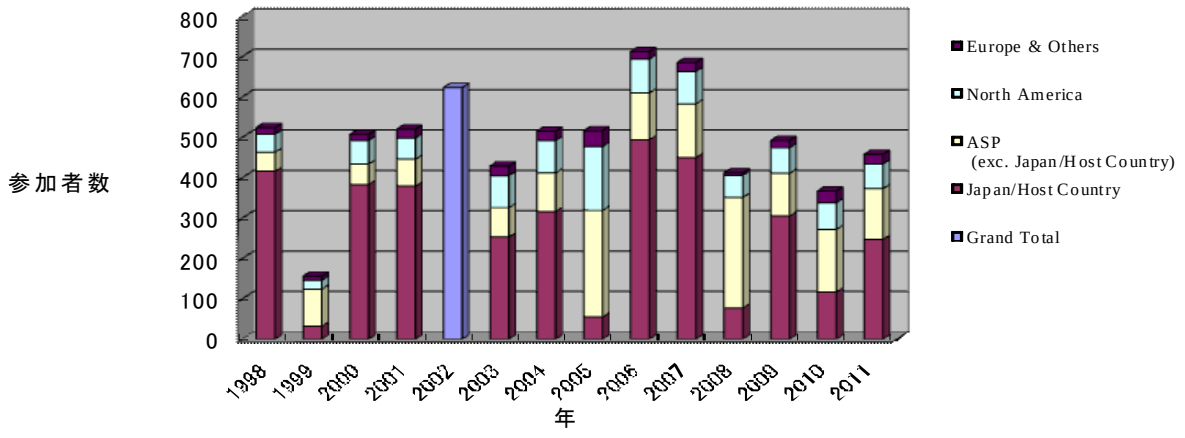


表 10:　日本開催年の日本からの参加者数と全体に占める割合

（チュートリアルのみの参加者を除く）

年 地域	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011
日本 (割合)	383 (76%)	379 (73%)	--	253 (59%)	316 (61%)	--	494 (70%)	450 (66%)	--	305 (62%)	--	247 (53.9 %)
全体	507	520	623	429	515	516	708	685	412	492	367	458

3.3.6 今後の展望

ASP-DAC の今後の開催予定を表 11 に示す。引き続き、隔年で日本開催になる模様である。

表 11:　ASP-DAC の今後の開催予定

年	開催予定地	開催時期	実行委員長
2012 年	Sydney （オーストラリア）	2012 年 1 月 30 日（月）～2 月 2 日（木）	Sri Parameswaran 氏 （Univ. of New South Wales）
2013 年	未定（日本）	未定	木村晋二 氏 （Waseda Univ.）

４．添付資料

1. 配線ばらつき感度 / SSPEF フォーマットの適用範囲調査

目次

- 背景と目的
- 配線RCトレンド
- Sensitivity - SPEFとは
- 感度見積もり精度評価
  - 配線RC抽出
  - 配線RC応答曲面算出
  - 配線RCモデル精度解析
- 遅延へのインパクト調査
- まとめ

背景と目的

- 背景
  - 配線コーナー数増大
    - 配線Globalばらつきを STAで扱った場合、温度未考慮としても配線だけで4コーナー必要(Cworst, RCworst, Cbest, RCbest)

  → SSTAで統一的に扱うことを期待
  - SSTA対応Sensitivity - SPEF : 線型感度モデルがIEEE1394 - 2009で規定

    → 適用範囲、制約の検証必要
- 目的
  - 22nmプロセスを想定し、現実的な配線、回路構造で線型感度モデルの精度検証を行い、信号伝播遅延精度への影響を評価
  - 上記結果に基き、SSTAにおける設計要求精度維持に必要な配線RCのばらつき感度モデル化手法を提案する。

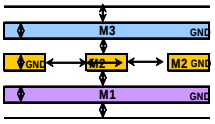
配線RCトレンド

1. ITRS2007各プロセス世代の構造パラメタに基づき、配線RCのトレンドを調査
2. 構造パラメータのばらつき量を仮定し、配線RCのばらつき量の見積もり実施

Tmt

Parameters			
65nm	45nm	32nm	22nm
122.40	81.00	60.80	44.00
122.40	81.00	60.80	44.00
122.40	81.00	60.80	44.00

TYPICAL



Wmt S

65nm	68.00	68.00
45nm	45.00	45.00
32nm	32.00	32.00
22nm	22.00	22.00

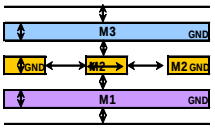
Tox

Parameters			
65nm	45nm	32nm	22nm
1E+05	1E+05	1E+05	1E+05
108.80	72.00	54.40	39.60
108.80	72.00	54.40	39.60
115.60	81.00	60.80	44.00

+ 24.4nm

Parameters			
65nm	45nm	32nm	22nm
146.80	105.40	85.20	68.40
146.80	105.40	85.20	68.40
146.80	105.40	85.20	68.40

WORST



+ 3.4nm - 3.4nm

65nm	71.40	64.60
45nm	48.40	41.60
32nm	35.40	28.60
22nm	25.40	18.60

- 21.8nm

Parameters			
65nm	45nm	32nm	22nm
1E+05	1E+05	1E+05	1E+05
87.04	50.24	32.64	17.84
87.04	50.24	32.64	17.84
92.48	64.80	48.64	35.20

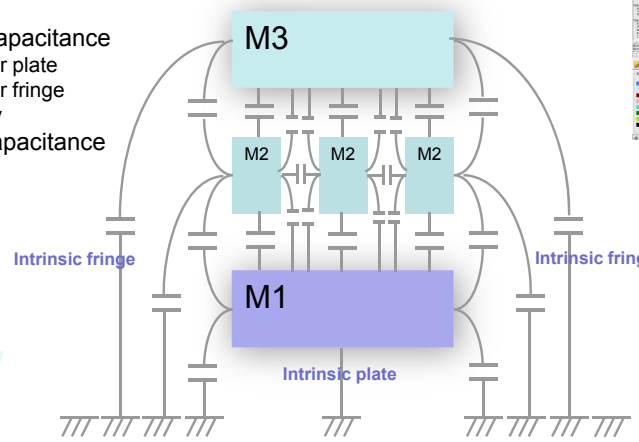
3D構造抽出条件

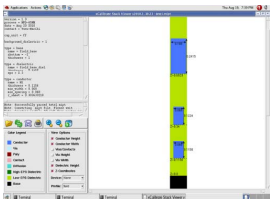
- 抽出環境
  - プロセス・パラメタ校正：Calibre xCalibrate v.2011.1_15.11
  - レイアウト抽出：Calibre xACT 3D v.2011.1_15.11
  - OS: Red Hat Enterprise Linux 5 64bit
  - Workstation 6.5 - 7,x 仮想マシン
- データ・ベース作成手順
  - 各プロセスノードに対応する配線膜厚Mt(t)と絶縁膜厚Tox(t)に関わる条件ファイル（.mipt)を作成、プロセス縦構造のパラメータ補正係数を抽出
  - 想定したプロセスノードに対応するプロセスの変動分を加味したレイアウトをそれぞれ作成
- RC抽出
  - 上記で抽出した補正係数を使用しExtraction実行
  - もとのネットリストにバック・アノテーションされた結果から、単位長さ当たりの寄生容量値、寄生抵抗値を算出

使用3D配線断面構造

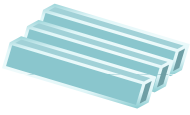
- Parasitic Capacitances

1. Intrinsic Capacitance
   - Plate
   - Fringe
2. Crossover Capacitance
   - Crossover plate
   - Crossover fringe
   - Nearbody
3. Nearbody Capacitance







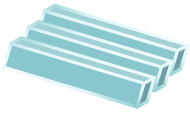


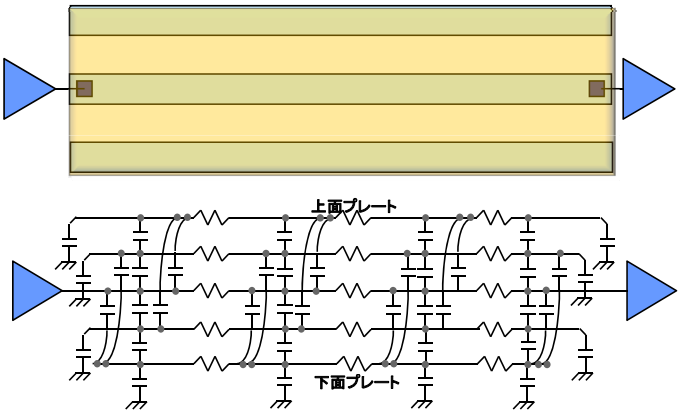
-複雑な寄生容量は、単純な面間容量計算だけでは推定困難
- Field Solver Baseの高精度容量抽出を使用

使用3D配線レイアウト形状

- Parasitic Capacitances

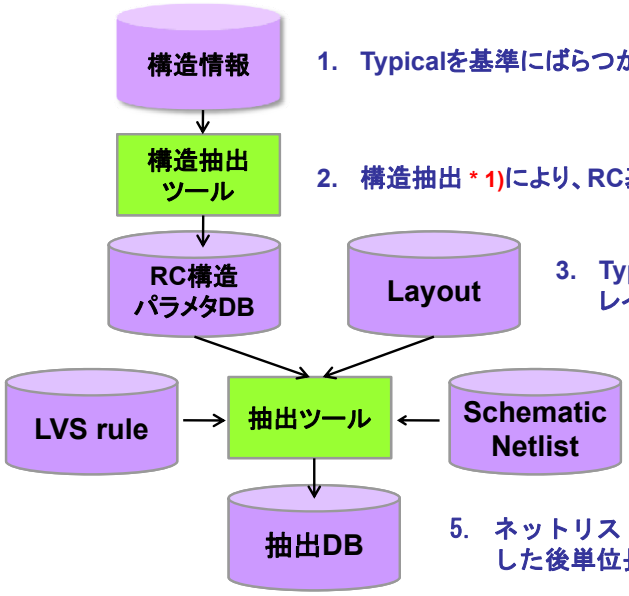
1. Intrinsic Capacitance
   - Plate
   - Fringe
2. Crossover Capacitance
   - Crossover plate
   - Crossover fringe
   - Nearbody
3. Nearbody Capacitance





-複雑な寄生RC値は、単純な面間容量計算だけでは推定困難
- Field Solver Baseの高精度容量抽出を使用

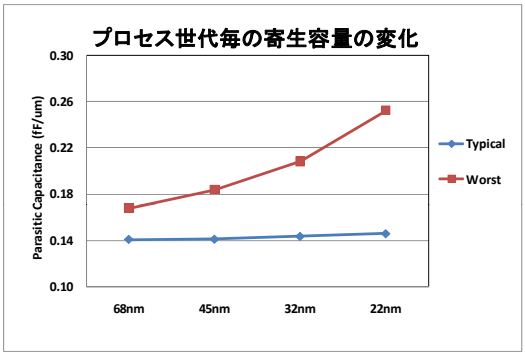
3D構造抽出フロー

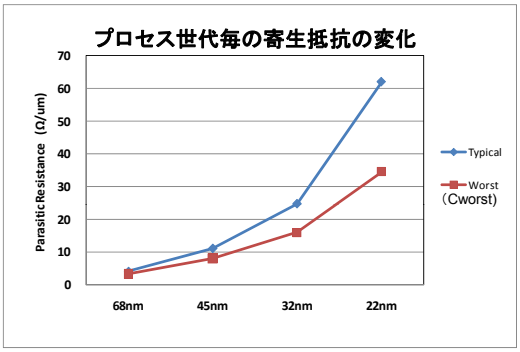


1. Typicalを基準にばらつかせたM(t), - Tox(t)の立て構造を作成
2. 構造抽出 * 1)により、RC基本パラメタ作成
3. Typical値を基準にばらつかせたM(w) レイアウトを作成
4. Field Solver * 2) による寄生RC抽出
5. ネットリスト上へのバック・アノテーションした後単位長さ辺りのRC(/ um)を求める

* 1) Calibre xCalibrate v.2011, * 2) Calibre xACT 3D v.2011

プロセスノード毎の容量・抵抗トレンド結果

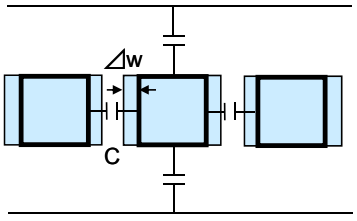




1. プロセス・ノードの微細化につれて、単位長さ当たりの寄生容量および抵抗の増加を確認
2. 微細化によりばらつき量を抑えなければ、配線RCのばらつき幅は増大する

Sensitivity SPEFとは

- ばらつきパラメータの変動に対するRLC感度情報を付加したSPEF(Standard Parasitic Exchange Format)ネットリスト



Interconnect cross - section

$$C(p)=C_0\times\left(\left(1+\sum_j(cn_j\Delta v_j)\right)/\left(1+\sum_i(cd_i\Delta v_i)\right)\right)$$

$$L(p)=L_0\times\left(\left(1+\sum_j(ln_j\Delta v_j)\right)/\left(1+\sum_i(ld_i\Delta v_i)\right)\right)$$

$$R(p,T)=R_0\times\left(1+a\times\Delta T+b\times\Delta T^2\right)\times\left(\left(1+\sum_j(rn_j\Delta v_j)\right)/\left(1+\sum_i(rd_i\Delta v_i)\right)\right)$$

$$cn_j=((\partial C(p)/\partial p_j)/C_0)\times NF(p_j) \qquad cd_i=((\partial C^{-1}(p)/\partial p_i)/(1/C_0))\times NF(p_i)$$

$$ln_j=((\partial L(p)/\partial p_j)/L_0)\times NF(p_j) \qquad ld_i=((\partial L^{-1}(p)/\partial p_i)/(1/L_0))\times NF(p_i)$$

$$rn_j=((\partial R(p)/\partial p_j)/R_0)\times NF(p_j) \qquad rd_i=((\partial R^{-1}(p)/\partial p_i)/(1/R_0))\times NF(p_i)$$

$$\Delta v_i=VC(p_i)\times VM(p_i)$$

$$VC(p_i)=\sigma(p_i)/NF(p_i)$$

$$a=(\partial R/\partial T)/R_0$$

$$b=(\partial^2 R/\partial T^2)/R_0$$

$$\Delta T=T-T_0$$

- 複数の感度パラメータの併用が可能
- 各感度パラメータは、RLCに対して比例もしくは反比例として定義され、比例 / 反比例毎の線形和として表現される

感度見積もり精度評価

変数感度係数モデルは以下の関係を前提とした。

・容量変動量　　：

（3変数）$C = C_{typ}(1 + \alpha W_{mt} + \beta T_{mt} + \gamma T_{ox})$

（2変数）$C = C_{typ}(1 + \delta W_{mt} / T_{ox} + \varepsilon T_{mt} / S)$

・抵抗変動量　　：

（2変数）$R = R_{typ}(1 + \varphi / W_{mt} + \lambda / T_{mt})$

（1変数）$R = R_{typ}(1 + \mu / W_{mt}\, T_{mt})$

（W_{mt}：配線幅，T_{mt}：配線膜厚，T_{ox}：配線層間絶縁膜厚，S：配線間隔）

感度算出方法：　以下の3通りの方法を比較、検証した。

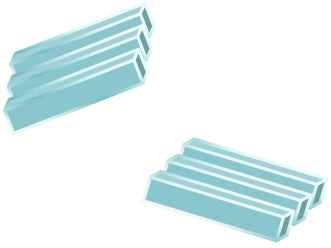
1．近傍近似：容量はW_{mt}, T_{ox}, - T_{mt} それぞれの原点、および + 0.2σ の点、抵抗は - W_{mt}, - T_{mt}それぞれの原点、および + 0.2σ の点でRSF(Response Surface Function)を求める。

2．最大近似：容量はW_{mt}, T_{ox}, - T_{mt} それぞれの原点、および + 3σ の点、抵抗は - W_{mt}, - T_{mt}それぞれの原点、および + 3σ の点でRSFを求める。

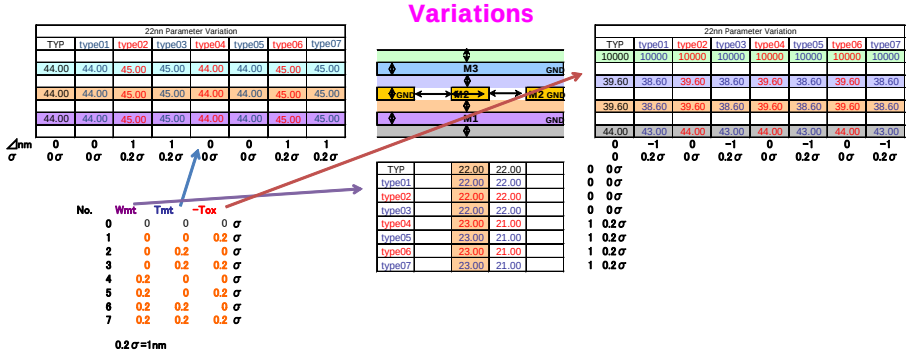
3．全体近似：容量はW_{mt}, T_{ox}, - T_{mt} 、抵抗は - W_{mt}, - T_{mt}を変数として、+ 1σ、+ 2σ、+ 3σ 3水準のL9直交表の試行を行い、RSFを求める。

3D配線抽出条件と抽出結果（近傍近似）

No.	A	B	C			
	Wmt	Tmt	-Tox		x3d_600	Rmt2
TYP0	0	0	0	σ	0.145881	62.086323
1	0	0	-0.2	σ	0.147702	62.086323
2	0	0.2	0	σ	0.148797	60.706789
3	0	0.2	-0.2	σ	0.149662	60.706789
4	0.2	0	0	σ	0.152328	59.385313
5	0.2	0	-0.2	σ	0.153090	59.385313
6	0.2	0.2	0	σ	0.154525	58.065794
7	0.2	0.2	-0.2	σ	0.154978	58.065794
	1σ＝5nm				fF/um	Ohms/um

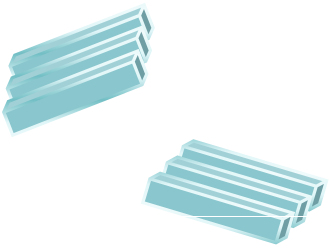


Variations



3D配線抽出条件と抽出結果（最大近似）

No.	A Wmt	B Tmt	C −Tox		x3d_600	Rmt2
TYP0	0	0	0	σ	0.145881	62.086323
1	0	0	3	σ	0.161024	62.086323
2	0	3	0	σ	0.174628	46.301448
3	0	3	3	σ	0.188963	46.301448
4	3	0	0	σ	0.352387	36.916412
5	3	0	3	σ	0.367074	36.916412
6	3	3	0	σ	0.438114	27.530755
7	3	3	3	σ	0.453172	27.530755

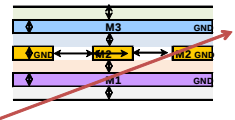


1σ＝5nm　　fF/um　　Ohms/um

Variations

22nn Parameter Variation

TYP	type01	type02	type03	type04	type05	type06	type07
44.00	44.00	59.00	59.00	44.00	44.00	59.00	59.00
44.00	44.00	59.00	59.00	44.00	44.00	59.00	59.00
44.00	44.00	59.00	59.00	44.00	44.00	59.00	59.00
⊿nm	0	0	15	15	0	0	15	15
σ	0σ	0σ	3σ	3σ	0σ	0σ	3σ	3σ



TYP		22.00	22.00	
type01		22.00	22.00	0
type02		22.00	22.00	0
type03		22.00	22.00	0
type04		37.00	7.00	15
type05		37.00	7.00	15
type06		37.00	7.00	15
type07		37.00	7.00	15

22nn Parameter Variation

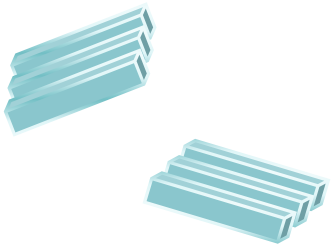
TYP	type01	type02	type03	type04	type05	type06	type07
10000	10000	10000	10000	10000	10000	10000	10000
39.60	24.60	39.60	24.60	39.60	24.60	39.60	24.60
39.60	24.60	39.60	24.60	39.60	24.60	39.60	24.60
44.00	29.00	44.00	29.00	44.00	29.00	44.00	29.00
	0	−15	0	−15	0	−15	0	−15
	0	3σ	0σ	3σ	0σ	3σ	0σ	3σ

No.	Wmt	Tmt	−Tox	
0	0	0	0	σ
1	0	0	3	σ
2	0	3	0	σ
3	0	3	3	σ
4	3	0	0	σ
5	3	0	3	σ
6	3	3	0	σ
7	3	3	3	σ

3σ=15nm

3D配線抽出条件と抽出結果（全体近似）

No.	A Wmt	B Tmt	C −Tox		x3d_600	Rmt2
TYP0	0	0	0	σ	0.145881	62.086323
1	1	1	1	σ	0.193973	45.426128
2	1	2	2	σ	0.210628	41.219916
3	1	3	3	σ	0.229241	37.726652
4	2	1	2	σ	0.257329	38.328525
5	2	2	3	σ	0.283201	34.779514
6	2	3	1	σ	0.286918	31.832054
7	3	1	3	σ	0.395352	33.149394
8	3	2	1	σ	0.411856	30.079940
9	3	3	2	σ	0.446184	27.530755
WST	3	3	3	σ	0.453172	27.530755
MABIKI	0	0	0	σ	0.094923	62.086323

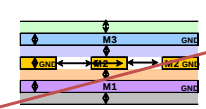


1σ＝5nm　　fF/um　　Ohms/um

Variations

22nn Parameter Variation

TYP	type01	type02	type03	type04	type05	type06	type07	type08	type09	WST
44.00	49.00	54.00	59.00	49.00	54.00	59.00	49.00	54.00	59.00	59.00
44.00	49.00	54.00	59.00	49.00	54.00	59.00	49.00	54.00	59.00	59.00
44.00	49.00	54.00	59.00	49.00	54.00	59.00	49.00	54.00	59.00	59.00
	0	5	10	15	5	10	15	5	10	15	15
	0	1σ	2σ	3σ	1σ	2σ	3σ	1σ	2σ	3σ	3σ



TYP		22.00	22.00	
type01		27.00	17.00	5
type02		27.00	17.00	5
type03		27.00	17.00	5
type04		32.00	12.00	10
type05		32.00	12.00	10
type06		32.00	12.00	10
type07		37.00	7.00	15
type08		37.00	7.00	15
type09		37.00	7.00	15
WST		37.00	7.00	15

22nn Parameter Variation

TYP	type01	type02	type03	type04	type05	type06	type07	type08	type09	WST
10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
39.60	34.60	29.60	24.60	29.60	24.60	34.60	24.60	34.60	29.60	24.60
39.60	34.60	29.60	24.60	29.60	24.60	34.60	24.60	34.60	29.60	24.60
44.00	39.00	34.00	29.00	34.00	29.00	39.00	29.00	39.00	34.00	29.00
	0	−5	−10	−15	−10	−15	−5	−15	−5	−10	−15
	0	1σ	2σ	3σ	2σ	3σ	1σ	3σ	1σ	2σ	3σ

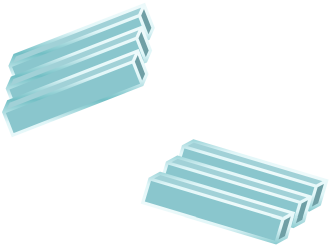
No.	Wmt	Tmt	−Tox	
1	1	1	1	σ
2	1	2	2	σ
3	1	3	3	σ
4	2	1	2	σ
5	2	2	3	σ
6	2	3	1	σ
7	3	1	3	σ
8	3	2	1	σ
9	3	3	2	σ

1σ=5nm

3D配線抽出条件と抽出結果（個別変動）

No.	A	B	C			
	Wmt	Tmt	−Tox		x3d 600	Rmt2
TYP0	0	0	0	σ	0.145881	62.086323
1	−3	0	0	σ	0.096083	195.105800
2	−1.5	0	0	σ	0.117102	94.197676
3	1.5	0	0	σ	0.201694	46.300645
4	0	−3	0	σ	0.119752	94.199699
5	0	−1.5	0	σ	0.133358	74.843948
6	0	1.5	0	σ	0.160827	53.044576
7	0	0	−3	σ	0.141975	62.086323
8	0	0	−1.5	σ	0.144146	62.086323
9	0	0	1.5	σ	0.152306	62.086323

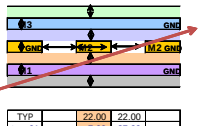
1σ＝5nm　　fF/um　　Ohms/um



Variations

22nm Parameter Variation

TYP	type01	type02	type03	type04	type05	type06	type07	type08	type09
44.00	44.00	44.00	44.00	29.00	36.50	51.50	44.00	44.00	44.00
44.00	44.00	44.00	44.00	29.00	36.50	51.50	44.00	44.00	44.00
44.00	44.00	44.00	44.00	29.00	36.50	51.50	44.00	44.00	44.00
Δnm	0	0	0	0	−15	−7.5	7.5	0	0	0
σ	0σ	0σ	0σ	0σ	−3σ	−1.5σ	1.5σ	0σ	0σ	0σ



TYP		22.00	22.00
type01		7.00	37.00
type02		14.50	29.50
type03		29.50	14.50
type04		22.00	22.00
type05		22.00	22.00
type06		22.00	22.00
type07		22.00	22.00
type08		22.00	22.00
type09		22.00	22.00

22nm Parameter Variation

TYP	type01	type02	type03	type04	type05	type06	type07	type08	type09
10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
39.60	39.60	39.60	39.60	39.60	39.60	39.60	54.60	47.10	32.10
39.60	39.60	39.60	39.60	39.60	39.60	39.60	54.60	47.10	32.10
44.00	44.00	44.00	44.00	44.00	44.00	44.00	59.00	51.50	36.50
0	0	0	0	0	0	0	15	7.5	−7.5
0	0σ	0σ	0σ	0σ	0σ	0σ	−3σ	−1.5σ	1.5σ

No.	Wmt	Tmt	−Tox	
0	0	0	0	
1	−3	0	0	σ
2	−1.5	0	0	σ
3	1.5	0	0	σ
4	0	−3	0	σ
5	0	−1.5	0	σ
6	0	1.5	0	σ
7	0	0	−3	σ
8	0	0	−1.5	σ
9	0	0	1.5	σ

配線RC応答曲面算出

- 各パラメータのばらつき量と、対応するRC抽出結果から、RCばらつきモデル式を算出
- モデル式の算出は、RSMaker for Excel（東京工業大学 大学院 理工学研究科 機械物理工学専攻 轟研究室）を使用
- 算出モデル式は以下の通り

配線容量モデル算出結果

Wmt, Tmt, Tox, S : [nm]
C : [fF/um]

■1次式3変数(1次3項)モデル
(全体近似)
C = − 0.261 + 0.0177 * Wmt + 0.000919 * Tmt − 1.16 / Tox
(最大近似)
C = − 0.357 + 0.0153 * Wmt + 0.00341 * Tmt + 0.577/ Tox
(近傍近似)
C = − 0.139 + 0.00594 * Wmt + 0.00246 * Tmt + 1.82/ Tox

■ 1次式2変数(1次2項)モデル
(全体近似)
C = 0.0395 + 0.0391 * Wmt / Tox + 0.0425 * Tmt / S
(最大近似)
C = 0.0400 + 0.0356 * Wmt / Tox + 0.0433 * Tmt / S
(近傍近似)
C = 0.0146 + 0.0667 * Wmt / Tox + 0.0471 * Tmt / S

■ 2次式2変数(2次2項)モデル
C = 0.0347 + 0.0371 * Wmt / Tox + 0.0467 * Tmt / S + 0.00481 * (Wmt / Tox)^2
− 0.00270 * (Wmt / Tox) * (Tmt / S) − 0.0000914 * (Tmt / S)^2

配線抵抗モデル算出結果

Wmt, Tmt, Tox, S : [nm]
R : [Ω/um]

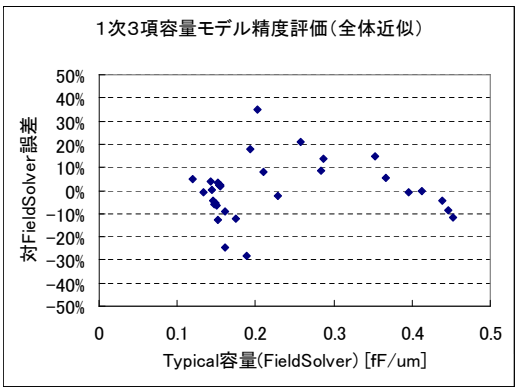
■ 1次式2変数(1次2項)モデル
(全体近似)
R = − 44.6 + 1242.0259 / Wmt + 2200 / Tmt
(最大近似)
R = − 47.3 + 1240 / Wmt + 2320 / Tmt
(近傍近似)
R = − 60.6 + 1360 / Wmt + 2690 / Tmt

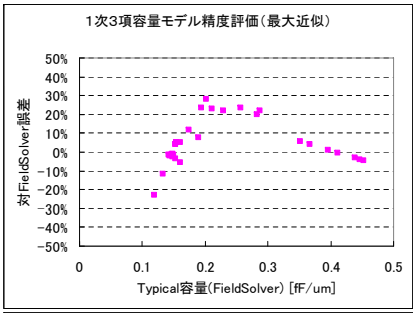
■ 1次式1変数(1次1項)モデル
(全体近似)
R = − 0.000376 + 60100 / (Wmt ・Tmt)
(最大近似)
R = 0.000117 + 60100 / (Wmt ・Tmt)
(近傍近似)
R = −0.0248 + 60100 / (Wmt ・Tmt)

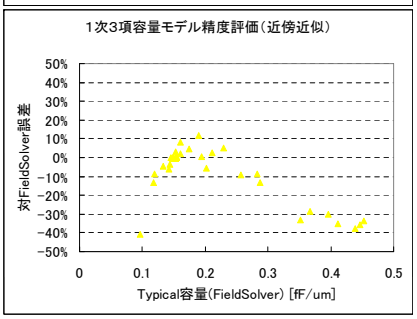
■ 2次式2変数(2次2項)モデル
R = − 0.000495 + 0.0562 / Wmt − 0.0522 / Tmt − 1.978 / Wmt^2 +
60100 / (Wmt * Tmt) − 1.93 / Tmt^2

容量モデル精度評価結果

•1次3項モデル(W,Tmt,Tox)の場合、全体近似(0~3σ)、最大近似(3σ)に関わらず、30%以上のモデル誤差が発生する。

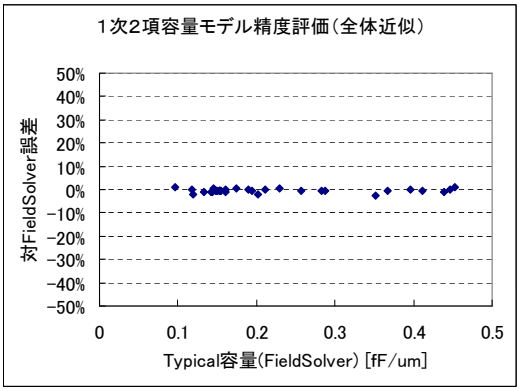




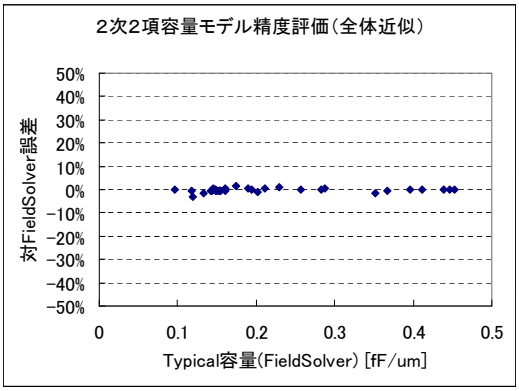


容量モデル精度評価結果

•2項モデル(W / Tox, Tmt / S)を用いることにより、平均誤差1%以下に削減が可能である
•また2項モデルを用いることにより、1次式モデルでも十分2次式モデルと同等の精度を得ることが可能である



最大誤差	2.6%
平均誤差	0.9%

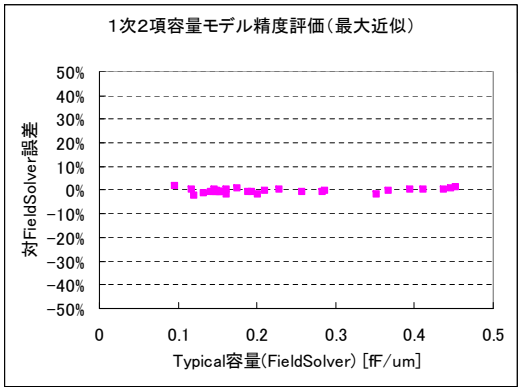


最大誤差	3.0%
平均誤差	0.8%

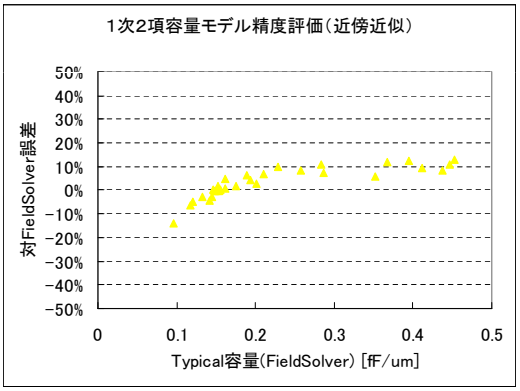
※平均誤差は自乗平均で定義

容量モデル精度評価結果

•1次2項モデル(W / Tox, S / Tmt)において最大近似(3σ)を用いた場合、全体近似(0~3σ)と同等の精度を得ることが可能である。
•ただし、近傍近似(0．2σ)では、10%を超えるモデル誤差を含んでいる。



最大誤差	2.4%
平均誤差	1.0%



最大誤差	14.2%
平均誤差	6.9%

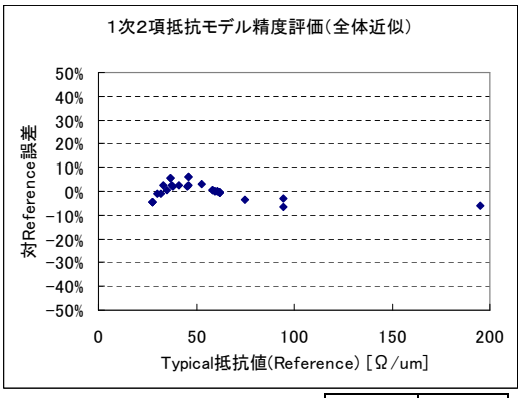
容量モデル精度評価考察

- 配線幅(W)、配線膜厚(Tmt)、絶縁膜厚(Tox)の3変数をパラメータとした場合、近似方法(全体近似、最大近似、近傍近似)に関わらず、30%を超える容量誤差が発生する。
- 配線幅 / 配線間膜厚(Wmt / Tox)、配線膜厚 / 配線間隔(Tmt / S)の2変数をパラメータとすることにより、最大近似(3σ)もしくは全体近似(0~3σ)において、容量誤差を最大3%に抑えることが可能となる。
- ただし近傍近似(0．2σ)では、Wmt / Tox 、Tmt / Sの2変数をパラメータとした場合でも、10%を超えるモデル誤差が発生する。

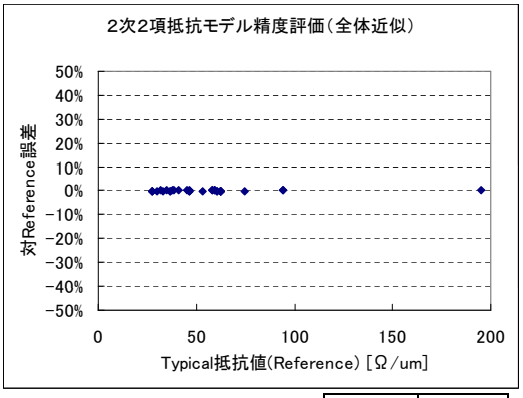
- 1次式モデルでも上記モデルを用いることにより、十分2次式モデルと同精度のモデル化が可能である。

抵抗モデル精度評価結果

•1次式モデルの場合、2項モデルであれば、最大7%程度の誤差が存在した。
•2次式モデルを用いれば、誤差をほぼゼロにすることが可能となる。



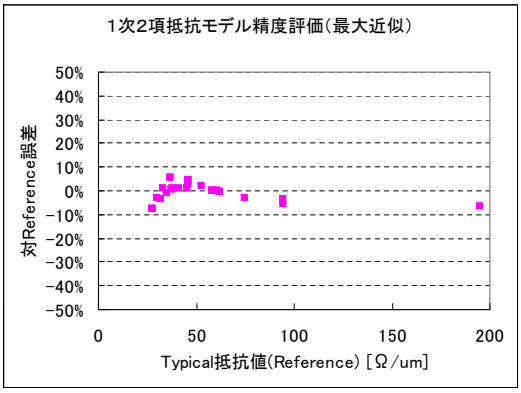
最大誤差	6.8%
平均誤差	3.3%



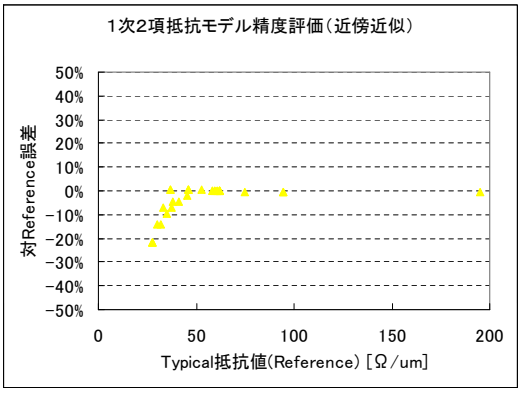
最大誤差	0.002%
平均誤差	0.001%

抵抗モデル精度評価結果

•1次2項モデル(W / Tox, S / Tmt)において最大近似(3σ)の精度は、全体近似(0~3σ)と同様、最大7%程度の誤差が存在する。
•近傍近似(0．2σ)では、さらに精度は低下し、20%以上のモデル誤差が存在する。



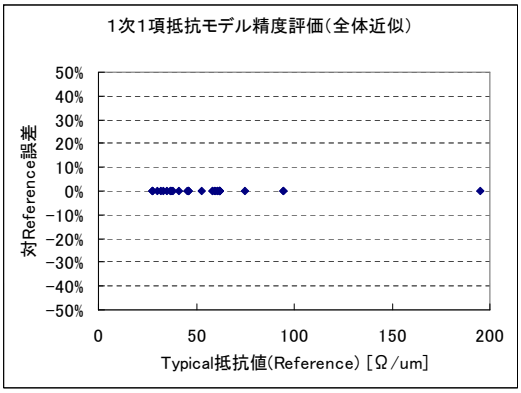
最大誤差	7.3%
平均誤差	3.5%



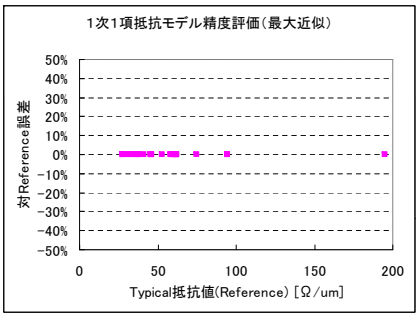
最大誤差	21.6%
平均誤差	7.9%

抵抗モデル精度評価結果

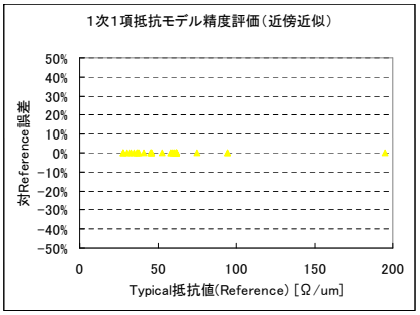
•1変数モデル(1 / Wmt Tmt)を用いることにより、1次式モデルであっても、モデル誤差をほぼゼロにすることが可能となる。



最大誤差	0.012%
平均誤差	0.002%



最大誤差	0.011%
平均誤差	0.002%



最大誤差	0.050%
平均誤差	0.024%

抵抗モデル精度評価考察

- 1次式モデルにおいて、1 / 配線幅(1 / Wmt) 、1 / 配線膜厚(1 / Tmt)の2変数をパラメータとした場合、全体近似(0~3σ)および最大近似(3σ)では最大7%、近傍近似(0. 2σ)では最大20%以上の抵抗誤差が発生する。

- ただし1次式モデルでも、1 / 配線幅・配線膜厚(1 / Wmt Tmt)をパラメータとすることにより、全体近似(0~3σ)、最大近似(3σ)、近傍近似(0. 2σ)に関わらず、ほぼモデル誤差をゼロにすることが可能となる。

- 1次式モデルでも上記モデルを用いることにより、十分2次式モデルと同精度のモデル化が可能である。

遅延に対するインパクト評価

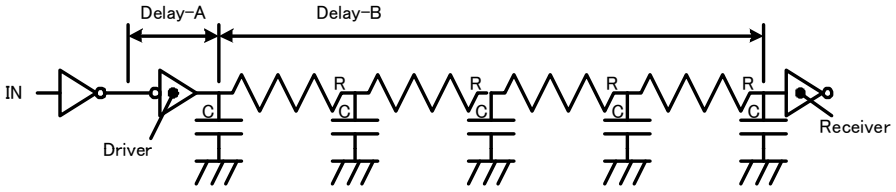
- 先に評価した、Sensivity - SPEFによる配線RCばらつきモデル精度評価結果を用いて、遅延誤差へのインパクトを評価
- 実験計画表に基づき、ばらつき成分(W,Tmt,Tox)を変動させ、シミュレーション実施
- 配線長は、Typical条件で　配線遅延＝セル遅延　となる配線長(Lw)とし、前記配線長(Lw)から配線RC値を決定

遅延評価条件

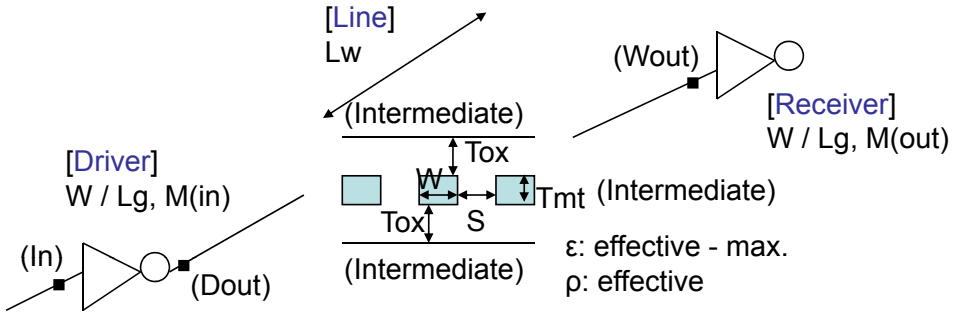
SPICEモデル	:PTM 22nm LP (Bulk)
SPICE電流モデル	:BSIM4
電源電圧(VDD)	:0.95V
基本INV（INV - X32）	:PMOS (W=135nm * 22 / 45,L=22nm)
	:NMOS (W=90nm * 22 / 45,L=22nm)
	～Nangate社45nmライブラリの基本INVを使用

下図の等価回路を用いて、RCのばらつき成分が遅延時間に及ぼす影響について調査

Delay - A	:セル遅延（Driver）
Delay - B	:配線遅延（Line）



評価フロー



1. Typical のRC値から22nm相当のW / Lgに換算した Nangate45nm(LP) ライブラリ、およびPTM22nmパラメータを使用
2. 下記定義により、配線遅延およびセル遅延が等しくなるLw(配線長）および M(Multiplier)(in, out) を算出

   (1) Min = Mout
   (2) Delay(In - >Dout)=Delay(Dout - >Mout)

3. 抽出により求まるRCと、近似式により得られたRCに対して、それらの誤差が遅延に及ぼす影響について評価実施

遅延評価結果(2次2項近似)

	Variation			Refer.(ps)			(2次2項)		
	Wmt	Tmt	−Tox	Driver	Line	D+L	Driver	Line	D+L
TYP	0	0	0	16.0	16.0	32.1	16.1	16.1	32.1
全体	1	1	1	18.0	14.2	32.2	18.0	14.2	32.2
	1	2	2	18.7	13.7	32.3	18.7	13.7	32.4
	1	3	3	19.4	13.2	32.7	19.5	13.3	32.8
	2	1	2	20.0	14.4	34.4	20.0	14.4	34.4
	2	2	3	21.1	14.0	35.1	21.1	14.0	35.1
	2	3	1	21.6	13.0	34.6	21.6	13.0	34.6
	3	1	3	23.9	17.0	40.9	23.9	17.0	40.9
	3	2	1	25.0	15.8	40.9	25.0	15.9	40.9
	3	3	2	26.5	15.4	41.9	26.5	15.4	41.9
	3	3	3	26.7	15.5	42.3	26.7	15.6	42.3
最大	0	0	3	16.3	16.8	33.1	16.3	16.8	33.0
	0	3	0	17.5	13.7	31.1	17.5	13.8	31.3
	0	3	3	17.7	14.3	32.0	17.8	14.3	32.1
	3	0	0	22.3	17.3	39.6	22.2	17.2	39.3
	3	0	3	22.6	17.8	40.5	22.6	17.8	40.4
	3	3	0	26.3	15.1	41.4	26.3	15.2	41.5
	3	3	3	26.7	15.5	42.3	26.7	15.6	42.3
近傍	0	0	0.2	16.1	16.1	32.2	16.1	16.1	32.2
	0	0.2	0	16.1	15.9	32.0	16.1	15.9	32.0
	0	0.2	0.2	16.1	15.9	32.1	16.1	15.9	32.1
	0.2	0	0	16.3	15.8	32.0	16.3	15.8	32.0
	0.2	0	0.2	16.3	15.8	32.1	16.3	15.8	32.1
	0.2	0.2	0	16.4	15.6	32.0	16.4	15.6	31.9
	0.2	0.2	0.2	16.4	15.6	32.0	16.4	15.6	32.0
単独	−3	0	0	13.5	35.5	49.0	13.5	35.5	49.0
	−1.5	0	0	14.7	20.5	35.2	14.7	20.5	35.2
	1.5	0	0	18.0	14.8	32.8	18.0	14.7	32.7
	0	−3	0	14.7	20.7	35.4	14.6	20.5	35.1
	0	−1.5	0	15.4	17.9	33.3	15.3	17.8	33.1
	0	1.5	0	16.8	14.7	31.5	16.8	14.8	31.6
	0	0	−3	16.0	15.8	31.8	15.9	15.8	31.7
	0	0	−1.5	16.0	15.9	31.9	16.0	15.9	31.9
	0	0	1.5	16.1	16.4	32.5	16.1	16.3	32.5

遅延評価結果(1次2項近似)

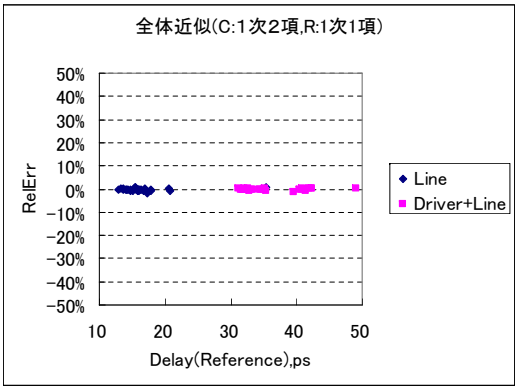
	Variation			Refer.(ps)			全体(1次2項)			最大(1次2項)			近傍(1次2項)		
	Wmt	Tmt	-Tox	Driver	Line	D+L	Driver	Line	D+L	Driver	Line	D+L	Driver	Line	D+L
TYP	0	0	0	16.0	16.0	32.1	16.1	16.0	32.1	16.1	16.0	32.1	16.0	16.0	32.1
全体	1	1	1	18.0	14.2	32.2	17.9	14.2	32.1	17.9	14.2	32.1	18.1	14.6	32.7
	1	2	2	18.7	13.7	32.3	18.7	13.6	32.3	18.6	13.6	32.3	19.0	14.2	33.2
	1	3	3	19.4	13.2	32.7	19.5	13.3	32.7	19.4	13.3	32.7	19.9	14.1	34.0
	2	1	2	20.0	14.4	34.4	19.9	14.4	34.3	19.9	14.4	34.3	20.5	15.3	35.8
	2	2	3	21.1	14.0	35.1	21.0	14.0	35.0	21.0	14.0	35.0	21.8	15.0	36.8
	2	3	1	21.6	13.0	34.6	21.6	12.9	34.5	21.6	12.9	34.5	22.2	13.6	35.8
	3	1	3	23.9	17.0	40.9	23.9	17.0	41.0	24.0	17.1	41.0	25.2	18.5	43.7
	3	2	1	25.0	15.8	40.9	25.0	15.8	40.7	25.0	15.9	40.9	26.0	16.9	42.9
	3	3	2	26.5	15.4	41.9	26.5	15.4	41.9	26.6	15.4	42.1	27.8	16.7	44.5
	3	3	3	26.7	15.5	42.3	26.8	15.6	42.4	26.9	15.7	42.5	28.3	17.1	45.4
最大	0	0	3	16.3	16.8	33.1	16.3	16.8	33.0	16.2	16.7	33.0	16.4	17.3	33.7
	0	3	0	17.5	13.7	31.1	17.5	13.7	31.2	17.5	13.7	31.2	17.5	13.8	31.3
	0	3	3	17.7	14.3	32.0	17.7	14.3	32.0	17.7	14.2	31.9	18.0	14.8	32.8
	3	0	0	22.3	17.3	39.6	22.1	17.0	39.1	22.1	17.1	39.2	22.7	18.1	40.8
	3	0	3	22.6	17.8	40.5	22.6	17.8	40.4	22.6	17.8	40.4	23.6	19.5	43.0
	3	3	0	26.3	15.1	41.4	26.2	15.0	41.2	26.3	15.1	41.5	27.3	16.1	43.4
	3	3	3	26.7	15.5	42.3	26.8	15.6	42.4	26.9	15.7	42.5	28.3	17.1	45.4
近傍	0	0	0.2	16.1	16.1	32.2	16.1	16.1	32.1	16.1	16.1	32.1	16.1	16.1	32.2
	0	0.2	0	16.1	15.9	32.0	16.1	15.9	32.0	16.1	15.9	32.0	16.1	15.9	32.0
	0	0.2	0.2	16.1	15.9	32.1	16.1	15.9	32.0	16.1	15.9	32.0	16.1	15.9	32.0
	0.2	0	0	16.3	15.8	32.0	16.2	15.7	32.0	16.2	15.7	32.0	16.3	15.8	32.0
	0.2	0	0.2	16.3	15.8	32.1	16.3	15.8	32.0	16.3	15.8	32.0	16.3	15.8	32.1
	0.2	0.2	0	16.4	15.6	32.0	16.3	15.5	31.9	16.3	15.5	31.9	16.4	15.6	32.0
	0.2	0.2	0.2	16.4	15.6	32.0	16.4	15.6	31.9	16.4	15.6	31.9	16.4	15.6	32.0
単独	-3	0	0	13.5	35.5	49.0	13.5	35.7	49.2	13.5	35.8	49.3	13.4	33.4	46.8
	-1.5	0	0	14.7	20.5	35.2	14.7	20.5	35.2	14.7	20.6	35.2	14.6	19.9	34.5
	1.5	0	0	18.0	14.8	32.8	17.9	14.7	32.6	17.9	14.7	32.6	18.1	15.1	33.2
	0	-3	0	14.7	20.7	35.4	14.7	20.5	35.2	14.7	20.5	35.2	14.6	20.3	34.9
	0	-1.5	0	15.4	17.9	33.3	15.3	17.8	33.1	15.3	17.8	33.1	15.3	17.7	33.0
	0	1.5	0	16.8	14.7	31.5	16.8	14.7	31.5	16.8	14.7	31.5	16.8	14.8	31.6
	0	0	-3	16.0	15.8	31.8	15.9	15.7	31.6	15.9	15.7	31.7	15.9	15.5	31.3
	0	0	-1.5	16.0	15.9	31.9	16.0	15.8	31.8	16.0	15.9	31.9	15.9	15.7	31.6
	0	0	1.5	16.1	16.4	32.5	16.1	16.3	32.5	16.1	16.3	32.4	16.2	16.5	32.7

遅延評価結果(1次3項近似)

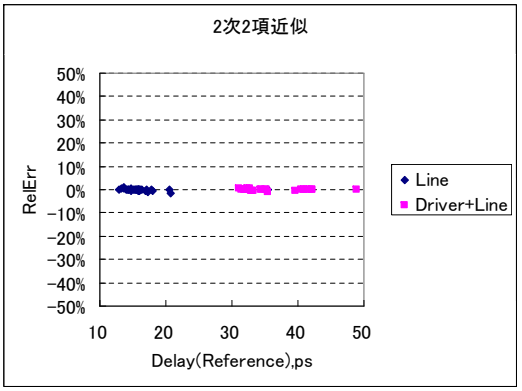
	Variation			Refer.(ps)			全体(1次3項)			最大(1次3項)			近傍(1次3項)		
	Wmt	Tmt	-Tox	Driver	Line	D+L	Driver	Line	D+L	Driver	Line	D+L	Driver	Line	D+L
TYP	0	0	0	16.0	16.0	32.1	15.9	15.6	31.6	16.0	15.8	31.8	16.0	16.0	32.0
全体	1	1	1	18.0	14.2	32.2	18.5	16.0	34.5	18.8	16.3	35.1	18.0	14.0	32.0
	1	2	2	18.7	13.7	32.3	18.9	14.6	33.5	19.6	15.7	35.3	19.0	13.3	32.3
	1	3	3	19.4	13.2	32.7	19.2	13.4	32.6	20.5	15.2	35.7	20.0	12.7	32.7
	2	1	2	20.0	14.4	34.4	21.1	16.8	37.9	21.2	16.9	38.2	19.7	13.0	32.7
	2	2	3	21.1	14.0	35.1	21.6	14.9	36.5	22.4	15.8	38.2	20.9	11.9	32.8
	2	3	1	21.6	13.0	34.6	22.7	14.0	36.7	23.3	14.4	37.7	21.3	10.1	31.4
	3	1	3	23.9	17.0	40.9	23.8	17.2	41.0	23.9	17.3	41.2	21.5	12.2	33.7
	3	2	1	25.0	15.8	40.9	25.1	15.6	40.7	25.2	15.2	40.4	22.0	9.9	31.9
	3	3	2	26.5	15.4	41.9	25.8	13.7	39.5	26.6	13.8	40.4	23.4	8.7	32.1
	3	3	3	26.7	15.5	42.3	25.6	13.5	39.1	26.6	13.9	40.5	23.7	8.9	32.6
最大	0	0	3	16.3	16.8	33.1	15.6	14.6	30.3	16.2	16.3	32.5	16.5	17.6	34.1
	0	3	0	17.5	13.7	31.1	16.8	13.4	30.3	17.7	15.1	32.8	17.6	14.1	31.7
	0	3	3	17.7	14.3	32.0	16.5	12.6	29.1	17.9	15.5	33.4	18.2	15.3	33.5
	3	0	0	22.3	17.3	39.6	23.0	20.3	43.3	22.4	19.0	41.4	19.6	13.2	32.8
	3	0	3	22.6	17.8	40.5	22.7	19.6	42.3	22.6	19.3	42.0	20.2	14.2	34.4
	3	3	0	26.3	15.1	41.4	26.1	14.0	40.1	26.4	13.6	40.1	22.9	8.3	31.2
	3	3	3	26.7	15.5	42.3	25.6	13.5	39.1	26.6	13.9	40.5	23.7	8.9	32.6
近傍	0	0	0.2	16.1	16.1	32.2	15.9	15.6	31.5	16.0	15.8	31.8	16.1	16.1	32.2
	0	0.2	0	16.1	15.9	32.0	16.0	15.4	31.4	16.1	15.8	31.9	16.1	15.9	32.0
	0	0.2	0.2	16.1	15.9	32.1	16.0	15.4	31.4	16.1	15.8	31.9	16.1	15.9	32.0
	0.2	0	0	16.3	15.8	32.0	16.3	16.1	32.4	16.4	16.1	32.5	16.3	15.8	32.1
	0.2	0	0.2	16.3	15.8	32.1	16.3	16.0	32.4	16.4	16.1	32.5	16.3	15.8	32.1
	0.2	0.2	0	16.4	15.6	32.0	16.4	15.9	32.3	16.5	16.0	32.5	16.4	15.6	32.0
	0.2	0.2	0.2	16.4	15.6	32.0	16.4	15.8	32.2	16.5	16.0	32.5	16.4	15.6	32.0
単独	-3	0	0	13.5	35.5	49.0	0.0	0.0	0.0	0.0	0.0	0.0	13.1	29.5	42.6
	-1.5	0	0	14.7	20.5	35.2	17.6	12.0	29.7	18.0	13.7	31.7	14.5	19.2	33.7
	1.5	0	0	18.0	14.8	32.8	24.1	18.5	42.6	23.9	17.8	41.7	17.8	14.4	32.2
	0	-3	0	14.7	20.7	35.4	19.7	20.5	40.2	19.2	18.2	37.4	14.6	19.8	34.4
	0	-1.5	0	15.4	17.9	33.3	20.3	17.8	38.0	20.1	16.7	36.8	15.3	17.5	32.8
	0	1.5	0	16.8	14.7	31.5	21.3	14.6	35.8	21.7	15.6	37.3	16.8	14.9	31.7
	0	0	-3	16.0	15.8	31.8	20.8	16.4	37.2	20.8	15.9	36.7	15.8	15.3	31.1
	0	0	-1.5	16.0	15.9	31.9	20.9	16.2	37.1	20.9	15.9	36.7	15.9	15.6	31.5
	0	0	1.5	16.1	16.4	32.5	20.7	15.5	36.2	20.9	16.3	37.2	16.2	16.6	32.8

遅延精度評価

•容量1次2項ル(Wmt / Tox, Tmt / S)および抵抗1次1項(1 / Wmt Tmt)モデルを用いた場合、遅延平均誤差は約1%であり、2次式モデル(容量:Wmt / Tox, Tmt / S; 抵抗:1 / Wmt, 1 / Tmt)とほぼ同等の精度を得ることが可能である。



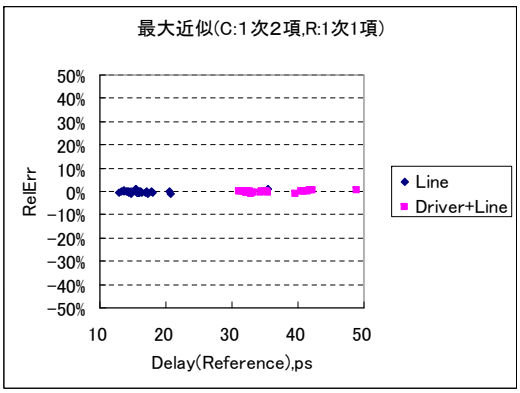
最大誤差	1.4%
平均誤差	0.4%



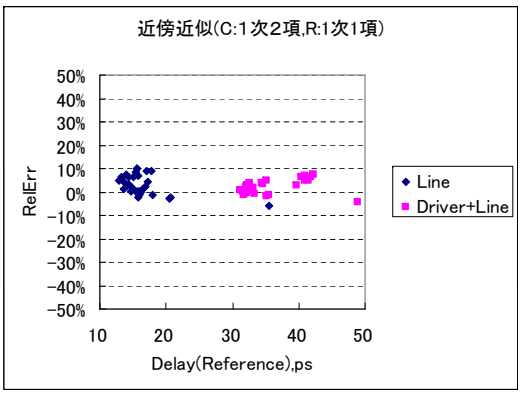
最大誤差	0.9%
平均誤差	0.3%

遅延精度評価(最大近似,近傍近似)

•容量1次2項モデル(Wmt/Tox, Tmt/S)抵抗1次1項モデル(1/WmtTmt)の場合、最大近似(3σ)であれば、最大誤差は約1%であり、2次モデルとほぼ同等の精度を得ることができる。
•ただし、近傍近似の場合、最大7%の遅延誤差が発生する。



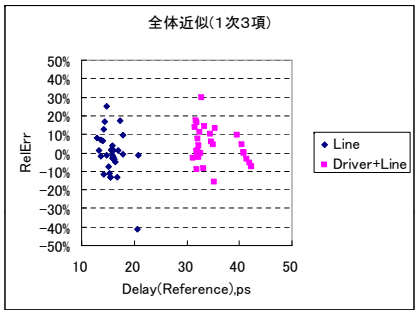
最大誤差	1.0%
平均誤差	0.4%



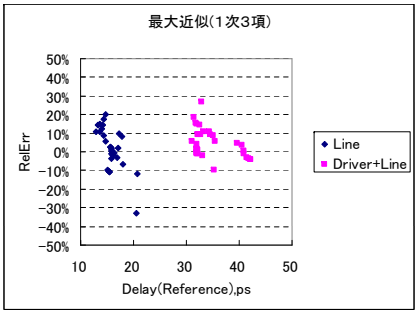
最大誤差	7.3%
平均誤差	3.4%

遅延精度評価(1次3項近似)

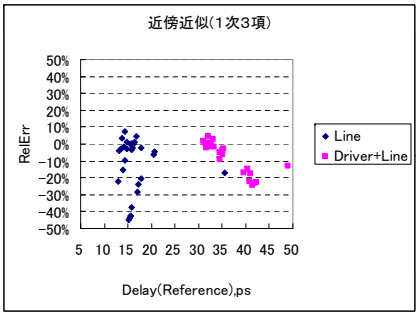
•容量1次3項モデル(Wmt,Tmt,Tox)、抵抗1次2項モデル(1/Wmt,1/Tmt)を用いた場合、全体近似(0~3σ)、最大近似(3σ)、近傍近似(0．2σ)に関わらず、20%を超える遅延モデル誤差が発生する。



最大誤差	29.7%
平均誤差	9.5%



最大誤差	27.0%
平均誤差	8.9%



最大誤差	24.6%
平均誤差	10.7%

まとめ

- 22nmプロセスを想定し、配線ばらつき込みネットリストフォーマットSensitivity - SPEF(IEEE1394 - 2009)の誤差評価を行った結果、以下の知見を得た。
  - 配線幅、配線膜厚、絶縁膜厚の3変数をパラメータとした場合、近似方法に関わらず、30%を超える容量誤差が発生する。
  - 配線幅 / 配線間膜厚(W / Tox)、配線膜厚 / 配線間隔(Tmt / S)の2変数をパラメータとすることで、容量誤差を3%まで抑えることができ、Sensitivity-SPEFの1次感度式モデルを用いた場合の実設計適用可能性を得た。

- 上記Sensitivity-SPEFモデルを使用した場合、22nmプロセスを想定したテスト回路においての遅延誤差は、最大1%程度であった。

残された検討項目

- 複数レイアウトパターンでの評価
  - 今回の評価では、単純レイアウトパターンでの評価を行った。複数のレイアウトパターンに対して、今回のモデル精度の確認を行う。
- 感度式の評価
  - Sensitivity-SPEFで定義された感度モデルと物理モデルとの差異を解析的に評価し、誤差の要因を明確化する。
- Sensitivity-SPEFフォーマットの改善検討
  - Sensitivity-SPEFに関する他の課題調査、および改善検討を行う。

2.プロセスばらつきを考慮した集積回路のNBTI故障率解析

目次

- 背景と目的
- NBTIしきい値電圧変動ばらつきの実態
  - 平均値
  - ばらつき分布
- 22nmプロセスでのNBTIしきい値電圧ばらつき解析
  - デバイスレベルの解析
  - 回路レベルの遅延解析
- 結果の検討
  - 回路遅延分布の評価
  - 1FITレベルのワースト遅延特性
- まとめ

背景と目的

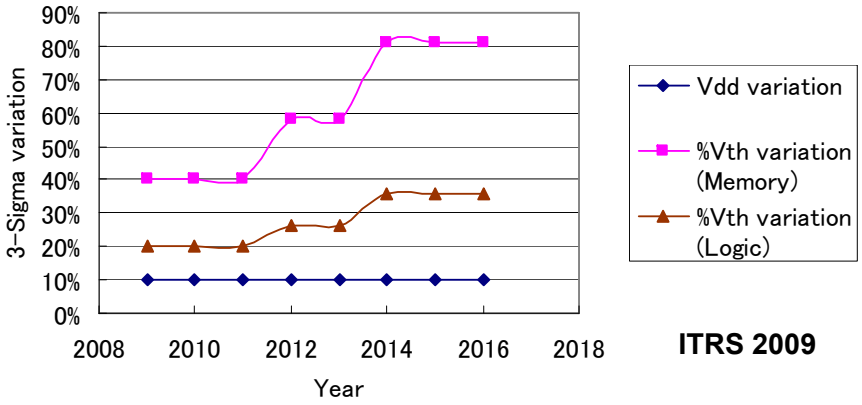
- 背景:
  - 車載用SOCの信頼性設計では1FITレベルの品質保証が要求される。
  - PMOSのNBTI特性劣化とその分布形状は、その品質保証に影響しうる。
  - さらに、NBTIのしきい値電圧劣化の分布形状が正規分布と異なり、しきい値電圧のシフトの大きい領域ですそ引きが見られることが明らかになってきた。
- 目的:
  - NBTI劣化特性のばらつき分布を正確に表現すること
  - 22nmCMOSでのNBTI劣化特性を予測すること
  - 22nmCMOSインバータ回路の遅延のワーストケース特性を定量的に評価すること
- 開発技術:
  - 遅延ワースト特性のモンテカルロ解析の高速化手法
  - NBTI回路故障率を予測するメソドロジー

取り扱うばらつき量

- 回路遅延特性のばらつき要因:
  - RDF(Random dopant fluctuation)
    - NMOSおよびPMOS
  - NBTIしきい値電圧変動とばらつき
    - PMOS
  - 簡単化のため、チップ間のしきい値電圧ばらつきはなしと仮定
- ばらつき分布の形状
  - RDF:正規分布でフィッティング
  - NBTI:正規分布及び対数正規分布でフィッティング

RDFのトレンドと22nmプロセスVthばらつき予測

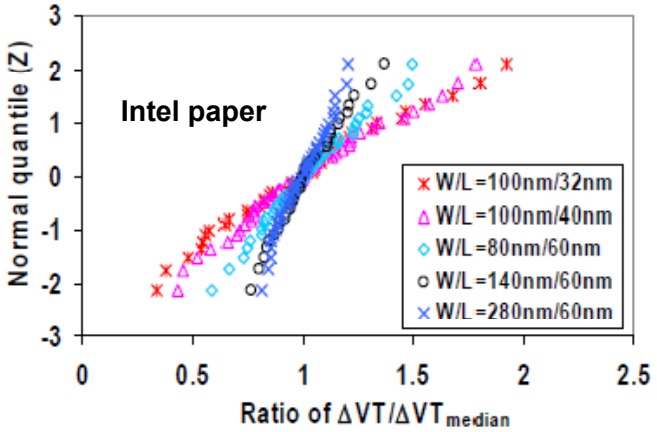
- ITRS 2009より、しきい値電圧(Vth)ばらつきを予測する。
- 22nmプロセス(2015年)のVthばらつき3σ(Vth)値＝36%として、σ(Vth)＝12%。したがって、Vth中心値=0.2V、を仮定するとσ(Vth)＝24mV。
- 22nmプロセスのNMOS＆PMOSのRDFによるVthばらつきはσ(Vth)＝24mV、とした。



ITRS 2009

NBTIしきい値電圧シフト量の分布(1)

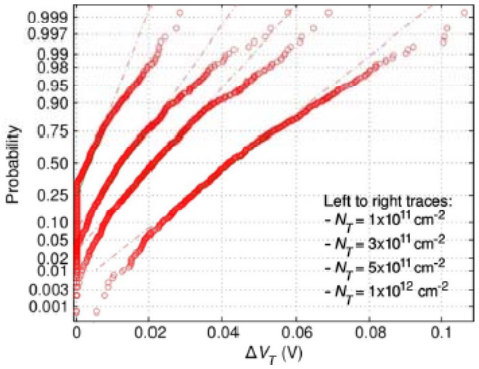
正規確率紙にプロットした、NBTIしきい値電圧シフト量分布の実測値。
正規確率紙上で、直線(正規分布)に乗らず、分布が歪んでいる。特に、±3σ近傍でずれが大きい。
NBTIしきい値電圧シフト量分布≠正規分布！

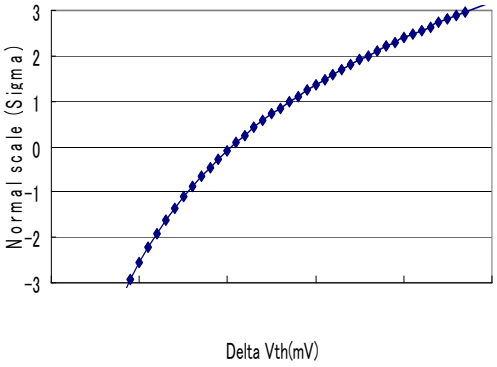


S. Pae et al., EFFECT OF NBTI DEGRADATION ON TRANSISTOR VARIABILITY IN ADVANCED TECHNOLOGIES, pp. 18-21, 2007 IIRW FINAL REPORT

NBTIしきい値電圧シフト量の分布(2)

左図:NBTIシフト量分布の3次元デバイスシミュレーションの正規確率紙上の表示
右図:対数正規確率分布の正規確率紙上の表示
両者はよく形状が一致している
NBTI Vthシフト量分布は対数正規分布で近似できる可能性が高い。



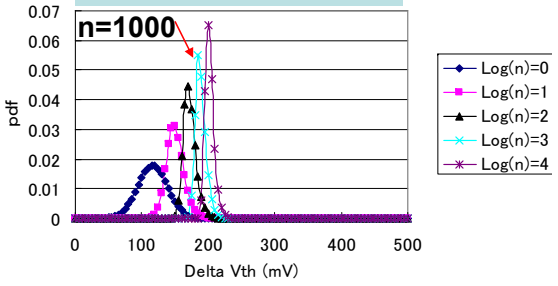


Asenov et al.:
IEEE TRANSACTIONS ON ELECTRON DEVICES, VOL. 57, NO. 4, APRIL 2010

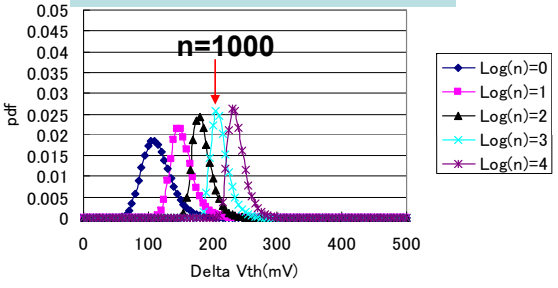
モデルの違いによるワースト分布の違い

- 同じ平均値と標準偏差を持つ正規分布と対数正規分布は、その裾野(ワーストケース)の挙動に差が見える。
  - 信頼性などにおいてワーストケースを議論するときは使われるモデルに注意。
- 正規＆対数正規分布のMax(n)分布「**n個のサンプリングの中の最大値(ワースト値)をとったものの分布**」をプロットしたものが下図。
- n=1000の場合、n=1に対してVthがシフトする。
  - 正規分布で220mV, 対数正規分布で290mV。モデル種により約70mVの差が生じる。

正規分布のMAX(n)分布

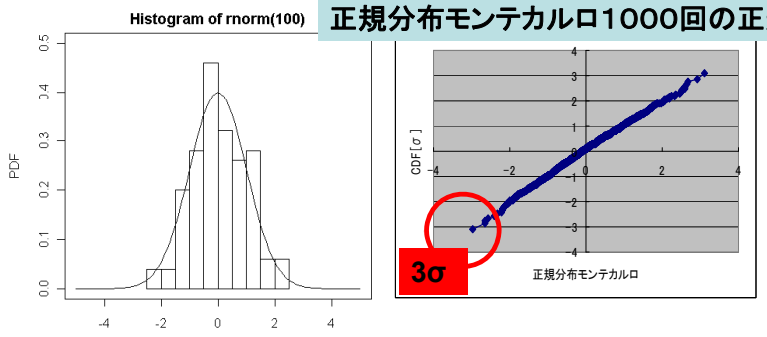


対数正規分布のMAX(n)分布



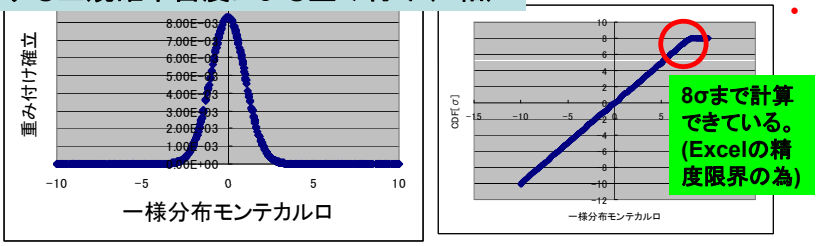
モンテカルロの課題と重み付け

正規分布モンテカルロ1000回の正規確率紙プロット



- 通常の正規分布を用いたモンテカルロでは、1000回でも、3σを表現することさえ難しい。

一様分布[±10]モンテカルロ1000回(X軸)と、対応する正規確率密度による重み付け(Y軸)

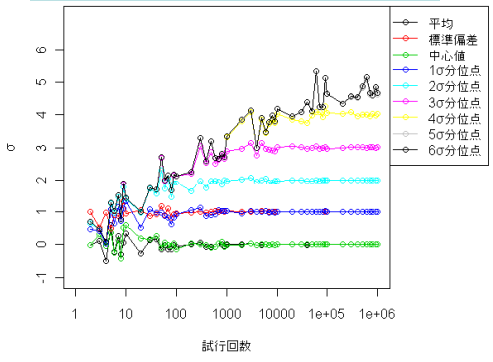


8σまで計算できている。(Excelの精度限界の為)

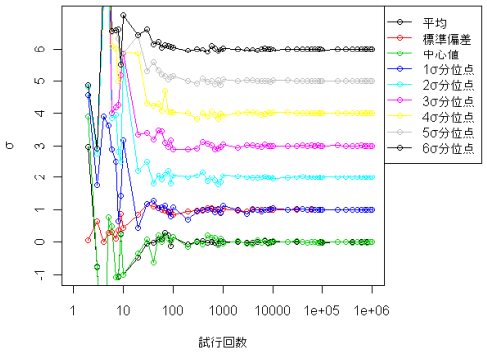
- 着目するところに特化したモンテカルロ(Important Sampling)を行い、**出現確率に重み付け**を行うことで解析精度の向上が可能

モンテカルロの試行回数精度評価

通常の正規分布のモンテカルロ



一様分布と重み付けによるモンテカルロ



- モンテカルロの試行回数と、各分位点(quantile)の変動を評価
  - 正規分布はN(μ=0,σ=1)
  - 一様分布は±10とし、正規確率密度による重み付けを行う。
- 一様分布を用いると、数百回の試行で分布の裾野(6σ)でも値が収束することがわかる。
  - その分、数十回の試行では、平均値や中心値の精度が落ちる。
- この手法を用いるには、**目的に合わせた分布の選択**と、結果に対して、**出現確率の重み付けを扱える仕組みやツール**が必要。

Max分布を用いたモンテカルロ高速化

- アイデア:
  - NBTIのばらつき現象を表現する確率密度分布 f(x)と、その分布のMax(n)分布を組み合わせて、Important Samplingを行う。
- アルゴリズム:
  - 確率密度分布 f(x) でm回のランダム・サンプリングを実施する
  - f(x)のMax(n)分布 $g_n(x)$ でm回のランダム・サンプリングを実施する
  - f(x)の重み付けを"n"、$g_n(x)$ の重み付けを"1"として、統計解析を実施
- アルゴリズムの正当性
  - Max(n)分布 $g_n(x)$は、f(x) 上でn個のサンプリングした結果の最大値の1個をとったものである。
  - 最大値以外の、残りのn-1個のサンプルは、 f(x)をサンプリングして得たデータ全体から最大より1/nを取り除いたデータで代用すると考える
    - nが十分大きい場合は、f(x)をそのまま利用する。
  - したがって、$g_n(x)$ に対して、f(x) をn倍重み付けすることで、全体のサンプリングによるヒストグラム形状が復元できる。
  - $g_n(x)$により、MAX値近傍にデータ点が取れるため、+6σレベルのデータ点の精度向上が見込まれる。

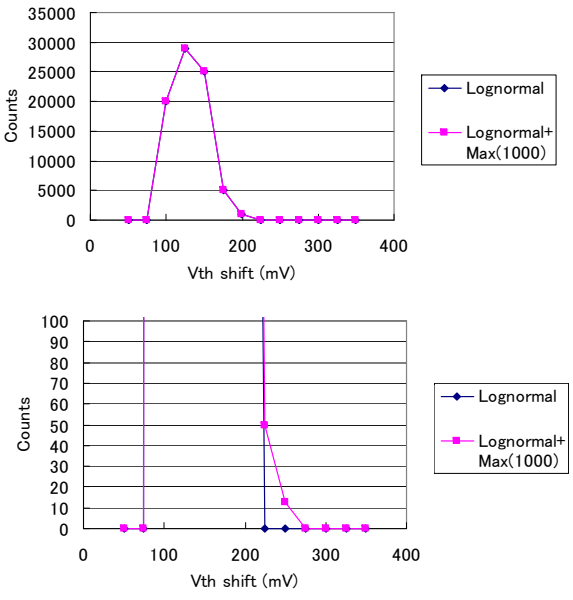
すそ野分布のモンテカルロ高速化のイメージ

Important Samplingによって、分布のすそ野の部分が、正確にサンプリングされていることが分かる。

モンテカルロ条件
- f(x) サンプル数＝80、重み係数=1000
- $g_{1000}(x)$ サンプル数=80 、重み係数=1

シミュレーション結果
- 青印:
  f(x) サンプル数＝80、重み係数=1000
- 赤印:
  (f(x) サンプル数＝80、重み係数=1000)+
  ($g_{1000}(x)$ サンプル数=80 、重み係数=1)

図面
- 上図:ヒストグラム全体図
- 下図:カウント数の小さい部分拡大



NBTIモデルと22nmプロセスでのパラメータ

- NBTIによるしきい値電圧の変動量(ΔVth)の平均値 μのモデル

$$\mu(\Delta V_{th}) = At^{n},\quad n = 1/6$$

- NBTIによるしきい値電圧の変動量(ΔVth)のばらつき（標準偏差）σのモデル

$$\sigma(\Delta V_{th}) = Bt^{m},\quad m = 1/12$$

- 正規分布ばらつきを仮定した場合のパラメータ

$$f(y) = \frac{1}{\sqrt{2\pi}\sigma_m}\exp\left[-\frac{(y-\mu_m)^2}{2\sigma_m^2}\right]$$

$$\mu_m = 19.6\times t(hr)^{1/6}\,mV,\ \sigma_m = 9.16\times t(hr)^{1/12}\,mV$$

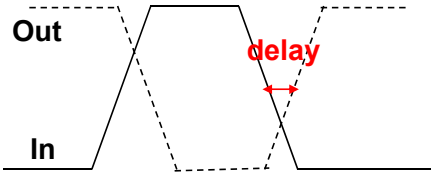
- 対数正規分布ばらつきを仮定した場合のパラメータ

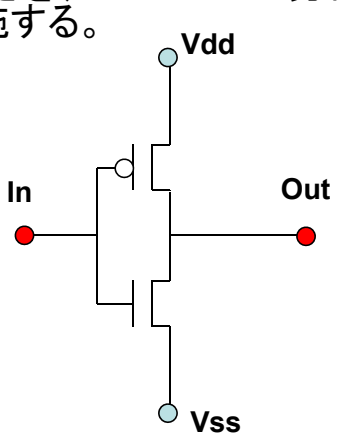
$$f(y) = \frac{1}{y\sqrt{2\pi}\sigma}\exp\left[-\frac{(\ln(y)-\mu)^2}{2\sigma^2}\right]$$

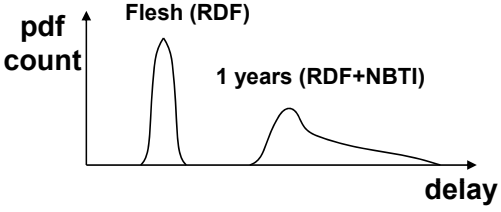
$$\mu = \ln\mu_m - \frac{1}{2}\ln\left(\frac{\sigma_m^2}{\mu_m^2}+1\right),\ \sigma = \sqrt{\ln\left(\frac{\sigma_m^2}{\mu_m^2}+1\right)}$$

遅延劣化解析用回路（インバータ）

- 立ち上がり遅延時間のばらつきを、RDF＋NBTI劣化を考慮し、回路シミュレーションを実施する。

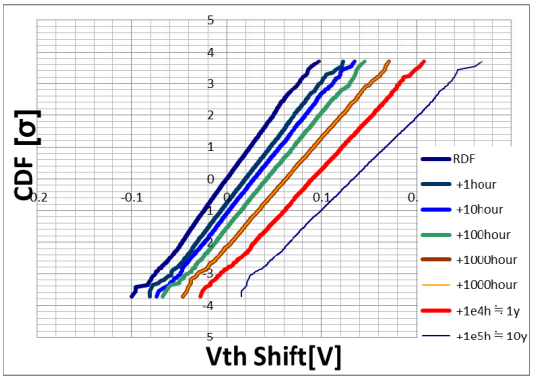


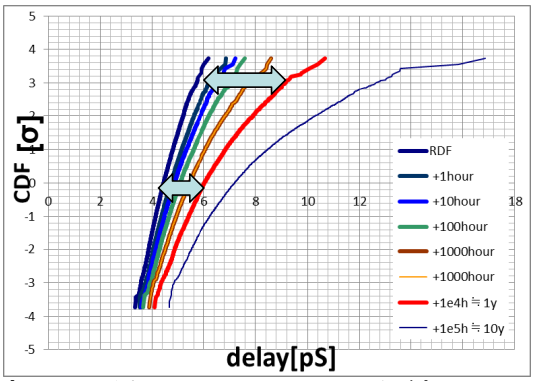




項目	値
SPICEモデルパラメータ	PTM 22nm HP
SPICE電流モデル	BSIM4
電源電圧	0.8V
基本のトランジスタサイズ	P W/L=400nm/22nm
	N W/L=300nm/22nm

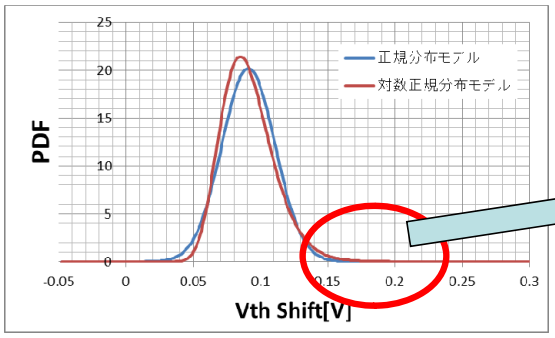
遅延劣化解析

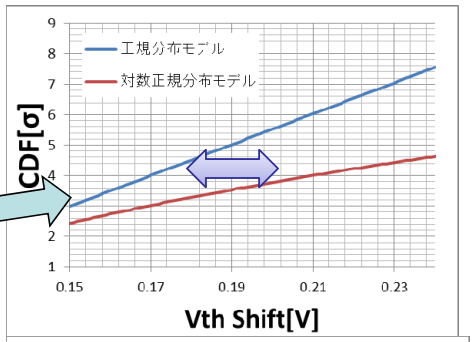


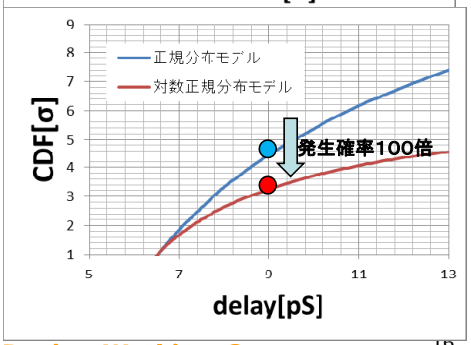


- NBTIのばらつきを正規分布で仮定し、遅延シミュレーションした結果(モンテカルロ1万回)
- 1万時間後の劣化量
  - Vth Shift：約90mV
  - Delay：
    - +36%（typical）
    - +60%(+3σ)

NBTIモデルの違いと1Fitレベルの解析







- RDF+NBTI(10000hour)
  - RDFは正規分布
  - NBTIの分布は、正規分布または対数正規分布
  - 解析は、一様分布と確率密度の重み付けを使用
- NBTIモデルの違いによる、1Fit(≒4.3σ@1万hour)レベルの特性差
  - Vth Shift ⇒ +50mV
  - Delay ⇒ +34%
- 正規分布仮定では1Fit(4.3σ)であったものが、**対数正規分布であった場合、約100Fit(3.2σ)相当になる。**

まとめ

- RDFとNBTIを考慮し、22nm世代のCMOSにおける劣化特性のばらつき分布を回路遅延として表現し、解析を行った。
- NBTIによる劣化のばらつきモデルとして、対数正規分布を提案し、正規分布による見積もりとの差を、1Fitレベルで定量的に評価を行った。
  - 対数正規分布である場合、そのワーストケースにおいて、故障率の差が大きく見えてくる。
- 信頼性のばらつき設計技術・解析技術を開発していく上で、分布の裾野の微小確立ケース（ワーストケース）を高速に解析する手法の開発が今後必要になってくると思われる。

参考文献

- ITRS 2009; http://www.itrs.net/links/2009ITRS/Home2009.htm
- S. Pae, J. Maiz and C. Prasad,: Effect of NBTI Degradation on Transistor Variability in Advanced Technologies, pp. 18-21, 2007 IIRW Final Report.
- Muhammad Faiz Bukhori, Scott Roy, and Asen Asenov, ; Simulation of Statistical Aspects of Charge Trapping and Related Degradation in Bulk MOSFETs in the Presence of Random Discrete Dopants, IEEE T-ED, VOL. 57, NO. 4, pp.795-803, APRIL 2010.
- Kunhyuk Kang, Sang Phin Park, and Muhammad A.Alam,; Estilna On of Statistical variation in Temporal NBTI Degrada On and its Impact On Lifetime Circuit Performances, ICCAD 2007, pp.730-734, 2007.
- Kanj, R.; Joshi, R.; Nassif, S.; Mixture Importance Sampling and Its Application to the analysis of SRAM designs in the presence of rare failure events, Design Automation Conference, 2006 43rd ACM/IEEE, pp.69-72, 2006.

Documentation on design automation subjects –

The Bird's-eye View of Design Languages (BVDL)

Version 1.0

素案

The EDA Standardization Technical Subcommittee
The EDA Technical Committee
Japan Electronics and Information Technology Industries Association

CONTENTS

1. Introduction
1.1. Background
1.2. Objectives
2. Scope
3. Structure and Content of the Bird's-eye View for Design Languages
3.1. Structure of the Bird's-eye View for Design Languages
3.2. Chart of Design Processes
3.3. Table of "Electronic System Design"
3.4. Table of "SoC Design"
3.5. Table of "Mixed-Signal Verification and Analog Block Design"
3.6. Table of "Characterization and IP Preparation"
3.7. Reading the Bird's-eye View for Design Languages
4. Use Case of the Bird's-eye View for Design Languages
5. The Bird's-eye View of Design Languages (BVDL) Version 1.0

Figures

Figure 1 Electronic Design Ecosystem
Figure 2 Chat and table in BVDL.
Figure 3 Structure of the table.
Figure 4 Chart of design processes.
Figure 5 “Electronic System Design” table
Figure 6 Part of “SoC Design” table
Figure 7 Part of “Mixed-Signal Verification and Analog Block Design” table
Figure 8 “Characterization and IP Preparation” table
Figure 9 Multiple marks in one design object
Figure 10 Multiple marks in the same design objects in the different processes

1. Introduction

1.1. Background

The automation of design and manufacturing technologies in electronic industries has been evolving world-wide for over three decades with remarkable development speed. Electronic design automation (EDA) technology enables the concept, implementation and validation of electronic systems, that is, transforms ideas and objectives of the system designers into manufacturable and testable representations in a cost-effective way. It is classified into three key factors such as design methodologies, design libraries and design tools. Standardization involves computer-sensible representations throughout overall design processes which integrate design libraries and design tools to build design ecosystem.

In the semiconductor industry EDA technologies have been substantially contributing to the unprecedented industry growth for three decades. To emerging new product-lines such as microcontroller, microprocessor, ASIC, FPGA , Memories, Analog & Mixed-signal and System on a chip (SoC) it has been continuously providing a wide-range of solutions to meet critical requirements on design productivity enhancement and design quality improvement.
EDA standards provide a mechanism to define common semantics for electronic design ecosystems among various design tools depicted in Figure 1. The state-of-the-art standards are classified into hardware description languages, hardware verification languages, electronic system level design languages, library formats, design constrain formats, interface formats with manufacturing and testing, design data exchange formats, data models and application procedure interfaces (API), etc. Therefore they are generally called standard design languages in a narrow sense. The semiconductor industry has been facing new design complexity barriers and is today facing unprecedented complexities brought by convergence of product features in terms of silicon process technology, system technology, high gate count and embedded software incorporation. This new design complexity requires integrated EDA solutions and at the same time impacts design ecosystem and standard design languages as well. So a new design language development or new features enhancement to an existing design language is needed. In result tens of design languages, which might be classified into De Jure standard language, De Facto standard language, Forum standard language and common language used in some community, are developed, enhanced or actually used in the industries, academia and communities world-wide.

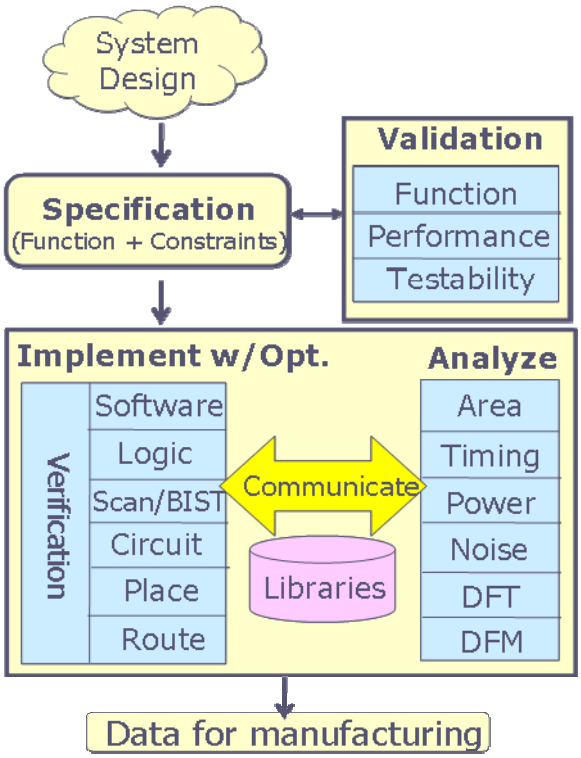


Figure 1 Electronic Design Ecosystem

The EDA technical committee (EDA-TC) was formed in JEITA in 1990 in order to take initiatives of international EDA standardization in Japan. Since then, they have been contributing design language standardization such as EDIF, VHDL, Verilog HDL, Delay and Power Calculation (DPC), System C, System Verilog and Power Format, to newly form the working group inviting experts form the industry and academia and work with IEC TC93, IEEE DASC, Accellera, Open SystemC Initiatives (OSCI) and others. Throughout those activities for over two decades they do feel necessity of bird's-eye view of existing, enhancing or developing tens of design languages in order to set strategy and the following action plans towards international EDA standardization. EDA-TC initiated the project in early 2009 to develop the Bird's-eye View of Design Languages (BVDL) spreadsheet documentation. They developed the first version in March, 2010, to have a lot of participation of design technology experts from the semiconductor industry and academia. They finalized BVDL documentation combined with the spreadsheet as a JEITA technical report in March, 2011.

1.2. Objectives

The BVDL originally aims to make full use of planning and decision-making on EDA standardization activities for a technical expert as well as a management in JEITA. It does facilitate understanding of various design languages to show positioning and features. Also it helps to easily know overviews of each design language for a new comer to EDA standards community and/or a designer as a user of EDA design ecosystem. Especially for a design language developer to directly join design language development and voting for standardization, it provides metrics to check duplication among similar languages, consistency to develop design ecosystem and future challenge for design languages.

2. Scope

This technical report describes features for existing, enhancing and newly developing design languages belonging to the defined design processes of System-on -a Chip (SoC) which ranges from system level design, SoC design implementation & verification, IP block creation and analog block design down to interface data preparation for manufacturing. This simplified design processes might not be obsolete by impact from electronic design automation evolution with the remarkable speed and seem easier to understand for a non-EDA expert.

Design languages handled in the report show in the following:
UML, Esterel, Rosetta, SystemC, SystemC-AMS, IBIS, CITI, TouchStone, BSDL, System Verilog, VHDL, Verilog HDL, UPF, CPF, e, PSL, FSDB, SDC, DEF, Open Access, SDF, GDS II, OASIS, STIL, WGL, Verilog-A, Verilog-AMS, SPICE, VHDL-AMS, LEF, Liberty, CDL, IP-XACT.

3. Structure and Content of the Bird's-eye View for Design Languages

3.1. Structure of the Bird's-eye View for Design Languages

In this sub-section, the overall structure of the bird's-eye view for design languages (BVDL). BVDL consists of one chart and four tables (see Figure 2).

Chart Table

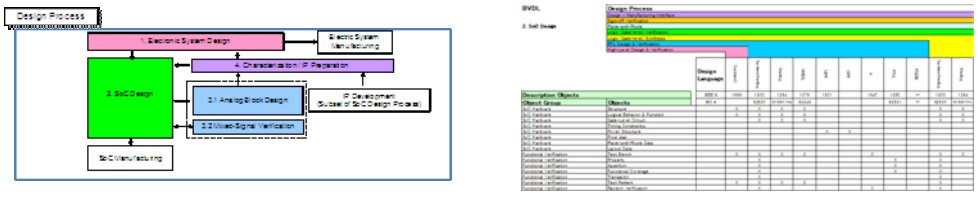


Figure 2 Chat and table in BVDL.

The purpose of BVDL is to show the design languages' positions and features in the design processes. To help to recognize them, major design processes are defined and design processes are classified into them. They are "Electronic System Design", "SoC Design", "Mixed-Signal Verification and Analog Block Design", and "Characterization and IP Preparation". The char of BVDL shows the relations between the major design processes.

The design processes which belong to each major design process are listed in the four tables. Each table has the structure in which it is suitable to recognize the positions and features of design languages. The design languages which are grouped according to design flow are in columns of the tables. Design objects are in rows of the tables. The design objects are what designers design in the design processes. For example, they are hardware description, verification description, design constraint, and so on. Furthermore, they are grouped into "Object Group" which represents category. The tables have marks which show the positions and features of languages. One language may appear in only one column or may appear in several columns (see Figure 3).

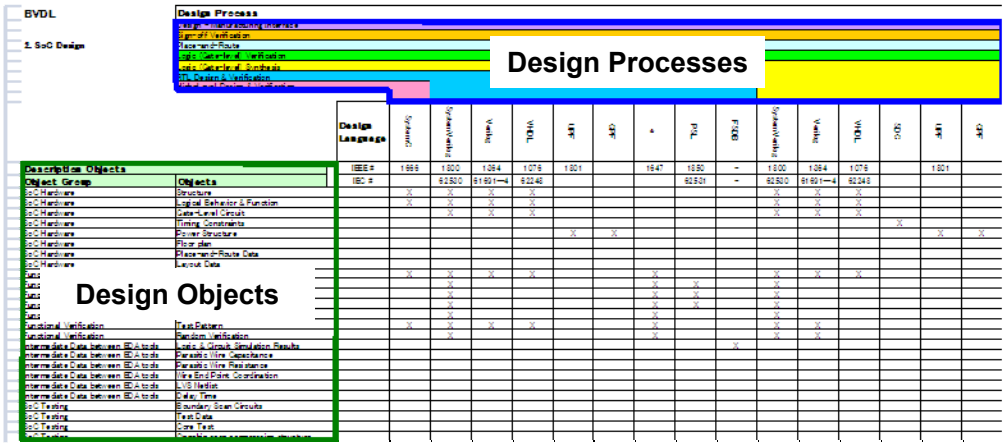


Figure 3 Structure of the table.

3.2. Chart of Design Processes

In this sub-section, the content of the chart of design processes is explained (see Figure 4).

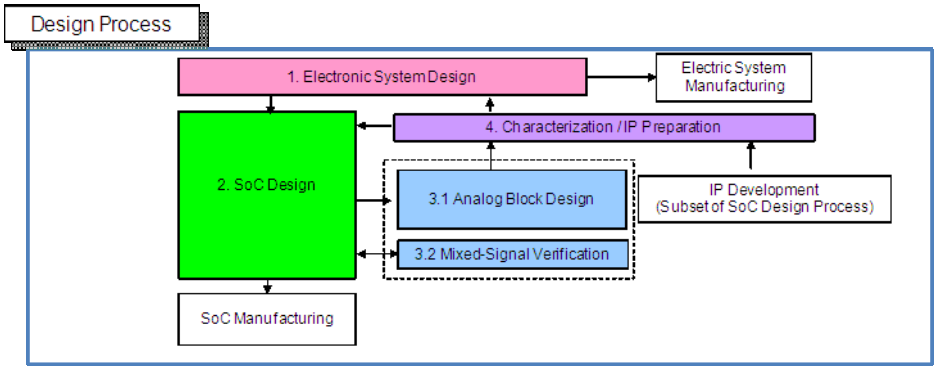


Figure 4 Chart of design processes.

In this chart, four major design processes, “Electronic System Design”, “SoC Design”, “Mixed-Signal Verification and Analog Block Design", and “Characterization and IP Preparation” are defined.

The “Electronic System Design” process is a design process to develop electronic systems. In the design process, system requirements are analyzed and then the system specifications are defined. According to the specifications, printed wiring boards, packages, system architectures, and algorithms to implement the specifications.

The “SoC Design” process is a design process to develop System-on-Chips (SoC). They are

designed according to the specifications, system architectures, and algorithms which are designed in the “Electronic System Design” process. In the BVDL, only the digital parts of SoC designs are focused. Although SoCs include analog circuits, design processes of analog circuits are considered in the “Analog Block Design” process.

The “Analog Block Design” process is a design process to develop analog blocks. They are provided to the SoC design process as IPs through the “Characterization and IP Preparation” process. In the “Mixed-Signal Verification” process, interfaces between digital parts and analog parts in SoC are verified.

In the “Characterization and IP Preparation” process, digital IPs and analog IPs are prepared. These IPs are provided to the other design processes.

3.3. Table of “Electronic System Design”

In this sub-section, the content of the “Electronic System Design” table is explained (see Figure 5).

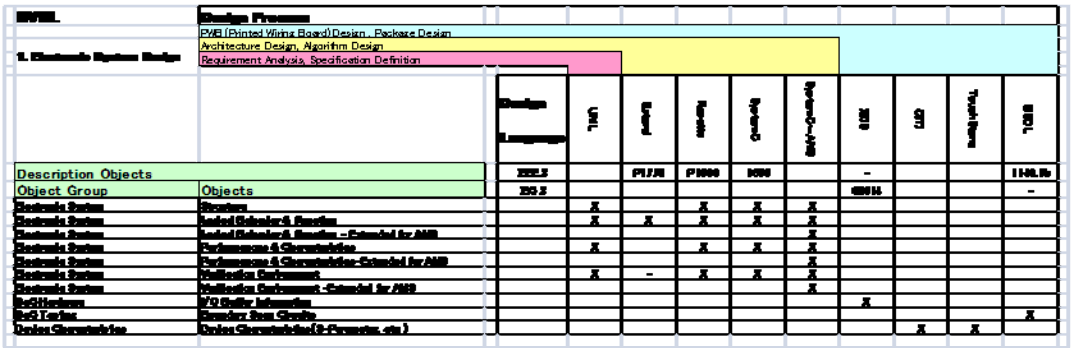


Figure 5 “Electronic System Design” table

The columns show the design processes in the “Electronic System Design” major design process. The design processes are printed wiring board designs and package designs, architecture designs and algorithm designs, and requirement analyses and specifications definition. The design languages related to the design processes are listed in the columns.

The rows show design objects which are designed in the design processes in the columns. They are structures, logical behaviors and functions, performances and characteristics, verification environments, I/O buffer information, boundary scan circuits, and device characteristics. Their granularity should be small enough to make clear the deference

between languages.

The design objects are grouped into four object groups, electronic system, SoC hardware, SoC testing, and device characteristics. The design objects, structures, logical behaviors and functions, performances and characteristics, and verification environments belong to the electronic system object group. The design object, I/O buffer information, belongs to the SoC hardware object group. The design object, boundary scan circuits, belongs to the SoC testing object group. The deign object, device characteristics, belongs to the device characteristics object group.

3.4. Table of “SoC Design”

In this sub-section, the content of the “SoC Design” table is explained (see Figure 6).

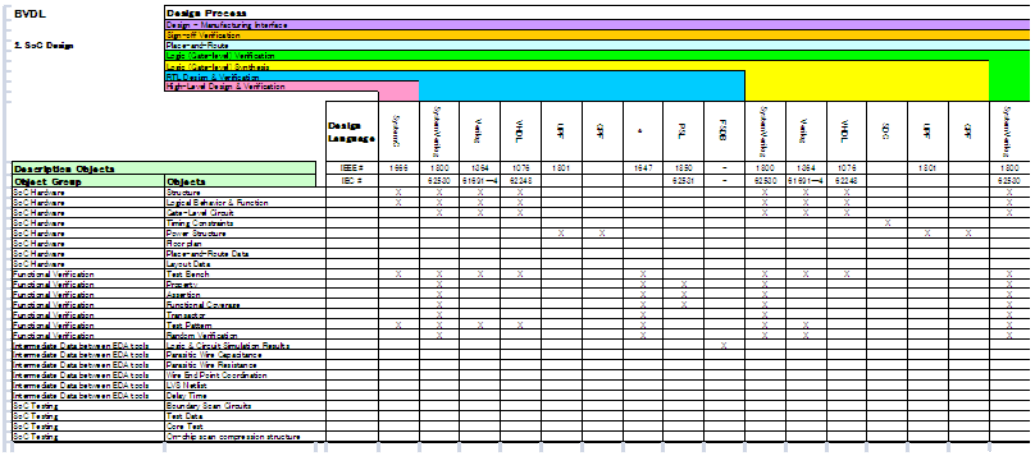


Figure 6 Part of “SoC Design” table

The columns show the design processes in the “Soc Design” major design process. The design processes are high-level designs and verifications, RTL designs and verifications, logic synthesis, logic verifications, place and route, sign-off verification, and design manufacturing interface. The design languages related to the design processes are listed in the columns.

The rows show design objects which are designed in the design processes in the columns. They are structures, logical behaviors and functions, gate-level circuits, timing constraints, power structures, floor plans, place and route data, layout data, test benched, properties, assertions, functional coverage, transactors, test patterns, random verifications, logic and circuit simulation results, parasitic wire capacitance and resistance, wire end point

coordinations, LVS netlist, delay time, boundary scan circuits, test data, core tests, and on-chip scan compression structures. Their granularity should be small enough to make clear the deference between languages.

The design objects are grouped into four object groups; SoC hardware, functional verification, intermediate data between EDA tools, and SoC testing. In each object group, the design objects are as follows:
In the SoC hardware object group, structures, logical behaviors and functions, gate-level circuits, timing constraints, power structures, floor plans, place-and-route data, and layout data; in the functional verification object group, test benches, properties, assertions, functional coverage, transactors, test patterns, and random verifications; in the intermediate data between EDA tools object group, logic and circuit simulation results, parasitic wire capacitances and resistances, wire end point coordinations, LVS netlists, and delay time; in the SoC testing object group, boundary scan circuits, test data, core tests, and on-chip scan compression structures.

3.5. Table of “Mixed-Signal Verification and Analog Block Design”

In this sub-section, the content of the “Mixed-Signal Verification and Analog Block Design” is explained (see Figure 7).

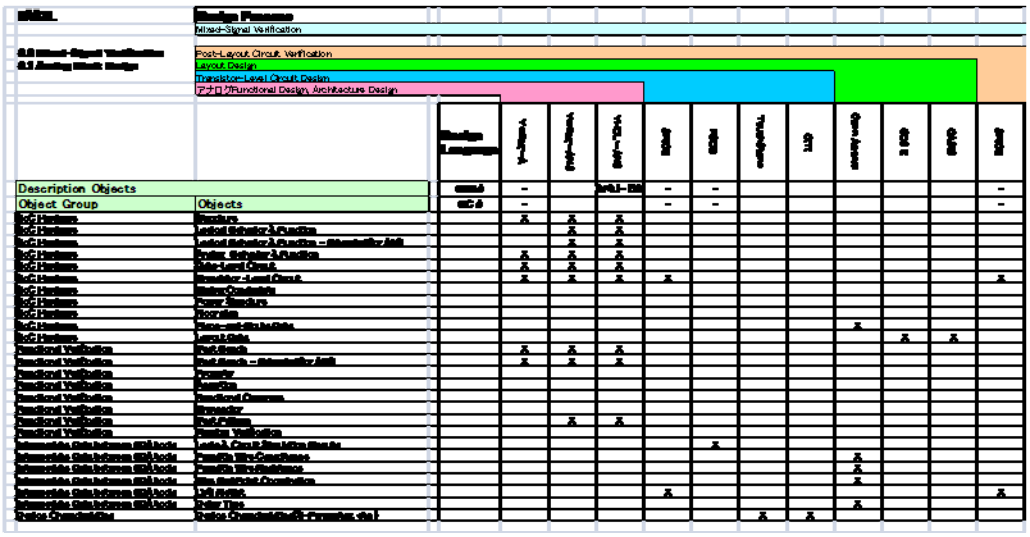


Figure 7 Part of “Mixed-Signal Verification and Analog Block Design” table

The columns show the design processes in the “Mixed-Signal Verification and Analog Block Design” major design process. The design processes are analog functional designs,

architecture designs, transistor-level circuit designs, layout designs, post-layout circuit verifications, and mixed-signal verifications. The design languages related to the design processes are listed in the columns.

The rows show design objects which are designed in the design processes in the columns. They are structures, logical behaviors and functions, logical behaviors and functions (extended for AMS), analog behaviors and functions, gate-level circuits, transistor-level circuits, timing constraints, power structures, floor plans, place-and-route data, layout data, test benches (also extended for AMS), properties, assertions, functional coverage, transactors, test patterns, random verifications, logic and circuit simulation results, parasitic wire capacitances and resistances, wire end point coordinations, LVS netlists, delay time, and device characteristics. Their granularity should be small enough to make clear the deference between languages.

The design objects are grouped into four design groups; SoC hardware, functional verification, intermediate data between EDA tools, and device characteristics. In each design group, the design objects are as follows:
In the SoC hardware design group, structures, logical behaviors and functions, logical behaviors and functions (extended for AMS), analog behaviors and functions, gate-level circuits, transistor-level circuits, timing constraints, power structures, floor plans, place-and-route data, and layout data; in the functional verification object group, test benches (also extended for AMS), properties, assertions, functional coverage, transactors, test patterns, and random verifications; in the intermediate data between EDA tools object group, logic and circuit simulation results, parasitic wire capacitances and resistances, wire end point coordinations, LVS netlists, and delay time; in the device characteristics object group, device characteristics.

3.6. Table of “Characterization and IP Preparation”

In this sub-section, the content of the “Characterization and IP Preparation” is explained (see Figure 8).

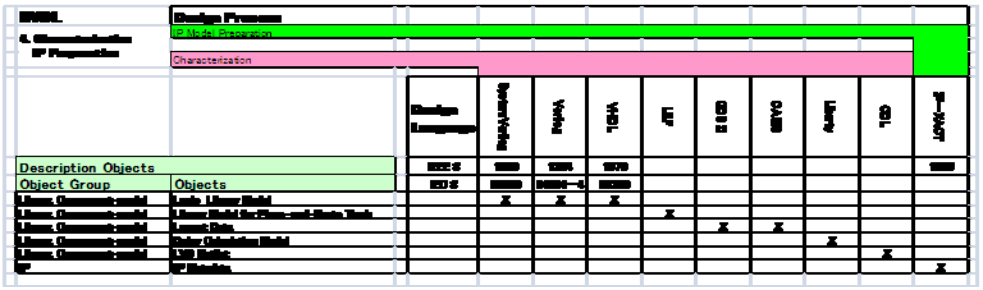


Figure 8 “Characterization and IP Preparation” table

The columns show the design processes in the “Characterization and IP Preparation” major design process. The design processes are characterization and IP model preparation. The design languages related to the design processes are listed in the columns.

The rows show design objects which are designed in the design processes in the columns. They are logic library models, library models for place-and-route tools, layout data, delay calculation models, LVS netlists, and IP metadata. Their granularity should be small enough to make clear the deference between languages.

The design objects are grouped into two object group: libraries, component-models and IP. In each object group, the design objects are as follows:
In the libraries, component-models object group, logic library models, library model for place-and-route tools, layout data, delay calculation models, and LVS netlists; in the IP object group, IP metadata.

3.7. Reading the Bird’s-eye View for Design Languages

In this sub-section, what you can read from the BVDL is explained.

Case 1) In the case where more than one language are marked in one design object in one design process, you can read that these languages are used to develop the same design objects in the same design process. In the example of Figure 9, in the “RTL Design & Verification” design process, the design languages, SystemVerilog, Verilog, and VHDL are used to write “Logical Behavior & Function” objects.

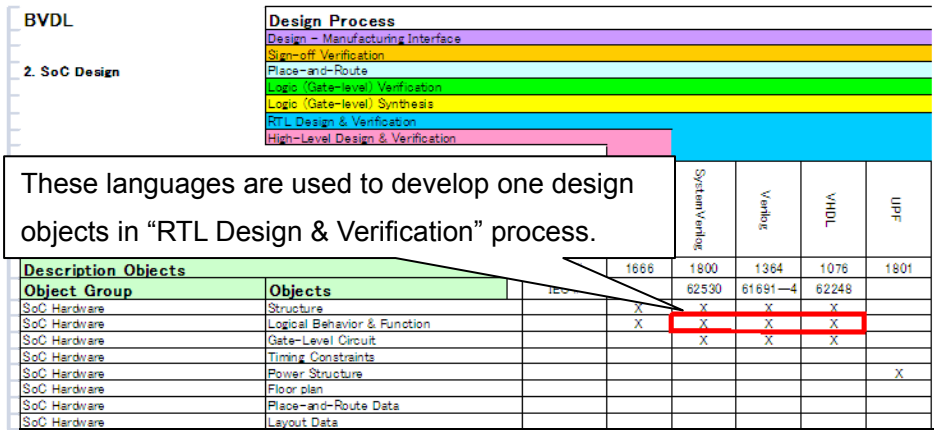


Figure 9 Multiple marks in one design object

Case 2) In the case where one language are marked in the same design objects in different design processes, you can read that one language is used to develop the same design objects in the different design processes. In the example of Figure 10, Verilog is used to express the same objects, “Structure”, “Logical Behavior & Function”, and “Gate-Level Circuit” design objects in three different design processes, “RTL Design & Verification”, “Logic (Gate-level) Synthesis”, “Logic (Gate-level) Verification”, and “Place-and-Route” design processes.

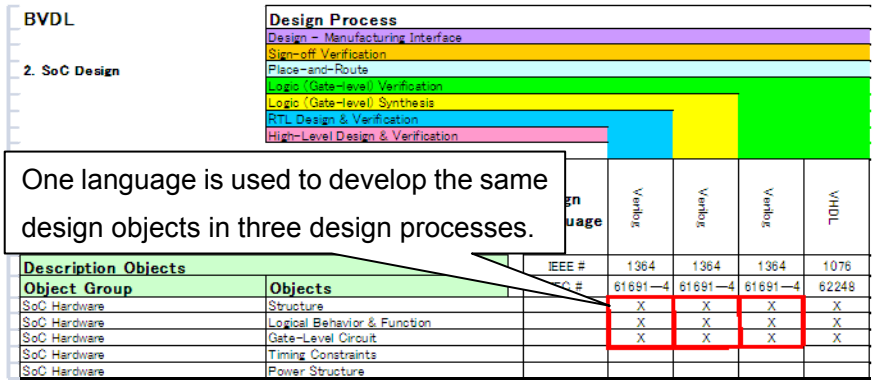


Figure 10 Multiple marks in the same design objects in the different processes

4. Use Case of the Bird’s-eye View for Design Languages

Case 1) Investigation of Consistency of Flow

When you focus on marks of one language in the different design processes, you can recognize its continuance or discontinuance in a design flow. From this recognition, you can

make clear problems emerged from that one language is used to design the same objects in the different processes, and requirements for design languages from the design flow view.

Case 2) Evolution of Language and Standardization

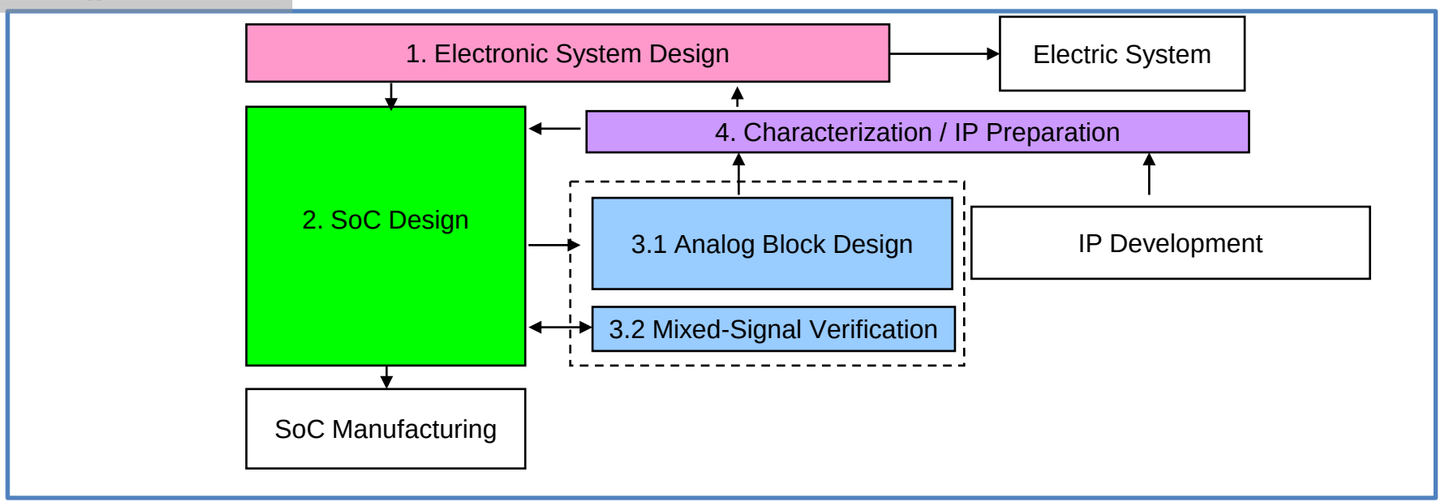
Marks in the BVDL will change through evolution of design languages and design processes. By keeping track of change the mark in the BVDL, you can recognize the changes in roles of languages. These data can help you make decisions on activities about the standardization.

Case 3) Emergence of New Technology

The design processes in the BVDL are to be changed when new technologies emerge. In this case, new design languages may be added in the BVDL. When you mark the new BVDL, you can recognize roles of new languages and consistency of flow. These recognitions can help you make decisions on activities about the standardization.

5. The Bird's-eye View of Design Languages (BVDL) Version 1.0

Design Process



BVDL

1. Electronic System Design

Design Process

PWB(Printed Wiring Board)Design , Package Design

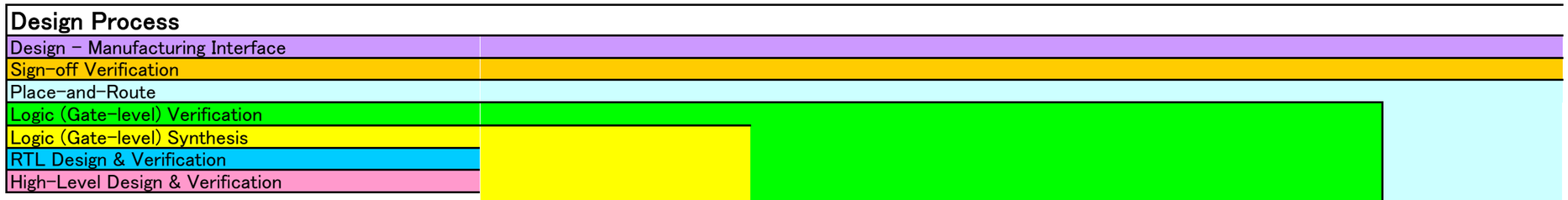
Architecture Design, Algorithm Design

Requirement Analysis, Specification Definition

Description Objects		Design Language	UML	Esterel	Rosetta	SystemC	SystemC—AMS	IBIS	CITI	TouchStone	BSDL
		IEEE #		P1778	P1699	1666		–			1149.1b
Object Group	Objects	IEC #						62014			–
Electronic System	Structure		X		X	X	X				
Electronic System	Logical Behavior & Function		X	X	X	X	X				
Electronic System	Logical Behavior & Function – Extended for AMS						X				
Electronic System	Performances & Characteristics		X		X	X	X				
Electronic System	Performances & Characteristics–Extended for AMS						X				
Electronic System	Verification Environment		X	–	X	X	X				
Electronic System	Verification Environment –Extended for AMS						X				
SoC Hardware	I/O Buffer Information							X			
SoC Testing	Boundary Scan Circuits										X
Device Characteristics	Device Characteristics(S-Parameter, etc)								X	X	

Design Process
Design – Manufacturing Interface
Sign-off Verification
Place-and-Route
Logic (Gate-level) Verification
Logic (Gate-level) Synthesis
RTL Design & Verification
High-Level Design & Verification

Description Objects		Design Language	SystemC	SystemVerilog	Verilog	VHDL	UPF	CPF	e	PSL	FSDB	SystemVerilog	Verilog	VHDL
Object Group	Objects	IEEE #	1666	1800	1364	1076	1801		1647	1850	–	1800	1364	1076
		IEC #		62530	61691—4	62248				62531	–	62530	61691—4	62248
SoC Hardware	Structure		X	X	X	X						X	X	X
SoC Hardware	Logical Behavior & Function		X	X	X	X						X	X	X
SoC Hardware	Gate-Level Circuit			X	X	X						X	X	X
SoC Hardware	Timing Constraints													
SoC Hardware	Power Structure						X	X						
SoC Hardware	Floor plan													
SoC Hardware	Place-and-Route Data													
SoC Hardware	Layout Data													
Functional Verification	Test Bench		X	X	X	X			X			X	X	X
Functional Verification	Property			X					X	X		X		
Functional Verification	Assertion			X					X	X		X		
Functional Verification	Functional Coverage			X					X	X		X		
Functional Verification	Transactor			X					X			X		
Functional Verification	Test Pattern		X	X	X	X			X			X	X	
Functional Verification	Random Verification			X					X			X	X	
Intermediate Data between EDA tools	Logic & Circuit Simulation Results										X			
Intermediate Data between EDA tools	Parasitic Wire Capacitance													
Intermediate Data between EDA tools	Parasitic Wire Resistance													
Intermediate Data between EDA tools	Wire End Point Coordination													
Intermediate Data between EDA tools	LVS Netlist													
Intermediate Data between EDA tools	Delay Time													
SoC Testing	Boundary Scan Circuits													
SoC Testing	Test Data													
SoC Testing	Core Test													
SoC Testing	On-chip scan compression structure													



Description Objects		Design Language	SDC	UPF	CPF	SystemVerilog	Verilog	VHDL	SDC	UPF	CPF	FSDB	SystemVerilog	Verilog
Object Group	Objects	IEC #				62530	61691—4	62248				–	62530	61691—4
		IEEE #		1801		1800	1364	1076		1801		–	1800	1364
SoC Hardware	Structure					X	X	X					X	X
SoC Hardware	Logical Behavior & Function					X	X	X					X	X
SoC Hardware	Gate-Level Circuit					X	X	X					X	X
SoC Hardware	Timing Constraints		X						X					
SoC Hardware	Power Structure			X	X					X	X			
SoC Hardware	Floor plan													
SoC Hardware	Place-and-Route Data													
SoC Hardware	Layout Data													
Functional Verification	Test Bench					X	X	X					X	X
Functional Verification	Property					X							X	
Functional Verification	Assertion					X							X	
Functional Verification	Functional Coverage					X							X	
Functional Verification	Transactor					X							X	
Functional Verification	Test Pattern					X							X	X
Functional Verification	Random Verification					X							X	
Intermediate Data between EDA tools	Logic & Circuit Simulation Results											X		
Intermediate Data between EDA tools	Parasitic Wire Capacitance													
Intermediate Data between EDA tools	Parasitic Wire Resistance													
Intermediate Data between EDA tools	Wire End Point Coordination													
Intermediate Data between EDA tools	LVS Netlist													
Intermediate Data between EDA tools	Delay Time													
SoC Testing	Boundary Scan Circuits													
SoC Testing	Test Data													
SoC Testing	Core Test													
SoC Testing	On-chip scan compression structure													

BVDL

2. SoC Design

Design Process

- Design – Manufacturing Interface
- Sign-off Verification
- Place-and-Route
- Logic (Gate-level) Verification
- Logic (Gate-level) Synthesis
- RTL Design & Verification
- High-Level Design & Verification

Description Objects		Design Language	VHDL	SDC	UPF	CPF	DEF	Open Access	SDF	GDS II	OASIS	SystemVerilog	Verilog	VHDL
Object Group	**Objects**	IEC #	62248				–					62530	61691—4	62248
SoC Hardware	Structure		X									X	X	X
SoC Hardware	Logical Behavior & Function		X									X	X	X
SoC Hardware	Gate-Level Circuit		X				X					X	X	X
SoC Hardware	Timing Constraints			X										
SoC Hardware	Power Structure				X	X								
SoC Hardware	Floor plan						X	X						
SoC Hardware	Place-and-Route Data						X	X						
SoC Hardware	Layout Data							X		X	X			
Functional Verification	Test Bench		X									X	X	X
Functional Verification	Property											X		
Functional Verification	Assertion											X		
Functional Verification	Functional Coverage											X		
Functional Verification	Transactor											X		
Functional Verification	Test Pattern		X									X	X	X
Functional Verification	Random Verification											X		
Intermediate Data between EDA tools	Logic & Circuit Simulation Results													
Intermediate Data between EDA tools	Parasitic Wire Capacitance							X						
Intermediate Data between EDA tools	Parasitic Wire Resistance							X						
Intermediate Data between EDA tools	Wire End Point Coordination							X						
Intermediate Data between EDA tools	LVS Netlist													
Intermediate Data between EDA tools	Delay Time							X	X					
SoC Testing	Boundary Scan Circuits													
SoC Testing	Test Data													
SoC Testing	Core Test													
SoC Testing	On-chip scan compression structure													

BVDL

2. SoC Design

Design Process

- Design – Manufacturing Interface
- Sign-off Verification
- Place-and-Route
- Logic (Gate-level) Verification
- Logic (Gate-level) Synthesis
- RTL Design & Verification
- High-Level Design & Verification

Description Objects		Design Language	SDC	UPF	CPF	DSPF	SPEF	SPF	Open Access	SDF	CDL	GDS II	OASIS	GDS II
Object Group	**Objects**	IEC #				–								
SoC Hardware	Structure													
SoC Hardware	Logical Behavior & Function													
SoC Hardware	Gate-Level Circuit													
SoC Hardware	Timing Constraints		X											
SoC Hardware	Power Structure			X	X									
SoC Hardware	Floor plan								X					
SoC Hardware	Place-and-Route Data								X					
SoC Hardware	Layout Data								X			X	X	X
Functional Verification	Test Bench													
Functional Verification	Property													
Functional Verification	Assertion													
Functional Verification	Functional Coverage													
Functional Verification	Transactor													
Functional Verification	Test Pattern													
Functional Verification	Random Verification													
Intermediate Data between EDA tools	Logic & Circuit Simulation Results													
Intermediate Data between EDA tools	Parasitic Wire Capacitance					X	X	X	X					
Intermediate Data between EDA tools	Parasitic Wire Resistance						X	X	X					
Intermediate Data between EDA tools	Wire End Point Coordination							X	X					
Intermediate Data between EDA tools	LVS Netlist										X			
Intermediate Data between EDA tools	Delay Time								X	X				
SoC Testing	Boundary Scan Circuits													
SoC Testing	Test Data													
SoC Testing	Core Test													
SoC Testing	On-chip scan compression structure													

BVDL

2. SoC Design

Design Process
Design – Manufacturing Interface
Sign-off Verification
Place-and-Route
Logic (Gate-level) Verification
Logic (Gate-level) Synthesis
RTL Design & Verification
High-Level Design & Verification

Description Objects		Design Language	OASIS	STIL	WGL
Object Group	Objects	IEEE #		1450	
		IEC #			–
SoC Hardware	Structure				
SoC Hardware	Logical Behavior & Function				
SoC Hardware	Gate-Level Circuit				
SoC Hardware	Timing Constraints				
SoC Hardware	Power Structure				
SoC Hardware	Floor plan				
SoC Hardware	Place-and-Route Data				
SoC Hardware	Layout Data		X		
Functional Verification	Test Bench				
Functional Verification	Property				
Functional Verification	Assertion				
Functional Verification	Functional Coverage				
Functional Verification	Transactor				
Functional Verification	Test Pattern				
Functional Verification	Random Verification				
Intermediate Data between EDA tools	Logic & Circuit Simulation Results				
Intermediate Data between EDA tools	Parasitic Wire Capacitance				
Intermediate Data between EDA tools	Parasitic Wire Resistance				
Intermediate Data between EDA tools	Wire End Point Coordination				
Intermediate Data between EDA tools	LVS Netlist				
Intermediate Data between EDA tools	Delay Time				
SoC Testing	Boundary Scan Circuits				
SoC Testing	Test Data			X	X
SoC Testing	Core Test			X	X
SoC Testing	On-chip scan compression structure			X	X

BVDL

Design Process

Mixed-Signal Verification

3.2 Mixed-Signal Verification

Post-Layout Circuit Verification

3.1 Analog Block Design

Layout Design

Transistor-Level Circuit Design

Analog Functional Design, Architecture Design

Description Objects		Design Language	Verilog—A	Verilog—AMS	VHDL—AMS	SPICE	FSDB	TouchStone	CITI	Open Access	GDS II	OASIS
		IEEE #	–		076.1-199	–	–					
Object Group	**Objects**	IEC #	–			–	–					
SoC Hardware	Structure		X	X	X							
SoC Hardware	Logical Behavior & Function			X	X							
SoC Hardware	Logical Behavior & Function – Extended for AMS			X	X							
SoC Hardware	Analog Behavior & Function		X	X	X							
SoC Hardware	Gate-Level Circuit		X	X	X							
SoC Hardware	Transistor -Level Circuit		X	X	X	X						
SoC Hardware	Timing Constraints											
SoC Hardware	Power Structure											
SoC Hardware	Floor plan											
SoC Hardware	Place-and-Route Data									X		
SoC Hardware	Layout Data										X	X
Functional Verification	Test Bench		X	X	X							
Functional Verification	Test Bench – Extended for AMS		X	X	X							
Functional Verification	Property											
Functional Verification	Assertion											
Functional Verification	Functional Coverage											
Functional Verification	Transactor											
Functional Verification	Test Pattern			X	X							
Functional Verification	Random Verification											
Intermediate Data between EDA tools	Logic & Circuit Simulation Results						X					
Intermediate Data between EDA tools	Parasitic Wire Capacitance									X		
Intermediate Data between EDA tools	Parasitic Wire Resistance									X		
Intermediate Data between EDA tools	Wire End Point Coordination									X		
Intermediate Data between EDA tools	LVS Netlist					X						
Intermediate Data between EDA tools	Delay Time									X		
Device Characteristics	Device Characteristics(S-Parameter, etc)							X	X			

BVDL

Design Process

Mixed-Signal Verification

3.2 Mixed-Signal Verification

Post-Layout Circuit Verification

3.1 Analog Block Design

Layout Design

Transistor-Level Circuit Design

Analog Functional Design, Architecture Design

Description Objects		Design Language	SPICE	FSDB	DSPF	SPF	TouchStone	CITI		Verilog—AMS	VHDL—AMS
		IEEE #	–	–	–						
Object Group	**Objects**	IEC #	–	–	–						
SoC Hardware	Structure									X	
SoC Hardware	Logical Behavior & Function									X	X
SoC Hardware	Logical Behavior & Function – Extended for AMS									X	X
SoC Hardware	Analog Behavior & Function									X	X
SoC Hardware	Gate-Level Circuit										
SoC Hardware	Transistor -Level Circuit		X								
SoC Hardware	Timing Constraints										
SoC Hardware	Power Structure										
SoC Hardware	Floor plan										
SoC Hardware	Place-and-Route Data										
SoC Hardware	Layout Data										
Functional Verification	Test Bench									X	
Functional Verification	Test Bench – Extended for AMS										
Functional Verification	Property										
Functional Verification	Assertion										
Functional Verification	Functional Coverage										
Functional Verification	Transactor										
Functional Verification	Test Pattern										
Functional Verification	Random Verification										
Intermediate Data between EDA tools	Logic & Circuit Simulation Results			X							
Intermediate Data between EDA tools	Parasitic Wire Capacitance				X	X					
Intermediate Data between EDA tools	Parasitic Wire Resistance					X					
Intermediate Data between EDA tools	Wire End Point Coordination					X					
Intermediate Data between EDA tools	LVS Netlist		X								
Intermediate Data between EDA tools	Delay Time										
Device Characteristics	Device Characteristics(S-Parameter, etc)						X	X			

BVDL

4. Characterization
IP Preparation

Design Process											
IP Model Preparation											
Characterization											
		Design Language	SystemVerilog	Verilog	VHDL	LEF	GDS II	OASIS	Liberty	CDL	IP—XACT
Description Objects		IEEE #	1800	1364	1076						1685
Object Group	**Objects**	IEC #	62530	61691—4	62248						
Library, Component-model	Logic Library Model		X	X	X						
Library, Component-model	Library Model for Place-and-Route Tools					X					
Library, Component-model	Layout Data						X	X			
Library, Component-model	Delay Calculation Model								X		
Library, Component-model	LVS Netlist									X	
IP	IP Metadata										X

4.3 SystemCワーキンググループ 2010年度活動報告

JEITA EDA技術専門委員会

標準化小委員会

SystemCワーキンググループ

Copyright 2011 JEITA, All rights reserved

JEITA

SystemCワーキンググループメンバー

主査	今井 浩史（東芝）
副主査	中西 早苗（ルネサスエレクトロニクス）
委員	西園寺 修（日本シノプシス）
	竹村 和祥（パナソニック）
	立岡 真人（富士通セミコンダクター）
	旦木 秀和（ソニー）
	牧野 潔　（メンター）
	村上 和隆（リコー）
特別委員	吉永 和弘（半導体理工学研究センター）
客員	今井 正治（大阪大学）

（計10名）

Copyright 2011 JEITA, All rights reserved

JEITA

目次

4.3.1 SystemCワーキンググループ概要
- SystemCとは
- SystemCの標準化の階層
- SystemCワーキンググループについて
- SystemCワーキンググループの活動
- これまでの活動内容と主な成果
- 2010年度報告資料について

4.3.2 TLM 2.0 LRM 日本語抄訳
4.3.3 SystemC Draft LRM 差分解説
4.3.4 IEEE P1666 WGへのフィードバック
4.3.5 SystemC Japan 2010 アンケート報告

Copyright 2011 JEITA, All rights reserved

JEITA

4.3.1 SystemCワーキンググループ概要

Copyright 2011 JEITA, All rights reserved

JEITA

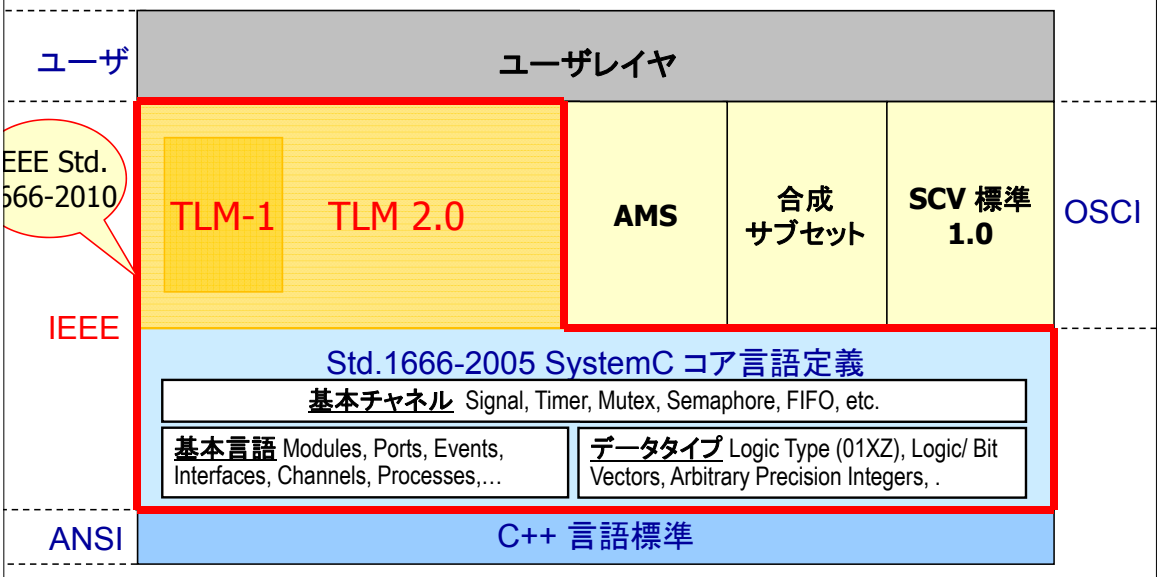
SystemC とは

- C++言語をベースとした、システムレベル設計言語の代表的な言語である
  - Open SystemC Initiative（OSCI）という標準化組織により、言語仕様（LRM）とリファレンスシミュレータが策定され、無償提供されている
    http://www.systemc.org/
  - 2005年12月に、SystemC LRMがIEEE Std. 1666として標準化された
  - 2011年には、IEEE Std. 1666-2010として更新される。この更新にはTLM-2.0が含まれる
  - 現在、合成サブセットやSystemCモデル内部へのアクセス手順の標準化案が検討されている
- C++の文法を保持したまま、クラスライブラリの形で以下のような言語拡張がなされている
  - 並列動作を可能とするシミュレーションエンジン（クロック、イベント、等）
  - 抽象化された通信手段（Channels, Interfaces）
  - ハード実装に必要なデータタイプ（固定小数点、固定長ビット、等）
  - 0, 1, Z, X 等の信号値等

Copyright 2011 JEITA, All rights reserved

JEITA

SystemCの標準化の階層



OSCI資料（2007）に加筆

Copyright 2011 JEITA, All rights reserved

JEITA

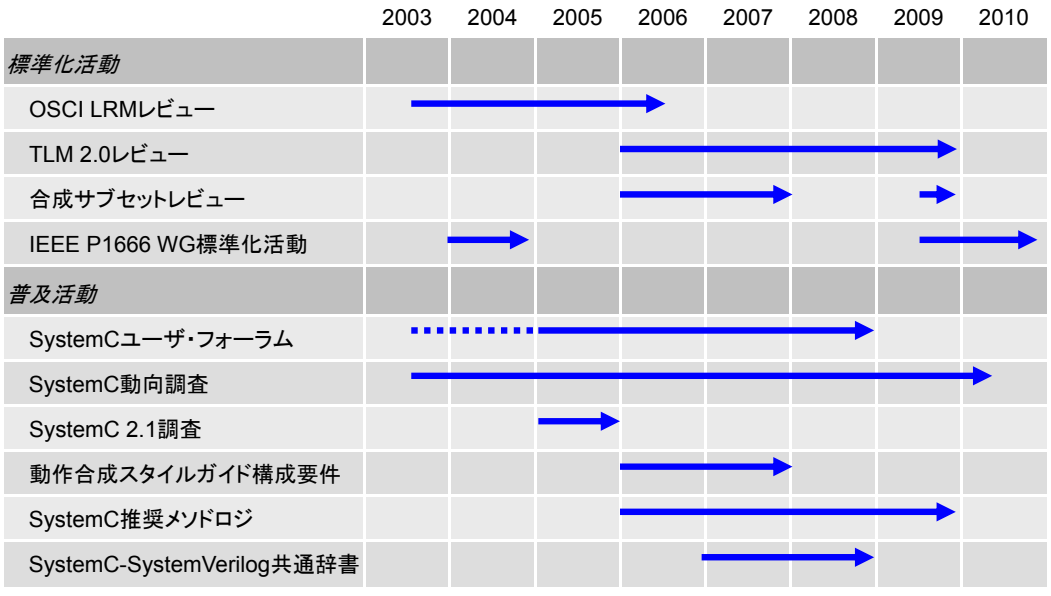
SystemCワーキンググループについて

- 設立と名称変更
  - 2003年10月に、JEITA EDA技術専門委員会 標準化小委員会内にSystemCタスクグループとして設置
  - 2007年4月度よりSystemCワーキンググループに名称変更して活動継続
- 目的
  - 日本国内における唯一のSystemCの標準化関連組織として、OSCIやIEEE P1666ワーキンググループと連携しつつ、日本国内の事情・要求事項を取り込むべくSystemCの国際標準化を進めていく。
  - SystemCに関連した調査結果を積極的に情報発信を行うことで、国内普及を図る。これらにより日本の産業界の国際競争力を高めることを目指す。

Copyright 2011 JEITA, All rights reserved

SystemCワーキンググループの活動

- 活動の歩み



Copyright 2011 JEITA, All rights reserved

これまでの活動内容と主な成果

- SystemC言語標準化活動
  - IEEE P1666ワーキンググループに投票権のあるメンバーとして言語仕様の策定に参加し、IEEE Std.1666-2005の標準化に直接貢献（2005年度）しました
  - 2006～2008年度にOSCIより公開されたTLM 2.0（Transaction Level Modeling 2.0）ドラフト版および正式版についてレビューを実施し、計14件をフィードバック（2006～2008年度）
  - 2009年7月に公開されたTLM2.0 LRM（Language Reference Manual）についてレビューを実施し、IEEE P1666 SystemC WGへフィードバック。抄訳を本アニュアルレポートに掲載
  - 2009年8月に公開された合成サブセットDraft1.3のレビューを実施し、OSCIへフィードバック
  - 2009年12月からIEEE P1666ワーキンググループに投票権のあるメンバーとして言語仕様の更新とTLM-2.0の標準化に参加し、2011年1月にP1666 WG内での投票を行いました
- SystemC動向・技術調査
  - 2000年～2010年度における国内外でのSystemCの利用状況について調査を実施
  - SystemC 2.1について調査を行い、その特長を日本語で紹介（2005年度）
  - 国内外におけるTLMの利用状況調査を実施（2006-2007年度）
  - OSCI公開の合成サブセットのドキュメントのレビュー及び抄訳の作成を実施（2006年度）
  - 「動作合成スタイルガイド構成要件」を公開（2007年度）
  - 「SystemC推奨設計メソドロジ 合成編」を公開（2007年度）
  - 「SystemC推奨設計メソドロジ 2008年度版」を公開

Copyright 2011 JEITA, All rights reserved

これまでの活動内容と主な成果

- SystemC普及活動
  - SystemCユーザフォーラムを開催（2005～2008年度）し、OSCIによるSystemCの最新情報の発表、本ワーキンググループの活動成果、SystemCの活用事例等を紹介
  - SystemC Japan 2010（2010年7月）でSystemCワーキンググループの活動を紹介
  - JEITA EDA-TCのWEBページで、これまでの活動成果を一挙掲載
    - http://www.jeita-edatc.com/index-jp.html

Copyright 2011 JEITA, All rights reserved

2010年度報告資料について

4.3.2 TLM 2.0 LRM 日本語抄訳

- OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL（JA32）の日本語抄訳（2009年度更新版）

4.3.3 SystemC Draft LRM 差分解説

- 2011年に改訂されるIEEE Std. 1666-2011と現標準のIEEE Std. 1666-2005の違いについて解説

4.3.4 IEEE P1666 WGへのフィードバック

- IEEE P1666ワーキンググループへ提出したフィードバック

4.3.5 SystemC Japan 2010 アンケート報告

- SystemC Japan 2010で実施したアンケート集計結果

Copyright 2011 JEITA, All rights reserved

4.3.2 TLM-2.0 LRM の日本語抄訳

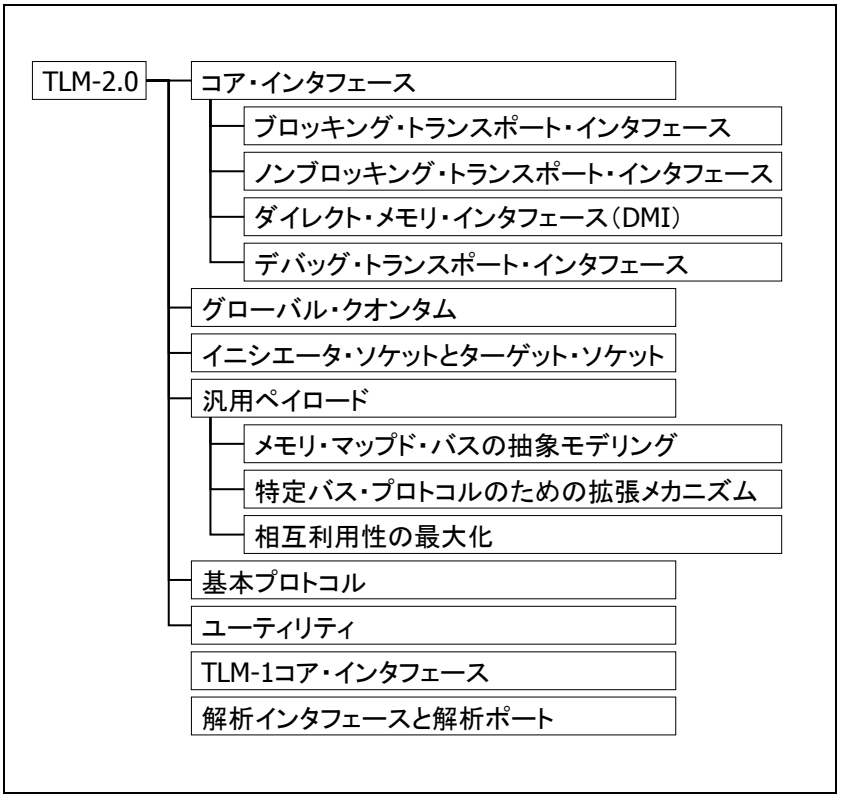
本節は、OSCI の TLM2 USER MANUAL Software version: TLM-2.0.1、Document version: JA32 を日本語抄訳したものである。

IEEE Std 1666™-2010 では一部変更されている。変更内容については、04-3-03 を参照。

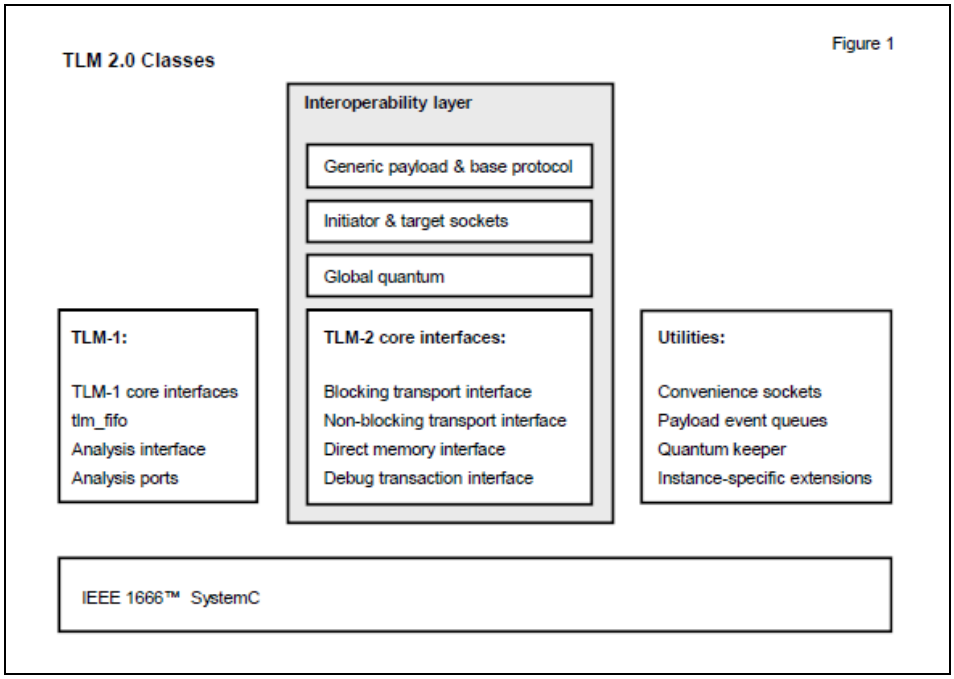
1. あらまし

本ドキュメントは OSCI トランザクション・レベル・モデリング標準バージョン 2.0 のリファレンス・マニュアルである。OSCI トランザクション・レベル・モデリング標準バージョン 2.0 は、バージョン 2.0 ドラフト 1 とドラフト 2 に取って代わるもので、これらと一般に互換性はない。この標準バージョンは、TLM-1 のコア・インタフェースを含む。

- TLM-2.0 の内容



- TLM-2 クラス



TLM-2.0 のクラスは SystemC クラス・ライブラリのレイヤの上にある。相互利用性を最大にするため（特にメモリ・マップド・バスのモデリングで）、TLM-2.0 コア・インタフェース、ソケット、汎用ペイロード、基本プロトコルを一緒に使用することを推奨する。これらのクラスは、まとめてインタオペラビリティ・レイヤと呼ばれる。汎用ペイロードが適さない場合は、代わりのトランザクション型とコア・インタフェース、イニシエータ・ソケット、ターゲット・ソケット、もしくは、その代わりのトランザクション型とコア・インタフェースを使うことができる。技術的には、イニシエータ・ソケットとターゲット・ソケットを使わずに、汎用ペイロードをコア・インタフェースと使うことができるが、このアプローチは推奨しない。

バスモデル間の相互利用性達成のためにユーティリティクラス（Figuer1 参照）を使う必要はまったくない。しかしながら、スタイルの一貫性のため、可能ならばこれらのユーティリティ・クラスを使うことが推奨されており、ユーティリティ・クラスは TLM-2.0 標準の一部としてドキュメント化され、保守されている。

- 汎用ペイロード

  メモリ・マップド・バスのモデリングを主なターゲット（非バス・プロトコルにも使用可能）とする。特定のプロトコルをモデリングするため、汎用ペイロードのアトリビュート、フェーズは拡張可能。ただし、このような拡張は、標準の拡張なし汎用ペイロードからどれだけ逸脱しているかによるが、相互利用性を低減する。

- コーディング・スタイル

  高速なルーズリー・タイムド・モデルは概して、ブロッキング・トランスポート・インタフェース、ダイレクト・メモリ・インタフェース、テンポラル・デカップリングを使うと期待される。より精度の高いアプロキシメイト・タイムド・モデルは概して、ノンブロッキング・トランスポート・インタフェースとペイロード・イベント・キューを使うと期待される。これらはコーディング・スタイルの提案であり、TLM-2.0 標準の規範を定める部分ではない。

1.1 スコープ

本ドキュメントはTLM-2.0標準のリファレンス・マニュアルの決定版である。本ドキュメントの主な焦点は、ユーティリティを含むTLM-2.0クラスのキー・コンセプトとセマンティックである。よって本ドキュメントでは、TLM2.0 リリースのソースコードの全体や例題やユニット・テストは記載していない。TLM-1 コア・インタフェースを載せるが、セマンティックは定義しない。

1.2 ソースコードとドキュメント

- TLM-2.0 リリースのディレクトリ構造

include/tlm/	:ソースコード、readme ファイル、リリース・ノート
include/tlm/tlm_h	:TLM-2.0 インタオペラビリティ・レイヤ
include/tlm/tlm_h/tlm_2_interfaces	:TLM-2 コア・インタフェース
include/tlm/tlm_h/tlm_generic_payload	:TLM-2 汎用ペイロード
include/tlm/tlm_h/tlm_sockets	:TLM-2 ソケット
include/tlm/tlm_h/tlm_quantum	:TLM-2 グローバル・クォンタム
include/tlm/tlm_1	:TLM-1 と解析
include/tlm/tlm_1/tlm_req_rsp	:TLM-1 標準
include/tlm/tlm_1/tlm_req_rsp/tlm_1_interfaces	:TLM-1 コア・インタフェース
include/tlm/tlm_1/tlm_req_rsp/tlm_channels	:TLM-1 fifo と req-rsp チャネル
include/tlm/tlm_1/tlm_req_rsp/tlm_ports	:TLM-1 イベント・ファインダーを持ったノンブロッキング・ポート
include/tlm/tlm_1/tlm_req_rsp/tlm_adapters	:TLM-1 スレーブ-トランスポートとトランスポート-マスタ・アダプタ
include/tlm/tlm_1/tlm_analysis	:解析インタフェースとポート
include/tlm/tlm_utils	:TLM-2 標準ユーティリティ・クラス。相互利用性の本質的ではない。
docs	:ドキュメント（ユーザ・マニュアル、ホワイト・ペーパー、Doxygen）
examples	:アプリケーション向けの例とそのドキュメント
unit_test	:リグレッション・テスト

2. リファレンス

- 本標準は以下の文書とともに使用すること。
    - ISO/IEC 14882:2003, Programming Languages—C++
    - IEEE Std 1666-2005, SystemC Language Reference Manual
    - Requirements Specification for TLM 2.0, Version 1.1, September 16, 2007

2.1 参考文献

- Transaction-Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems, edited by Frank Ghenassia, published by Springer 2005, ISBN 10 0 387-26232-6(HB), ISBN 13 978-0-387-26232-1(HB)
- Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms, by Tim Kogel, Rainer Leupers, and Heinrich Meyr, published by Springer 2006, ISBN 10 1-4020-4825-4(HB), ISBN 13978-1-4020-4825-4(HB)
- ESL Design and Verification, by Brian Bailey, Grant Martin and Andrew Piziali, published by Morgan Kaufmann/Elsevier 2007, ISBN 10 0 12 373551-3, ISBN 13 978 0 12 373551-5

3. イントロダクション

3.1 背景

TLM-1 標準では value 型もしくは const 参照型によってトランザクションを転送するコア・インタフェースを定義していた。これらのインタフェースは幾つかのアプリケーションでは成功しているが、メモリ・マップド・バス、オンチップ・コミュニケーション・ネットワークのモデリングに関して 3 つの欠点がある。

a) TLM-1 は標準のトランザクション・クラスを持たないため、各アプリケーションはそれ自身の標準でないクラスを作る必要がある。その結果、異なるソースのモデル間の相互利用性が低くなる。TLM-2.0 は、この欠点を汎用ペイロードで解決している。

b) TLM-1 はタイミング・アノテーションをサポートしない。その結果、モデル間でタイミング情報を交換する方法の標準がない。TLM-1 モデルは概して wait 文を呼ぶことで遅延を実装している。これがシミュレーション速度を遅くする。TLM-2.0 は、この欠点をブロッキング/ノンブロッキング・トランスポート・インタフェースにタイミング・アノテーションを追加することで解決している。

c) TLM-1 インタフェースはすべてのトランザクション・オブジェクトとデータが value 型もしくは const 参照型で渡されることを要求する。これがシミュレーション速度を遅くする。幾つかのアプリケーションでは、トランザクション・オブジェクトにポインタを埋め込むことでこの制限を回避している。しかし、この方法は標準ではなく、相互利用可能ではない。TLM-2.0 でサポートされた新しいトランスポート・インタフェースでは、この欠点をトランザクション・オブジェクトのライフタイムを複数のトランスポート・コールに渡るように延長することにより解決している。

3.2 トランザクション・レベル・モデリング、ユースケース、抽象度

ESL コミュニティでは、トランザクション・レベル・モデリングの抽象度の分類法について最も適したものは何かについて長い間議論されてきた。モデルは様々な基準、時間の粒度、モデル評価の頻度、機能の抽象度、コミュニケーションの抽象度、ユースケースで分類されてきた。トランザクション・レベル・モデリングには様々なユースケースがあることが分かっている（「TLM-2.0 Requirements Specification」を参照）。そこで、TLM-2.0 では、抽象度を定義する代わりに、インタフェース（API）とコーディング・スタイルを区別するというアプローチをとる。TLM-2.0 標準は、トランザクション・レベル・モデルを実装するための下位レベルのプログラミングの仕組みとしてインタフェースのセットを定義し、様々なユースケースに適する（固定するわけではない）幾つかのコーディング・スタイルを記述する。

標準の TLM-2.0 インタフェースの定義はコーディング・スタイルの記述とは別物である。TLM-2.0 インタフェースはこの標準の規範を構成するもので、相互利用性を保証する。各コーディング・スタイルは機能、タイミング、コミュニケーションの様々な抽象化をサポートする。原理的には、ユーザは自分自身のコーディング・スタイルを作ることができる。

アンタイムド機能モデルは一つのソフトウェア・スレッドからなり、C 関数もしくは一つの SystemC プロセスで書かれる。これはときにアルゴリズム・モデルとも呼ばれる。このモデルは

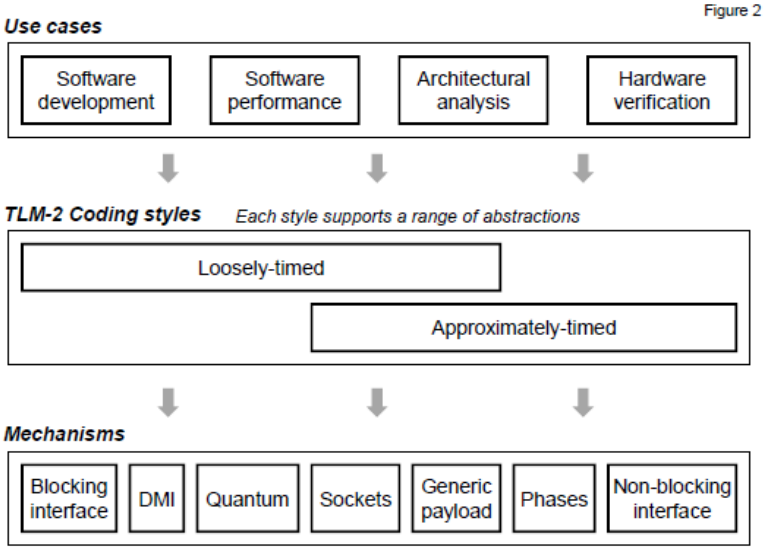
本質的にトランザクション・レベルではない。なぜならば、トランザクションとはコミュニケーションの抽象度の一つと定義されており、シングル・スレッド・モデルにはプロセス間通信がないからである。トランザクション・レベル・モデルは並列実行とコミュニケーションをシミュレートするために複数の SystemC プロセスを必要とする。

複数プロセス（複数ソフトウェア・スレッド）を持つトランザクション・レベル・モデルでは、スレッドが他のスレッドに制御を譲る仕組が必要である。なぜならば、SystemC は協調的マルチタスクモデルを採用しており、そこでは実行中のプロセスは他のプロセスに実行を譲ることは出来ない。SystemC プロセスは、スレッド・プロセスでは wait 文を呼ぶことで、メソッド・プロセスではカーネルに戻ることで制御を譲る。wait 文の呼び出しは通常、プログラミング・インタフェース（API）の中に隠されており、そのプログラミング・インタフェースを用いて、特定の抽象度、または特定のプロトコル（タイミング情報に依存するかしないかに関わらず）をモデル化する。

同期には 2 種類ある。強い同期は一連のコミュニケーション・イベントがあらかじめ正確に決まっているもの。弱い同期は一連のコミュニケーション・イベントが個々のプロセスのタイミングによって、部分的に決まっているもの。強い同期は FIFO もしくはセマフォと使って SystemC で容易に実装できる。これは完全にアンタイムド・モデルリング・スタイルにでき、原理的にシミュレーション時間を進めることなくシミュレーションできる。この意味で、アンタイムド・モデリングは TLM-2.0 の範囲外である。一方、複数の組込みソフトウェア・スレッドの並列動作を許す高速なバーチャル・プラットフォーム・モデルでは、強いあるいは弱い同期を使うだろう。本標準では、このようなモデルに適したコーディング・スタイルはルーズリー・タイムドと呼ばれる。

より詳細なトランザクション・レベル・モデルでは、それぞれのトランザクションに対して複数のプロトコルに特有なタイミング・ポイント（プロトコルの各フェーズの開始と終了を表すタイミング・ポイント）によって構成される必要があるかもしれない。適切な数のタイミング・ポイントを選ぶことによって、毎クロック・サイクルでコンポーネント・モデルを動作させることなく、高いタイミング精度のコミュニケーションをモデリングすることができる。本標準では、このようなコーディング・スタイルをアプロキシメイトリー・タイムドと呼ぶ。

Figure 2



3.3　コーディング・スタイル

コーディング・スタイルとは、プログラム言語の表現のセット（いっしょに使うとうまく動く）であり、特定の抽象度、ソフトウェア・プログラミング・インタフェースということではない。簡潔さのため、本ドキュメントでは2つのコーディング・スタイル、ルーズリー・タイムドとアプロキシメイトリー･タイムド、に限定する。コーディング・スタイルという性質上、これらは正確に定義されるものではない。TLM-2.0 コア・インタフェースを規定する規則は、これらのコーディング・スタイルとは独立に定義される。原理的には、TLM-1 および TLM-2.0 の仕組に基づいた別のコーディング・スタイルを定義することも可能である。

3.3.1　アンタイムド・コーディング・スタイル

TLM-2.0 はアンタイムド・コーディング・スタイルを明確には規定しない。なぜならば、現代のバス・ベース・システムは一つ以上の組込みプロセッサ上で動作するソフトウェアをモデル化するためには何らかの時間概念が必要だからである。しかしながら、アンタイムド・モデリングは TLM-1 コア・インタフェースでサポートされる。（アンタイムドという言葉は、しばしば、限られた量の時間情報を持つモデルを指して使われてきた。TLM-2.0 では、このようなモデルはルーズリー・タイムドと呼ばれる。）

3.3.2　ルーズリー・タイムド・コーディング・スタイルとテンポラル・デカップリング

ルーズリー・タイムド・コーディング・スタイルはブロッキング・トランスポート・インタフェースを使用する。このインタフェースは各トランザクションに対して2つのタイミング・ポイント、(ブロッキング・トランスポート関数の呼出しと戻りに対応）のみを許す。基本プロトコルでは、最初のタイミング・ポイントはリクエストの開始を示し、2 番目のタイミング・ポイントはレスポンスの開始を示す。これらの2つのタイミング・ポイントはシミュレーション時間で同時でも異なる時刻でもよい。

ルーズリー・タイムド・コーディング・スタイルは、1つ以上の OS を持つような MPSoC のバーチャル・プラットフォーム・モデルを使ったソフトウェア開発に適する。ルーズリー・タイムド・コーディング・スタイルは、ターゲット・マシン上の OS のブートと任意のコードを実行させるに十分な、タイマー、割り込みのモデリングをサポートする。

ルーズリー・タイムド・コーディング・スタイルは、テンポラル・デカップリングもサポートする。テンポラル・デカップリングでは、SystemC プロセスはシステムの残りの部分と同期が必要なポイントまで、実際にシミュレーション時刻を進めることなく、ローカルな“タイム・ワープ”まで実行を進めることができる。テンポラル・デカップリングは、データとコードの局所性を高め、シミュレータのスケジューリングのオーバーヘッドを低減するので、あるシステムでは、とても高速なシミュレーションを可能とする。各プロセスは、あるタイム・スライス間もしくはクォンタム間動作することが許される。または、明示的な同期ポイントに到達して制御を譲る。

SystemCそれ自体を考えると､SystemCスケジューラはシミュレーション時刻を厳格に管理する。スケジューラは次のイベントの時刻までシミュレーション時刻を進め、その時刻に実行すべきプロセスまたはそのイベントにセンシティブなプロセスを動作させる。SystemC プロセスは現シミュレーション時刻（sc_time_stamp 関数を読んで得られる時刻）で動作し、SystemC プロセスが変数を読み書きするときは、現シミュレーション時刻の変数の値にアクセスする。プロセスが実行を終える時、シミュレーション・カーネルに制御を返さなければならない。シミュレーション・モデルが精度の高いレベルで書かれていると、イベント・スケジューリングとコンテクスト・ス

イッチのオーバーヘッドがシミュレーション・スピードについての支配的な要因となる。シミュレーション速度を上げる 1 つの方法は、プロセスがシミュレーション時刻に先行して実行できることを許すこと、すなわち、テンポラル・デカップリングである。

SystemC においてテンポラル・デカップリングを実装すると、プロセスは、他のプロセスと相互作用する必要のあるときまで、例えば、他のプロセスが持つ変数を読んだり更新したりするときまで、シミュレーション時刻に先行して動作できるようになる。そのポイントで、プロセスは現在の値にアクセスして実行を続けるか（タイミング精度を失う可能性あり）、シミュレーション・カーネルに制御を戻してローカルな“タイム・ワープ”に追いつくまでその他のプロセスを実行する。各プロセスはモデルの機能を壊すことなしにシミュレーション時刻に先行して動作できるかどうかを決める責任を負う。プロセスが外部依存に出会ったときは 2 つの選択肢がある：一つは、強制同期、つまり、シミュレーション時刻が追いつくまで他のプロセスが通常動作するように実行権を譲る。もう一つは、現在の値をサンプリングまたは更新して実行を続けるかである。この同期についての選択肢は標準 SystemC シミュレーション・セマンティクスと機能の一致を保証する。現在の値での実行継続は、モデル化したシステムのコミュニケーションとタイミングにどのような仮定を置いているかを当てにしている。つまり、値のサンプリング、更新が早すぎても遅すぎてもダメージが無いと仮定している場合に実行継続される。ソフトウェア・スタックがハードウェアのタイミングの下位レベルに依存しない、バーチャル・プラットフォーム・シミュレーションでは通常この仮定は成り立つ。

テンポラル・デカップリングはルーズリー・タイムド・コーディング・スタイルの特徴である。

もしプロセスが制限無くシミュレーション時刻に先行して実行することが許されたとすると、SystemC スケジューラは動作することができず、他のプロセスは決して実行されない。これは、グローバル・クォンタムを参照することで避けられる。グローバル・クォンタムはプロセスがシミュレーション時刻に先行して実行できる上限を設定する。クォンタムはアプリケーションが設定し、クォンタム値はシミュレーションのスピードと精度のトレードオフを表す。小さすぎる値ではプロセスが非常に頻繁に制御を譲り同期し、シミュレーション速度が落ちる。大きすぎる値ではシステムにタイミングに関する矛盾が入り、システムが機能しなくなる可能性がある。

例えば、プロセッサ、メモリ、タイマー、幾つかの遅い外部周辺回路から成るシステムのシミュレーションを考える。プロセッサ上で動作しているソフトウェアはメモリから命令をフェッチし、実行するのにほとんどの時間を使っており、システムの残りの部分とはタイマーからの割込み、例えば、1 ms 毎、でのみ相互作用する。ISS モデルはクォンタムを使うと SystemC シミュレーション時間を 1 ms まで先行してメモリ・モデルを直接アクセスして動作でき、周辺モデルとはタイマー割込みの割合で同期する。ポイントは、ISS は従来のハードウェア・ソフトウェア協調シミュレーションのように、システムのハードウェア部分のシミュレーション・クロックに縛られる必要があないということである。モデルの詳細によるが、テンポラル・デカップリングだけで、おおよそ 10 倍、DMI といっしょに使うとおおよそ 100 倍のシミュレーション速度の改善が可能である。幾つかのプロセスはテンポラル・デカップリングを使い、他は使わない、プロセスは異なる値のタイム・クォンタムを使っている場合でも、これは可能である。しかしながら、テンポラル・デカップリングを使わないプロセスは、シミュレーション速度のボトルネックとなりやすい。

TLM-2.0 では、テンポラル・デカップリングは tlm_global_quantum クラスとブロッキングとノンブロッキング・トランスポート・インタフェースのタイミング・アノテーションでサポートされる。ユーティリティ・クラス tlm_quantumkeeper はグローバル・クォンタムにアクセスする便利な

方法を提供する。

3.3.3 ルーズリー・タイムド・モデルでの同期

アンタイムド・モデルは、実行中にイニシエータ間で制御を渡すために明示的な同期ポイント（wait 文呼出し、または、ブロッキング・メソッド呼出し）の存在に依存する。ルーズリー・タイムド・モデルもまた、実行の決定を保証するため明示的な同期の恩恵を受けることができる。しかし、ルーズリー・タイムド・モデルでは明示的な同期（wait 文呼出し）がなくても時間の経過が可能である。なぜならば、各イニシエータは制御を譲るまでタイム・クォンタムの終わりに到達するまで先行して動作するからである。ルーズリー・タイムド・モデルは要求ベース同期、すなわち、タイム・クォンタムの終了に到達する前にスケジューラに制御を譲る、によってタイミング精度を向上させることができる。

タイム･クオンタム･メカニズムは正確なシステム同期を保証することを意図したわけではなく、スケジューラベースのシミュレーション環境で、複数のシステム・イニシエータが時間に経過を許すシミュレーション・メカニズムである。タイム・クオンタム・メカニズムは、システム・レベルでの明示的同期機構の代替ではない。

3.3.4 アプロキシメイトリー・タイムド・コーディング・スタイル

アプロキシメイトリー・タイムド・コーディング・スタイルは、ノンブロッキング・トランスポート・インタフェースでサポートされており、アーキテクチャ探索と性能解析のユースケースに適する。ノンブロッキング・トランスポート・インタフェースは、タイミング・アノテーションと、トランザクションのライフタイム間の複数のフェーズとタイミング・ポイントを提供する。アプロキシメイトリー・タイムド・モデリングでは、トランザクションはフェーズ間の遷移のしるしをつける明示的タイミング・ポイントを持った複数フェーズに分けられる。基本プロトコルには、リクエストの開始と終了、レスポンスの開始と終了のしるしであるタイミング・ポイントがちょうど 4 つある。特定のプロトコルではさらにタイミング・ポイントを追加する必要があるかもしれないが、これは汎用ペイロードとの直接的な互換性を失う結果となるかもしれない。トランザクションの開始と終了を示す 2 フェーズでノンブロッキング・トランスポート・インタフェースを使うこともできるが、ルーズリー・タイムド・モデリングでは、ブロッキング・トランスポート・インタフェースが一般的に望ましい。

アプロキシメイトリー・タイムド・コーディング・スタイルは、タイミング精度が必要なため、テンポラル・デカップリングを使うことは一般的にできない。代わりに、各プロセスは SystemC スケジューラの決められた手順に従って動作する。プロセス間のやりとりでは決められた遅延がアノテートされる。アプロキシメイトリー・タイムド・モデルを作るには、ライトとリード・コマンドのデータ転送時間とターゲットのレイテンシを表す遅延をアノテートするだけで一般的には十分である。基本プロトコルでは、データ転送時間は実質的に 2 つの連続したリクエストの最小間隔もしくは 2 つの連続したレスポンスの受信時の遅延と同じである。アノテートされた遅延は SystemC スケジューラの呼出し、すなわち、wait(delay)もしくは notify(delay)、で実装される。

3.3.5 ルーズリー・タイムドとアプロキシメイトリー・タイムド・コーディング・スタイルの特徴

コーディング・スタイルはタイミング・ポイントとテンポラル・デカップリングで特徴づけられる。

- ルーズリー・タイムド

  各トランザクションは2つのタイミング・ポイントだけ（トランザクションの開始と終了を表す）を持つ。シミュレーション実行中にシミュレーション時間は経過していくが、プロセスがシミュレーション時間から一時的に分離されていることがある。プロセスはシミュレーション時間からどれだけ先行したかを記憶し、明示的同期ポイントに到達もしくはタイム・クォンタムを消費して制御を譲る。

- アプロキシメイトリー・タイムド

  各トランザクションは複数のタイミング・ポイントを持つ。プロセスは一般にSystemCシミュレーション時間に決められて動作する必要がある。プロセス相互作用にアノテートされた遅延はタイムアウト（wait）もしくはタイムド・イベント発行（notify）を使って実装される。

- アンタイムド

  シミュレーション時間の概念は必要ない。プロセスは予め決まっている明示的同期ポイントで制御を譲る。

3.3.6 ルーズリー・タイムドとアプロキシメイトリー・タイムドのモデリング手法の切換

モデルはシミュレーション中にルーズリー・タイムドとアプロキシメイトリー・タイムド・コーディング・スタイル間で切り替わっても良い。目的は、ルーズリー・タイムド・レベルでリセットとブートシーケンスを実行し、シミュレーションがある興味あるステージに到達したら、より詳細な解析のためアプロキシメイトリー・タイムド・モデルリングに切り替えるというものである。

3.3.7 サイクル精度モデリング

現在、サイクル精度モデリングはTLM-2.0のスコープ外である。SystemCとTLM-1そのままでサイクル精度モデルを作ることは可能であるが、サイクル精度コーディング・スタイルの標準化は未解決課題であり、おそらく、将来のOSCI標準で解決されるであろう。

原理的にではあるが、適当なフェーズとルールの組を定義することでアプロキシメイトリー・タイムド・コーディング・スタイルがサイクル精度モデリングまで拡張されるかもしれない。TLM-2.0リリースはこのために十分な仕組みを含んでいるが、現状はうまくいっていない。

3.3.8 ブロッキング対ノンブロッキング・トランスポート・インタフェース

ブロッキングとノンブロッキング・トランスポート・インタフェースは、異なるレベルのタイミングの詳細をサポートするためにTLM-2.0に存在する、別のインタフェースである。ブロッキング・トランスポート・インタフェースはトランザクションの開始と終了をモデル化することのみ可能で、トランザクションは1回の関数コールで完了する。ノンブロッキング・トランスポート・インタフェースは1つのトランザクションを複数タイミング・ポイントへ分割し、典型的には1回のトランザクションに対して複数回の関数コールが要求される。

相互利用性のために、ブロッキングとノンブロッキング・トランスポート・インタフェースは1つのインタフェースに統合される。トランザクションを開始するモデルは、コーディング・スタイルにしたがって、ブロッキングもしくはノンブロッキング・トランスポート・インタフェース（または両方）を使用する。TLM-2.0トランスポート・インタフェースを提供するどのようなモ

デルも相互利用性を最大にするためにブロッキングとノンブロッキングの両インタフェースを提供しなければならない。しかし、そのようなモデルは内部で両インタフェースを実装しなければならないわけではない。

TLM-2.0 はブロッキング・トランスポート・コールをノンブロッキング・トランスポート・コールにおよび逆に自動変換する仕組み（いわゆる便利ソケット）を提供する。トランスポート・コール変換はあるコスト(特にノンブロッキング・コールをブロッキング実装へ変換時)が生じる。しかしながら、このコストのオーバーヘッドは、アプロキシメイトリー・タイムド・モデル自体がシミュレーション速度に負の影響を及ぼしやすいとういう事実により、和らげられる。

C++の静的な型決定ルールは両インタフェースの実装を強要するが、イニシエータはブロッキングまたはノンブロッキング・トランスポート・メソッドのどちらをコールするか、動的に選択することができる。別々のイニシエータは別々のメソッドをコールすることも可能だし、イニシエータは動作中にブロッキングとノンブロッキング・コールを切り替えることも可能。本標準は同一ターゲットへのブロッキングとノンブロッキング・トランスポート・コールの混在、順序付けを規定するルールが含まれる。

ブロッキング・トランスポート・インタフェースの強みは、1 回の関数コールでトランザクションを完了するモデルの単純なコーディング・スタイルを許すということである。ノンブロッキング・トランスポート・インタフェースの強みは、1 つのトランザクションに複数のタイミング・ポイントを結びつけることをサポートすることである。原理的に、どちらのインタフェースもテンポラル・デカップリングをサポートするが、アプロキシメイトリー・タイムド・モデルにおける複数タイミング･ポイントの存在がおそらくテンポラル･デカップリングの利点を無効化する。

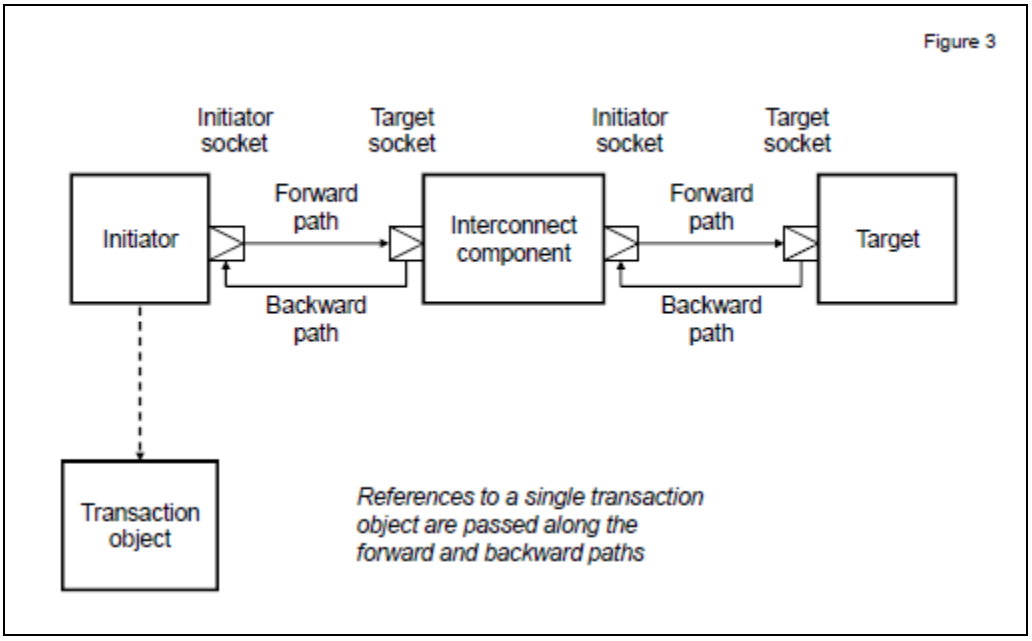
3.3.9 ユースケースとコーディング・スタイル

ユースケース	コーディング・スタイル
ソフトウェア・アプリケーション開発	ルーズリー・タイムド
ソフトウェア性能解析	ルーズリー・タイムド
ハードウェア・アーキテクチャ解析	ルーズリー・タイムド アプロキシメイトリー・タイムド
ハードウェア・パフォーマンス検証	アプロキシメイトリー・タイムド サイクル精度
ハードウェア機能検証	アンタイムド（検証環境） ルーズリー・タイムド アプロキシメイトリー・タイムド

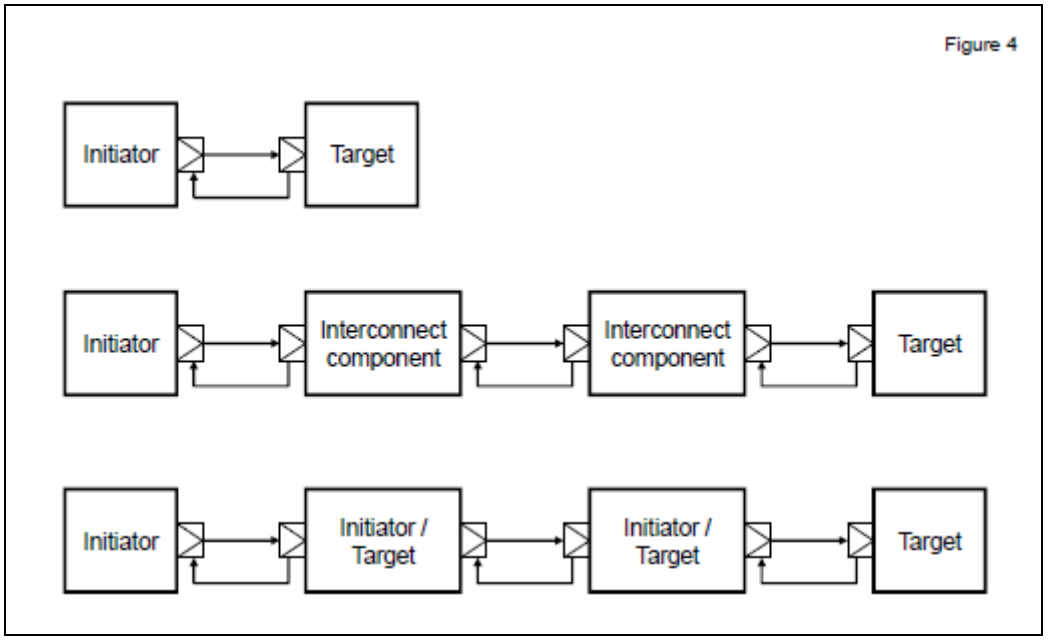
3.4 イニシエータ、ターゲット、ソケット、トランザクション・ブリッジ

TLM-2.0 コア・インタフェースはイニシエータとターゲット間でトランザクションを渡す。イニシエータはトランザクションを開始できるモジュールである。すなわち、新しいトランザクション・オブジェクトを生成し、コア・インタフェースの 1 つのメソッドをコールしてそれを渡す。ターゲットはトランザクションの最終到達地として動作するモジュールである。ライト・トランザクションの場合、イニシエータ（プロセッサなど）はデータをターゲット（メモリなど）へ書き込む。リード・トランザクションの場合、イニシエータはターゲットからデータを読み出す。インターコネクト・コンポーネントはトランザクションにアクセスするが、そのトランザクショ

ンのイニシエータとしてもターゲットとしても動作しないモジュールで、例えば、アービタとルータである。イニシエータ、インターコネクト、ターゲットの役割は動的に変わることができる。例えば、コンポーネントはあるトランザクションに対してはインターコネクトとして振舞い、他のトランザクションに対してはターゲットとして振舞うかもしれない。



この考え方を説明するために、このパラグラフでは典型的なトランザクション・オブジェクトのライフタイムについて述べる。トランザクション・オブジェクトはイニシエータによって生成され、トランスポート・インタフェース（ブロッキングまたはノンブロッキング）のメソッドの引数として渡される。そのメソッドはアービタのようなインターコネクト・コンポーネントで実装される。インターコネクト・コンポーネントでのメソッドの実装では、さらにトランスポート・コールによりトランザクション・オブジェクトを次モジュールへ渡す前にそのアトリビュートを読むかもしれない。2 回目のトランスポート・メソッドはルータのような 2 番目のインターコネクト・コンポーネントで実装される。2 番目のインターコネクト・コンポーネントはメモリのようなターゲット、すなわち、そのトランザクション・オブジェクトの最終目的地、への 3 回目のトランスポート・コールでトランザクションを渡す。（インターコネクト・コンポーネントの数はトランザクション毎に異なるし、まったくないかもしれない。）このような一連のメソッド・コールはフォワード・パスと呼ばれる。そのトランザクションはターゲットで処理され、次の 2 つの方法のどちらかの方法でイニシエータへ戻される。1 つは、リターン・パスと呼ばれるトランスポート・メソッド・コールの返り値として戻す。もう 1 つは、バックワード・パスと呼ばれるターゲットからイニシエータへの逆方向のトランスポート・メソッド・コールで戻す。これらの選択はノンブロッキング・トランスポート・メソッドの返り値によって決定される。（厳密に言うと、フォワードとバックワード・パスに対応する 2 つのリターン・パスがあることになる。しかし、それらがどちらを指すかは通常、文脈から明らかである。）



フォワード・パスは、イニシエータもしくはインターコネクト・コンポーネントが他のインターコネクト・コンポーネントもしくはターゲットの方向へのインタフェース・メソッド・コールである。バックワード・パスは、ターゲットもしくはインターコネクト・コンポーネントが他のインターコネクト・コンポーネントもしくはイニシエータの方向へのインタフェース・メソッド・コールである。イニシエータとターゲット間のパスは幾つかのホップからなる。ホップとは隣り合った 2 つのコンポーネントの接続のことである。イニシエータとターゲット間のホップ数はそのパス上のインターコネクト･コンポーネント数より 1 だけ大きい。汎用ペイロードを使う場合、フォワード・パスとバックワード・パスは常に同じコンポーネントとソケットを通過する。

フォワード・パスとバックワード・パスの両方をサポートするために、コンポーネント間の接続はポートとエクスポートを必要とする。ポートとエクスポートは接続されなければならない。これはイニシエータ・ソケットとターゲット・ソケットで容易化される。イニシエータ・ソケットはフォワード・パスのインタフェース・メソッド・コールのためのポートと、バックワード・パスのインタフェース・メソッド・コールのためのエクスポートを提供する。ターゲット・ソケットはその逆のものを提供する。（イニシエータ・ソケットは sc_port クラスを継承し、sc_export を持つ。ターゲット・ソケットは sc_export クラスを継承し、sc_port を持つ。）イニシエータ・ソケットとターゲット・ソケット・クラスは、フォワード・パスとバックワード・パス両方をバインドする SystemC port のバインド演算子をオーバーロードする。

トランスポート・インタフェースと同様に、ソケットは DMI とデバッグ・トランスポート・インタフェースを隠ぺいする（以下を参照）。

ソケットと使う場合、イニシエータ・コンポーネントは少なくとも 1 つのイニシエータ・ソケットを持ち、ターゲット・コンポーネントは少なくとも 1 つのターゲット・ソケットを持つ。インターコネクト・コンポーネントは両ソケットを少なくとも 1 つずつ持つ。コンポーネントは別々のトランザクション型を転送する幾つかのソケットを持つかもしれない。複数の独立したトランザクションに対してイニシエータもしくはターゲットとして動作するコンポーネントはそのようになる。

バス・ブリッジをモデリングするには 2 つのやり方がある。1 つは、バス・ブリッジをインター

コネクト・コンポーネントとしてモデリングするやり方であり、もう1つは、2つの個別のTLM-2.0トランザクション間のトランザクション・ブリッジとしてモデリングするやり方である。インターコネクト・コンポーネントは1つのトランザクション・オブジェクトへのポインタを渡す。これはシミュレーション速度に対して最善のアプローチである。トランザクション・ブリッジはトランザクション・オブジェクトをコピーすることを求められる。これは、2 つのトランザクションが別々のアトリビュートを持てるとので、より柔軟性がある。

最大限の相互利用性、便利性、一貫したコーディング・スタイルのためTLM-2.0 ソケットの使用が推奨される。TLM-2.0 コア・インタフェースと SystemC ポートとエクスポートを直接使うことも可能だが、推奨しない。

3.5 DMIとデバッグ・トランスポート・インタフェース

ダイレクト・メモリ・インタフェース（DMI）とデバッグ・トランスポート・インタフェースはトランスポート・インタフェースとは別個の特殊なインタフェースで、ターゲットが持つメモリ領域への直接アクセスとデバッグ・アクセスを提供する。一旦 DMI 要求が許可されると、DMIインタフェースはイニシエータがトランスポート・インタフェースで使われるインターコネクト・コンポーネントを通過する通常のパスをバイパスすることを可能にする。DMI は、ルーズリー・タイムド・シミュレーションで通常のメモリ・トランザクションを高速化することを意図する。デバッグ・トランスポート・インタフェースは、デバッグ・アクセスを通常のトランザクションに伴う遅延あるいは副作用から解放する。

DMI はフォワード（イニシエータからターゲットへ）インタフェースとバックワード（ターゲットからイニシエータへ）インタフェースがある。デバッグ・インタフェースはフォワード・インタフェースのみである。「4.2 ダイレクト・メモリ・インタフェース」と「4.3 デバッグ・トランスポート・インタフェース」を参照。

3.6 統合インタフェースとソケット

ブロッキングとノンブロッキング・トランスポート・インタフェースは、DMI とデバッグ・トランスポート・インタフェースとともに標準のイニシエータ・ソケット、ターゲット・ソケットに統合されている。4 つのインタフェース（2 つのトランスポート・インタフェース、DMI、デバッグ）すべては 1 つのターゲットへ並列にアクセスできる（本標準で記述されるルールに従う限りは）。これらの統合インタフェースは TLM-2.0 標準を使ったコンポーネント間の相互利用性を保証するキーの 1 つである。他のキーは汎用ペイロードである。「6.1 統合インタフェース」を参照。

標準ターゲット・ソケットは 4 つのインタフェースすべてを提供する。したがって、各ターゲット・コンポーネントは事実上 4 つすべてのインタフェースのメソッドを実装しなければならない。しかしながら、ブロッキングとノンブロッキング・トランスポート・インタフェースの設計は、これらの間の変換対策としての便利ソケットとともに、ターゲットはモデルの速度と精度の要求によってブロッキングもしくはノンブロッキング・トランスポート・メソッドのどちらかの（両方ではなく）実装を必要とする。

イニシエータは、速度と精度の要求に応じて、コア・インタフェースの幾つかもしくはすべてのメソッドをコールすることを選択する。コーディング・スタイルは、インタフェースの持つ特徴の適切な組み合わせを選択する助けになる。典型的には、ルーズリー・タイムド・イニシエータはブロッキング・トランスポート、DMI、デバッグをコールし、アプロキシメイトリー・タイムド・イニシエータはノンブロッキング・トランスポートとデバッグをコールする。

3.7 ネームスペース

TLM-2.0 クラスは 2 つのトップレベルの C++ネームスペース、tlm と tlm_utils の下で宣言されなければならない。TLM-2.0 クラスの特定な実装はこれら 2 つのネームスペース内でさらにネストしたネームスペース下でもよいが、このようなネストしたネームスペースはアプリケーション内で使われてはいけない。

ネームスペース	クラス
tlm	メモリ・マップド・バス・モデリング用の相互利用性インタフェースを含む。
tlm_utils	メモリ・マップド・バス・モデル間のインタフェースでの相互利用性に厳密には必要ないユーティリティ・クラスを含む。これらは、TLM-2.0 標準の一部である。

3.7.1 ヘッダファイルとバージョン

アプリケーションは配布されたソフトウェアの include/tlm ディレクトリ下のヘッダファイル tlm.h をインクルードすべきである。また、必要に応じて include/tlm/tlm_utils ディレクトリ下のヘッダファイルもインクルードすべきである。

シンプル・ソケットを OSCI シミュレータの現リリースバージョンでコンパイルするには、SystemC ヘッダファイルをインクルードする前に、マクロ SC_INCLUDE_DYNAMIC_PROCESSES を定義する必要がある。

3.8 ソフトウェアのバージョン情報

ヘッダファイル include/tlm/tlm_h/tlm_version.h は、配布された OSCI TLM-2.0 ソフトウェアのバージョン情報を示すマクロ、定数、関数を含まなければならない。

3.8.1 クラス定義

```
namespace tlm
{
  #define TLM_VERSION_MAJOR              implementation_defined_number // For example, 2
  #define TLM_VERSION_MINOR              implementation_defined_number // 0
  #define TLM_VERSION_PATCH              implementation_defined_number // 1
  #define TLM_VERSION_ORIGINATOR         implementation_defined_string  // “OSCI”
  #define TLM_VERSION_RELEASE_DATE       implementation_defined_date    // “20090329”
  #define TLM_VERSION_PRERELEASE         implementation_defined_string  // “beta”
  #define TLM_IS_PRERELEASE              implementation_defined_bool    // TRUE
  #define TLM_VERSION                    implementation_defined_string  // “2.0.1_beta-OSCI”
  #define TLM_COPYRIGHT                  implementation_defined_string
  const unsigned int tlm_version_major;
  const unsigned int tlm_version_minor;
  const unsigned int tlm_version_patch;
  const std::string tlm_version_originator;
  const std::string tlm_version_release_date;
  const std::string tlm_version_prerelease;
```

```
    const bool tlm_is_prerelease;
    const std::string tlm_version_string;
    const std::string tlm_copyright_string;
    inline const char* tlm_release();
    inline const char* tlm_version();
    inline const char* tlm_copyright();
} // namespace tlm
```

3.8.2 ルール

a) implementation_defined_number は文字セット[0-9]からなる 10 進数であり、引用符で囲まれていないこと。

b) originator と pre-release strings は文字セット[A-Z][a-z][0-9]_から成る文字列であり、引用符に囲まれていること。

c) version release date は YYYYMMDD の形、8 個の文字は 10 進数、の ISO 8601 basic format calendar date で、引用符に囲まれていること。

d) IS_PRERELEASE flag は FALSE か TRUE のどちらかで、引用符に囲まれていないこと。

e) version string は、”major.minor.patch_prerelease-originator”または”major.minor.patch-originator”の値であること。major、minor、patch、prerelease、originator はそれぞれに対応した文字列の値である（引用符で囲まれない）。prerelease 文字列があるかないかは IS_PRERELEASE flag の値による。

f) copyright string は著作権表示であるべきである。

g) 各定数は適切なデータ型に変換されたマクロで定義された値で初期化されなければならない。

h) tlm_release と tlm_version メソッドは C string へ変換された version string の値を返さなければならない。

i) tlm_copyright メソッドは C string へ変換された copyright string を返さなければならない。

4. TLM-2 コア・インタフェース

TLM-1 からのコア・インタフェースに加えて、TLM-2.0 ではブロッキングとノンブロッキングのトランスポート・インタフェース、ダイレクト・メモリ・インタフェース（DMI）、およびデバッグ用トランスポート・インタフェースが追加された。

4.1 トランスポート・インタフェース

トランスポート・インタフェースはイニシエータとターゲット、そしてインターコネクト・コンポーネントの間のトランザクションを転送するのに使用される主要なインタフェースである。ブロッキング及びノンブロッキングのトランスポート・インタフェースの両方ともタイミング・アノテーションとテンポラル・デカップリングをサポートするが、ノンブロッキング・トランスポート・インタフェースだけがトランザクションのライフタイム中の複数フェーズをサポートする。ブロッキング・トランスポートは明示的なフェーズ引数を持たず、ブロッキング・トランスポートとノンブロッキング・トランスポート・インタフェースのフェーズの間のどんな関連でも純粋に抽象的である。ノンブロッキング・トランスポートのメソッドだけが、リターン・パスが使用されたかどうかを示す値を返す。

トランスポート・インタフェースと汎用ペイロードは、メモリ・マップド・バスの抽象モデリングで高速に実行するために設計された。トランスポート・インタフェース・テンプレートは、それらが汎用ペイロードとは独立して使うトランザクション型と一緒に特殊化されるが、相互利用性の利益の多くを失う。

トランザクション・オブジェクトのメモリ管理、トランザクションの順序、及び許可された関数呼び出しのシーケンスの規則は、トランスポート・インタフェースのテンプレート引数として渡された特定のトランザクション型に依存しており、同様にソケットのテンプレート引数として渡されたプロトコル・トレイツ・クラスに依存している。（もしソケットが使用されているならば）

4.1.1 ブロッキング・トランスポート・インタフェース

4.1.1.1 イントロダクション

新しい TLM-2.0 のブロッキング・トランスポート・インタフェースは、ルーズリー・タイムド・コーディング・スタイルをサポートする事を目的としている。ブロッキング・トランスポート・インタフェースの利用が適当な場合は、イニシエータが 1 回の関数呼び出しで、ターゲットと一緒にトランザクションを完了したい場合、すなわち、観測するべきタイミング・ポイントが、トランザクションの開始と終了だけである場合である。

ブロッキング・トランスポート・インタフェースは、イニシエータからターゲットまではフォワード・パスだけを使用する。

TLM-2.0 のブロッキング・トランスポート・インタフェースは、TLM-2.0 の規格の一部である TLM-1 からのトランスポート・インタフェースと意図的に類似性を持たせているが、TLM-1 のトランスポート・インタフェースと TLM-2.0 のブロッキング・トランスポート・インタフェースは同一ではない。特に、新しい b_transport メソッドは、非 const 参照渡しによる単一のトランザクション引数とタイミングをアノテートする為の第2引数を持っているのに対し、TLM-1 の transport

メソッドは、タイミング・アノテーションを持っていない単一の const 参照のリクエスト引数を取り、レスポンスの値を返す。TLM-1 は、リクエストとレスポンスで異なるオブジェクトを持ち、値で渡されるのに対し（もしくは const の参照による）、TLM-2.0 ではブロッキング・インタフェースを使用するかノンブロッキング・インタフェースを使用するかに関わらず、参照渡しによる単一のトランザクション・オブジェクトを仮定する。

b_transport メソッドは、タイミング・アノテーション引数を持つ。この単一の引数は、b_transport の呼び出しと b_transport からの応答の両方で使用されるもので、現在のシミュレーション時間に対する相対時間でトランザクションの開始及び終了の各々の時間を示す。

4.1.1.2 クラス定義

```
namespace tlm {
  template <typename TRANS = tlm_generic_payload>
  class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
  public:
    virtual void b_transport(TRANS& trans, sc_core::sc_time& t) = 0;
  };
} // namespace tlm
```

4.1.1.3 TRANSテンプレート引数

このコア・インタフェースは、あらゆる型のトランザクションを転送するのに使用できる。特別なトランザクション型である tlm_generic_payload は、トランザクションのアトリビュートの正確性や詳細があまり重要でない場合のモデル間の相互利用性の簡便さの為に提供される。

最大限の相互利用性の為には、アプリケーションは基本プロトコルと一緒に標準のトランザクション型である tlm_generic_payload を使うべきである。「8.2 基本プロトコル」を参照。特定のプロトコルをモデル化するために、アプリケーションは独自のトランザクション型で置き換えることができる。異なるトランザクション型で特殊化されたインタフェースを使用するソケットは、バインド出来ない。そして、コンパイル時チェックが提供されるが、相互利用性を制限する。

4.1.1.4 規則

a) b_transport メソッドは、wait を直接もしくは間接的に呼び出すことができる。

b) b_transport メソッドは、メソッド・プロセスからは呼び出されないものとする。

c) イニシエータは、1 つの呼び出しから次の呼び出し、トランスポート・インタフェース、DMI、及びデバッグ・トランスポート・インタフェースに渡る呼び出しでトランザクション・オブジェクトを再利用することができる。

d) b_transport の呼び出しは、トランザクションの最初のタイミング・ポイントである事を示す。b_transport からの応答は、トランザクションの最終的なタイミング・ポイントである事を示す。

e) タイミング・アノテーション引数は、関数呼び出し及び応答が実行されるシミュレーション時間（sc_time_stamp()で返された値）からのオフセットのタイミング・ポイントである事を示す。

f) 呼び出し先関数は、トランザクション・クラス TRANS によって課せられたあらゆる制約にも従うようにトランザクション・オブジェクトを変更もしくはアップデートしてもかまわない。

g) トランザクション・オブジェクトにはタイミング情報を含まないことを推奨する。タイミングは b_transport の sc_time 引数を使ってアノテートするべきである。

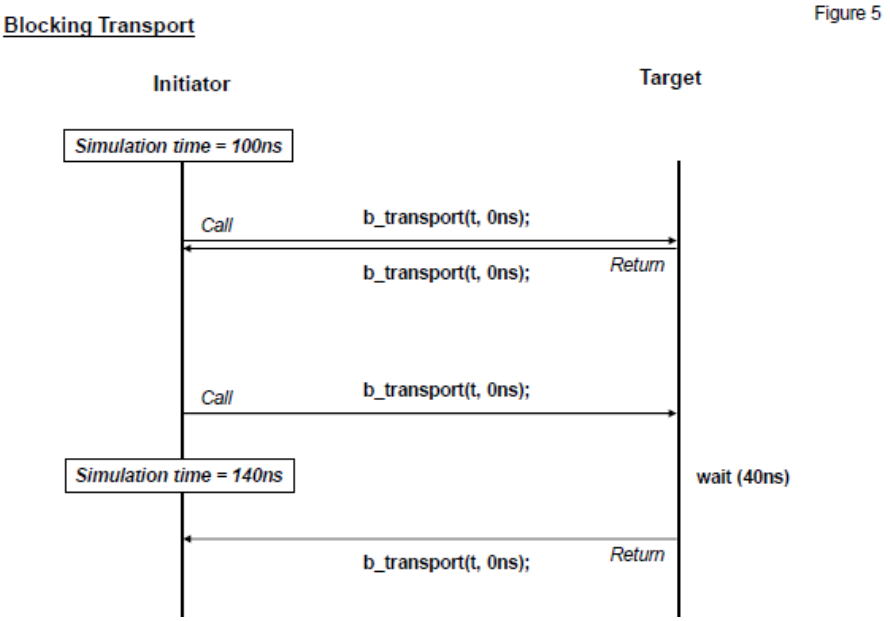
h) 通常、インターコネクト・コンポーネントはフォワード・パスに沿って b_transport の呼び出しをイニシエータからターゲットまで受け渡すべきである。言い換えれば、インターコネクト・コンポーネントのターゲット・ソケットにおける b_transport の実装は、イニシエータ・ソケットの b_transport メソッドを呼び出すことができる。

i) b_transport の実装が nb_transport_fw を呼び出す事を許可されるかどうかは、プロトコルと連携したルールに依存する。基本プロトコルにおいては、便利ソケットの simple_target_socket が自動的にこの変換を行う事が出来る。「12.1.2 シンプル・ソケット」を参照の事。

j) b_transport の実装は nb_transport_bw を呼び出してはならない。

4.1.1.5 メッセージ・シーケンス図ーブロッキング・トランスポート

ブロッキング・トランスポート・メソッドは、すぐに戻ることもできるし（すなわち、現在の SystemC 評価フェーズで）、スケジューラに制御を譲渡し、シミュレーション時間上の後のポイントでイニシエータに戻ることもできる。イニシエータのスレッドはブロックされるかもしれないが、イニシエータの他のスレッドは、プロトコルに依存して、最初の関数呼び出しが応答する前に b_transport を呼び出すことが許可されるかもしれない。

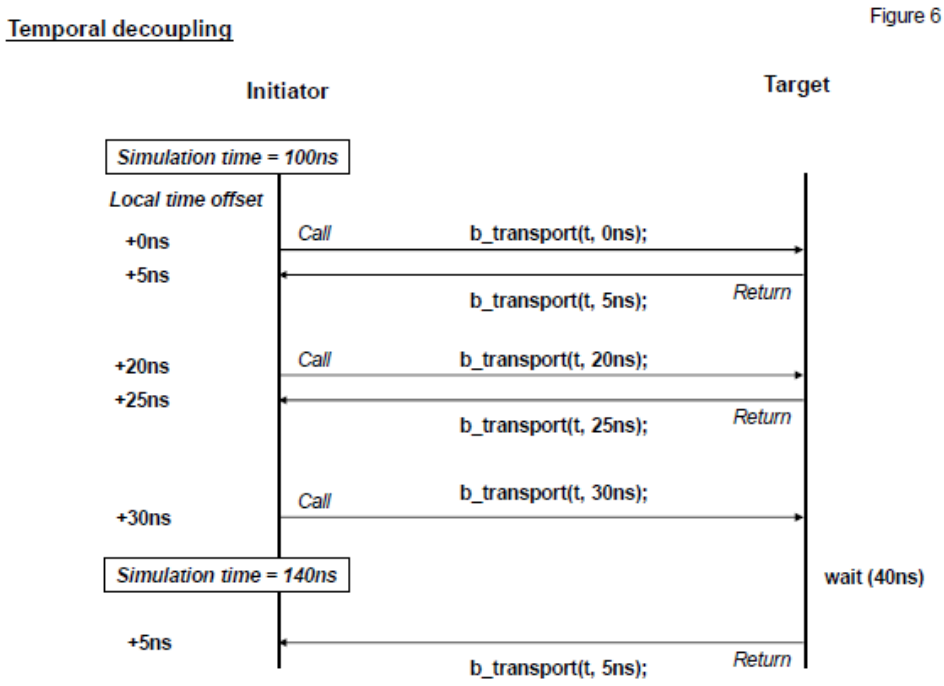


4.1.1.6 メッセージ・シーケンス図ーテンポラル・デカップリング

テンポラル・デカップリングされたイニシエータは、現在のシミュレーション時間に先立って抽象的なローカル・タイムで実行されるかもしれないが、その場合、以下に示すように、時間引数として非ゼロの値を b_transport に渡すべきである。イニシエータとターゲットは、時間引数の値

を増加させる事で、各々ローカル・タイム・オフセットを進めるであろう。呼び出し先から返された時間引数を加えた現在のシミュレーション時間は、各々のトランザクションが完了するまでの名目上の時間を表すが、シミュレーション時間はイニシエータのスレッドが制御をカーネルに譲渡するまで進むことが出来ない。b_transport の本体は、wait を呼び出してもよいが、その場合はローカル・タイム・オフセットをゼロにリセットすべきである。以下のダイアグラムでは、イニシエータからの最後の応答は 140ns のシミュレーション時間後に起こるが、5ns の遅延がアノテートされたため、実効ローカル・タイムとしては 145ns となる。

Figure 6



4.1.1.7 メッセージ・シーケンス図ータイム・クォンタム

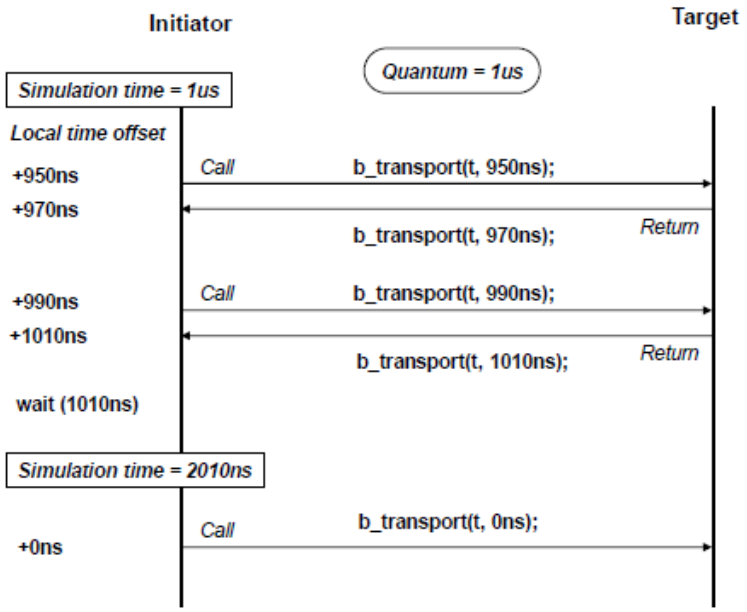
テンポラル・デカップリングされたイニシエータは、タイム・クォンタムが超えるまでローカル・タイムで進み続ける。その時点でイニシエータは、直接または間接的に引数としてローカル・タイムを引数に wait メソッドを呼び出すことにより、実行を中断し、同期しなければならない。これにより、モデル中の他のイニシエータが動作して、追いつくことができる。そして、それは効果的に、イニシエータが順番に実行することを意味し、各々が自身のローカル・タイムの経過を管理することによって、いつ制御を返すべきかを決定する役割を果たす。オリジナルのイニシエータは、シミュレーション時間が次のクォンタムに達した後にだけ、再開するべきである。

ルーズリー・タイムド・コーディング・スタイル中の遅延の第一の目的は、各々のイニシエータがいつ制御を返すべきかを決定する事を考慮する為のものである。もし正確に機能するためのタイミングの詳細を当てにしないのであれば、それが最良である。

各クォンタム中、所定のイニシエータによって生成されたトランザクションは厳密な連続した順序で起こるが、シミュレーション時間は進まない。ローカル・タイムは SystemC のスケジューラによって追跡されない。

Figure 7

The time quantum



4.1.2 ノンブロッキング・トランスポート・インタフェース

4.1.2.1 イントロダクション

ノンブロッキング・インタフェースは、アプロキシメイトリー・タイムドによるコーディング・スタイルのサポートを意図しており、各トランザクションが通過するイニシエータとターゲット間の相関関係の詳細な流れをモデリングするのに適している。言い換えれば、トランザクションを複数のフェーズに分解することであり、各々のフェーズはタイミング・ポイントに関連付けられる。ノンブロッキング・トランスポート・メソッドからの各呼び出しと戻りは、フェーズ・トランジションに対応するかもしれない。タイミング・ポイント数を 2 つに制限する事によってルーズリー・タイムドにも使用する事は出来るが、一般的には推奨されない。ルーズリー・タイムドにおいてはブロッキング・インタフェースの容易性が好まれる。ノンブロッキング・インタフェースは、ブロッキング・トランスポート・インタフェースを使用したモデリングが難しい、パイプライン化されたトランザクション・モデリングに適している。

ノンブロッキング・インタフェースはイニシエータからターゲット（フォワード・パス）と、ターゲットからイニシエータ（バックワード・パス）の両方で使用する。逆方向パスで使用される tlm_fw_nonblocking_transport_if と tlm_bw_nonblocking_transport_if の 2 つの異なるインタフェースが用意されている。ノンブロッキング・インタフェースは、ノンブロッキング・インタフェース・メソッドがトランザクション・オブジェクトとタイミング・アノテーションの非 const 参照を渡す点だけがブロッキング・インタフェースの引数渡しのメカニズムと良く似ている。ノンブロッキング・インタフェース・メソッドはまた、トランザクションの状態を示す為のフェーズを渡し、関数からの戻りがフェーズ・トランジションを表すかを示す為に列挙型の値を返す。ブロッキングとノンブロッキングの両方のトランザクションがタイミング・アノテーションをサポートするが、ノンブロッキングだけがトランザクションのライフタイムの間、複数フェーズをサポートしている。ブロッキング、ノンブロッキング・トランザクション・インタフェースと汎用ペイロー

ドは、高速で抽象モデリングされるメモリ・マップド・バスと共に使用する事を前提に設計されている。しかし、トランスポート・インタフェースは特定のプロトコルをモデル化する為の汎用ペイロードとは独立して使用出来る。トランザクションの型とフェーズの型はいずれもノンブロッキング・トランスポート・インタフェースのテンプレート引数である。

4.1.2.2 クラス定義

```
namespace tlm {
  enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };
  template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
  class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
  public:
    virtual tlm_sync_enum nb_transport_fw(TRANS& trans, PHASE& phase,
    sc_core::sc_time& t) = 0;
  };
  template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
  class tlm_bw_nonblocking_transport_if : public virtual sc_core::sc_interface {
  public:
    virtual tlm_sync_enum nb_transport_bw(TRANS& trans, PHASE& phase,
    sc_core::sc_time& t) = 0;
  };
} // namespace tlm
```

4.1.2.3 TRANSとPHASEテンプレート引数

ノンブロッキング・トランスポート・インタフェースは、任意のフェーズ数やタイミング・ポイントと共に、あらゆる型のトランスポート・トランザクションで使用しても良い事が意図されている。詳細なトランザクションのアトリビュートが重要にならないようなモデル間の相互利用性向上用に、特別なトランザクション型である tlm_generic_payload 型が提供され、また基本プロトコルで使用される特別な型である tlm_phase 型も提供されている。「8.2 基本プロトコル」を参照の事。

相互利用性を最大限に活用するには基本プロトコルとしてデフォルト・トランザクション型の tlm_generic_payload とデフォルト・フェーズ型の tlm_phase を使用すべきである。特定のプロトコルをモデル化するために、アプリケーションはそれら自身のトランザクション型とフェーズ型を代用するかもしれない。異なったトランザクション型と共に定義されたインタフェースを使うソケットは相互に接続できず、コンパイル時のチェックを提供するが、しかし相互利用性は制限される。

4.1.2.4 nb_transport_fwとnb_transport_bw呼び出し

a) フォワード・パス上で使用する nb_transport_fw と、バックワード・パス上で使用する nb_transport_bw の二つのノンブロッキング・トランスポート・メソッドが用意されている。名前と呼び出しの方向は異なるが、これら二つのメソッドのセマンティクスはよく似ている。本書ではイタリック体で書かれている nb_transport は、二つのメソッドを区別する必要が無い状況において、両方のメソッドを説明する際に使用される。

b) 基本プロトコルの場合、フォワードとバックワードのパスは逆並びで同じコンポーネントとソケットのシーケンスを通じて渡すべきである。各コンポーネントは、そのトランザクション最初に受けとったターゲット・ソケットを使用して、どんなトランザクションもイニシエータへ返送する責任がある。

c) nb_transport_fw はフォワード・パスのみで呼ばれるものとし、nb_transport_bw はバックワード・パスのみで呼ばれるものとする。

d) フォワード・パス上の nb_transport_fw 呼び出しは、バックワード・パス上の nb_transport_bw を直接的または間接的に呼び出さないものとする。また逆も同様である。

e) nb_transport メソッドは直接、または間接的に wait を呼ばないものとする。

f) nb_transport メソッドはスレッド・プロセスまたはメソッド・プロセスから呼ばれる。

g) nb_transport では b_transport を呼び出す事が許可されていない。一つの解決策は、オリジナルの nb_transport_fw メソッドによって生成または通知された別のスレッド・プロセスから b_transport を呼び出す事である。基本プロトコルでは便利ソケットの simple_target_socket が提供されており、この変換を自動的に行わせる事が出来る。詳細は「12.1.2 シンプル・ソケット」を参照。

h) ノンブロッキング・トランスポート・インタフェースは、明確にパイプライン化されたトランザクションのサポートを意図している。言い換えれば、最初のトランザクションの完了を待つ必要はなく、同じプロセスから nb_transport_fw が連続して呼ばれ、それぞれ個別のトランザクションを開始する事が出来る。

i) 原則として、トランザクションの最終タイミング・ポイントは、フォワード・パスまたはバックワード・パスのいずれかにおける nb_transport を呼び出しか、そこから戻る事によって検出される。

4.1.2.5 trans引数

a) 与えられたトランザクション・オブジェクトのライフタイムは、nb_transport への一連の呼び出しがイニシエータ、インターコネクト・コンポーネント、ターゲット間で一つのトランザクション・オブジェクトをフォワードとバックワードへ渡すことができるよう、nb_transport からの戻りを超えて拡張しても良い。

b) 特定のトランザクション・インスタンスに関連している nb_transport への複数の呼び出しがあれば、全く同じトランザクション・オブジェクトは引数としてそのようなあらゆる呼び出しに渡される。言い換えれば、特定のトランザクション・インスタンスは単一のトランザクション・オブジェクトによって表される。

c) イニシエータは、1 つ以上のトランザクション・インスタンスを表す場合、またはトランスポート・インタフェース、DMI、およびデバッグ・トランスポート・インタフェースに渡る呼び出しで特定のトランザクション・オブジェクトを再利用してもよい。

d) トランザクション・オブジェクトのライフタイムは nb_transport の呼び出し毎に拡張できる為、呼び出し元関数と呼び出し先関数のいずれかは、トランザクション・クラスの TRANS によって課せられたなんらかの制約条件の下に、そのトランザクション・オブジェクトの変更または更新を行っても良い。例えば、汎用ペイロードでは、そのターゲットはリード・コ

マンドの場合にそのトランザクション・オブジェクトのデータ配列をアップデートしても良いが、コマンド・フィールドを更新すべきではない。「7.7 アトリビュートのデフォルト値と変更可否」を参照の事。

4.1.2.6 phase引数

a) 各 nb_transport 呼び出しはフェーズ・オブジェクトの参照を渡す。基本プロトコルの場合、同一のフェーズにおける nb_transport の連続的な呼び出しは許可されない。各フェーズ・トランジションには、関連するタイミング・ポイントがある。sc_time 引数を使用したタイミング・アノテーションは、フェーズ・トランジションに関連したタイミング・ポイントを遅らせるものとする。

b) フェーズ引数は参照渡しされる。呼び出し元関数または呼び出し先関数のいずれかがそのフェーズを変更してもよい。

c) 意図として、フェーズ引数が、トランザクションのアトリビュートを読むもしくは変更するのがいつかそして変更されたかどうかを、コンポーネントを知らせる為に使用されているべきであるということである。もしプロトコルの規則であるコンポーネントが特定のフェーズの間トランザクションのアトリビュート値を変更できるものならば、そのコンポーネントはそのフェーズの時にいつでもそして何回でも値を変更するかもしれない。プロトコルは、値が次のフェーズ・トランジションの後に読まれることを許可するだけにし、他のコンポーネントがそのフェーズの間アトリビュート値を読むのを禁じるべきである。

d) フェーズ引数の値は、特定のホップのためにプロトコル・ステートマシンの現在の状態を表す。単一トランザクション・オブジェクトは二つ以上のコンポーネント（イニシエータ、インターコネクト、ターゲット）間のいずれかに渡され、各ホップは、別々のプロトコル・ステートマシンを（少なくとも概念的には）要求する。

e) トランザクション・オブジェクトは一つの nb_transport 呼び出しを超えるライフタイムとスコープを持つのに対して、フェーズ・オブジェクトは一般的に呼び出し元関数においてローカルである。トランザクションを与える為の各 nb_transport 呼び出しは異なるフェーズ・オブジェクトを持っても良い。異なるホップ上に対応するフェーズ・トランジションは、シミュレーション時間内の異なるポイントで発生しても良い。

f) デフォルト・フェーズ型の tlm_phase は基本プロトコル向けである。その他プロトコルについては、拡張型の tlm_phase、またはそれ自身のフェーズ型（相互利用性低下への対応と共に）を代入出来る。「8.1 フェーズ」を参照。

4.1.2.7 tlm_sync_enumの戻り値

a) 同期の概念は方々で示されている。同期する事とは、いくつかプロセスが動作している中でSystemC スケジューラに制御を譲渡する事であるが、テンポラル・デカップリング用に追加的な意味も含まれる。これは別の場所でより多くの議論がされている。「12.2.4 テンポラル・デカップリングを使用するプロセスに関する一般ルール」を参照。

b) 原則として同期は制御の譲渡によって実行可能であるが（スレッド・プロセスの場合 wait を呼ぶか、またはメソッド・プロセスでカーネルに戻る）、テンポラル・デカップリングされたイニシエータは、tlm_quantumkeeper クラスの sync メソッドを呼ぶことによって同期する。一般的に他の SystemC プロセスが実行される事を許す一方で、時々同期するイニシエー

タにはそれが必要である。

c) 下記のルールはフォワードとバックワード・パスの両方に適用する。

d) nb_transport の戻り値の意味は固定されており、トランザクション型またはフェーズ型によって変わらない。したがって、以下の規則は基本プロトコルの制限を受けないが、しかしノンブロッキング・トランスポート・インタフェースをパラメタライズする際に使用される、あらゆるトランザクションとフェーズ型に適用する。

e) TLM_ACCEPTED： 呼び出し先関数は呼び出し中のトランザクション・オブジェクト、フェーズ、または時間引数の状態を変更しないものとする。言い換えると、TLM_ACCEPTED はリターン・パスが使用されていない事を示す。呼び出し元関数は、呼び出し先関数がそれらを変わりがないままにするのが強いられるので、呼び出しに続く nb_transport の引数の値を無視するかもしれない。一般に、呼び出し先関数を含むコンポーネントがトランザクションに応じることができる前に、呼び出し元関数は制御を譲渡しなければなければならないだろう。基本プロトコルのために、無視可能なフェーズを無視している呼び出し先関数は TLM_ACCEPTED を返すべきである。

f) TLM_UPDATE： 呼び出し先関数はトランザクション・オブジェクトを更新する。呼び出し先関数は、呼び出し中にフェーズ引数の状態変更、トランザクション・オブジェクトの状態変更、時間引数の値の増加を行える。言い換えると、TLM_UPDATED はリターン・パスが使用されている事を示し、そして呼び出し先関数は、トランザクションと共に連想されるプロトコル・ステートマシンの状態を進める。呼び出し先関数が各々引数を変更する事を実際に行う義務があるかどうかはそのプロトコルに依存する。nb_transport 呼び出しに続き、その呼び出し先関数は、フェーズ・トランジション、または時間引数を検査し、適切な動作をすべきである。

g) TLM_COMPLETED： 呼び出し先関数はトランザクション・オブジェクトを更新して、そしてそのトランザクションは完了する。呼び出し先関数は、呼び出し中にフェーズ引数の状態変更、トランザクション・オブジェクトの状態変更、時間引数の値の増加を行える。フェーズ引数の値は未定義である。言い換えると、TLM_COMPLETED は、リターン・パスが使用されており、トランザクションが特定のソケットに関して完了していることを示す。nb_transport 呼び出しに続くその呼び出し先関数は、フェーズ・トランジション、または時間引数を確認し、適切な動作をすべきであるが、フェーズ引数は無視するべきである。これ以上フォワードまたはバックワードのいずれかに沿って、この現在のソケットを通過したトランザクションに関連して呼び出されるトランスポートはない。この意味上の完了は、必ずしも無事に完了したという意味ではなく、その為トランザクション型に依存して呼び出し元関数はトランザクション・オブジェクト内に埋め込まれたレスポンス状態の検査を必要とする可能性がある。

h) 一般に、nb_transport に TLM_COMPLETED を返させることによってトランザクションを完了する義務は全くない。トランザクションは、プロトコルの最終フェーズが nb_transport の引数として渡されるとき、特定のソケットに関してどのような場合でも完了する。（基本プロトコルでは、最終フェーズは END_RESP である。） 言い換えれば、TLM_COMPLETED は義務ではない。

i) ３つの戻り値のいずれであってもプロトコルによって、nb_transport の呼び出しに続く呼び出し先関数はトランザクション・オブジェクトのレスポンスを生成するか、またはリリースす

る為に呼び出し先関数を含むコンポーネントを許可する上で制御の譲渡を必要とする可能性がある。

4.1.2.8 tlm_sync_enumのまとめ

tlm_sync_enum	Transaction object	Phase on return	Timing annotation on return
TLM_ACCEPTED	Unmodified	Unchanged	Unchanged
TLM_UPDATED	Updated	Changed	May be increased
TLM_COMPLETED	Updated	Ignored	May be increased

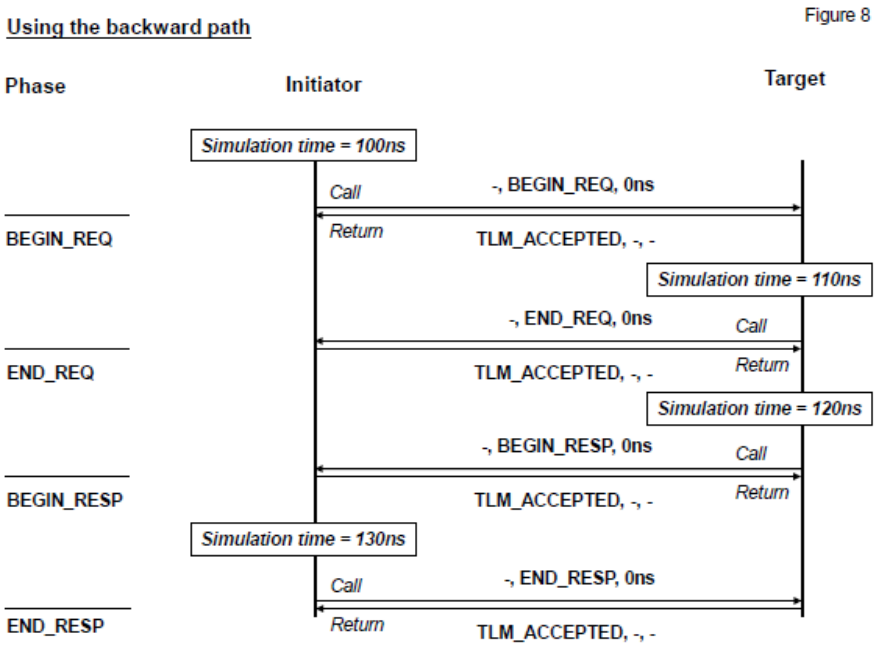
4.1.2.9 メッセージ・シーケンス・チャート – バックワード・パスを使用しているケース

記のメッセージ・シーケンス・チャートは、nb_transport のシーケンスを呼び出すバリエーションを図示している。nb_transport へ、または nb_transport から渡される引数と戻り値は、return、phase、delay の表記で表される。return は関数呼び出しからの戻り値、phase はフェーズ引数の値、delay は sc_time 引数の値を意味し、“ー”の表記は値が使用されない事を示す。

下記のメッセージ・シーケンス・チャートを例として、BEGIN_REQ や END_REQ 等の基本プロトコルのフェーズを使用する。アプロキシメイトリー・タイムド・コーディング・スタイルと基本プロトコルで、トランザクションはイニシエータとターゲット間を前後２回渡される。他のプロトコルではフェーズ数とその名前は異なる可能性がある。

nb_transport 呼び出しの受け取り側が、その次のトランザクションの状態、または次のタイミング・ポイントの遅延を即座に計算出来ない場合、それは TLM_ACCEPTED の値を戻すべきである。呼び出し元関数は、スケジューラへ制御を譲渡し、呼び出し先関数が応答する準備が完了した時に、反対の経路から nb_transport の呼び出しを受ける事を想定すべきである。ルーズリー・タイムドの場合と異なったこの場合、呼び出し元関数がイニシエータかターゲットであるかもしれないのに注意すべきである。

トランザクションはパイプラインされても良い。イニシエータは前のトランザクションの最終フェーズ・トランジションを確認する以前に、ターゲットに対して他のトランザクションを送る為の nb_transport を呼び出す事が出来る。プロセスはシミュレーション時間を進める事を許可する為の一般的な制御の譲渡をしているのであり、アプロキシメイトリー・タイムド・コーディング・スタイルは、ルーズリー・タイムド・コーディング・スタイルより遅いシミュレーションを実行する事が想定されている。



4.1.2.10 メッセージ・シーケンス・チャート ― リターン・パスを使用しているケース

nb_transport 呼び出しの受け取り側が、そのトランザクションの次の状態および次のタイミング・ポイントの遅延を即座に計算出来る場合、それは反対の経路を使用するのではなく nb_transport からのリターンにより新しいステートを戻す可能性がある。

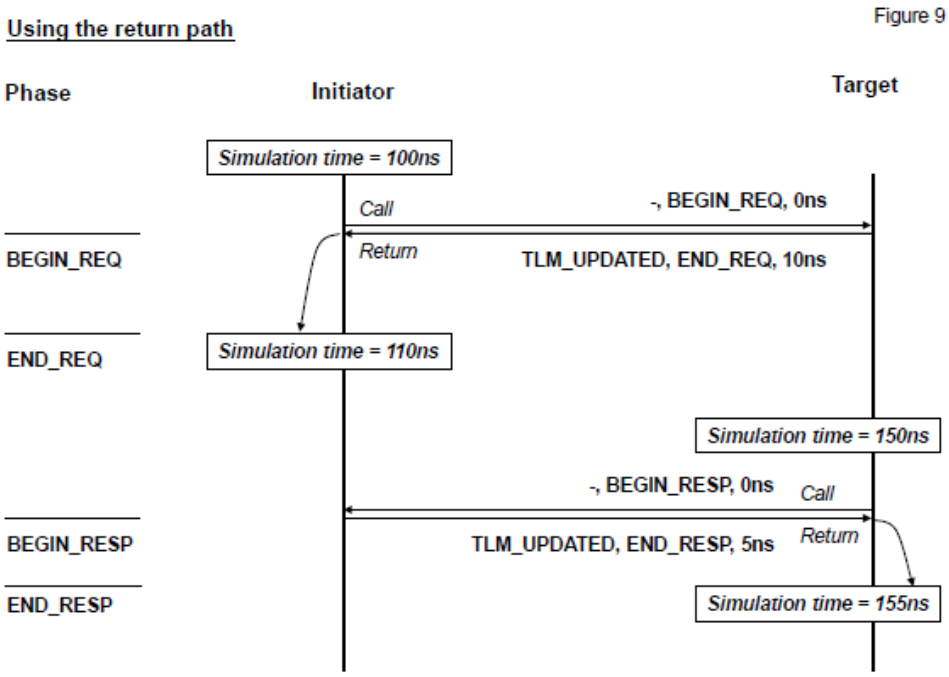
次のタイミング・ポイントがトランザクションの終わりを示すなら、受け取り側は TLM_UPDATED か TLM_COMPLETED のどちらかを返すことができる。

トランザクションの完了時に呼び出し先関数は、それが他のフェーズを先取りして最終フェーズまでジャンプした事を呼び出し元関数に示す為に、(プロトコルの規則を条件として) どのステージでも TLM_COMPLETED を返すことが出来る。

これはイニシエータとターゲットのいずれも適用される。

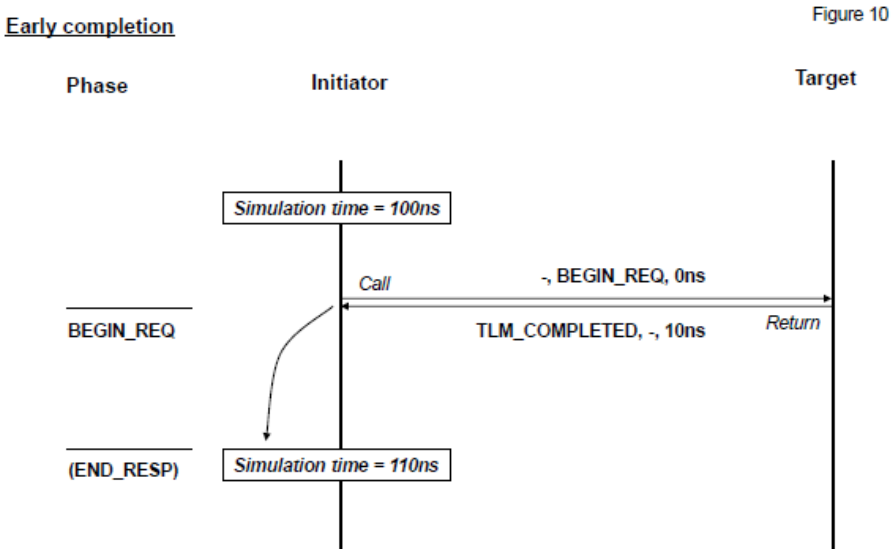
TLM_UPDATED と共に、呼び出し先関数はトランザクション、フェーズ、およびタイミング・アノテートを更新すべきである。

以下のダイアグラムでは、リターン時に関数呼び出しから渡された非ゼロのタイミング・アノテーション引数は、ホップの上のフェーズ・トランジションと対応するタイミング・ポイントの間の遅れを呼び出し元関数に示す。



4.1.2.11 メッセージ・シーケンス・チャート ― 早期完了

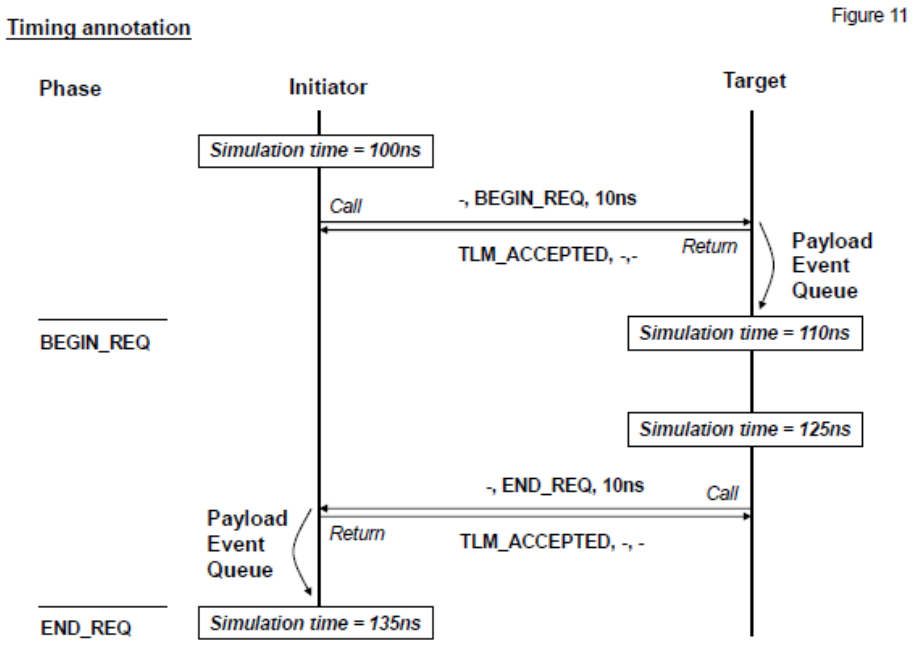
プロトコルに依存して、イニシエータやターゲットは早期にトランザクションを完了する為に nb_transport から TLM_COMPLETED を任意のタイミング・ポイントで返しても良い。イニシエータとターゲットのいずれも、このトランザクション・インスタンスの nb_transport 呼び出しを行わなくても良い。イニシエータやターゲットは、このようなトランザクションのショートカットを受け入れられるか否かは、その特定のプロトコルのルールに依存する。以下のダイアグラムのリターン・パスにおけるタイミング・アノテーションは、与えられた遅延後に最終タイミング・ポイントが起こる事をイニシエータに指示している。BEGIN_REQ から END_REQ 及び BEGIN_RESP を通し END_RESP までのフェーズ・トランジションは、引数として明示的に nb_transport に渡されるものより暗黙的である。



4.1.2.12 メッセージ・シーケンス・チャート ― タイミング・アノテーション

呼び出し元関数は、nb_transport 呼び出しに遅延をアノテートしても良い。これは、呼び出し先関数へのトランザクションがまるで特定の遅れの後にそれを受け取ったかのように、処理されるべきであるという指示である。アプロキシメイトリー・タイムドな呼び出し先関数は、このシミュレーション時間がアノテートされた時間に到達した際、処理する為のペイロード・イベント・キューにトランザクションを入れる事によって、通常この状況をハンドルする。ペイロード・イベント・キューの実現によって、この処理はペイロード・イベント・キューからのイベント通知による SystemC のプロセス・センシティブか、もしくはペイロード・イベント・キューに登録されたコールバックによって発生させるかもしれない。遅延はフォワードまたはバックワード・パスまたはそれに対応するリターン・パスのいずれかの呼び出しにアノテートさせる事が出来る。アプロキシメイトリー・タイムドのイニシエータはフォワード・リターン・パスとバックワード・パスの両方で入って来るトランザクションを同様に扱うはずである。同じく、アプロキシメイトリー・タイムドのターゲットはバックワード・リターン・パスとフォワード・パスの両方で入って来るトランザクションを同様に扱うはずである。

Figure 11



4.1.3 トランスポート・インタフェースにおけるタイミング・アノテーション

タイミング・アノテーションはブロッキングまたはノンブロッキング・トランスポート・インタフェースの共有の機能であり、b_transport、nb_transport_fw、nb_transport_bw メソッドに sc_time 引数を使用して表される。このドキュメントでは、イタリック体の transport は、b_transport、nb_transport_fw、nb_transport_bw の３つのメソッドの意味で用いられる。トランザクションの順番はコア・インタフェース規則とプロトコル規則の組み合わせで管理される。以下の節の規則はプロトコルの選択にかかわらずコア・インタフェースに適用される。基本プロトコルにおいて、ここに与えられた規則は「11.4.6 タイミング・アノテーション関連の基本プロトコル・ルール」に関連して読まれるべきである。

4.1.3.1 sc_time引数

a) トランザクション・オブジェクトは、タイミング情報を含まない事が推奨される。あらゆるタイミング・アノテーションは transport の sc_time 引数を使用してアノテートされるべきである。

b) 時間引数はマイナスにはならず、常にカレント・シミュレーション時間の sc_time_stamp()に連動して表される。

c) 時間引数は transport の呼び出しと transport からの戻りの両方で適用されるものとする。（tlm_sync_enum のルールに従って nb_transport の値を戻す）

d) nb_transport メソッドはそれ自身に時間引数の値を増加する可能性があるが値は減少しない。b_transport メソッドは wait が呼ばれる場合に時間引数の値を増加または減少させる事でシミュレーション時間を同期するが、それはプロセスが中断した時間に達する時間以下までである。このルールは SystemC シミュレーションにおいて時間が戻らない事と同様である。

e) 以下の説明では、transport 呼び出しにおけるトランザクションの受け取り側は呼び出し先関数であり、そして transport からの戻りにおけるトランザクションの受け取り側は呼び出し元関数である。

f) 受け取り側は、sc_time_stamp()+t（t は時間引数の値）の実効ローカル・タイムでトランザクションを受け取るように振舞うものとする。言い換えると、受け取り側はインタフェース・メソッド・コールに関連しているタイミング・ポイントが実効ローカル・タイムに起こるかのように振舞うものとする。

g) トランザクションが処理される事により一般の時間順番を増やす可能性のある実効ローカル・タイムの transport を呼び出すシーケンスが与えられる。異なったイニシエータによって作成されたトランザクションにおいて、それはインタフェース・メソッド・コールの順番が実効ローカル・タイムの順番と異なるかもしれないテンポラル・デカップリングに必要である。アウト・オブ・オーダーのタイミング・アノテーション付きトランザクションをハンドルする為の責任は受け取り側が持つ。

h) 非ゼロのタイミング・アノテーションと共にトランザクションを受け取る上で、どのような受け取り側もいつもスピードと精度間におけるトレードオフをモデリングに反映させる選択肢を持つ。受け取り側はトランザクションにより引き起こされたどんな状態変更もすぐに実行することが出来、タイミング・アノテーションを渡し、あるいは増加することができるか、または何らかの内部プロセスがアノテートされた時間の一部またはすべてが経過した後に再開して、その時だけ状態変更の実行をスケジューリングすることができる。その選択はトランスポート・インタフェースによって実施されるのでは無く、プロトコル・トレイツ・クラスや、コーディング・スタイルの部分に記載される。

i) もしタイミングの精度、またはそれらタイミング・アノテーションによって与えられた順番に来ているトランザクションのシーケンスを処理する事が受け取り側に関係がない場合、それは遅延を除いて各トランザクションを即座に処理出来る。またその場合受け取り側は、トランザクションを処理するのに必要な時間をモデルする為の、タイミング・アノテーションの値を増やす事を選択出来る。このシナリオは、システム設計が正しいイベントの連続性を実施する為の明確的なメカニズム（上記の TLM-2.0 インタフェースの上で）が存在する為、アウト･オブ･オーダー実行を許容出来るとみなせる。

j) もし受け取り側がタイミング並びに実行順番に正確なモデルを実装されるのならば、それはトランザクションが相互作用の可能性があるその他のSystemCプロセスに基づいた正しい時間に処理される事を確実にするべきである。SystemC における将来の時間に起きるイベントのスケジュールを適切に行うメカニズムは時間イベント通知である。容易性を考慮し、TLM-2.0 では SystemC の通常のセマンティクス（「12.3 ペイロード・イベント・キュー」を参照）に従った適切なシミュレーション時間に処理する為、トランザクションをキューイング出来るペイロードのイベント・キューを知るユーティリティ・クラスのファミリを提供する。言い換えると、アプロキシメイトリー・タイムドの受け取り側は、ペイロードのイベント・キューに通常トランザクションを置くべきである。

k) 直接トランザクションを処理するよりむしろ、受け取り側はトランザクションの変更と、同一のフェーズとタイミング・アノテーション（または増加されたタイミング・アノテーション）を使用している事を除き、更なる transport メソッドからの呼び出し、またはリターンと共にトランザクションを渡しても良い。

l) ルーズリー・タイムド・コーディング・スタイルでは、トランザクションが通常直ちに実行され、実行順はインタフェース・メソッド・コールの順と一致するので、b_transport メソッドが推薦される。

m) アプロキシメイトリー・タイムド・コーディング・スタイルでは、トランザクションは通常遅延し、実行順は実効ローカル・タイムの順と一致するので、nb_transport メソッドが推薦される。

n) 各コンポーネントは呼び出しごとのベースでダイナミックに上記の選択をすることができる。例えば、ルーズリー・タイムドのコンポーネントは呼び出し順の一連のトランザクションを直ちに実行し、タイミング・アノテーションを渡すかもしれないが、タイミング・アノテーションで与えられた遅延が経過した（オンデマンド同期として知られている）後にだけ、実行のためのまさしく次のトランザクションのスケジュールする事を選ぶかもしれない。これはコーディング・スタイルの問題である。

o) 上記の選択はブロッキング及びノンブロッキング・トランスポートの両方のために存在している。例えば、b_transport は、タイミング・アノテーションを増加させすぐに戻るか、もしくは戻る前にタイミング・アノテーションが経過するのを wait するかもしれない。nb_transport は、タイミング・アノテーションを増加させ TLM_COMPLETED を返すか、もしくは TLM_ACCEPTED を返して後で実行するためにトランザクションのスケジュールをするかもしれない。

p) 上記の規則の結果として、もしコンポーネントが実効ローカル・タイムの順番と異なってインタフェース・メソッド・コールの順番のトランザクションのシリーズの受け取り側ならば、コンポーネントは自由にそれらの特定のトランザクションの互いの実行順を選ぶことができる。推奨は呼び出し順に全てのトランザクションを実行するか、または全てのトランザクションを実効ローカル・タイムの順で実行することだが、これは義務ではない。

q) 事実上、受信するトランザクションがコンポーネントによって実行される順番がインタフェース・メソッド・コールの順番といつも同じであるべきであることに注意する必要がある。なぜならば、コンポーネントがタイミング・アノテーションにかかわらずインタフェース・メソッド・コールから戻る前に受信したトランザクションを実行するか（ルーズリー・タイムド）、または適切な将来の時間に実行のためにトランザクションのスケジュールし

TLM_ACCEPTED を返すか（アプロキシメイトリー・タイムド）のどちらかであり、従って次のトランザクションを発行する前にそれが待つべきである事を呼び出し元関数に示すだろう。（TLM_ACCEPTED だけが次のトランザクションの発行から呼び出し元関数を禁止しないが、基本プロトコルの場合では、リクエストとレスポンスの排他規則はそうするかもしれない。）

r) またタイミング・アノテーションについては、テンポラル・デカップリングの観点でも説明される。ゼロでないタイミング・アノテーションは、“タイム・ワープ”が受け取り側に要請されたと見なす事が出来る。受け取り側は、タイム・ワープに入るか、後処理と譲渡の為にキューにトランザクションを置く事か選択する事が出来る。ルーズリー・タイムド・モデルにおいて、タイム・ワープは一般的に有効である。もう一方では、もしターゲットが他の非同期イベントに依存関係を持つ場合、そのターゲットは確実に先のトランザクションの状態を予測する事が出来るまでは、シミュレーション時間の更新を待たなければならない可能性がある。

s) テンポラル・デカップリングの説明については、「3.3.2 ルーズリー・タイムド・コーディング・スタイルとテンポラル・デカップリング」を参照。

t) クォンタムの説明については、「12.2 クォンタム・キーパー」を参照。

例

以下の中間コードの断片は多くの可能なコーディング・スタイルについてのほんのいくつかを説明するものである:

```
// ----------------------------------------------
// Various interface method definitions
// ----------------------------------------------

void b_transport( TRANS& trans, sc_core::sc_time& t )
{
  // Loosely-timed coding style
  execute_transaction( trans );
  t = t + latency;
}

void b_transport( TRANS& trans, sc_core::sc_time& t )
{
  // Loosely-timed with synchronization at the target or synchronization-on-demand
  wait( t );
  execute_transaction( trans );
  t = SC_ZERO_TIME;
}

tlm_sync_enum nb_transport_fw( TRANS& trans, PHASE& phase, sc_core::sc_time& t )
```

```
{
   // Pseudo-loosely-timed coding style using non-blocking transport (not recommended)
   execute_transaction( trans );
   t = t + latency;
   return TLM_COMPLETED;
}

tlm_sync_enum nb_transport_fw( TRANS& trans, PHASE& phase, sc_core::sc_time& t )
{
   // Approximately-timed coding style
   // Post the transaction into a payload event queue for execution at time sc_time_stamp() + t
   payload_event_queue->notify( trans, phase, t );
   // Increment the transaction reference count
   trans.acquire();
   return TLM_ACCEPTED;
}

tlm_sync_enum nb_transport_fw( TRANS& trans, PHASE& phase, sc_core::sc_time& t )
{
   // Approximately-timed coding style making use of the backward path
   payload_event_queue->notify( trans, phase, t );
   trans.acquire();
   // Modify the phase and time arguments
   phase = END_REQ;
   t = t + accept_delay;
   return TLM_UPDATED;
}

// ------------------------------------------------------------------------
// b_transport interface method calls, loosely-timed coding style
// ------------------------------------------------------------------------

initialize_transaction( T1 );
socket->b_transport( T1, t );                                   // t may increase
process_response( T1 );

initialize_transaction( T2 );
socket->b_transport( T2, t );                                   // t may increase
process_response( T2 );

// Initiator may sync after each transaction or after a series of transactions
```

```
quantum_keeper->set( t );
if ( quantum_keeper->need_sync() )
   quantum_keeper->sync();

// ---------------------------------------------------------------------------
// nb_transport interface method call, approximately-timed coding style
// ---------------------------------------------------------------------------

initialize_transaction( T3 );
status = socket->nb_transport_fw( T3, phase, t );
if ( status == TLM_ACCEPTED )
{
   // No action, but expect an incoming nb_transport_bw method call
}
else if ( status == TLM_UPDATED )                    // Backward path is being used
{
   payload_event_queue->notify( T3, phase, t );
}
else if ( status == TLM_COMPLETED )                  // Early completion
{
   // Timing annotation may be taken into account in one of several ways
   // Either (1) by waiting, as shown here
   wait ( t );
   process_response( T3 );
   // or (2) by creating an event notification
   // response_event.notify( t );
   // or (3) by being passed on to the next transport method call (code not shown here)
}
```

4.1.4 TLM-1 からのマイグレーション・パス

旧 TLM-1 と新 TLM-2.0 インタフェースはどちらも TLM-2.0 標準の一部である。TLM-1 ブロッキングとノンブロッキング・インタフェースは現在も使用出来る。例えばベンダの何社かは、HDL 設計向けに機能検証環境の構築にこれらインタフェースを使用している。新旧ブロッキング・トランスポート・インタフェース間の類似性は、TLM-1 インタフェースを使用した古いモデルと、新しいTLM-2.0インタフェース間におけるアダプタを構築する作業を容易にする事を意図している。

4.2　ダイレクト・メモリ・インタフェース

4.2.1　イントロダクション

ダイレクト・メモリ･インタフェース（DMI）は、イニシエータがターゲット内のメモリ領域に直接アクセスできる手段を提供する。つまりトランスポート・インタフェースを使わずに直接ポインタを使ってメモリアクセスが可能である。DMI によって、イニシエータからインターコネクト･コンポーネントに接続されたターゲットへの複数の b_transport 関数や nb_transport 関数の呼び出しをバイパスできるため、イニシエータとターゲット間のメモリアクセスを高速化することが可能となる。

ダイレクト・メモリ・インタフェースには 2 つのインタフェースがあり、ひとつはイニシエータからターゲットへのフォワード・パス呼び出しともうひとつはターゲットからイニシエータへのバックワード･パス呼び出しである。フォワード･パスは、ある指定されたアドレスに対する DMI アクセス（読み込みまたは書き込み）の特定モードを要求するために利用され、DMI 領域の境界を含む tlm_dmi 型の DMI デスクリプタへのリファレンスを返す。バックワード・パスは、フォワード･パスによって確立されたＤＭＩポインタを無効にするためにターゲットから呼び出される。フォワード･パスとバックワード･パスはゼロ、ひとつ、または複数のインターコネクト・コンポーネントを通ってもよいが、同じソケットを通る対応するトランスポート呼び出しと同一であるべきである。

DMI ポインタは、フォワード・パスにそってトランザクションを通過することによって要求される。デフォルトの DMI トランザクション型は、tlm_generic_payload であり、トランザクション･オブジェクトのコマンドとアドレス・アトリビュートのみ使用する。DMI は、トランスポート・インタフェースの拡張と同じ仕組みである。つまり、DMI 要求には、無視可能または必須拡張があってもよいが、すべての無視不可能拡張のときには、新しいプロトコル・トレイツ・クラス定義する必要がある。（tlm_generic_payload の型定義が記載されている「7.2.2tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義」を参照）

DMI デスクリプタは、イニシエータで消費されるレイテンシ値を返し、その結果ルーズリー・タイムド・モデリングに十分なタイミング精度を確保することができる。

DMI ポインタはデバッグ用途にも使えるが、通常デバッグのトラフィックよりもメモリアクセスのトラフィックの方が支配的であるため、デバッグ用途では、DMI ではなくデバッグ・トランザクション・インタフェースで十分である。DMI ポインタをデバッグ用に使う場合、レイテンシ値は無視すべきである。

4.2.2　クラス定義

```
namespace tlm {

  class tlm_dmi
  {
  public:
    tlm_dmi() { init(); }
    void init();
```

```
    enum dmi_access_e {
      DMI_ACCESS_NONE = 0x00,
      DMI_ACCESS_READ = 0x01,
      DMI_ACCESS_WRITE = 0x02,
      DMI_ACCESS_READ_WRITE = DMI_ACCESS_READ | DMI_ACCESS_WRITE
    };
    unsigned char* get_dmi_ptr() const;
    sc_dt::uint64 get_start_address() const;
    sc_dt::uint64 get_end_address() const;
    sc_core::sc_time get_read_latency() const;
    sc_core::sc_time get_write_latency() const;
    dmi_access_e get_granted_access() const;
    bool is_none_allowed() const;
    bool is_read_allowed() const;
    bool is_write_allowed() const;
    bool is_read_write_allowed() const;

    void set_dmi_ptr(unsigned char* p);
    void set_start_address(sc_dt::uint64 addr);
    void set_end_address(sc_dt::uint64 addr);
    void set_read_latency(sc_core::sc_time t);
    void set_write_latency(sc_core::sc_time t);
    void set_granted_access(dmi_access_e t);
    void allow_none();
    void allow_read();
    void allow_write();
    void allow_read_write();
};

template <typename TRANS = tlm_generic_payload>
class tlm_fw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
    virtual bool get_direct_mem_ptr(TRANS& trans, tlm_dmi& dmi_data) = 0;
};
class tlm_bw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range) = 0;
};
```

```
} // namespace tlm
```

4.2.3 get_direct_mem_ptrメソッド

a) get_direct_mem_ptr メソッドは、ターゲットではなく、イニシエータまたはインターコネクト・コンポーネントからのみ呼び出されるべきである。

b) trans 引数には、イニシエータで生成された DMI トランザクション・オブジェクトへの参照値を渡す。

c) dmi_data 引数には、イニシエータで生成された DMI デスクリプタへの参照値を渡す。

d) すべてのインターコネクト・コンポーネントでは、get_direct_mem_ptr 呼び出しをイニシエータからターゲットへのフォワード・パスに沿って通過させる必要がある。別の言い方でいうと、インターコネクト・コンポーネントのターゲット・ソケットで、get_direct_mem_ptr の実装は、イニシエータ・ソケットの get_direct_mem_ptr メソッドを呼び出してもよい。

e) それぞれの get_direct_mem_ptr 呼び出しは、イニシエータからターゲットへのトランスポート呼び出しの対応するセットと全く同一のパスを辿る。言い換えれば、それぞれの DMI 要求は、一つのイニシエータと一つのターゲット間の相互作用を含む。これら二つのコンポーネント（イニシエータとターゲット）もまた、トランスポート・インタフェースを通る一つのトランザクション・オブジェクトのイニシエータとターゲットとしての役割をする。DMI は、重要なビット幅変換器のような、二番めのトランザクション・オブジェクトを生成するようなコンポーネントを通るパス上では使うことができない。（もし DMI でシミュレーション速度が非常に要求される場合には、シミュレーション・モデルを再構成する必要があるかもしれない）

f) すべてのインターコネクト・コンポーネントは、フォワード方向での trans と dmi_data 引数を渡す必要がある。唯一許されている修正は、以下のように記述される DMI トランザクション・オブジェクトのアドレス・アトリビュート値である。トランザクションと DMI デスクリプタのアドレス・アトリビュートは両方とも get_direct_mem_ptr メソッドからのリターン時に修正されるかもしれない。つまり、ターゲットからイニシエータへ関数呼び出しが順にリターンされるときである。

g) もし、指定されたアドレスへの DMI アクセスをターゲットがサポートしていれば、以下に記述するように、DMI デスクリプタのメンバをセットし、関数の返り値として true をセットする必要がある。ターゲットが DMI アクセスを許可するときは、DMI デスクリプタは許可されたアクセスの詳細を示すために使用される。

h) もし、指定されたアドレスへの DMI アクセスをターゲットがサポートしていなければ、以下に記述するように、アドレス範囲と DMI デスクリプタの型メンバのみセットし、関数の返り値として false をセットする必要がある。ターゲットが DMI アクセスを拒否するときは、DMI デスクリプタは拒否されたアクセスの詳細を示すために使用される。

i) ターゲットは、この節で与えられるルールに従って、メモリ領域（不連続な領域含む）への DMI アクセスを許可または拒否してもよい。

j) ターゲットが DMI アクセスを許可し、関数の返り値に true をセットする場合、インターコネクト・コンポーネントは、get_direct_mem_ptr メソッドから返ってきたときに関数の返り値に false をセットすることによって DMI アクセスを拒否してもよい。その逆は許可されてい

ない。ターゲットが DMI アクセスを拒否した場合、インターコネクト・コンポーネントは DMI アクセスを許可するべきではない。

k) get_direct_mem_ptr を複数回呼び出した場合、ターゲットは同じメモリ領域、同じ時刻に DMI アクセスを複数のイニシエータに許可するかもしれない。アプリケーションは同期とデータの一貫性を保証する必要がある。

l) それぞれの get_direct_mem_ptr の呼び出しは隣り合うメモリ領域への単一 DMI ポインタのみを返すので、実際にはそれぞれの DMI 要求は一つのターゲットでのみ実行可能である。別の言い方をすれば、もしメモリ領域が複数のターゲットに分散している場合、例えアドレス範囲が隣りあっていても、それぞれのターゲットは分割された DMI リクエストを要求するだろう。

m) あるメモリ領域への読み込み又は書き込みアクセスが、ターゲットに副作用（つまり、メモリの状態だけではなく、ターゲットの状態にまで変化を与えること）が発生する場合は、ターゲットはメモリ領域への与えられた型の DMI アクセスを許可すべきではない。しかし、例えば、書き込みによってのみターゲットに副作用が発生するなら、ターゲットは与えられた領域への読み込みの DMI アクセスは許可してもよい。

n) get_direct_mem_ptr は、invalidate_direct_mem_ptr 呼び出しを実装してもよい。

o) get_direct_mem_ptr は、直接的または間接的に wait 読み出しを実装してはいけない。

4.2.4 テンプレート引数とtlm_generic_payloadクラス

a) tlm_fw_direct_mem_if テンプレートは DMI トランザクション・クラスの型でパラメタライズされる。

b) トランザクション・オブジェクトはダイレクト・メモリ・アクセスが要求するアドレスを指し示すアトリビュート及び、要求されるアクセスの型、つまり指定されたアドレスに対する読み込みアクセスか、書き込みアクセスかを含む。基本プロトコルの場合、これらは汎用ペイロードのコマンドとアドレスのアトリビュートである。

c) TRANS テンプレートの引数のデフォルト値は tlm_generic_payload クラスである。

d) 相互利用性を最大化するために、DMI トランザクション・クラスは tlm_generic_payload にすべきである。無視不可能拡張または他のトランザクション型を使うことは相互利用性を制限することになる。

e) イニシエータは DMI トランザクション・オブジェクトを生成及び管理する必要があり、また get_direct_mem_ptr への引数として渡される前に適切なアトリビュートを設定する必要がある。

f) トランザクション・オブジェクトのコマンド・アトリビュートは、要求されている DMI アクセスの種類を示すために、イニシエータによって設定され、インターコネクト・コンポーネントまたはターゲットによって修正されるべきでない。基本プロトコルでは、許される値は読み込みアクセスの TLM_READ_COMMAND と書き込みアクセスの TLM_WRITE_COMMAND である。

g) 基本プロトコルでは、コマンド・アトリビュートの値に TLM_IGNORE_COMMAND は禁止されている。しかし、この値は他のプロトコルでは使用してよい。

h) トランザクション・オブジェクトのアドレス・アトリビュートは、ダイレクト・メモリ・アクセスが要求しているアドレスを指し示すために、イニシエータによって設定される。

i) フォワード・パスに沿って DMI トランザクション・オブジェクトが通過するインターコネクト・コンポーネントは、同じソケットの対応するトランスポート・インタフェースにトランザクションのアドレス・アトリビュートを、デコード及び、必要に応じて修正すべきである。例えば、インターコネクト・コンポーネントによって、ターゲットのアドレス幅やシステム・メモリ・マップにおける場所に応じて、アドレスをマスク（上位ビットの数を削減）する必要がある。

j) インターコネクト・コンポーネントは,アドレスエラーを検知した場合、get_direct_mem_ptr 呼び出しを通過させる必要はない。

k) 基本プロトコルの場合、イニシエータは、コマンドとアドレスを除いて汎用ペイロードのアトリビュートを設定する必要はない。また、ターゲットとどんなインターコネクト・コンポーネントでも、他のすべてのアトリビュートを無視してもよい。特に、応答ステータスのアトリビュートや許可されている DMI アトリビュートは無視してもよい。許可されている DMI アトリビュートは、トランスポート・インタフェースとともに利用することだけを想定している。

l) イニシエータは、一つの DMI 呼び出しから次の DMI 呼び出し、さらにはトランスポート・インタフェース、デバッグ・トランスポート・インタフェースの呼び出しを含めて、トランザクション・オブジェクトを再利用しても良い。

m) もしアプリケーションが、ターゲットが DMI アクセスの種類を判別する目的で、DMI トランザクション・オブジェクトにさらにアトリビュートを追加する必要があるならば、無関係なトランザクション・クラスに置き換えるよりも、汎用ペイロードを拡張することを推奨する。例えば、DMI トランザクションは、CPU の種類に応じて異なった DMI 要求をターゲットが処理するために、CPU Id が必要かもしれない。そのような拡張が無視不可能な場合は、新しいプロトコル・トレイツ・クラスを定義する必要がある。

4.2.5 tlm_dmiクラス

a) DMI デスクリプタは tlm_dmi クラスのオブジェクトである。DMI デスクリプタは、イニシエータで生成されるが、そのメンバはインターコネクト・コンポーネントまたはターゲットで設定される。

b) DMI デスクリプタは、次のアトリビュートを持つ：DMI ポインタ・アトリビュート、許可されたアクセス型アトリビュート、開始アドレス・アトリビュート、終了アドレス・アトリビュート、読み込みレイテンシ・アトリビュート、そして書き込みレイテンシ・アトリビュート。これらアトリビュートのデフォルト値：DMI pointer attribute = 0,access type = DMI_ACCESS_NONE, start address = 0, end address = the maximum value of type sc_dt::uint64, read latency = SC_ZERO_TIME, and write latency = SC_ZERO_TIME

c) init メソッドは、DMI 拡張子のメンバをデフォルト値に初期化する。

d) DMI 拡張子は、イニシエータによって get_direct_mem_ptr への引数として渡されるときには必ず、デフォルト状態でなければならない。もし DMI 拡張子オブジェクトが蓄積されると、イニシエータは get_direct_mem_ptr への引数として渡される前に、DMI 拡張子をそのデフォ

ルト状態にリセットしなければならない。init メソッドはこの目的のために呼び出される。

e) インターコネクト・コンポーネントは DMI デスクリプタをターゲット方向に渡される際に修正してはいけないので、DMI デスクリプタはターゲットで受け取ったときに初期状態にあるべき。

f) set_dmi_ptr メソッドは DMI ポインタ・アトリビュートを引数で渡される値に設定される。get_dmi_ptr メソッドは DMI ポインタ・アトリビュートの現在の値を返す。

g) DMI ポインタ・アトリビュートは開始アドレス・アトリビュートの値に対応するストレージ場所を示すためにターゲットによって設定される。これは get_direct_mem_ptr 呼び出しで要求されるアドレス以下でなければならない。初期値は 0 である。

h) DMI 領域のストレージは unsigned char*の型で表現される。ストレージは汎用ペイロードのデータ配列と構成を持つ。もしターゲットがその構成のメモリ領域へのポインタを返すことが出来ない場合は、ターゲットは DMI をサポートできず、get_direct_mem_ptr は false 値を返すべきである。TLM-2.0 におけるメモリ構成とエンディアンの扱い方の詳細は、「7.17 エンディアン」を参照。

i) インターコネクト・コンポーネントは、get_direct_mem_ptr 関数呼び出しからのリターン・パスにおいて DMI アクセスを許可する領域を制限するために、DMI ポインタ・アトリビュートを修正することが許されている。

j) set_granted_access メソッドは、許可されたアクセス型アトリビュートに引数で渡された値を設定する。get_granted_access メソッドは、許可されたアクセス型アトリビュートの現在の値を返す。初期値は DMI_ACCESS_NONE でなければならない。

k) allow_none、allow_read、allow_write そして allow_read_write メソッドは、許可されたアクセス型アトリビュートをそれぞれ DMI_ACCESS_NONE 、 DMI_ACCESS_READ 、DMI_ACCESS_WRITE、または DMI_ACCESS_READ_WRITE の値に設定する。

l) is_none_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_NONE の値の場合及びその場合のみ真の値を返す。is_read_allowed メソッドは、 許可されたアクセス型アトリビュートが DMI_ACCESS_READ または DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。is_write_allowed メソッドは、 許可されたアクセス型アトリビュートが DMI_ACCESS_WRITE または DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。is_read_write_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。

m) ターゲットは、許可されたまたは拒否されたアクセス型アトリビュートに、許可されたアクセスの型を設定する。ターゲットは、読み込みまたは読み込み／書き込みアクセスが許可される（または拒否される）ことよって読み込みアクセスの要求に答えることができ、同様に書き込みまたは読み込み／書き込みアクセスが許可される（または拒否される）ことよって書き込みアクセスの要求に答えることができる。インターコネクト・コンポーネントは、get_direct_mem_ptr 関数呼び出しのリターン・パスにおいて、DMI_ACCESS_READ_WRITE の値を DMI_ACCESS_READ または DMI_ACCESS_WRITE で上書きすることによって許可されたアクセスを制限ことができる。

n) DMI 領域への読み込み･書き込みアクセスを拒否したいターゲットは､DMI_ACCESS_NONE ではなく、DMI_ACESS_READ_WRITE にアクセス型を許可すべきである。

例）

```
bool get_direct_mem_ptr( TRANS& trans, tlm::tlm_dmi& dmi_data )
{
// Deny DMI access to entire memory region
dmi_data.allow_read_write();
dmi_data.set_start_address( 0x0 );
dmi_data.set_end_address( (sc_dt::uint64)-1 );
return false;
}
```

o) ターゲットは、次の場合には DMI_ACCESS_NONE を許可されたアクセス型に設定すべきである。すなわち、イニシエータへの読み込みや書き込みまたは読み込み／書き込みアクセスが許可されていない（または拒否されていない）が、その一方で、DMI トランザクション・オブジェクトへの拡張によって要求されるようないくつか他の種類のアクセスを許可している（または拒否している）ことを示す場合である。この値は、DMI トランザクション・オブジェクトの拡張によって事前に定義されたアクセス型（読み込み、書き込みと読み込み／書き込み）が不要または意味がなくなる場合にのみ利用されるべきである。この値は基本プロトコルの場合に利用されるべきではない。

p) イニシエータは、許可されたアクセス型アトリビュートを使ってターゲットによって許可されている（または基本プロトコルとは別に、汎用ペイロードの拡張やまたほかの DMI トランザクション型を使って許可されている）DMI アクセスのこれらのモード（インターコネクトによって修正される可能性はあるが）のみを利用する責任がある。

q) set_start_address と set_end_address メソッドは、開始と終了アドレス・アトリビュートをそれぞれ引数で渡される値を設定する。get_start_address と get_end_address メソッドは、開始と終了アドレス・アトリビュートの現在の値をそれぞれ返す。

r) 開始と終了アドレス・アトリビュートは DMI 領域における最初と最後のバイトのアドレスを示すためにターゲットによって設定（またはインターコネクトによって修正）される。DMI 領域が許可されているか拒否されているかは、get_direct_mem_ptr メソッド返り値（true または false）によって決まる。全メモリ領域へのアクセスを拒否したいターゲットは、start_address に 0 を end_address に sc_dt::uint64 型の最大値を設定してもよい。

s) ターゲットは、それぞれの get_direct_mem_ptr 呼び出しに対して一つの隣接するメモリ領域を許可または拒絶のみできる。ターゲットは、開始と終了アドレス・アトリビュートを同じにすることによってメモリ領域を一つのアドレスに設定することができ、DMI 領域を任意の大きさに設定することもできる。

t) 指定された領域への指定された型の DMI アクセスが許可されているとき、イニシエータはその領域が無効になるまでその領域のどこでも指定された型のアクセスを実行するかもしれない。別の言い方でいうと、DMI 要求によって指定されたアドレスはアクセス制限されない。

u) get_direct_mem_ptr 呼び出しが通過するいかなるインターコネクト・コンポーネントでも、アドレス引数を変換するように開始と終了アドレス・アトリビュートを変換する必要がある。DMI デスクリプタにおけるアドレスのいかなる変換も、デスクリプタが get_direct_mem_ptr

関数呼び出しのリターン・パスを通過するときに実行される。例えば、ターゲットは開始アドレス・アトリビュートをそのターゲットが知っているメモリマップ内での相対アドレスに設定する。その場合インターコネクト・コンポーネントは相対アドレスをシステム上のメモリマップにおける絶対アドレスに変換する必要がある。初期値はそれぞれ 0 と sc_dt::uint64 の取り得る最大値である。

v) インターコネクト・コンポーネントは、DMI アクセスが許可されている領域を制限するため、または DMI アクセスが許可されていない領域を拡張するために、開始と終了アドレス・アトリビュートを修正することができる。

w) もし get_direct_mem_ptr が true を返す場合は、開始と終了アドレス・アトリビュートで示される DMI 領域が DMI アクセスできる領域である。もし get_direct_mem_ptr が false を返す場合は、DMI アクセスできない領域である。

x) ターゲットまたはインターコネクト・コンポーネントが二つまたはそれ以上の get_direct_mem_ptr 呼び出しを受け取る場合は、二つまたはそれ以上の重なり合う DMI アクセス許可領域、または二つまたはそれ以上の重なり合う DMI アクセス禁止領域を返すかもしれない。

y) ターゲットまたはインターコネクト・コンポーネントは、(最初の領域を無効にするために invalidate_direct_mem_ptr を途中で呼び出さない限り)、重なりあう領域に対し、一方が許可され、もう一方が禁止されるような DMI 領域は返さない。例えば、どちらも読み込み及び読み込み／書き込みの両方であるか、書き込み及び読み込み／書き込みの両方である。

z) 別の言葉で言えば、DMI 領域の定義は invalidate_direct_mem_ptr を途中呼び出しによって最初の領域が無効化されない限り、生成された順番に依存しない。特に禁止 DMI 領域の生成は、同じアクセス型ですでに許可された DMI 領域に穴をあけることができない。逆もまた真。

aa) ターゲットは DMI アクセスをすべてのアドレス領域（開始アドレスを 0、終了アドレスを最大値）で禁止してもよい。多分ターゲットは DMI アクセスを全くサポートしないからだろう。その場合、インターコネクト・コンポーネントはこの禁止領域をターゲットが占有しているメモリマップの領域にとどめるべきである。もしインターコネクト・コンポーネントがアドレス領域をとどめることに失敗したら、イニシエータはすべてのシステム上のアドレス空間のすべてが DMI 禁止されていると勘違いをするだろう。

bb) set_read_latency と set_write_latency メソッドは、それぞれ読み込み及び書き込みレイテンシ・アトリビュートに引数で渡された値を設定する。get_read_latency と get_write_latency メソッドは、それぞれ読み込み及び書き込みレイテンシ・アトリビュートの現在の値を返す。

cc) 読み込み及び書き込みレイテンシ・アトリビュートは、それぞれ読み込み及び書き込みのメモリ・トランザクションにかかるバイトあたりの平均レイテンシを設定する。言い換えれば、直接的なメモリ操作を実行するイニシエータが等価な transport トランザクションによって転送されるバイト数と DMI 拡張子からの読み込みまたは書き込みレイテンシとを掛け合わせることによって実際のレイテンシを計算しなければならない。初期値は SC_ZERO_TIME。インターコネクト・コンポーネントとターゲットの両方が読み込み、書き込みのどちらかのレイテンシの値を増やし、その結果 DMI デスクリプタが get_direct_mem_ptr メソッドからのリターン・パスにおいてイニシエータからターゲットへ戻るときにレイテンシが積算される。

許可されたアクセス型アトリビュートの値によって、一つまたは両方のレイテンシが有効になる。

dd) イニシエータはダイレクト・メモリ・ポインタを使ってメモリにアクセスするときはいつでも、レイテンシを考慮する必要がある。もしイニシエータがレイテンシを無視すれば、タイミングが不正確になる。

4.2.6 invalidate_direct_mem_ptrメソッド

a) invalidate_direct_mem_ptr メソッドはターゲットまたはインターコネクト・コンポーネントだけによって呼び出される。

b) ターゲットは存在する DMI 領域の有効性やアクセス型を修正する変更の前に invalidate_direct_mem_ptr メソッドを呼び出す必要がある。例えば、存在する DMI 領域のアドレス領域を制限する前や、アクセス型を読み込み／書き込みから読み込みに変更する前や、アドレス空間をリマップする前などである。

c) start_range と end_range の引数は、DMI 領域が無効化されるアドレス範囲の最初と最後のアドレスを示す。

d) イニシエータが invalidate_direct_mem_ptr を受け取ったらすぐに指定されたアドレス範囲と重なり合う DMI 領域（get_direct_mem_ptr で事前に獲得していたもの）を無効にして廃棄する。

e) 部分的に重なり合う場合、つまり存在する DMI 領域の一部分だけ無効な場合には、イニシエータは存在する領域の境界を調整するか、または全体の領域を無効化する。

f) 各 DMI 領域は、ターゲットが invalidate_direct_mem_ptr 呼び出しで明示的に無効化するまでは有効のままである。各イニシエータは、有効な DMI 領域の管理表を管理し、無効化されるまで各領域を利用し続ける。

g) すべてのインターコネクト･コンポーネントは、デコードや必要に応じて対応するトランスポート・インタフェースのためにアドレス引数を修正するときに、invalidate_direct_mem_ptr 呼び出しをターゲットからイニシエータへのバックワード・パスに沿って通過させる義務がある。トランスポート・インタフェースがフォワード・パスのアドレスとバックワード・パスの DMI を変換するので、トランスポートと DMI 変換は相互に逆であるべきである。

h) もし一つの invalidate_direct_mem_ptr をターゲットから呼び出すならば、インターコネクト・コンポーネントは複数の invalidate_direct_mem_ptr をイニシエータへ呼び出してもよい。複数のイニシエータが存在しそれぞれが同じターゲットに対してダイレクト・メモリ・ポインタを得るかもしれないので、インターコネクト・コンポーネントがすべてのイニシエータに対して invalidate_direct_mem_ptr を呼び出すのが安全な実装方法である。

i) インターコネクト・コンポーネントは、start_range を 0、end_range を sc_dt::uint64 の最大値に設定することによって、イニシエータにあるすべてのダイレクト・メモリ・ポインタを無効化することができる。

j) TLM-2.0 コア・インタフェース・メソッドは、invalidate_direct_mem_ptr を呼び出す実装でもよい。

k) invalidate_direct_ptr は get_direct_mem_ptr を直接的または間接的に呼び出す実装ではない。

l) invalidate_direct_mem_ptr は wait を直接的または間接的に呼び出す実装ではない。

4.2.7 DMIとtransportの比較

a) 定義によれば、ダイレクト・メモリ・インタフェースによって、イニシエータとターゲット間を直接アクセスすることが可能である。このとき、インターコネクト・コンポーネントはバイパスされる。一方、トランスポート・インタフェースでは、インターコネクト・コンポーネントをバイパス不可能である。

b) インターコネクト・コンポーネントが状態を保持しているまたは副作用がある場合でも(例えばバッファ機能を有するインターコネクト・コンポーネントやキャッシュ・メモリをモデル化しているインターコネクトコネクト・コンポーネント)、正しく動作するように注意が必要である。ダイレクト・メモリ・インタフェースとは違い、トランスポート・インタフェースは、インターコネクト・コンポーネントの状態にアクセスしたり、アップデートしたりすることが可能である。ダイレクト・メモリ・インタフェースの場合は、インターコネクト・インタフェースをバイパスする。最も安全な方法は、このようなインターコネクト・コンポーネントの場合には、常に DMI アクセスを拒否することである。

c) イニシエータは、トランスポート・インタフェースを呼び出すことと、ダイレクト・メモリ・ポインタを使うことを、相互に切り替えることが可能である。あるイニシエータが、トランスポート・インタフェースを利用している間に、もう一方のイニシエータで DMI を使うことも可能である。トランスポート呼び出しがタイミング・アノテーションを転送している場合は、正しく振舞うように保証するように注意が必要である。これはアプリケーション側がケアする必要がある。例えば、あるターゲットが DMI とトランスポートを同時にサポートしている場合や、トランスポートが呼び出されている間にすべての DMI ポインタが無効化されている場合である。

4.2.8 DMIとテンポラル・デカップリング

a) ターゲットやインターコネクト・コンポーネントで invalidate_direct_mem_ptr を呼び出すことによってのみ、DMI 領域を無効化することができる。

b) イニシエータは、次の考慮すべき条件において DMI ポインタを使う前に DMI 領域がまだ有効かどうかをチェックする責任がある。

c) 一度イニシエータが起動されたら、そのイニシエータが制御を手放すまで、他の SystemC プロセスは実行できないことは、SystemC のコ・ルーチン・セマンティクスによって保証される。特に、他の SystemC プロセスは DMI ポインタを無効化できない（現プロセスは可能かもしれないが)。結果として、テンポラル・デカップリングされたイニシエータが、DMI ポインタを使う度に与えられ DMI 領域がまだ有効かどうかを繰り返し確認する必要は必ずしもない。

d) イニシエータに呼び出されるインタフェース・メソッド・コールによって、他のコンポーネントが invalidate_direct_mem_ptr を呼び出し、これによってそのイニシエータに利用されている DMI 領域を無効化する場合がある。これは、テンポラル・デカップリングされたイニシエータが制御を譲り渡さずに実行する場合に当てはまる。

e) イニシエータが、他のコンポーネントと通信を行わない、あるいは、制御を保持して実行される間は、有効な DMI 領域は有効なままである。

f) DMI を使うテンポラル・デカップリングされたイニシエータが制御をカーネルに渡した後、他のテンポラル・でカップリングされたイニシエータによって同じ DMI 領域が、現クォンタム時間内で無効化することがあり得る。これは、一般的にテンポラル・デカップリングに本来ある基本的な不正確さを表しているが、この節で与えているルールに違反するものではない。

4.2.9 DMIヒントを使った最適化

a) DMI ヒント、つまり DMI 許可アトリビュートは、DMI アクセスの繰り返しポーリングをなくすことによってシミュレーション・スピードを最適化するメカニズムである。DMI ポインタが使える状態か確認するために get_direct_mem_ptr を呼び出すかわりに、イニシエータがトランスポート・インタフェースを通過する通常のトランザクションの DMI 許可アトリビュートを確認することができる。

b) 汎用ペイロードによって DMI 許可アトリビュートは提供される。ユーザ定義のトランザクションによって同じメカニズムを実装できる。その場合ターゲットによって DMI 許可アトリビュートの値を適切に設定すべきである。

c) DMI 許可アトリビュートを使うかどうかは自由に選択できる。イニシエータは汎用ペイロードの DMI 許可アトリビュートを自由に無視できる。

d) イニシエータが DMI 許可アトリビュートを有効にするための推奨手順は下記に示す。

   i. イニシエータが有効な DMI 領域のキャッシュに対するアドレスを確認する。

   ii. もし DM ポインタが存在しないなら、イニシエータはトランスポート・インタフェースを通して通常のトランザクションを実行する。

   iii. その後、イニシエータはトランザクションの DMI 許可アトリビュートを確認する。

   iv. もしアトリビュートが DMI を許可していることを示していれば、イニシエータは get_direct_mem_ptr を呼び出す。

   v. イニシエータはその呼び出しからの戻り値により有効な DMI 領域のキャッシュを修正する。

4.3 デバッグ・トランスポート・インタフェース

4.3.1 イントロダクション

デバッグ・トランスポート・インタフェースは、ターゲットのストレージに対して読み込み及び書き込みを行うための手段を提供する。このとき、通常のトランスポート・インタフェースで使われるイニシエータからターゲットへのフォワード・パスを経由するが、遅延、wait、イベント通知、副作用は一切発生させない。言い換えるとデバッグ・トランスポート・インタフェースは邪魔にはならない。デバッグ・トランスポート・インタフェースは、トランスポート・インタフェースと同じパスを通るため、デバッグ・トランスポート・インタフェースの実装は通常のトランザクションと同じアドレス変換を実行する。

例えばデバッグ・トランスポート・インタフェースによって、ISS に接続されたソフトウェアデバッガが、シミュレーション上の CPU モデルからアクセス可能なシステム内のメモリデータを読み出したり、書き込んだりできる。またデバッグ・トランスポート・インタフェースによって、

イニシエータがシステムメモリの内容をシミュレーション中に解析目的でスナップショットを取ることや、エラボレーション終了時にシステム内のメモリのある領域を初期化することも可能である。

デフォルトのデバッグ・トランザクション型は tlm_generic_payload であり、使われているトランザクション・オブジェクトのコマンド、アドレス、データ長及びデータ・ポインタ・アトリビュートのみを利用する。デバッグ・トランザクションは、トランスポート・インタフェースにおける拡張と同じアプローチを取る。つまり、デバッグ・トランザクションは無視可能拡張を含むかもしれないが、無視不可能あるいは必須拡張の場合は新しいプロトコル・トレイツ・クラスを定義する必要がある（tlm_generic_payload の型定義の内容を含む「7.2.2 tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義」を参照）。

4.3.2 クラス定義

```
namespace tlm {

  template <typename TRANS = tlm_generic_payload>
  class tlm_transport_dbg_if : public virtual sc_core::sc_interface
  {
  public:
    virtual unsigned int transport_dbg(TRANS& trans) = 0;
  };

} // namespace tlm
```

4.3.3 TRANテンプレート引数とtlm_generic_payloadクラス

a) tlm_transport_dbg_if テンプレートはデバッグ・トランザクション・クラスの型でパラメタライズされている。

b) デバッグ・トランザクション・クラスには、デバッグ・アクセスに必要なコマンド、アドレス、データ長及びデータ・ポインタをターゲットに教えるためのアトリビュートが含まれる。基本プロトコルの場合、これらは汎用ペイロードの対応するアトリビュートである。

c) TRANS テンプレート引数のデフォルト値は tlm_generic_payload クラスである。

d) 相互利用性を最大化するために、デバッグ・トランザクション・クラスは tlm_generic_payload であるべき。無視不可能拡張や他のトランザクション型を使えば、相互利用性が制限される。

e) もしアプリケーションがデバッグ・トランザクションにさらにアトリビュートを追加する必要がある場合には、関連性のないトランザクション・クラスに置き換えるよりも汎用ペイロードを拡張する方法を推奨する。その拡張が無視不可能または必須の場合には、新しいプロトコル・トレイツ・クラスを定義する必要がある。

4.3.4 ルール

a) transport_dbg への呼び出しは、通常のトランザクションで利用されるトランスポート・インタフェースと同じフォワード･パスに従う。

b) trans 引数はデバッグ・トランザクション・オブジェクトへの参照渡しである。

c) イニシエータはデバッグ・トランザクション・オブジェクトを生成し、管理する必要がある。また transport_dbg への引数で渡す前にオブジェクトの適切なアトリビュートを設定する必要がある。

d) トランザクション・オブジェクトのコマンド・アトリビュートは要求されているデバッグ・アクセスの種類を示すためにイニシエータによって設定され、インターコネクト・コンポーネントやターゲットから修正されない。基本プロトコルで、許可されている値は、ターゲットへの読み込みアクセスの TLM_READ_COMMAND 、ターゲットと TLM_IGNORE_COMMAND への書き込みアクセスの TLM_WRITE_COMMAND である。

e) TLM_IGNORE_COMMAND と等しいコマンド・アトリビュートをもったトランザクションを受け取ったときは、ターゲットは読み込みや書き込みを実行すべきではないが、拡張されたデバッグ・トランザクションを実行する際の拡張を含む汎用ペイロードにおいてアトリビュートの値を使ってもよい。

f) トランスポート・トランザクションの場合のように、デバッグ・トランスポート・インタフェースを持つ無視不可能または必須の汎用ペイロードの拡張を使うためには、新しいプロトコル・トレイツ・クラスを定義する必要がある。

g) アドレス・アトリビュートはイニシエータによって読み込みまたは書き込みされる領域の最初のアドレスに設定される。

h) フォワード・パスに沿ってデバッグ・トランザクション・オブジェクトを通るインターコネクト・コンポーネントで、同じソケットの対応するトランスポート・インタフェースに、トランザクション・オブジェクトのアドレス・アトリビュートをデコード及び必要に応じて修正されるべきである。例えば、インターコネクト・コンポーネントで、ターゲットアドレス幅とシステム・メモリマッ上の場所に従ってアドレスをマスク（上位ビットを削減）する必要がある。

i) インターコネクト・コンポーネントでは、アドレッシング・エラーを検知した場合 transport_dbg 呼び出しを通過させる必要はない。

j) アドレス・アトリビュートは、もしデバッグ・ペイロードが複数のインターコネクト・コンポーネントを通って転送されれば、複数回修正されているかもしれない。デバッグ・ペイロードがイニシエータに戻ってきた時には、アドレス・アトリビュートのもともとの値は上書きされているかもしれない。

k) データ長アトリビュートはイニシエータによって読み込むまたは書き込むバイト数に設定され、インターコネクト・コンポーネンツやターゲットによって修正されない。データ長アトリビュートは、ターゲットがバイトを読み込みまたは書き込みしない場合は、0 でもよい。データ・ポインタ・アトリビュートは null でもよい。

l) データ・ポインタ・アトリビュートはイニシエータによってターゲットにコピーされる値（書き込み時）またはターゲットからコピーされる値（読み込み時）の配列のアドレスに設定され、インターコネクト・コンポーネントやターゲットによって修正されない。この配列はイニシエータによって割り当てられ、transport_dbg から戻ってくる前に削除されない。配列サイズは少なくともデータ長アトリビュートの値と同じである。データ長アトリビュートが 0 なら、データ・ポインタ・アトリビュートは null ポインタでもよく、配列をアロケートする

必要はない。

m) もし可能であれば、ターゲットにおける transport_dbg の実装は、指定されたアドレスを使って（インターコネクトを通してアドレス変換後に）指定されたバイト数を読み込みまたは書き込むようにする。書き込みコマンドの場合、ターゲットはデータ配列のコンテンツを修正してはいけない。

n) トランスポート・インタフェースと一緒に利用されたときに、データ配列は汎用ペイロードのデータ配列と同じ構成を持つ。transport_dbg の実装は、ターゲット内のローカル・データ・ストレージの構成と汎用ペイロードの構成間で変換する必要がある。

o) 基本プロトコルの場合、イニシエータはコマンド、アドレス、データ長及びデータ・ポインタ以外の汎用ペイロードのアトリビュートを設定する必要はない。ターゲットとインターコネクト・コンポーネントは他のすべてのアトリビュートを無視してもよい。特に、応答ステータス・アトリビュートは無視される。

p) イニシエータは一つの呼び出しから次の呼び出し、デバッグ・トランスポート・インタフェース、トランスポート・インタフェース及び DMI に渡る呼び出しでトランザクション・オブジェクトを再利用してもよい。

q) transport_dbg は、実際に読み込みまたは書き込みしたバイト数の回数を返す。それはデータ長アトリビュートの値よりも少ない。もしターゲットが実行できない時には、0 の値を返す。

r) 直接的または間接的に transport_dbg は、wait を呼んではいけない。デバッグ書き込みコマンドを実行後するに反映される場合を置いておいて、インターコネクト・コンポーネントやターゲットの状態を修正してはいけない。

5. グローバル・クォンタム

5.1 イントロダクション

テンポラル・デカップリング機能は、SystemC のプロセスに対して、クォンタムと呼ばれる時間を実際のシミュレーション時刻より先に進めるもので、ルーズリー・タイムドコーディング・スタイルと関連付けて用いられる。テンポラル・デカップリング機能を用いると、スレッドのコンテクスト・スイッチとイベントを減らすことにより非常に高速なシミュレーション速度が実現できる。

テンポラル・デカップリングを行うプロセス間で明示的な同期を行う上で、クォンタム時間の使用は、厳密には必須ではない。プロセスは次の同期ポイントまで先行的に実行可能であるが、クォンタム時間を必要とするプロセスは、グローバル・クォンタムを使用すべきである。

テンポラル・デカップリング機能を用いる時には、b_transport()および nb_transport()メソッドの引数によって渡される時間は、通常、sc_time_stamp()の戻り値である現在のシミュレーション時刻とクォンタム境界からの相対時間として定義される。グローバル・クォンタムは、連続するクォンタム境界間のデフォルト時間間隔である。グローバル・クォンタム時間の値は、唯一の tlm_global_quantum クラスによって管理される。各プロセスはグローバル・クォンタム時間を使うべきだが、1 つのプロセスは、自身のローカル・クォンタム時間の計算を行うことだけが許可されている。

テンポラル・デカップリングの詳細については、「3.3.2 ルーズリー・タイムド・コーディング・スタイルとテンポラル・デカップリング」も参照のこと。

タイミング・アノテーションに関しては、「4.1.3 トランスポート・インタフェースにおけるタイミング・アノテーション」も参照のこと。

tlm_quantumkeeper ユーティリティ・クラスには、クォンタム時間を管理するためのメソッドが用意されている。クォンタム・キーパーの使い方に関しては、「12.2 クォンタム・キーパー」を参照のこと。

5.2 ヘッダファイル

グローバル・クォンタムのクラス定義は、tlm.h ヘッダファイルにある。

5.3 クラス定義

```
namespace tlm {
  class tlm_global_quantum
  {
  public:
    static tlm_global_quantum& instance();
    virtual ~tlm_global_quantum();
    void set( const sc_core::sc_time& );
    const sc_core::sc_time& get() const;
    sc_core::sc_time compute_local_quantum();
```

```
  protected:
    tlm_global_quantum();
  };
} // namespace tlm
```

5.4 tlm_global_quantumクラス

a) tlm_global_quantum クラスによって単一のグローバルなクォンタム時間が定義される。これがクォンタム時間のデフォルトとなる。テンポラル・デカップリングをサポートしているイニシエータは、グローバル・クォンタム時間の整数倍、あるいは必要であればより頻繁に同期を図るために用意されている。

b) それぞれのイニシエータはそれぞれ別のクォンタム時間を持つことも可能であるが、一般的には全てのイニシエータはグローバルなクォンタム時間を持つ。同期の回数が少ないイニシエータは他よりクォンタム時間を長く設定できるわけではあるが、一般的には、クォンタム時間が短いとシミュレーション時間に悪い影響を与える。

c) instance()メソッドは、唯一のグローバル・クォンタム・オブジェクトの参照を返す。

d) set()メソッドは、引数で受け取った値をグローバル・クォンタム値として設定する。

e) get()メソッドは、グローバル・クォンタムの値を返す。

f) compute_local_quantum()メソッドは、唯一のグローバル・クォンタムを元に、ローカル・クォンタムの値を計算して返す。ローカル・クォンタムはグローバル・クォンタムの次の整数倍数から、sc_time_stamp の値を引くことにより計算する。compute_local_quantum()がグローバル・クォンタムの整数倍のシミュレーション時間でコールされる場合は、ローカル・クォンタムはグローバル・クォンタムと等しい。そうでない場合は、ローカル・クォンタムはグローバル・クォンタムよりも小さい。

6. 統合インタフェースとソケット

6.1 統合インタフェース

6.1.1 イントロダクション

フォワード・トランスポート・インタフェース 、バックワード・トランスポート・インタフェースは、TLM-2.0 コア・インタフェースをグループ化している。これは、イニシエータ/ターゲット・ソケットによりグループ化される。

統合インタフェースには、トランスポート、DMI、デバッグ・トランスポート・インタフェースを含むが、TLM-1 コア・インタフェースは含まない。

フォワード・インタフェースは、イニシエータ・ソケットからターゲット・ソケットへのフォワード・パスにおけるメソッド呼び出しを提供する。バックワード I/F は、ターゲット・ソケットからイニシエータ・ソケットへのバックワード・パスにおけるメソッド呼び出しを提供する。ブロッキング・トランスポート・インタフェースとデバッグ・トランスポート・インタフェースはバックワード呼び出しパスを必要としない。

標準のイニシエータ・ソケットやターゲット・ソケットに関連のない新しいソケットクラスを定義し、結合インタフェースを使ってこれらのテンプレートクラスをテンプレート引数としてインスタンス化することは、技術的に可能であるが、相互利用性の観点からは推奨しない。一方、標準のソケットクラスから新しいソケットクラスを派生させる方法は、推奨する。

結合インタフェースのテンプレートは、プロトコル型クラスでパラメタライズされる。そのプロトコル・トレイツ・クラスは、フォワード・インタフェース、バックワード・インタフェースで使われる型、すなわちペイロード型とフェーズ型を定義する。プロトコル・トレイツ・クラスは、特定のプロトコルに関連付けられる。デフォルトのプロトコル型は tlm_base_protocol_types である。基本プロトコルに関しては、「8.2 基本プロトコル」を参照。

6.1.2 クラス定義

```
namespace tlm {
  // The default protocol traits class:
  struct tlm_base_protocol_types
  {
  typedef tlm_generic_payload tlm_payload_type;
  typedef tlm_phase tlm_phase_type;
  };
  // The combined forward interface:
  template< typename TYPES = tlm_base_protocol_types >
  class tlm_fw_transport_if
  : public virtual tlm_fw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
  typename TYPES::tlm_phase_type>
  , public virtual tlm_blocking_transport_if<typename TYPES::tlm_payload_type>
```

```
  , public virtual tlm_fw_direct_mem_if<    typename TYPES::tlm_payload_type>
  , public virtual tlm_transport_dbg_if<    typename TYPES::tlm_payload_type>
  {};                                                    // The combined backward interface:
  template < typename TYPES = tlm_base_protocol_types >
  class tlm_bw_transport_if
  : public virtual tlm_bw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
                                                   typename TYPES::tlm_phase_type >
  , public virtual tlm_bw_direct_mem_if
  {};
} // namespace tlm
```

6.2 イニシエータとターゲット・ソケット

6.2.1 イントロダクション

ソケットはポートとエクスポートを組み合わせたものである。イニシエータ・ソケットはフォワード・パス用のポートとバックワード・パスのためのエクスポートがあり、ターゲット・ソケットにはフォワード・パス用のエクスポートとバックワード・パス用のポートがある。また、ソケットは対向するソケットのポートとエクスポートの両方をそれぞれエクスポートとポートに割り当てるために SystemC ポートのバインディング・オペレータをオーバーロードする。ソケットを親から子または子から親のように階層的にバインドする場合、バインドする順番を慎重に考慮することが重要である。

イニシエータとターゲット・ソケットの両方が、C++の継承階層を使用することでコード化される。最上位の派生クラスの tlm_initiator_socket と tlm_target_socket のみが通常、直接アプリケーションで使用される。これら 2 つのソケットは、フォワードとバックワード・インタフェースによって使用される型を定義するプロトコル・トレイツ・クラスでパラメタ化される。ソケットが同一のプロトコル型である場合だけ、それらを一緒に割り当てることができる。デフォルト・プロトコル型は tlm_base_protocol_types クラスである。アプリケーションが新しいプロトコルを定義する場合、新しいプロトコルが汎用ペイロードに基づいているか否かに関係なく、新しいプロトコル・トレイツ・クラスによる統合インタフェース・テンプレートをインスタンス化するべきである。

イニシエータとターゲット・ソケットは以下の利点を提供する:

a) フォワード・パスとバックワード・パス両方の為に、トランスポート、ダイレクト・メモリ、デバッグ・トランスポート・インターフェースを 1 つのオブジェクトにグループ化する。

b) フォワード・パスとバックワード・パス両方のポートとエクスポートを、 1 回の呼び出しでバインドするメソッドを提供する。

c) 互換性のないプロトコル型でパラメタライズされたソケットに対する、強力な型チェック機能を提供する。

d) トランザクションを解釈するために利用できるバス幅パラメタを含む。

tlm_initiator_socket と tlm_target_socket クラスは、TLM2.0 標準のインタオペラビリティ・レイヤに

属する。更に、ユーティリティ・ネームスペース内にひとまとめにして、便利ソケットとして知られる派生ソケットクラスが提供されている。

6.2.2 クラス定義

```
namespace tlm {
  // Abstract base class for initiator sockets
  template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>
  >
  class tlm_base_initiator_socket_b
  {
  public:
    virtual ~tlm_base_initiator_socket_b() {}
    virtual sc_core::sc_port_b<FW_IF>     & get_base_port() = 0;
    virtual                       BW_IF   & get_base_interface() = 0;
    virtual sc_core::sc_export<  BW_IF>   & get_base_export() = 0;
  };
  // Abstract base class for target sockets
  template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>
  >
  class tlm_base_target_socket_b
  {
  public:
    virtual ~tlm_base_target_socket_b();
    virtual sc_core::sc_port_b<BW_IF>     & get_base_port() = 0;
    virtual sc_core::sc_export<FW_IF>     & get_base_export() = 0;
    virtual                       FW_IF   & get_base_interface() = 0;
  };
  // Base class for initiator sockets, providing binding methods
  template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>,
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
  >
```

```
class tlm_base_initiator_socket : public tlm_base_initiator_socket_b<BUSWIDTH, FW_IF, BW_IF>,
                                  public sc_core::sc_port<FW_IF, N, POL>
{
public:
  typedef FW_IF      fw_interface_type;
  typedef BW_IF      bw_interface_type;
  typedef sc_core::sc_port<fw_interface_type, N, POL>      port_type;
  typedef sc_core::sc_export<bw_interface_type>      export_type;
  typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
  base_target_socket_type;
  typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
  base_type;
  tlm_base_initiator_socket();
  explicit tlm_base_initiator_socket(const char* name);
  virtual const char* kind() const;

  unsigned int get_bus_width() const;

  void bind(base_target_socket_type& s);
  void operator() (base_target_socket_type& s);
  void bind(base_type& s);
  void operator() (base_type& s);
  void bind(bw_interface_type& ifs);
  void operator() (bw_interface_type& s);

  // Implementation of pure virtual functions of base class
  virtual sc_core::sc_port_b<FW_IF> & get_base_port() { return *this; }
  virtual      BW_IF      & get_base_interface() { return m_export; }
  virtual sc_core::sc_export<BW_IF> & get_base_export() { return m_export; }
  protected:
  export_type m_export;
};

// Base class for target sockets, providing binding methods
template <
  unsigned int BUSWIDTH = 32,
  typename FW_IF = tlm_fw_transport_if<>,
  typename BW_IF = tlm_bw_transport_if<>,
  int N = 1,
  sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
```

```
class tlm_base_target_socket : public tlm_base_target_socket_b<BUSWIDTH, FW_IF, BW_IF>,
public sc_core::sc_export<FW_IF>
{
public:
   typedef FW_IF      fw_interface_type;
   typedef BW_IF      bw_interface_type;
   typedef sc_core::sc_port<bw_interface_type, N, POL>      port_type;
   typedef sc_core::sc_export<fw_interface_type>      export_type;
   typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
   base_initiator_socket_type;
   typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
   base_type;
   tlm_base_target_socket();
   explicit tlm_base_target_socket(const char* name);
   virtual const char* kind() const;

   unsigned int get_bus_width() const;

   void bind(base_initiator_socket_type& s);
   void operator() (base_initiator_socket_type& s);
   void bind(base_type& s);
   void operator() (base_type& s);
   void bind(fw_interface_type& ifs);
   void operator() (fw_interface_type& s) ;
   int size() const;
   bw_interface_type* operator-> ();
   bw_interface_type* operator[] (int i);                    // Implementation of pure virtual functions of base class
   virtual sc_core::sc_port_b<BW_IF> & get_base_port() { return m_port; }
   virtual      FW_IF      & get_base_interface() { return *this; }
   virtual sc_core::sc_export<FW_IF> & get_base_export() { return *this; }
protected:
   port_type m_port;
};

// Principle initiator socket, parameterized with protocol traits class
template <
   unsigned int BUSWIDTH = 32,
   typename TYPES = tlm_base_protocol_types,
   int N = 1,
   sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
```

```
  class tlm_initiator_socket : public tlm_base_initiator_socket <
    BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
  {
  public:
    tlm_initiator_socket();
    explicit tlm_initiator_socket(const char* name);
    virtual const char* kind() const;
  };

  // Principle target socket, parameterized with protocol traits class
  template <
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm_base_protocol_types,
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
  >
  class tlm_target_socket : public tlm_base_target_socket <
    BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
  {
  public:
    tlm_target_socket();
    explicit tlm_target_socket(const char* name);
    virtual const char* kind() const;
  };
} // namespace tlm
```

6.2.3 tlm_base_initiator_socket_bとtlm_base_target_socket_b

a) 抽象ベースクラス tlm_base_initiator_socket_b と tlm_base_target_socket_b は、ソケットと関連付けられたポート、エクスポート、およびインタフェース・オブジェクトを返す、すべての派生クラスで上書きされる純粋仮想関数を宣言する。

b) これらソケットは、通常アプリケーションから直接使われることはない。

6.2.4 tlm_base_initiator_socketとtlm_base_target_socket

a) tlm_initiator_socket クラスに関して、name 引数をとるコンストラクタはモジュール階層にインスタンス名の文字列を設定するため、sc_port ベースクラスに属するコンストラクタに文字列引数を渡して、また、バックワード・パスの sc_export に対して同じ文字列に”_export”を付加した名前を設定する。"_export"の付加と sc_gen_unique_name の呼び出しは名前の衝突を回避する。例えば、tlm_initiator_socket(“foo")の呼び出しは、ポート名を"foo"に、エクスポート名を"foo_export"に設定する。デフォルト・コンストラクタについては、名前は、ポートに対しては sc_gen_unique_name("tlm_base_initiator_socket")を呼び出し、エクスポートに対しては sc_gen_unique_name("tlm_base_initiator_socket_export")を呼び出すことによって生成される。

b) tlm_base_target_socket クラスに関して、name の引数をとるコンストラクタは、sc_export ベースクラスに属するコンストラクタに文字列引数を渡して、モジュール階層にインスタンス名の文字列を設定する。また、同じ文字列に”_port”を付加した名前をバックワード・パスの sc_port に対して設定する。"_port"の付加と sc_gen_unique_name の呼び出しは、名前の衝突を回避するために行う。例えば、tlm_target_socket(“foo")の呼び出しはエクスポート名を"foo"に、ポート名を"foo_port"に設定する。デフォルト・コンストラクタについては、名前の生成は、ポートに対しては sc_gen_unique_name("tlm_base_target_socket")を呼び出し、エクスポートに対しては sc_gen_unique_name("tlm_base_target_socket_port")を呼び出すことによって行われる。

c) kind メソッドは、クラス名を C 文字列として返す。すなわち、それぞれ、”tlm_base_initiator_socket”または、”tlm_base_target_socket”を返す。

d) get_bus_width メソッドは、BUSWIDTH テンプレート引数の値を返す。

e) テンプレート引数 BUSWIDTH は、ソケットを介して転送される、個々のデータ・ワードのワード長を決定するもので、ワード中のビット数で表される。バースト転送では、BUSWIDTH は各ビート中のビット数を決定する。この属性の厳密な解釈はトランザクション型に依存する。汎用ペイロードにおける BUSWIDTH の意味は、「7.11 データ長アトリビュート」を参照。

f) ソケットとソケットをバインドする場合、二つのソケットは同一の BUSWIDTH テンプレート引数値を持たなければならない。イニシエータまたはターゲットにおける実行可能コードは、BUSWIDTH を参照し、それに従い動作する。

g) ソケットを引数とする bind、operator（）メソッドは、メソッドへの引数として渡されたソケット・インスタンスと、それぞれのメソッドが属するソケット・インスタンスをバインドするものとする。

h) インタフェースを引数とする bind、operator（）メソッドは、メソッドへの引数として渡されたチャネル・インスタンスと、メソッドが属するソケット・インスタンスのエクスポートをバインドするものとする。（チャネルはインタフェースを実装するクラスを表す SystemC 用語。）

i) イニシエータ・ソケットをターゲット・ソケットにバインドする場合、bind、operator()メソッドは、イニシエータ・ソケットの port をターゲット・ソケットの export にバインドする。また、ターゲット・ソケットの port とイニシエータ・ソケットの export をバインドする。これは、双方のソケットが同じ階層に置かれている場合に適用される。

j) イニシエータ・ソケットまたはターゲット・ソケットの、どちらかのソケットのバインド・メソッドまたは operator()を呼ぶことによって、イニシエータ・ソケットをターゲット・ソケットに全く同じ効力でバインドすることができる。どちらの場合でも、フォワード・パスはイニシエータ・ソケットからターゲット・ソケットの向きとなる。

k) イニシエータ・ソケットをイニシエータ・ソケットにバインドする場合、または、ターゲット・ソケットをターゲット・ソケットにバインドする場合、bind、operator()メソッドは、それぞれもう片方のソケットの port と port、export と export をバインドする。これは、階層をまたぐバインド、すなわち、子モジュール上のソケットを親モジュール上のソケットにバインドする場合、または、親モジュール上のソケットを子モジュールのソケットにバインドする場合のように、トランザクションがモジュール階層上下に受け渡される場合に適用される。

l) 階層バインディングにおいては、正しい順序でソケットをバインドする必要がある。イニシエータ・ソケットをイニシエータ・ソケットにバインドする場合は、子ソケットが親ソケットに対してバインドされなければならない。ターゲット・ソケットをターゲット・ソケットにバインドする場合は、親ソケットが子ソケットに対してバインドされなければならない。このルールは、tlm_base_initiator_socket が sc_port から派生しており、tlm_base_target_socket が sc_export から派生していることと合致している。階層をさかのぼる場合は、port から port へ、トップ階層においては port から export へ、階層を下がる場合は export から export へバインドされなければならない。

m) tlm_base_initiator_socket と tlm_base_target_socket の二つのソケットを一緒にバインドするために、フォワード・インタフェース型、バックワード・インタフェース型、バス幅は同じでなければならない。

n) ターゲット・ソケットの size メソッドは、バックワード・パスのターゲット・ソケットのポートの size メソッドを呼び出し、そのポートの size で返される値を返す。

o) ターゲット・ソケットの operator->メソッドは、バックワード・パスのターゲット・ソケットのポートの operator->メソッドを呼び出し、そのポートの operator->で返される値を返す。

p) ターゲット・ソケットの operator[]メソッドは、バックワード・パスのターゲット・ソケットのポートの operator[]メソッドを同じ引数で呼び出し、そのポートの operator[]で返される値を返す。

q) tlm_base_initiator_socket と class tlm_base_target_socket は、マルチ・ソケットとしてそれぞれ動作する。すなわち、一つのイニシエータ・ソケットは、複数のターゲット・ソケットとバインドすることができ、一つのターゲット・ソケットは、複数のイニシエータ・ソケットとバインドすることができる。2 つのクラステンプレートは、テンプレート・パラメタとしてバインディングの数を示すパラメタとポートバインディングポリシーを指定するパラメタを取る。それらは、関連する sc_port テンプレートをインスタンス化する際にパラメタ化する実装中で使用される。

r) tlm_base_initiator_socket または tlm_base_target_socket が複数回バインドされた場合、operator[]メソッドは接続先の対応するオブジェクトを指定する目的に使用できる。インデックス値は bind または operator()メソッドがソケットをバインドするために呼び出された順序で決定される。しかしながら、そのようなソケットによって受信される着信インタフェース・メソッド呼び出しは、呼び出し側を特定するメカニズムを持たないという意味で匿名となる。一方、便利ソケットでは、そのようなメカニズムが提供される。「9.1.4 マルチ・ソケット」を参照。

s) 例えば、一つのソケットが、二つの別々のターゲットにバインドされる場合を考える。socket[0]->nb_transport_fw(...)と socket[1]->nb_transport_fw()は、2 個のターゲットを扱うことができるが、これら二つのターゲットから入ってくる nb_transport_bw()で呼び出し側を特定する方法はない。

t) 仮想メソッド get_base_port と get_base_export の実装は、ソケットの port と export のオブジェクトをそれぞれ返す。仮想メソッド get_base_interface の実装は、イニシエータ・ポートの場合は export オブジェクトを、ターゲット・ソケットの場合はソケット・オブジェクトそのものを返すものとする。

6.2.5 tlm_initiator_socketとtlm_target_socket

a) tlm_initiator_socket と tlm_target_socket は、プロトコル・トレイツ・クラスをテンプレート・パラメタとしてとる。アプリケーションは、通常ベース・ソケットよりこれらのソケット（またはこれらから派生する便利ソケット）を使用するべきである。

b) tlm_initiator_socket と tlm_target_socket クラスのコンストラクタは、対応するベースクラスのコンストラクタを呼ぶものとする。char*引数が存在する場合はそれを渡す。

c) tlm_initiator_socket と tlm_target_socket を相互にバインドする場合は、同じプロトコル・トレイツ・クラス（デフォルトは tlm_base_protocol_types）、および同じバス幅を持つ必要がある。新規プロトコル型に対しても、新しいプロトコル・トレイツ・クラスを定義することにより、ソケット間の強力な型チェックが可能である。これは汎用ペイロードに基づいたプロトコルでなくても可能である。

d) kind メソッドは、クラス名を C 文字列として返す。すなわち、それぞれ、”tlm_initiator_socket”または”tlm_target_socket”を返す。

例

```
#include <systemc>
#include "tlm.h"
using namespace sc_core;
using namespace std;
struct Initiator: sc_module, tlm::tlm_bw_transport_if<>          // Initiator implements the bw interface
{
  tlm::tlm_initiator_socket<32> init_socket;                     // Protocol types default to base protocol
  SC_CTOR(Initiator) : init_socket("init_socket") {
    SC_THREAD(thread);
    init_socket.bind( *this );                                   // Initiator socket bound to the initiator itself
  }
  void thread() {                                                // Process generates one dummy transaction
    tlm::tlm_generic_payload trans;
    sc_time delay = SC_ZERO_TIME;
    init_socket->b_transport(trans, delay);
  }
  virtual tlm::tlm_sync_enum nb_transport_bw(
    tlm::tlm_generic_payload& trans,
    tlm::tlm_phase& phase,
    sc_core::sc_time& t) {
    return tlm::TLM_COMPLETED;                                   // Dummy implementation
  }
  virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
   { }                                                           // Dummy implementation
```

```
};

struct Target: sc_module, tlm::tlm_fw_transport_if<>                    // Target implements the fw interface
{
  tlm::tlm_target_socket<32> targ_socket;                               // Protocol types default to base protocol
  SC_CTOR(Target) : targ_socket("targ_socket") {
    targ_socket.bind( *this );                                          // Target socket bound to the target itself
  }
  virtual tlm::tlm_sync_enum nb_transport_fw(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_core::sc_time& t) {
      return tlm::TLM_COMPLETED;                                        // Dummy implementation
    }
  virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
    { }                                                                 // Dummy implementation
  virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmi_data)
    { return false; }                                                   // Dummy implementation
  virtual unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
    { return 0; }                                                       // Dummy implementation
};

SC_MODULE(Top1)                         // Showing a simple non-hierarchical binding of initiator to target
{
  Initiator *init;
  Target *targ;
  SC_CTOR(Top1) {
    init = new Initiator("init");
    targ = new Target("targ");
    init->init_socket.bind(targ->targ_socket);                          // Bind initiator socket to target socket
  }
};

struct Parent_of_initiator: sc_module                                   // Showing hierarchical socket binding
{
  tlm::tlm_initiator_socket<32> init_socket;
  Initiator* initiator;
  SC_CTOR(Parent_of_initiator) : init_socket("init_socket") {
    initiator = new Initiator("initiator");
    initiator->init_socket.bind( init_socket );             // Bind initiator socket to parent initiator socket
  }
};
```

```
struct Parent_of_target: sc_module
{
   tlm::tlm_target_socket<32> targ_socket;
   Target* target;
   SC_CTOR(Parent_of_target) : targ_socket("targ_socket") {
      target = new Target("target");
      targ_socket.bind( target->targ_socket );          // Bind parent target socket to target socket
   }
};

SC_MODULE(Top2)
{
   Parent_of_initiator *init;
   Parent_of_target *targ;
   SC_CTOR(Top2) {
      init = new Parent_of_initiator("init");
      targ = new Parent_of_target("targ");
      init->init_socket.bind(targ->targ_socket);        // Bind initiator socket to target socket at top level
   }
};
```

7. 汎用ペイロード

7.1 イントロダクション

汎用ペイロードはコア・インタフェースを通過するトランザクション・オブジェクトのための TLM-2.0 が提供するクラス型である。汎用ペイロードは基本プロトコルと密接に関係する。基本プロトコル自身は汎用ペイロードを使ったときの相互利用性を保証するためのルールを定義している。「8.2 基本プロトコル」を参照。

汎用ペイロードはメモリ・マップド・バスモデルの相互利用性を向上することを目的としており、2 つのレベルでこれらを実現する。一つ目は、バス・プロトコルの正確な詳細が重要でないメモリ・マップド・バスの抽象モデルを作るとき、汎用ペイロードは当面の相互利用性を保証する既成の一般目的のペイロードを提供する。そして、同時に無視可能なアトリビュートのための拡張メカニズムも提供する。二つ目は、汎用ペイロードは特定のバス・プロトコルの詳細モデルのベースとなり得ることである。異なるプロトコル間のブリッジもしくはアダプタが必要なとき、ときにはブリッジを書くことが些細なことになるほど実装コスト削減での利点があり、シミュレーション速度を向上させる点での利点がある。

汎用ペイロードは特にメモリ・マップド・バスのモデリングを目的とする。これは、典型的なメモリ・マップド・バス・プロトコルのアトリビュートであるコマンド、アドレス、データ、バイト・イネーブル、シングルワード転送、バースト転送、ストリーミング、応答ステータスを持つ。また、汎用ペイロードはメモリ・マップド・バス以外のプロトコルのモデリングのベースとして使うこともできる。

汎用ペイロードは典型的なメモリ・マップド・バス・プロトコルの全てのアトリビュートを持つわけではない。しかし、拡張メカニズムを持つため、アプリケーションはそれ自身の特別のアトリビュートを追加することができる。

特殊なプロトコルがバスベースかどうかに関わらず、モデリングと相互利用性はプロトコルのオーナーが責任を持つもので、OSCI の範囲外である。それは、プロトコル・オーナーや、熟練者が量産しようとしているモデルの内容や、それらの特定のプロトコルの記述ガイドラインに期する問題である。しかしながら、汎用ペイロードはここでも利用可能である。なぜならば、汎用ペイロードは、モデル作成の共通のスターティング・ポイントを提供し、多くの場合、トランザクション・レベル・モデルで異なるプロトコル間のブリッジ開発のコストを削減する。

汎用ペイロードはイニシエータ・ソケット、ターゲット・ソケットとともに使うことを推奨する。ソケットは汎用ペイロードのデータ配列を解釈する際に使われるバス幅パラメタ、フォワード・パス、バックワード・パス、汎用ペイロードを使う使わないに関わらず異なるプロトコル間の強力な型チェッキングのメカニズムを提供する。

汎用ペイロードはブロッキング・トランスポート・インタフェースとノンブロッキング・トランスポート・インタフェースで使うことができる。また、汎用ペイロードはダイレクト・メモリ・トランスポート・インタフェース、デバッグ・トランスポート・インタフェースでも使うことができる。その場合は、制限されたアトリビュートだけが使われている。

7.2 拡張性と相互利用性

汎用ペイロードのゴールはメモリ・マップド・バス・モデル間の相互運用性を可能にすることであるが、すべてのバスが同様に作成されるというわけではない。2 つのトランザクション・レベル・モデルがあり、それらが異なるプロトコルを用い詳細なレベルでプロトコルがモデル化されているなら、通常の物理的システムで行なわれるように、2 つのモデル間のプロトコル変換を行い通信できるようにアダプタやブリッジが挿入されなければならない。これに対して、設計フローの初期段階で作られる多くのトランザクション・レベル・モデルは、特定のプロトコル規格の詳細を考慮することはない。そのようなモデルでは、特定のアドレスで始まる 1 ブロックのデータをコピーするので十分であり、それらのモデルにおいて、素晴らしい相互運用性を得るのに直接汎用ペイロードが使用できる。

汎用ペイロード拡張機能は、どんな型のいろいろな拡張もトランザクション・オブジェクトに定義して、追加することが可能である。各拡張は新しい属性の集合を表現し、トランザクション・オブジェクトと共に伝送される。イニシエータ、相互接続コンポーネント、およびターゲットは同様に拡張を作成し、追加し、書いて、読むことができる。拡張機能自体は少しの制限も課さない。もちろん、この拡張機能の規律のない使用が相互運用性を危うくするので、規律ある使用が強く奨励される。しかし、柔軟性を必要とするならそれは可能である!

拡張機能の使用には、ソケットをバインドする時のコード化の簡便さの増大とコンパイル時の型チェックが減少するというトレードオフがある。もし、汎用ペイロード拡張の規律のない使用が許されているとするなら、各アプリケーションは、各インターコネクト・コンポーネントおよびターゲット間の明示的なランタイム・チェックを行なうことにより拡張間のいかなる非互換性をも検出することが強いられる。しかし、特定の拡張の存在を強制するメカニズムが全くないであろう。TLM-2.0 規格は、これらの落とし穴を避けるために具体的なコード化ガイドラインを定める。

3 種類の、そして 3 種類のみの推奨される拡張手法がある。これらはブロッキングおよびノン・ブロッキング・トランスポート・インタフェースのトランザクション・テンプレート引数 TRANS、および統合インタフェースのテンプレート引数 TYPES に関する代替手段である。

a) 汎用ペイロードをそのまま、無視可能な拡張と共に使用する

b) tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義する

c) 新しいプロトコル・トレイツ・クラスと新しいトランザクション型を定義する

a→b→c の順に相互利用性が損なわれる。また、これらの手法を a, b, c の順に以下に説明する。

汎用ペイロードから新クラスを派生させる事は可能であるが、それが相互運用性のための推奨の手法でないということを強調しておかなければならない。

また、これらの 3 つのオプションが単一のシステム・モデルの中で混在するかもしれないということも強調しておかなければならない。特に、拡張機能が効率的な相互運用性を可能にするように設計されているので、最初の 2 つのオプションの混在は価値がある。

7.2.1 汎用ペイロードをそのまま、無視可能な拡張と共に使用

a) この場合、トランザクション型は tlm_generic_payload である。そして、フェーズ型は tlm_phase

である。そして、統合インタフェースへのプロトコル・トレイツ・クラスは tlm_base_protocol_types である。これらはそれぞれ、トランスポート・インタフェースに関する TRANS 引数と統合インタフェースに関する TYPES 引数に対するデフォルト値である。基本プロトコルの標準のイニシエータとターゲット・ソケットを使用するモデルは、それらのモデルが汎用ペイロードの意味論とベース・プロトコルを尊重する限り、そのようなモデルと共に相互利用できるだろう。「8.2 基本プロトコル」を参照。

b) この場合、汎用ペイロードの拡張や拡張されたフェーズは"無視可能"とする。"無視可能"とは拡張されたコンポーネント以外のコンポーネントが、この拡張が存在しないものとして動作することを許容することである。「7.20.1.1 無視可能拡張」を参照。

c) 拡張が無視可能であるなら、定義上、ターゲットでその拡張のサポートを強制するコンパイル時のチェックは必要とされない。そして、実際、無視可能拡張機能はコンパイル時のチェックをサポートしない。

d) 汎用ペイロードは、本質的にプロトコルの小さなバリエーションをサポートする。一般的な原則として、ターゲットは汎用ペイロードのあらゆる機能をサポートすることが推奨される。しかし、例えば、特定のコンポーネントはバイト・イネーブルをサポートするかもしれないし、しないかもしれない。汎用ペイロードの特定の機能をサポートできないターゲットは、標準のエラー応答を発生させなければならない。これは汎用ペイロードの仕様の一部であるとして考えられるべきである。

7.2.2 tlm_generic_payloadに対するtypedefを含む新しいプロトコル・トレイツ・クラスを定義

a) この場合、トランザクション型は tlm_generic_payload、フェーズ型は tlm_phase であるが、ソケットを特殊化したプロトコル・トレイツ･クラスは、デフォルトの tlm_base_protocol_types ではなく、新しいアプリケーションで定義されたクラスである。これは、拡張汎用ペイロードが異なった型として扱われるのを保証し、イニシエータとターゲット・ソケットがバインドされたとき、コンパイル時の型チェック機能を提供する。

b) 新しいプロトコル型は独自のルールを持つが、独自ルールは、汎用ペイロードのメモリ・マネージメント・ルール(7.5 汎用ペイロード・メモリ管理を参照)と属性の変更可能性のためのルール（7.7 デフォルト値と属性の更新性を参照）を含む、基本プロトコルのルールを拡張、またはそれと矛盾していてもよい。しかしながら、一貫性と相互運用性のために、基本プロトコルの規則とコーディング･スタイルにできるだけ従うことを推奨する。8.2 ベース・プロトコルを参照のこと。

c) 汎用ペイロード拡張メカニズムは、無視可能または無視不可能、必須拡張に制限なく利用される。すべての拡張の動作は新しいプロトコル・トレイツ・クラスを用いて完全にドキュメント化すべきである。

d) トランザクション型が tlm_generic_payload であるので、トランザクションは汎用ペイロード型を使用する相互接続コンポーネントあるいはターゲットを通して転送でき、すべての拡張を含んだ形で、全体として複製を作ることができる。これは再利用できるコンポーネントを作成したり、アダプタを作成したり、異なるプロトコル間のブリッジを作成するための良い出発点を提供するが、ユーザは非常に慎重に拡張汎用ペイロードのセマンティックスを検討しなければならない。

e) イニシエータからゼロまたは何個かの相互接続コンポーネントを通ってターゲットへのトランザクションが後に続く経路の全長に沿って全く同じプロトコル・トレイツ・クラスを使用することは普通である。しかしながら、アダプタやバス・ブリッジを、あるプロトコルでトランザクションを受け取り、他のプロトコルの送出されるトランザクションとして変換するインターコネクト・コンポーネントとしてモデル化することも可能である。到着するトランザクションに対してはターゲットとして動作し、送出されるトランザクションに対してはイニシエータとして動作するトランザクション・ブリッジを作成することも可能である。

f) 異なるプロトコル・トレイツ・クラスを利用するソケット間で汎用ペイロードを渡す場合、ユーザは拡張としてではなく、汎用ペイロードとして認識しているコンポーネントを通してトランザクションを転送できることを保証するために、非常に慎重にそれぞれの拡張のセマンティックスを検討しなければならない。一般的な規則はない。不運な事故なしで、拡張を認識しないコンポーネントを通してある種の拡張を転送できる場合もある。例えば、データのセキュリティー・レベルを指定する属性がある。他の拡張は、明示的な追加を必要とするか、または全くサポートできないであろう。例えば、インターコネクトがロックされること、と指定する属性がある。

7.2.3 新しいプロトコル・トレイツ・クラスと新しいトランザクション型を定義

a) この場合は、トランザクション型は汎用ペイロードとは関係がない。

b) 新しいプロトコル・トレイツ・クラスは統合インタフェースとソケットをパラメタライズするように定義する必要がある。

c) 新しいトランザクション型が汎用ペイロードとかなり異なるか、非常に特別なプロトコルを表すとき、この選択は正当化される。

d) 最大限の相互運用性のために汎用ペイロードを使用するつもりであるなら、お勧めの手法は、新しいクラスを定義して使用するより、前2つの節の1つで説明される汎用ペイロードを使用することである。

7.3 汎用ペイロード・アトリビュートとメソッド

汎用ペイロード・クラスはプライベートなアトリビュートとそれに値をセット/ゲットするパブリックなアクセス関数を持つ。それらのアクセス関数の正確な実装は、実装定義(実装依存)である。

アトリビュートの大部分の設定はイニシエータが行い、相互接続コンポーネントやターゲットで変更しないものとする。アドレスやDMI許可情報、レスポンス・ステータスと拡張属性はインターコネクトやターゲットが変更する場合がある。リード・コマンドの場合、ターゲットはデータ配列を変更する。

7.4 クラス定義

```
namespace tlm {
  class tlm_generic_payload;
  class tlm_mm_interface {
  public:
    virtual void free(tlm_generic_payload*) = 0;
    virtual ~tlm_mm_interface() {}
```

```
};
unsigned int max_num_extensions();    class tlm_extension_base
{
public:
   virtual tlm_extension_base* clone() const = 0;
   virtual void free() { delete this; }
   virtual void copy_from(tlm_extension_base const &) = 0;
protected:
   virtual ~tlm_extension_base() {}
};
template <typename T>
tlm_extension : public tlm_extension_base
{
public:
   virtual tlm_extension_base* clone() const = 0;
   virtual void copy_from(tlm_extension_base const &) = 0;
   virtual ~tlm_extension() {}
   const static unsigned int ID;
};
enum tlm_command {
   TLM_READ_COMMAND,
   TLM_WRITE_COMMAND,
   TLM_IGNORE_COMMAND
};
enum tlm_response_status {
   TLM_OK_RESPONSE = 1,
   TLM_INCOMPLETE_RESPONSE = 0,
   TLM_GENERIC_ERROR_RESPONSE = -1,
   TLM_ADDRESS_ERROR_RESPONSE = -2,
   TLM_COMMAND_ERROR_RESPONSE = -3,
   TLM_BURST_ERROR_RESPONSE = -4,
   TLM_BYTE_ENABLE_ERROR_RESPONSE = -5
};
#define TLM_BYTE_DISABLED 0x0
#define TLM_BYTE_ENABLED 0xff    class tlm_generic_payload {
public:
   // Constructors and destructor
   tlm_generic_payload();
   explicit tlm_generic_payload( tlm_mm_interface* );
   virtual ~tlm_generic_payload();
private:
```

```
    // Disable copy constructor and assignment operator
    tlm_generic_payload( const tlm_generic_payload& );
    tlm_generic_payload& operator= ( const tlm_generic_payload& );
  public:
    // Memory management
    void set_mm( tlm_mm_interface* );
    bool has_mm() const;
    void acquire();
    void release();
    int get_ref_count() const;
    void reset();
    void deep_copy_from( const tlm_generic_payload& ) const;
    void update_original_from( const tlm_generic_payload & , bool use_byte_enable_on_read = true );
    void update_extensions_from(const tlm_generic_payload & );
    void free_all_extensions();                                   // Access methods
    tlm_command get_command() const;
    void set_command( const tlm_command );
    bool is_read();
    void set_read();
    bool is_write();
    void set_write();
    sc_dt::uint64 get_address() const;
    void set_address( const sc_dt::uint64 );
    unsigned char* get_data_ptr() const;
    void set_data_ptr( unsigned char* );
    unsigned int get_data_length() const;
    void set_data_length( const unsigned int );
    unsigned int get_streaming_width() const;
    void set_streaming_width( const unsigned int );
    unsigned char* get_byte_enable_ptr() const;
    void set_byte_enable_ptr( unsigned char* );
    unsigned int get_byte_enable_length() const;
    void set_byte_enable_length( const unsigned int );           // DMI hint
    void set_dmi_allowed( bool );
    bool is_dmi_allowed() const;
    tlm_response_status get_response_status() const;
    void set_response_status( const tlm_response_status );
    std::string get_response_string();
    bool is_response_ok();
    bool is_response_error();                                     // Extension mechanism
    template <typename T> T* set_extension( T* );
```

```
    tlm_extension_base* set_extension( unsigned int , tlm_extension_base* );
    template <typename T> T* set_auto_extension( T* );
    tlm_extension_base* set_auto_extension( unsigned int index , tlm_extension_base* );
    template <typename T> void get_extension( T*& ) const;
    template <typename T> T* get_extension() const;
    tlm_extension_base* get_extension( unsigned int ) const;
    template <typename T> void clear_extension( const T* );
    template <typename T> void clear_extension();
    template <typename T> void release_extension(T* ext);
    template <typename T> void release_extension();
    void resize_extensions();
  };
} // namespace tlm
```

7.5 汎用ペイロードのメモリ管理

a) イニシエータは既存ストレージにデータ・ポインタ、バイトイネーブル・ポインタ・アトリビュートを設定する。ストレージはスタティックやオートマティック（stack）、またはダイナミック（new）で確保されている。トランザクションのライフタイムが完了するまで、イニシエータはこのストレージを削除しない。汎用ペイロードのデストラクタはこれら 2 つのアレイを削除しない。

b) この節に関連して「7.20 汎用ペイロードの拡張」のルールを読むこと。

c) 汎用ペイロードのメモリ管理には 2 つの異なる方法がある。1 つは独立してメモリを管理するメモリ・マネージャを利用する方法、もう一つはイニシエータによるアドホックなメモリ管理方法である。混在は可能。いずれの方法もトランザクション・オブジェクトとその拡張の両方を管理すること。

d) tlm_generic_payload オブジェクトの生成・削除は、拡張配列の確保が要因で処理時間が大きい。そのため汎用ペイロードの生成・削除を繰り返し行わないようにすべきである。推奨する方法としては、トランザクション・オブジェクトのプールを持つメモリ・マネージャを使用する方法と、アドホックなメモリ管理であれば同一の汎用ペイロードを b_transport コール時に繰り返し再利用（実質的にサイズが 1 のトランザクション・プール）する方法がある。特に transport メソッドをコールする毎に汎用ペイロードの生成と削除を行うと、大きな速度低下を招くため避けるべきである。

e) メモリ・マネージャは抽象クラス tlm_mm_interface を基底とするユーザ定義のクラスで、純粋仮想関数 free メソッドの実装が必要である。メモリ・マネージャはトランザクション・プールから汎用ペイロード・トランザクション・オブジェクトを割り当てるためのメソッドを提供する。同一のプールからトランザクション・オブジェクトを返すために free メソッドを実装し、プール全体を削除するためにデストラクタを実装する。free メソッドはトランザクション・オブジェクトの参照カウント数が 0 になった時、tlm_generic_payload クラスの release メソッドからコールされる。また、自動削除対象の拡張を全て削除するために、tlm_generic_payload クラスの reset メソッドからもコールされる。

f) 汎用ペイロードのメソッド set_mm、acquire、release、get_ref_count、reset は、メモリ・マネー

ジャーが存在するときにだけ利用する。デフォルトでは、汎用ペイロードのオブジェクトはメモリ管理セットを持っていない

g) イニシエータによるアドホックなメモリ管理の場合、TLM-2.0 コア・インタフェース呼び出しの前にイニシエータはトランザクション・オブジェクトのためのメモリを割り当てる。また、呼出し後にイニシエータはトランザクション・オブジェクトとその拡張オブジェクトを削除あるいはプールする。

h) 汎用ペイロードがブロッキング・トランスポート・インタフェース、ダイレクト・メモリ・インタフェースまたはデバッグ・トランスポート・インタフェースと共に使用されるとき、いずれのメモリ管理方法も使用される。イニシエータによるアドホックなメモリ管理でも十分である。メモリ・マネージャがなければ、b_transport、get_direct_mem_ptr、または transport_dbg メソッドは、トランザクション・オブジェクトとその拡張がリターンの時に無効にされるか、または削除されると仮定するべきである。

i) 汎用ペイロードがノンブロッキング・トランスポート・インタフェースと共に使用されるとき、メモリ・マネージャを使用すべきである。これは呼び出し元関数がイニシエータ、インターコネクト・コンポーネント、またはターゲットであることにかかわらない。

j) ブロッキングからノンブロッキングへのトランスポートアダプターは受け取ったトランザクションにメモリ・マネージャがなければ設定する必要がある。また呼び出し元に制御を返す前に、そのトランザクションに設定されているメモリ・マネージャを削除する必要がある。メモリ・マネージャは参照カウントが 0 になるまで削除できないので、メモリ・マネージャの free メソッドでは、トランザクション・オブジェクトを削除しないような実装が必要となる。simple_target_socket がそのようなアダプタの実装例となっている。

k) メモリ・マネージャを使用するとき、トランザクション・オブジェクトとその拡張オブジェクトはヒープ（結局は new か malloc）で割り当てる。

l) アドホックなメモリ管理を使用するとき、トランザクション・オブジェクトとその拡張オブジェクトはヒープまたはスタックに割り当てられるものとする。スタックに割り当てられた時、メモリ・リークやセグメンテーション・フォールトを回避するために、拡張オブジェクトのメモリ管理に注意を払うこと。

m) メソッド set_mm は汎用ペイロード・オブジェクトのメモリ・マネージャを設定し、そのオブジェクトのアドレスが引数として渡される。引数が null である場合、既存のメモリ・マネージャはトランザクション・オブジェクトから切り離される。また、それ自体は、削除されない。メモリ・マネージャと参照カウントが 0 以上を既に持つトランザクション・オブジェクトのために set_mm を呼んではいけない。

n) メモリ・マネージャが設定されている場合にだけ、メソッド has_mm が true を返すものとする。nb_transport メソッドから呼ばれると、has_mm は true を返すべきである。

o) b_transport、get_direct_mem_ptr、または transport_dbg メソッドから呼ばれると、has_mm は true か false を返す。インターコネクト・コンポーネントは、has_mm を呼び、トランザクションにメモリ・マネージャが設定されているかどうかによって適切な行動を取る。さもなければ、インターコネクト・コンポーネントはメモリ・マネージャ（例えば、ヒープ割り付け）をもつトランザクションのすべての義務を引き受けるものとする。そして、メモリ・マネージャの存在を必要とするメソッドのいずれも（例えば acquire）呼び出さない。

p) それぞれの汎用ペイロード・オブジェクトは参照カウントを持つ。そのデフォルト値はゼロ。

q) メソッド acquire は参照カウントの値を増加させる。メモリ・マネージャが存在しないとき、acquire が呼ばれると、ランタイムエラーが発生する。

r) メソッド release は参照カウントの値を減少させる。参照カウントの値が 0 となると、メモリ・マネージャ・オブジェクトの free メソッド呼んで、引数としてトランザクション・オブジェクトのアドレスを渡す。メモリ・マネージャが存在しないとき、acquire が呼ばれると、ランタイムエラーが発生する。

s) メソッド get_ref_count は参照カウントの値を返すものとする。メモリ・マネージャが不在のとき、返り値は 0。

t) メモリ・マネージャで、そのオブジェクトを最初にインタフェース・メソッド呼び出しの引数のとして、渡す前に、各イニシエータは、それぞれのトランザクション・オブジェクトの acquire メソッドを呼ぶ。そして、オブジェクトが不要になったときに、そのトランザクション・オブジェクトの release メソッドを呼ぶ。

u) メモリ・マネージャで、現在のインタフェース・メソッド呼び出しを超えてトランザクション・オブジェクトのライフタイムを延長するときは、各インターコネクト・コンポーネントとターゲットは acquire メソッドを呼ぶべきである。そして、オブジェクトが不要になったときに release メソッドを呼ぶ。

v) メモリ・マネージャを用いるコンポーネントは、あらゆるインタフェース・メソッド・コールやプロセスの中で release メソッドをコールしてよい。したがってコンポーネントは、以前に acquire メソッドをコールしていない限り、インタフェース・メソッド・コール後あるいは制御の譲渡後にトランザクション・オブジェクトがまだ有効であると想定してはいけない。例えば、イニシエータでは nb_transport_bw の実装で release メソッドをコールしてもよいし、ターゲットでは nb_transport_fw の実装で release メソッドをコールしてもよい。

w) 解析目的のためにトランザクション・オブジェクトのライフタイムを無期限に伸ばす場合は、各インターコネクト・コンポーネントとターゲットは参照カウント・メカニズムを使用するよりもトランザクション・オブジェクトのクローンを作るべきである。言い換えれば、プロトコルの正常なフェーズを超えてトランザクション・オブジェクトのライフタイムを伸ばすのに参照カウントを使用するべきでない。

x) メモリ・マネージャが存在するとき、トランザクション・オブジェクトを参照している、イニシエータ自身以外のコンポーネントがないと参照カウントが示すまで、トランザクション・オブジェクトを新しいトランザクションを表すのに再利用したり、または異なったインタフェースと共に再利用してはいけない。イニシエータがトランザクション・オブジェクトのために acquire を呼ぶとすると、参照カウントが 1 になるまでである。このルールは、同じインタフェースのトランザクションの再利用、トランスポートの転送、ダイレクト・メモリ、デバックトランスポート・インターフェイスの時、適用される。異なるトランザクション・インスタンスとしてトランザクション・オブジェクトを再利用する時、参照カウントが 0 になるまで、すなわちオブジェクトが解放されるまで再利用を行わないのが最も良い方法である。

y) メソッド reset は自動削除する拡張を削除して、対応する拡張へのポインタを null に設定する。各拡張は拡張オブジェクトのメソッド free と呼ぶことによって、削除される。ユーザが

拡張オブジェクトのための明示的にメモリ管理を提供したいなら、free メソッドをオーバーロードできる。reset メソッドは、トランザクションの生存期間終了時点で拡張を削除するために、一般的に tlm_mm_interface クラスの free メソッドからコールされるべきである。

z) 拡張オブジェクトは set_extension コールで追加され、release_extension コールで削除される。clear_extension コールは拡張のポインタのみをクリアし、拡張オブジェクトを削除しない。トランザクション・オブジェクトがメモリ・マネージャなしでスタックに割り当てられ、拡張オブジェクトがプールされる場合に、この後者の振舞いが必要である。

aa) メモリ・マネージャが存在しない場合、b_transport、get_direct_mem_ptr、または transport_dbg からコントロールを返す前に、コンポーネントが割り当てられるまたは設定されるどちらの場合であっても、与えられた拡張がその同じ拡張を削除またはクリアすべき。たとえば、b_transport で実装され、set_mm にてトランザクション・オブジェクトをメモリ・マネージャに追加した場合のインターコネクト・コンポーネントは、トランザクション・オブジェクトとその拡張が削除されるまで、b_transport から返らない。（以降のコンポーネントも同様に拡張が既に削除されていることを仮定する）

bb) メモリ・マネージャが存在する場合、拡張は set_auto_extension コールで追加され、メモリ・マネージャで自動的に削除あるいはプールされる。set_extension コールで追加された拡張は、いわゆるスティッキー拡張と呼ばれる。スティッキー拡張とは、トランザクション参照カウント数が 0 でプールされているが、トランザクションとの関連付けが維持されている場合に、自動的に削除されないものを指す。スティッキー拡張は、トランスポート呼び出し間で拡張オブジェクトを削除したり再作成したりする必要がないので、拡張オブジェクトを管理する場合には特に有効な方法である。

cc) メモリ・マネージャが存在するかどうか分からない場合、拡張は set_extension コールで追加し、release_extension コールで削除すべきである。この呼出し手順はメモリ・マネージャの存在の如何によらず安全である。この状況は has_mm コールしないインターコネクト・コンポーネントかターゲットの中で起こりうる。（イニシエータの中では、常に、メモリ・マネージャの存在するかどうか分かっている、且つ has_mm コールはメモリ・マネージャの有無に関わらず常に自明である）

dd) メソッド free_all_extensions は自動削除のためにマークしたものを含むすべての拡張を削除して、対応する拡張ポインタを null に設定する。各拡張は拡張オブジェクトの free メソッド・コールで削除される。ユーザが拡張オブジェクトのための明示的なメモリ管理を提供したいなら、free メソッドをオーバーロードできる。

ee) free_all_extensions はメモリ・マネージャを使用しないでプールされたトランザクション・オブジェクトから拡張を削除する時に便利である。メモリ・マネージャを使う時、自動削除のためにマークした拡張は削除される。スティッキー拡張は削除されない。

ff) deep_copy_from メソッドは、メソッドへの引数として渡される別のトランザクション・オブジェクトをコピーすることによって、現在のトランザクション・オブジェクトのアトリビュートと拡張を変更する。コマンド、アドレス、データ長、バイト・イネーブル長、ストリーミング幅、レスポンス・ステータス、DMI 許可情報がコピーされる。もし 2 つの（コピー元とコピー先）トランザクションのポインタが null でなければ、データとバイト・イネーブル配列はディープコピーされる。アプリケーションは現在のトランザクションで配列が十分大きいことを確認する必要がある。他のトランザクションの拡張が現在のトランザクションに

存在しているならば、拡張クラスの copy_from メソッドのコールでコピーされる。さもなければ、新しい拡張オブジェクトは拡張クラスの clone メソッド・コールで作成され、現在のトランザクションにセットされる。クローンの場合で、もしメモリ・マネージャが現在のトランザクションのために存在するならば、新しい拡張は自動削除のマークがされる。

gg) 言い換えれば、メモリ・マネージャが存在する場合で、deep_copy_from は現在のオブジェクトに存在しない新しい拡張の全てに自動削除をマークする。メモリ・マネージャが存在しない場合は、全ての拡張は、自動削除の対象にできない。

hh) update_original_from メソッドは、メソッドへの引数として渡される別のトランザクション・オブジェクトをコピーすることによって、現在のトランザクション・オブジェクトの特定のアトリビュートと拡張を変更する。update_original_from メソッドを用いる目的は、deep_copy_from メソッドで作成したトランザクションに対して、レスポンス情報を反映させることである。現在のトランザクション・オブジェクトのレスポンス・ステータスと DMI 許可情報がコピーされる。もし現在のトランザクションのコマンド・アトリビュートが TLM_READ_COMMAND で、2 つの（コピー元とコピー先）トランザクションのポインタが null ではなくかつ不一致ならば、データ配列はディープコピーされる。もしバイト・イネーブルのポインタが null ではなく use_byte_enable_on_read 引数が true ならば、バイト・イネーブル配列は read コマンドによる上記コピー時のマスクに使われる。そうでなければ完全なデータ配列がディープコピーされる。現在のトランザクション・オブジェクトの拡張情報は update_extensions_from メソッドにより更新されなければならない。

ii) メソッド update_extensions_from は、別のトランザクション・オブジェクトからそれらの現在のオブジェクトに存在している拡張をコピーすることによって、現在のトランザクション・オブジェクトの拡張を変更するものとする。拡張は拡張クラスの copy_from メソッド・コールによってコピーされる。

jj) deep_copy_from、update_original_from、update_extensions_from の典型的なユースケースは、トランザクション・ブリッジにおける使用である。到着したリクエストをディープコピーして、イニシエータ・ソケットを通してコピーを出す。そしてレスポンスを受け取ると適切なアトリビュートと拡張情報を元のトランザクション・オブジェクトにコピーバックする。トランザクション・ブリッジは、配列のディープコピーを行うか、あるいは単にポインタのコピーを行うかを選んでよい。

kk) これらの義務は汎用ペイロードに適用される。原則として、同様の義務は汎用ペイロードに関係ないトランザクション型にも適用される。

7.6 コンストラクタ、代入、デストラクタ

a) デフォルト・コンストラクタは、以下の手順で汎用ペイロード・アトリビュートにデフォルト値を設定する。

b) コンストラクタ tlm_generic_payload(tlm_mm_interface*)は、汎用ペイロード・アトリビュートにデフォルト値を設定し、汎用ペイロード・オブジェクトのメモリ・マネージャに、引数で渡されるオブジェクトを設定する。これは、デフォルト・コンストラクタを呼び出し、直後に set_mm を呼び出した場合と同じ結果である。

c) コピーコンストラクタと代入演算は無効である。

d) 仮想デストラクタ~tlm_generic_payload は、自動削除とマークされたものに限らず、すべての拡張を削除するものとする。各拡張は拡張オブジェクトの free メソッドの呼び出しで削除される。デストラクタはデータ配列、バイト・イネーブル配列を削除しない。

7.7 アトリビュートのデフォルト値と変更可否

汎用ペイロードに含まれるアトリビュートと配列の、デフォルト値と変更可否を以下の表に要約する。

アトリビュート	デフォルト値	インターコネクトでの変更	ターゲットでの変更
コマンド	TLM_IGNORE_COMMAND	No	No
アドレス	0	Yes	No
データ・ポインタ	0	No	No
データ長	0	No	No
バイト・イネーブル・ポインタ	0	No	No
バイト・イネーブル長	0	No	No
ストリーミング幅	0	No	No
DMI 許可	False	Yes	Yes
レスポンス・ステータス	TLM_INCOMPLETE_RESPONSE	No	Yes
拡張ポインタ	0	Yes	Yes

配列	デフォルト値	インターコネクトでの変更	ターゲットでの変更
データ配列	-	No	リード・コマンドのみ
バイト・イネーブル配列	-	No	No

a) 汎用ペイロードの各アトリビュートに対して値をセットするのは、イニシエータの役割である（拡張ポインタは除く）。これは、トランザクション・オブジェクトがインタフェース・メソッド・コールに渡される前に行う必要がある。トランザクション・オブジェクトがプールされ、再利用される場合においても、アトリビュートに正しい値が確実にセットされるよう、注意をはらう必要がある。

b) トランザクション・オブジェクトがプールに戻されるか、または再利用される場合、当該トランザクション・インスタンスのライフタイムの終わりに達した時点で、これらの変更可否ルールは無効となる。

c) トランザクション・オブジェクトをインタフェース・メソッド・コール（b_transport、nb_transport_fw、 get_direct_mem_ptr、 transport_dbg）に引数として渡した後、トランザクションのライフタイム中変更可能な汎用ペイロードのアトリビュートは、拡張ポインタのみである。

d) インターコネクト・コンポーネントは、アドレス・アトリビュートを変更することができる。ただし、これができるのは、当該トランザクションを、フォワード・パスの TLM-2.0 のコア・インタフェース・メソッドの引数として渡す前に限られる。一旦インターコネクト・コンポーネントが下流のコンポーネントに対してトランザクションの参照を渡した後は、当該トランザクションのライフタイム中、アドレス・アトリビュートを再設定することはできない。

e) 前記のルールにより、b_transport、get_direct_mem_ptr、transport_dbg のいずれかのフォワード・パス上のインタフェース・メソッドが呼び出された直後、アドレス・アトリビュートは有効となる。nb_transport_fw の場合は、フェーズが BEGIN_REQ の場合のみ、呼び出し直後にアドレス・アトリビュートが有効となる。各 TLM-2.0 のフォワード・パスのインタフェース・メソッド・コールから復帰した時点で、アドレス・アトリビュートは最終のインターコネクト・コンポーネントがセットした値に書き換えられている。このため、トランザクションの経路選択の目的としては未定義として扱うべきである。

f) インターコネクトとターゲットは、ライト・コマンドの場合はデータ配列を変更してはならない。リード・コマンドの場合は、ターゲットのみがデータ配列を修正することができる。

g) トランザクション・オブジェクトに対して、ターゲットは DMI 許可アトリビュート、レスポンス・ステータス・アトリビュート、（リード・コマンドに対して）データ配列を修正することができる。この変更は、トランザクション・オブジェクトを最初に受け取ってから、上流へレスポンスを返すまでの時間であれば、いつでも可能である。ターゲットは、上流に向けてレスポンスを発行した後、これらのアトリビュートを変更することはできない。レスポンスを発行するとは、ここでは次の場合に相当する。b_transport、get_direct_mem_ptr、transport_dbg の各メソッドから制御を返す時、nb_transport に対して BEGIN_RESP フェーズを渡した時、nb_transport から TLM_COMPLETED を返した時である。

h) DMI 許可アトリビュートが false の場合、インターコネクト・コンポーネントは DMI 許可アトリビュートを変更してはならない。しかし、ターゲットが DMI 許可アトリビュートに true を設定した場合は、インターコネクト・コンポーネントは上流方向にレスポンスを送る際に、DMI 許可アトリビュートを false にリセットすることができる。言い換えると、ターゲットが DMI 許可アトリビュートを設定した場合でも、インターコネクト・コンポーネントは、DMI 許可アトリビュートをクリアすることができる。

i) ターゲットによって修正された DMI 許可アトリビュート、レスポンス・ステータス・アトリビュート、（リード・コマンドに対する）データ配列の値をイニシエータが確認できるのは、レスポンスを受け取った後のみである。

j) 上記のルールに従ってあるコンポーネントが、特定の期間内にトランザクション・アトリビュートの値を修正する場合、その期間内であれば、アトリビュートはいつでも、何度でも変更することができる。その他のコンポーネントは、その期間を過ぎた後にのみ、アトリビュートの値を読むことができる（拡張は例外）。

k) イニシエータ、インターコネクト、ターゲットの役割は、ダイナミックに変化する場合がある。例えば、インターコネクト・コンポーネントはレスポンス・ステータス・アトリビュートを変更することはできないが、同じコンポーネントが当該トランザクションに対するターゲットとして振舞う場合は、そのレスポンスを変更することができる。その場合、当該トランザクションを下流のコンポーネントに転送することはできない。

l) 汎用ペイロードがダイレクト・メモリ・インタフェース、デバッグ・トランスポート・インタフェースのトランザクション型として使用される場合、本章の修正可否ルールは、適宜必要なアトリビュートに対して適用される。すなわち、ダイレクト・メモリの場合はコマンドとアドレス・アトリビュートへ、デバッグ・トランスポートの場合は、コマンド、アドレス、データ・ポインタ、データ長アトリビュートに対して適用される。

7.8 コマンド・アトリビュート

a) set_command メソッドは値渡しの引数でコマンド・アトリビュートを設定し、get_command メソッドは現在のコマンド・アトリビュートを値で返す。

b) set_read メソッドと set_write メソッドはそれぞれ TLM_READ_COMMAND、TLM_WRITE_COMMAND をコマンド・アトリビュートに設定する。is_read メソッドと is_write メソッドはそれぞれ現在のコマンド・アトリビュートの値が TLM_READ_COMMAND、TLM_WRITE_COMMAND かどうかを判定しそれを返す。

c) リード・コマンドはコマンド・アトリビュートが TLM_READ_COMMAND と等しい汎用ペイロード・トランザクションであり、ライト・コマンドはコマンド・アトリビュートが TLM_WRITE_COMMAND と等しい汎用ペイロード・トランザクションである。イグノア・コマンドは、コマンド・アトリビュートが TLM_IGNORE_COMMAND と等しい汎用ペイロード・トランザクションである。

d) リード・コマンドを受取った時、ターゲットはローカル配列の内容をデータ・ポインタ・アトリビュートとなるよう配列ポインタにコピーする。

e) ライト・コマンドを受取った時、ターゲットはデータ・ポインタ・アトリビュートにより指定されている配列の中身をターゲット内のローカル配列にコピーする。

f) もしターゲットがリードまたはライト・コマンドを実行することが出来ない場合は、標準エラー応答を生成する。推奨するレスポンス・ステータスは TLM_COMMAND_ERROR_RESPONSE である。

g) イグノア・コマンドはヌル・コマンドである。イグノア・コマンドは汎用ペイロードのリードもしくはライト・コマンドを実行することなく、汎用ペイロードの拡張を運ぶ手段として使うことを意図している。しかし、拡張に関するルールは、すべての 3 コマンドで同じである。

h) イグノア・コマンドを受け取った場合、ターゲットはライトまたはリード・コマンドを実行してはならない。

i) イグノア・コマンドを受け取った際、通常インターコネクト・コンポーネントとして動作するコンポーネントは、トランザクションをターゲットに向けて進める（インターコネクトとして振る舞う）か、もしくはエラー応答（ターゲットとして振る舞う）かのどちらかである。リードとライト・コマンドを別に接続するコンポーネントは、エラー応答を返すものとする。

j) トランザクションを受け取り、汎用ペイロード・アトリビュート値が満足されることをチェックしたら、ターゲットはイグノア・コマンドの実行が成功したものとみなす。「7.16 応答ステータス・アトリビュート」の Response status attribute を参照のこと。

k) コマンド・アトリビュートはイニシエータにより設定され、インターコネクトやターゲット

では書き換わらない。

l) コマンド・アトリビュートのデフォルト値は TLM_IGNORE_COMMAND である。

7.9 アドレス･アトリビュート

a) set_address は、アドレス・アトリビュートに引数で与えられた値を設定し、 get_address は現在値を返す。

b) read コマンドや write コマンドに対して、ターゲットはアドレス・アトリビュートの現在値が read/write される連続するデータ･ブロックの最初のバイトを指しているものと解釈する。ただし、データ・ポインタで示される配列の先頭バイトを指しているかどうかは、ホスト計算機のエンディアンに依存する。

c) データ配列中の特定のバイトのアドレスは、アドレス･アトリビュート、配列インデックス、ストリーミング幅・アトリビュート、ホスト計算機のエンディアン、ソケット幅で決まる。「7.17 エンディアン」 を参照のこと。

d) アドレス・アトリビュートの値は、ワード・アラインメントである必要はない。（もしアドレス・アトリビュートがバイトで表現されるローカルソケット幅の倍数であれば、アドレス計算が大幅に単純化できる）

e) ターゲットが指定されたアドレス・アトリビュートでトランザクションを実行できない場合は、標準エラーを生成する。推奨のエラーコードは、TLM_ADDRESS_ERROR_RESPONSE である。

f) アドレス・アトリビュートはイニシエータで設定されるが、一つ以上の接続コンポーネントにより上書きされることがある。上書きは、接続コンポーネントがメモリの絶対アドレスからターゲットが認識できる相対アドレスへ変換する場合などのために必要である。アドレス・アトリビュートが上書きされた場合、明示的に別の場所に退避しない限り、上書き前の値は失われる。

g) アドレス・アトリビュートのデフォルト値は‘0’である。

7.10 データ・ポインタ･アトリビュート

a) set_data_ptr はデータ・ポインタ・アトリビュートに、引数で与えられた値を設定し、get_data_ptr は現在値を返す。データ・ポインタ・アトリビュートは、データ配列へのポインタで、ポインタ値の設定や取得をするもので、データ配列の内容の設定や取得をするものではない。

b) ターゲットは read/write コマンドで、それぞれのデータ配列へのデータコピー、データ配列からのデータコピーを行う。

c) データとバイト・イネーブル配列用の領域はイニシエータが確保する。その領域は、イニシエータ内のレジスタ・ファイルやキャッシュ・メモリのようなデータ領域や、トランザクションレベル・インタフェースのデータ転送に使うテンポラリ･バッファである。

d) 一般的には、汎用ペイロードのデータ配列の構成は、イニシエータやターゲットのローカル・レジスタの構成とは独立である。しかしほとんどの場合、汎用ペイロードはターゲットとのデータ・コピーは memcopy 一回で行うよう設計されるので、ターゲットのレジスタは

汎用ペイロードと同じ構成であることを想定している。この前提はあくまでもシミュレーション速度のためであり、汎用ペイロードの表現能力を制約するものではない。ターゲットはデータ配列とのコピーについて任意のデータ変換をすることができる。

e) トランザクション･オブジェクトがヌル・ポインタでトランスポート・インタフェースを呼び出すとエラーとなる。

f) データ配列長はデータ長アトリビュートの値以上であること。（単位はバイト）

g) データ・ポインタ・アトリビュートはイニシエータが設定するもので、他のコンポーネントやターゲットによって上書きされない。

h) write コマンドや TLM_IGNORE_COMMAND では、データ配列はイニシエータが設定するもので、他のコンポーネントやターゲットによって上書きされない。

i) read コマンドでは、（データ配列はバイト・イネーブルの設定に従って）、ターゲットが応答を返す前にのみターゲットで上書きされ、他のコンポーネントから上書きはできない。b_transport、get_direct_mem_ptr もしくは transport_dbg メソッドから制御が返ったとき、もしくは BEGIN_RESP フェーズが nb_transport の引数としてに渡ったとき、もしくは nb_transport から TLM_COMPLETED が返ったときはすぐに、ターゲットは応答を返す。

j) データ・ポインタ・アトリビュートのデフォルト値は 0（null）である。

7.11 データ長アトリビュート

a) set_data_length はデータ長・アトリビュートに引数で与えられた値を設定し、 get_data_length は現在値を返す。

b) read コマンドや write コマンドに対して、ターゲットはデータ長アトリビュートの現在値を、バイト・イネーブル・アトリビュートでディスエーブルされているバイトも含めて、コピーされるデータ配列のバイト数であると解釈する。

c) 値はイニシエータが設定して、他のコンポーネントやターゲットによって上書きされない。

d) ‘０’に設定してはいけない。ゼロバイト転送の場合は、コマンド・アトリビュートを’TLM_IGNORE_COMMAND’とする。

e) インタオペラビリティ・レイヤの標準ソケットクラスもしくはその派生クラスを使って、バースト転送をする場合、転送のワード長はソケットの BUSWIDTH テンプレート・パラメタで指定される。BUSWIDTH はデータ長アトリビュートとは独立で、ビット数で表される。もし、データ長が BUSWIDTH/８と同じか小さい場合のトランザクションはシングルワード転送であり、それより大きい場合はバースト転送を効率的にモデリングしていることになる。一つのトランザクションを異なるバス幅のソケットで渡すことが出来る。BUSWIDTH は転送のレイテンシ計算に使用される。

f) ターゲットは、ターゲットのワード長より大きなデータ長のトランザクションをサポートするかもしれないし、サポートしないかもしれない。そのときのワード長は、BUSWIDTH テンプレート・パラメタで与えられるか、もしくは他の値で与えられる。

g) ターゲットは与えられたデータ長のトランザクションが実行できなかった場合は、標準エラーレスポンスを返して、データ配列の内容を変更してはいけない。推奨レスポンス・ステータスは、’TLM_BURST_ERROR_RESPONSE’である。

h) データ長アトリビュートのデフォルト値は’0’で、無効な値である。故に、データ長・アトリビュートはインタフェース・メソッド・コールでトランザクション・オブジェクトが渡される前に明示的に設定されるべきである。

7.12 バイト･イネーブル･ポインタ･アトリビュート

a) set_byte_enable_ptr で、バイト･イネーブル配列へのポインタに引数で与えられた値を設定し、get_byte_enable_ptr でバイト･イネーブル･ポインタ・アトリビュートの値を返す。

b) バイト･イネーブル配列の各要素は次のように解釈される。’0’は、対応するバイトがディスエーブルされていることを示し、‘0xff’は対応するバイトがイネーブルされていることを示す。その他の値は未定義である。‘0xff’が選ばれた理由は、バイト・イネーブル配列がそのままマスクとして使えるように値が選ばれた。TLM_BYTE_DISABLED と TLM_BYTE_ENABLED の 2 つのマクロが用意されているからである。

c) バイト・イネーブルは、それぞれのビートのアドレスの増加分がビートの最上位バイトより大きいときや、バスにおいて選ばれたバイト･レーンに複数ワードを設定するバースト転送に使われる。より抽象度の高いレベルでは、データ配列に歯抜けがある状態でバースト転送を行う時に使用する。

d) 小さなパターンを繰り返し使用する場合や、データ配列全体をカバーする大きなパターンがある場合に、byte enable mask が定義される。「7.13 バイト･イネーブル長アトリビュート」を参照のこと。

e) バイト･イネーブル配列のエレメント数は、バイト･イネーブル長アトリビュートで与えられる。

f) バイト･イネーブル･ポインタが’0’（null）に設定されている場合は、その転送ではバイト･イネーブルは使用されず、バイト･イネーブル長は無視される。

g) バイト･イネーブルが使われる場合は、バイト･イネーブル･ポインタ･アトリビュートはイニシエータが設定する。バイト･イネーブル配列領域はイニシエータが確保して、バイト･イネーブル配列の内容もイニシエータが設定し、イネーブル配列の内容やバイト・イネーブル・ポインタは他のコンポーネントやターゲットで上書きされない。

h) バイト・イネーブル・ポインタが null でないときは、ターゲットは以下に定義された動作を実装するか、もしくは標準エラー応答を生成する。推奨エラーは’TLM_BYTE_ENABLE_ERROR_RESPONSE’である。

i) write コマンドでは、データ配列のディスエーブルされたバイトのデータは接続コンポーネントやターゲットでは無視されなければならない。ディスエーブルされたバイトは、接続コンポーネントやターゲットの動作に影響を与えないことが推奨される。無視されるこれらのバイトに対しては、イニシエータはどのような値を書き込んでも良い。

j) write コマンドで、ターゲットがトランザクション･データ配列からローカル配列へバイト単位のコピーをしている場合には、ターゲットは、汎用ペイロードのディスエーブル･バイトに対応するローカル配列のバイト値を書き換えてはならない。

k) read コマンドでは、データ配列中のディスエーブルされたバイトの値を、接続コンポーネントやターゲットで変更してはならない。イニシエータは、これらのディスエーブルされたバ

イトは接続コンポーネントやターゲットで変更されないものとみなすことができる。

l) read コマンドでは、ターゲットがローカル配列からトランザクション･データ配列へバイト単位でコピーする場合、ターゲットは、汎用ペイロードのディスエーブル･バイトに対応するローカル配列の値は無視すべきである。

m) もし、アプリケーションでこれらの規定を破らなければならない場合や、本書の汎用ペイロードの規定を破らなければならない場合は、新しくプロトコル型･クラスを作ることを推奨する。「7.2.2 tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義」を参照のこと 。

n) イネーブル･ポインタ･アトリビュートのデフォルト値は'0'（null）である。

7.13 バイト･イネーブル長アトリビュート

a) set_byte_enable_length は、引数で与えられた値をバイト･イネーブル長アトリビュートに設定し、get_byte_enable_length は現在の値を返す。

b) read/write コマンドに対して、ターゲットは、バイト･イネーブル長アトリビュートをバイト・イネーブル配列の要素数として解釈する。

c) バイト･イネーブル長アトリビュートの値はイニシエータが設定して、接続コンポーネントやターゲットは上書きしない。

d) データ配列のあるエレメントに適用するバイト・イネーブルは byte_enable_array_index=data_array_index%byte_enable_length で与えられる。つまり、バイト・イネーブル配列はデータ配列に繰り返し適用される。

e) バイト･イネーブル長がデータ配列の長さより大きい場合、余分なバイト･イネーブルは read/write コマンドに影響しない。ただし、拡張で使用することはできる。

f) バイト･イネーブル･ポインタが 0、つまりヌル・ポインタの場合、バイト・イネーブル長の値は接続コンポーネントやターゲットから無視される。もし、バイト･イネーブル･ポインタが'0'でなければ、バイト･イネーブル長は'0'ではない。

g) ターゲットが、指定されたバイト・イネーブル長を使ってトランザクションが実行できないときは、エラーを生成する。推奨は、TLM_BYTE_ENABLE_ERROR_RESPONSE である。

h) バイト･イネーブル長アトリビュートのデフォルト値は、'0'である。

7.14 ストリーミング幅アトリビュート

a) set_streaming_attribute は、引数の値をストリーミング幅アトリビュートに設定し、get_streaming_attribute は現在値を返す。

b) read/write コマンドに対して、ターゲットはストリーミング幅の現在値に従って動作する。

c) ストリーミングは、コンポーネントがデータ配列をどのように解釈するかに影響する。ストリームは、継続するビートで発生するデータ転送のシーケンスを構成し、各々のビートは汎用ペイロードのアドレス・アトリビュートで与えられる同じスタートアドレスを持つ。ストリーミング幅アトリビュートはストリームの幅を決定し、それは、各々のビートで転送されるバイト数である。すなわち、データ配列の構成にストリーミングは影響しない。

d) データ配列中のバイトは、汎用ペイロード・トランザクションにアクセスしているコンポーネント内のローカルアドレスの順序に対応している。最下位アドレスはアドレス・アトリビュートで与えられる。最上位アドレスは、address_attribute+streaming_width-1 で与えられる。ターゲットにコピーされる各々のバイトのアドレスは、各々のビートのスタートでアドレス・アトリビュートの値として設定される。

e) データ配列の実装としては、ストリーミングの幅を持つ単一のトランザクションは、一連のトランザクションと機能的には等価である。そのトランザクションは、元のトランザクションと同じアドレスで、元のストリーミング幅と同じデータ長・アトリビュートをもち、それぞれのビートで元のデータ配列の異なるサブセットをデータ配列としたものである。このサブセットは、元のデータ配列のデータ順を維持したものである。

f) 0 のストリーミング幅は無効である。もし、ストリーミング転送が必要でなかったなら、ストリーミング幅アトリビュート は、データ長・アトリビュートと同じがそれ以上の値にすべきである。

g) ストリーミング幅アトリビュートはデータ配列の長さやデータ配列に格納されているバイト数には影響しない。

h) ストリーミング幅がソケットの幅と異なる場合には、幅変換の問題が生じる。「7.17 エンディアン」を参照のこと。

i) ターゲットが、指定されたストリーミング幅を使ってトランザクションが実行できないときは、標準エラーを生成する。推奨は、TLM_BURST_ERROR_RESPONSE である。

j) ストリーミングは、バイト･イネーブルと組み合わせて使われるときがあり、通常は、ストリーミング幅はバイト・イネーブル長 と同じ値になる。バイト･イネーブル長がストリーミング幅の倍数ならば、各々のビートで異なるバイト数がイネーブルされることを意味する。

k) ストリーミング幅アトリビュートはイニシエータによって設定され、どんなインターコネクト・コンポーネントやターゲットにも上書きされない。

l) ストリーミング幅アトリビュートのデフォルト値は、0 である。

7.15 DMI許可アトリビュート

a) set_dmi_allowed は、DMI 許可アトリビュートに引数の値を設定し、get_dmi_allowed は、現在の値を返す。

b) DMI 許可アトリビュートは、ダイレクト・メモリ・ポインタを取得できる可能性をイニシエータに示すもので、DMI 経由で現在のトランザクションを実行した場合は、ターゲットが DMI 許可アトリビュートを'true'にセットする。4.2.9 DMI ヒントを使った最適化を参照のこと。

c) DMI 許可アトリビュートのデフォルト値は、false である。

7.16 応答ステータス・アトリビュート

a) set_response_status は、引数の値を応答ステータス・アトリビュートへ設定し、get_response_status は、現在値を返す。

b) 応答ステータスの現在値が TLM_OK_RESPONSE の場合にのみ、is_response_ok は'true'を返

す。 is_response_error は、応答ステータスの現在値が TLM_OK_RESPONSE と異なる場合にのみ'true'を返す。

c) get_response_string は現在の状態を文字列で返す。

d) 原則としてターゲットは汎用ペイロードの全ての機能を実装することが推奨されているが、そのようになっていない場合はエラーを返す。「7.16.1 標準エラー応答」を参照のこと。

e) 応答ステータス･アトリビュートは、イニシエータが TLM_INCOMPLETE_RESPONSE を設定して、ターゲットで上書きすることも上書きしないこともできる。応答ステータス・アトリビュートは、どんなインターコネクト・コンポーネントでも上書きされない。TLM_INCOMPLETE_RESPONSE の値は、通常インターコネクトとして動作するコンポーネントが応答を返す場合のように、ターゲットとして動作するコンポーネントがコマンドを実行しようとしなかったことを示すのに使われる。

f) ターゲットは処理が成功した場合には、応答ステータス･アトリビュートに TLM_OK_RESPONSE を設定する。そうでなければ、ターゲットは以下のテーブルにある 6 つのエラー応答のいずれかを設定する。ターゲットは、エラーの原因に応じて適切なエラーを選ぶべきである。

Error response	Interpretation
TLM_INCOMPLETE_RESPONSE	ターゲットはコマンドを実行しようとしなかった
TLM_ADDRESS_ERROR_RESPONSE	ターゲットはアトリビュートで与えられた値では動作できない、もしくはアドレス範囲を超えている
TLM_COMMAND_ERROR_RESPONSE	ターゲットは与えられたコマンドを実行できない
TLM_BURST_ERROR_RESPONSE	ターゲットは与えられたデータ長、もしくはストリーミング幅では動作できない
TLM_BYTE_ENABLE_ERROR_RESPONSE	ターゲットは与えられたバイト･イネーブルでは動作できない
TLM_GENERIC_ERROR_RESPONSE	その他のエラー

g) ターゲットがエラーを検出しても、特定のエラーを選ぶことが出来ないときは、TLM_GENERIC_ERROR_RESPONSE を設定してよい。

h) 応答ステータス･アトリビュート のデフォルト値は、TLM_INCOMPLETE_RESPONSE である。

i) TLM_IGNORE_COMMAND の場合、トランザクションを受け取り、リードもしくはライト・コマンドを実行するであろうターゲットは TLM_OK_RESPONSE を返すべきである。そうでなければターゲットは、リードもしくはライト・コマンドに適用したのと同じ基準でエラー応答を設定することを選択することができる。例えば、バイト・イネーブルをサポートしないターゲットは、TLM_BYTE_ENABLE_ERROR_RESPONSE を返してもよい（必須ではない）。

j) 汎用ペイロードの拡張もしくはフェーズ拡張によって、ターゲットで無視可能拡張に関するルールに従うならば異なる応答ステータスを返すことができる。言い換えれば、基本プロトコル内で、拡張はコマンドを失敗にすることもできるし、ターゲットは拡張を無視し、コマンドを成功させることもできる。

k) ターゲットは、トランザクションのライフタイムの適当なところで、イニシエータは応答ステータス･アトリビュートを設定しなければならない。ブロッキング・トランスポート・インタフェースの場合、b_transport から制御が戻る前に設定する必要がある。ノンブロッキング・インタフェースと基本プロトコルでは、BIGIN_RESP フェーズを設定する前か、TLM_COMPLETE を返す前になる。

l) イニシエータは、BEGIN_RESP まで、もしくはトランザクションが完了するまで、受け取るトランザクションの応答ステータス･アトリビュートを常にチェックすることを推奨する。前もって、TLM_OK_RESPONSE であることが分かっている場合には、イニシエータは、応答ステータス･アトリビュートを無視することを選ぶかもしれない。たとえば、イニシエータが、常に TLM_OK_RESPONSE を返すターゲットにのみ接続していることを前もって知っている場合。ただし、常にそうであるとは限らないので、イニシエータは自らの責任で、応答ステータス･アトリビュートを無視することになる。

m) ターゲットはエラー応答を選択する際、若干の自由度を持つ。例えば、コマンドとアドレス・アトリビュートがエラーだった時、ターゲットは TLM_ADDRESS_ERROR_RESPONSE、TLM_COMMAND_ERROR_RESPONSE、あるいは TLM_GENERIC_ERROR_RESPONSE のいずれも正しい設定である。応答ステータスをその動作の決定に使うとき、イニシエータは単独でエラー応答の 6 つの区別に頼ってはいけない、しかしイニシエータはユーザのために表示される診断メッセージの内容を決めるために応答ステータスを使うことができる。

7.16.1 標準エラー応答

ターゲットが汎用ペイロードのトランザクションを受け取ったときには、次のいずれか一つのみの処理をすべきである。

a) 汎用ペイロード・アトリビュートのセマンティクスや、モデル化されたコンポーネントの公にドキュメント化されているセマンティックスに従って、トランザクションで示されるコマンドを実行して、レスポンス・ステータスを TLM_OK_RESPONSE に設定する。

b) 先に示した５つのエラーメッセージのうちの一つを汎用ペイロードのレスポンス・ステータスとして設定する。

c) SystemC の標準レポートハンドラを使って、コマンド実行失敗、コマンド無視を含む SystemC の４つの重要度レベルのひとつでレポートを生成して、レスポンス・ステータスを TLM_OK_RESPONSE に設定する。

ターゲットは上記の 3 つのうちのいずれか一つを実行することが推奨されるが、実装方法は、この推奨を押しつけられるものではない。

汎用ペイロード以外のトランザクション型に対しても、上記と同様の応答を推奨する。すなわち、コマンドを期待されるとおりに実行するか、トランザクション・アトリビュートを使ってエラー応答を生成するか、もしくは SystemC レポートを生成する。しかしながら、処理の詳細とエラー応答メカニズムはこの標準の範囲外である。

上記の a)を満たすための条件は、コンポーネントのユーザが見ることのできるターゲット・コンポーネントの期待されている動作によって決まる。汎用ペイロードの･アトリビュートはメモリ・マップド･バスにおける慣例的なセマンティックで規定されているが、それは、ターゲットが RAM のような動作をすることを必ずしも想定するものではない。現実には、多くの異なった状況があ

る。

i. ターゲットが write コマンドと read コマンドの両方をサポートしているメモリ・マップド・レジスタを持っていて、write コマンドがターゲットのステータスを変更しても、write コマンドの後の read コマンドが直前に書き込まれた値ではなく、ターゲットのステータスで決まる値を返すことがある。もしこれが、コンポーネントに期待される正常な動作であれば、a)でカバーされる。

ii. ターゲットの write コマンドがデータ・アトリビュートを無視してビットを設定するように実装されていることがある。もしこれが、期待されている正常な動作であれば、a)でカバーされる。

iii. ROM は、応答ステータス・アトリビュートを使ったエラー応答をイニシエータに出すことなく無視することがある。write コマンドはターゲットの状態を変更せず、無視されているが、ターゲットは、少なくとも SC_INFO か SC_WARNING の重要度レベルの SystemC レポートを生成すべきである。

iv. ターゲットは read コマンドを実行する write コマンドや、write コマンドを実行する read コマンドを実装してはならない。これは汎用ペイロードの基本的な使い方に違反するものである。

v. ターゲットは、汎用ペイロードのルールに対応はしているが、副作用を持つ read コマンドを実装するかもしれない。これは a)でカバーされる。

vi. アドレス指定可能なレジスタ・ファイルで構成したメモリ・マップド・レジスタを持つターゲットが、アドレス範囲外の write コマンドを受け取った場合には、トランザクションの応答ステータス・アトリビュートに TLM_ADDRESS_ERROR_RESPONSE を設定するか、SystemC レポートを生成すべきである。

vii. バスモニター・ターゲットが、バスの物理範囲を越えるアドレスのトランザクションを受け取った時には、a)の対応として、エラーの可能性があるトランザクションを後処理のためにログ記録して、b)や c)によるエラー生成はしないのがよい。代替として、c)に基づくレポートを生成しても良い。

言い換えれば、a)、b)、c)の選択は、最終的に具体的な判断はケースバイケースであるが、汎用ペイロードに対しては、いずれかの対応をすることが明確なルールとなっている。

例

```
// Showing generic payload with command, address, data, and response status

// The initiator
void thread() {
  tlm::tlm_generic_payload trans;                  // Construct default generic payload
  sc_time delay;
  trans.set_command(tlm::TLM_WRITE_COMMAND);       // A write command
  trans.set_data_length(4);                        // Write 4 bytes
  trans.set_byte_enable_ptr(0);                    // Byte enables unused
```

```
   trans.set_streaming_width(4);                                    // Streaming unused
   for (int i = 0; i < RUN_LENGTH; i += 4) {                        // Generate a series of transactions
      int word = i;
      trans.set_address(i);                                         // Set the address
      trans.set_data_ptr( (unsigned char*)(&word) );                // Write data from local variable 'word'
      trans.set_dmi_allowed(false);                                 // Clear the DMI hint
      trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE ); // Clear the response status

      init_socket->b_transport(trans, delay);

      if (trans.is_response_status() <= 0)                          // Check return value of b_transport
         SC_REPORT_ERROR("TLM-2", trans.get_response_string().c_str());
      ...
}
...
// The target
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{
   tlm::tlm_command          cmd = trans.get_command();
   sc_dt::uint64             adr = trans.get_address();
   unsigned char*            ptr = trans.get_data_ptr();
   unsigned int              len = trans.get_data_length();
   unsigned char*            byt = trans.get_byte_enable_ptr();
   unsigned int              wid = trans.get_streaming_width();

   if (adr+len > m_length) {                                        // Check for storage address overflow
      trans.set_response_status( tlm::TLM_ADDRESS_ERROR_RESPONSE );
      return;
   }
   if (byt) {                                                       // Target unable to support byte enable attribute
      trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
      return;
   }
   if (wid < len) {                                                 // Target unable to support streaming width attribute
      trans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
      return;
   }
   if (cmd == tlm::TLM_WRITE_COMMAND)                               // Execute command
      memcpy(&m_storage[adr], ptr, len);
   else if (cmd == tlm::TLM_READ_COMMAND)
      memcpy(ptr, &m_storage[adr], len);
```

```
    trans.set_response_status( tlm::TLM_OK_RESPONSE );      // Successful completion
}

// Showing generic payload with byte enables

// The initiator
void thread() {
    tlm::tlm_generic_payload trans;
    sc_time delay;
    static word_t byte_enable_mask = 0x0000ffffful;                    // MSB..LSB regardless of host-endianness
    trans.set_command(tlm::TLM_WRITE_COMMAND);
    trans.set_data_length(4);
    trans.set_byte_enable_ptr( reinterpret_cast<unsigned char*>( &byte_enable_mask ) );
    trans.set_byte_enable_length(4);
    trans.set_streaming_width(4);
    ...
    ...
// The target
virtual void b_transport(
    tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{
    tlm::tlm_command            cmd = trans.get_command();
    sc_dt::uint64               adr = trans.get_address();
    unsigned char*              ptr = trans.get_data_ptr();
    unsigned int                len = trans.get_data_length();
    unsigned char*              byt = trans.get_byte_enable_ptr();
    unsigned int                bel = trans.get_byte_enable_length();
    unsigned int                wid = trans.get_streaming_width();

    if (cmd == tlm::TLM_WRITE_COMMAND) {
        if (byt) {
            for (unsigned int i = 0; i < len; i++)              // Byte enable applied repeatedly up data array
                if ( byt[i % bel] == TLM_BYTE_ENABLED )
                    m_storage[adr+i] = ptr[i];                  // Byte enable [i] corresponds to data ptr [i]
        }
        else
            memcpy(&m_storage[adr], ptr, len);                  // No byte enables
    } else if (cmd == tlm::TLM_READ_COMMAND) {
        if (byt) {                                              // Target does not support read with byte enables
            trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
            return;
```

```
        }
      else
        memcpy(ptr, &m_storage[adr], len);
    }
  trans.set_response_status( tlm::TLM_OK_RESPONSE );
}
```

7.17 エンディアン

7.17.1 イントロダクション

イニシエータとターゲット間でデータ転送するために汎用ペイロードを使う時、ホストマシンのエンディアン（ホスト・エンディアン）とイニシエータのエンディアンとモデル化されるターゲット（モデル化エンディアン）の両方が関連する。この章では汎用ペイロードを使用しているイニシエータとターゲット間の相互利用性を確保するための規則を定義する。すなわち、汎用ペイロードのデータ配列とバイト・イネーブル配列の構造に特に関係する。しかしながら、ここで示す規則は汎用ペイロードの直接スコープを超えてエンディアンをモデル化するいくつかの選択肢に影響することがある。

TLM-2.0 アプローチの基本原則は汎用ペイロード・データ配列の構造がイニシエータ、インターコネクト・コンポーネントまたはターゲット内部で局所的に知りえる情報だけに依存する。特に、これはソケット、ホストマシンのエンディアンとモデル化されたコンポーネントの幅に依存する。

汎用ペイロードの構造とエンディアンのアプローチは、ある共通システムのシナリオにおいてシミュレーション速度を最大限に引き出すために選択された。これから述べる規則は、汎用ペイロードの構造を示している。そして、これはモデル化されたシステムの構造から独立である。例えば、汎用ペイロード内部の「ワード」は必ずしもモデル化されたアーキテクチャ内部のどんな「ワード」の内部表現とは対応しない。

マクロ視点では、主原則は汎用ペイロードが MSB 対 MSB と LSB 対 LSB で結ばれたエンディアン混在システムにおけるコンポーネントを仮定している。換言すると、1 ワードが異なったエンディアンのコンポーネント間で転送されるなら、MSB ... LSB 関係が保存される。しかし各コンポーネント内部で見られる各バイトのローカルアドレスはアドレス・スウィズリング（address swizzling）と一般的に呼ばれる変換を用いて必ず変換される。これは、モデル化されたシステムと TLM2.0 モデルの両方に当てはまる。一方、もしエンディアン混在システムが、各コンポーネント内部でローカルアドレスが変化しないように接続されていれば、(すなわち、各バイトがどんなコンポーネントから参照されても同じアドレスを持つ時)、明示的バイトスワップが TLM2.0 モデルに挿入される必要がある。

汎用ペイロード配列に関する相互運用性を達成するためには、この節で与えられる規則に従うだけで良い。Helper 関数群がデータ配列の構造を支援するために準備されている。「7.19 エンディアン変換用」を参照。

7.17.2 ルール

a) 以下の規則では、汎用ペイロード・データ配列を data と示し、汎用ペイロードバイトイネーブル配列を be と示す。

b) インタオペラビリティ・レイヤの標準ソケットクラス（または、これらの派生クラス）を使うとき、データとバイト・イネーブルの配列の中身は、ローカルに送受信されるトランザクションを通すソケットのテンプレート・パラメタ BUSWIDTH を使って解釈されなければならない。有効であるワード長は、(BUSWIDTH+7)/8 バイトと計算されなければならない。これを以下の規則で W と示す。

c) この W 量は、データ配列ワードの長さを定義する。ここのワードは、ローカルソケットを通して 1 ビート送ることのできるデータの量である。データ配列は 1 つのワード、パートワード、または一連のワード、パートワードを含んで良い。データ配列の中の最初または最後のワードだけが、パートワードであってよい。この記述は、モデル化されたアーキテクチャの組織ではなく、汎用ペイロードの内部組織を参照する。

d) もし与えられた汎用ペイロード・トランザクション・オブジェクトが異なる幅のソケットを通るなら、データ配列ワード長は、異なるソケットの見地から計算するとき異なって現れる。（以下の幅変換の記述を参照）

e) データ配列の個々のワードの中のバイトの順は、ホストのエンディアンでなければならない。つまり、リトルエンディアンのホストプロセッサでは、与えられたワードの中で data[n]は、data[n+1]よりも小さい桁であり、ビッグエンディアンのホストプロセッサでは、data[n]は data[n+1]よりも大きな桁である。

f) データ配列の中のワードの境界は、アドレスが揃っていなければならない。つまり、ワード長 W の整数倍のアドレスに揃わなければならない。しかしながら、アドレス・アトリビュートもデータ長アトリビュートもワード長の倍数であることを要求されない。したがって、データ配列の最初と最後のワードはパートワードである可能性がある。

g) データ配列の中のワードの順は、モデル化されたシステムのメモリマップのアドレスによって決められなければならない。ストリーミング幅アトリビュートの値より小さい配列インデックス値のために、連続するワードのローカルアドレスは、増加順でなければならない。そして、（続くパートワードを除いて）address_attribute – (address_attribute % W) + NW に等しくなければならない。ここで、N は負でない整数、%は割り算の余りを示す。

h) 一方で、配列インデックスの増加順でデータ配列の要素を一覧するために表記{a,b,c,d}を使い、N 番目のワードの最小桁のバイトを示す LSB_N を使うと、リトルエンディアンのホスト上ではバイトは{…, MSB_0, LSB_1, …, MSB_1, LSB_2, …}の順で蓄えられ、ビッグエンディアンのホスト上では{…, LSB_0, MSB_1, …, LSB_1, MSB_2, …}である。そこでは、個々の全ワード中のバイト数は W であり、バイトの総数は data_length アトリビュートによって与えられる。

i) 以上の規則は、実質的にイニシエータとターゲットは LSB-to-LSB、MSB-to-MSB に接続されることを意味する。規則は、もとのエンディアンが何であってもイニシエータとターゲットの大半がホストのエンディアンを使ってモデル化される場合には最適なシミュレーション速度が得られるように選ばれた。これはまた、「算術モード」として知られている。

j) アプリケーションは、ホストのエンディアンに依存しないことが強く推奨される。つまり、どちらのエンディアンのホスト上で動くときも同じ動作をするようにモデル化しなければならない。これは、ヘルパー関数や条件コンパイルの使用を要求するかもしれない。

k) もしイニシエータかターゲットがその元のエンディアンでモデル化され、それがホストのエンディアンと異なったら、汎用ペイロードのデータ配列から/へデータを転送した場合、ワー

ドの中のバイト順を入れ替える必要がある。ヘルパー関数は、この目的のために作られている。

l) たとえば、以下の SystemC コードの一部を考える。ここでは、汎用ペイロードのデータ配列を初期化する値として 0xAABBCCDD の文字列が使われている。

```
int data = 0xAABBCCDD;
trans.set_data_ptr( reinterpret_cast<unsigned char*>( &data ) );
trans.set_data_length(4);
trans.set_address(0);
socket->b_transport(trans, delay);
```

m) C++コンパイラは、ホストのエンディアンで文字列 0xAABBCCDD を解釈する。どちらの場合も、MSB の値が 0xAA で LSB の値が 0xDD である。このことからみて、このコードは正しくホストのエンディアンに依存しない。しかしながら、4 バイトの配列インデックスはホストのエンディアンに依存する点が異なる。リトルエンディアンのホストでは、data[0] = 0xDD で、ビッグエンディアンのホストでは、data[0] = 0xAA である。モデル化されたシステムのローカルアドレスと配列インデックスの間の整合性は、モデル化されたエンディアンとホストのエンディアンが等しいかどうかに依存する。

```
リトルエンディアンモデルとリトルエンディアンホスト: data[0] is 0xDD and local address 0
ビッグエンディアンモデルとリトルエンディアンホスト: data[0] is 0xDD and local address 3
リトルエンディアンモデルとビッグエンディアンホスト: data[0] is 0xAA and local address 3
ビッグエンディアンモデルとビッグエンディアンホスト: data[0] is 0xAA and local address 0
```

n) 上記のようなコードは、リトルエンディアンかビッグエンディアンのどちらを使うホストコンピュータでもポータブルではない。その場合、そのコードはバイトアドレスだけを使って汎用ペイロードのデータ配列をアクセスするように書き直さなければならない。

o) リトルエンディアンとビックエンディアンのモデルが与えられた汎用ペイロードのトランザクションを解釈するとき、定義によりワードのどちらが MSB か LSB かモデル間で同意するが、それぞれはワードのバイトにアクセルするために異なるローカルアドレスを使う。

p) データ長アトリビュートとアドレス・アトリビュートのどちらも W の整数倍であることを要求されない。しかしながら、アドレスとデータ長をワードの固まりで並べて W を 2 の累乗にすることは、データ配列へのアクセスをかなり単純にする。その点を強調すると、汎用ペイロードのトランザクションは、48 ビットソケットの真ん中の 3 バイトで示されたアドレスとデータ長を持つと申し分ない。もし特別なターゲットが与えられたアドレス・アトリビュートかデータ長をサポートできないのであれば、標準エラー応答を生成しなければならない。「7.16 応答ステータス・アトリビュート」を参照。

q) たとえば、リトルエンディアンのホストで W = 4、 address = 1、data_length = 4 ならば、最初のワードはアドレス 1…3 の 3 バイトに含まれ、2 番目のワードはアドレス 4 の 1 バイトに含まれる。

r) 単一バイトとパートワードの転送は、並ばないアドレスを使って表現されるかもしれない。たとえば、W = 8、address = 5、data = {1,2}が与えられると、ローカルアドレス 5 と 6 の 2 バイトはエンディアンに依存した順番でアクセスされる。

s) パートワードと並ばない転送は、常にバイト・イネーブルといっしょに W の整数倍を使っ

て表すことができる。これは与えられた転送はいくつかの等しく正しい汎用ペイロードの表現を持つことを意味する。たとえば、リトルエンディアンのホストとリトルエンディアンのイニシエータがあるとすると、

address = 2, W = 4, data = {1}は
address = 0, W = 4, data = {x, x, 1, x}, and be = {0, 0, 0xff, 0}と等しい。
address = 2, W = 4, data = {1,2,3,4} は、
address = 0, W = 4, data = {x, x, 1, 2, 3, 4, x, x}, and be = {0, 0, 0xff, 0xff, 0xff, 0xff, 0, 0} と等しい。

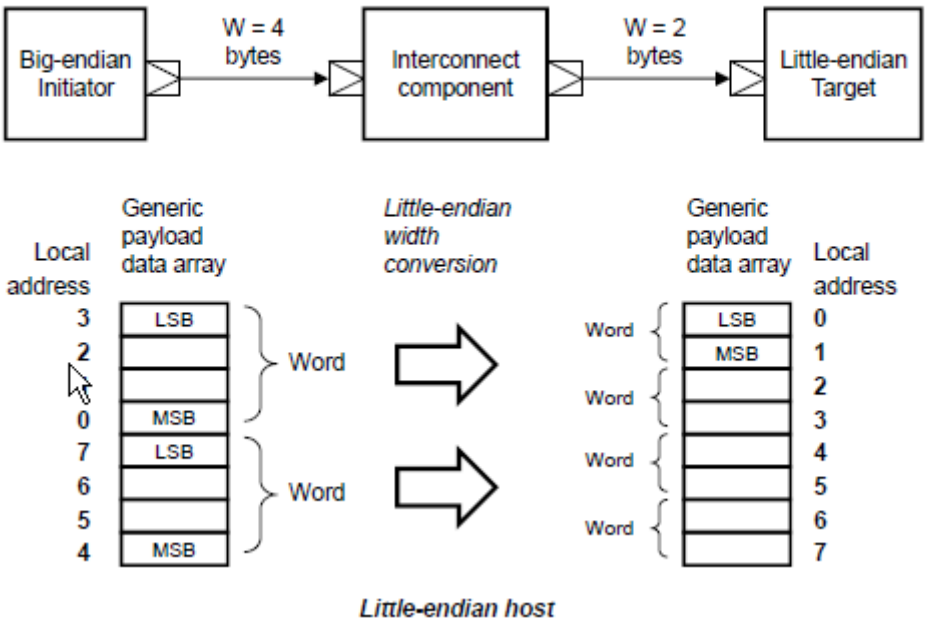
t) パートワードのアクセスのためにバイト・イネーブルを使うことの必要性はエンディアンに依存する。たとえば、最初のワードのすべてと第 2 のワードの LSB をアクセスする意図は、リトルエンディアンのホストでは次のように表現する。

address = 0, W = 4, data = {1,2,3,4,5}
ビッグエンディアンのホストでは、
address = 0, W = 4, data = {4,3,2,1,x,x,x,5}, be = {0xff, 0xff, 0xff, 0xff, 0, 0, 0, 0xff }

u) 2 つのソケットが接続されるとき、それらは同じ BUSWIDTH を持つことが必要である。しかしながら、トランザクションはターゲット・ソケットから異なるバス幅のイニシエータ・ソケットへ送ることができる。この場合、汎用ペイロードのトランザクションの幅変換は考慮されなければならない。幅の広い方のソケットの小さい桁のバイトと大きい桁のバイトのどちらを最初に取り出すかに依存して、どんな幅の変換もそれ自身の内在するエンディアンがある。

Width conversion

Figure 12



Little-endian host

v) 幅変換のために選ばれたエンディアンがホストのエンディアンと一致するとき、幅変換は事実上自由である。それは単一のトランザクション・オブジェクトは修正なしに socket-to-socket からフォワードされることができることを意味する。そうでなければ、2 つの分離した汎用

ペイロードのトランザクション・オブジェクトが要求される。図 12 で 4 バイトソケットと 2 バイトソケットの間の幅変換はホストのエンディアンを使い、個々のワードの中でホスト・エンディアンのバイト順で保つ間に小さい桁のバイトを低いアドレスに移動する。イニシエータとターゲットは共にデータ配列の中で同じバイトの順序でアクセスするが、それらのローカルアドレスのスキームはまったく異なる。

w) もし幅変換が狭いソケットから広いソケットに対して実行されるなら、送信するトランザクションのアドレスの整列 (alignment) を実行すべきかどうか選択がなされなければならない。アドレスの整列を実行することは常に新しい汎用ペイロードのトランザクション・オブジェクトの構築を必要とする。

x) 同様な幅の変換の問題は、ストリーミング幅アトリビュートがゼロでなく W と異なるときに生じる。ホストのエンディアンと幅変換で望まれるエンディアンに依存するデータ配列のデータを読む順序を決めなければならない。

7.18 ホスト・エンディアンを決定するHelper関数

7.18.1 イントロダクション

ヘルパー関数群がホストマシンのエンディアンを決めるために提供されている。これらは、汎用ペイロード・データ配列を作ったり、解釈したりする時に使うことを目的としている。

7.18.2 クラス定義

```
namespace tlm {
  enum tlm_endianness {
    TLM_UNKNOWN_ENDIAN, TLM_LITTLE_ENDIAN, TLM_BIG_ENDIAN };
  inline tlm_endianness get_host_endianness(void);
  inline bool host_has_little_endianness(void);
  inline bool has_host_endianness(tlm_endianness endianness);
} // namespace tlm
```

7.18.3 ルール

a) get_host_endianness 関数は、ホストのエンディアンを返す。

b) host_has_little_endianness 関数は、ホストがリトルエンディアンのときに限り true 値を返す。

c) has_host_endianness 関数は、ホストのエンディアンが引数で示された値と同じ場合のときに限り true 値を返す。

d) もしホストがリトルエンディアンでもビッグエンディアンでもない場合、以上 3 つの関数から返される値は未定義である。

7.19 エンディアン変換用ヘルパー関数

7.19.1 イントロダクション

汎用ペイロード・データ配列の構造を統括する規則は十分定義されているし、多くの簡単なケースでは、データ配列を作り解釈するためのホストとは独立なC＋＋コードを書くことは明確な作

業となる。しかしながら、規則はモデル化されたコンポーネントのエンディアンとホスト・エンディアン間の関係に依存する。そこでホストとは独立なコードを書くことは、ソケット幅と異なるアラインされないアドレスとデータ・ワード幅を含む場合では極めて複雑になる。ヘルパー関数群はこの作業を支援するために準備されている。

エンディアンに関して、相互利用性はエンディアン規則だけに依存する。ヘルパー関数は必ずしも相互利用性に重要ではない。

エンディアン変換関数の背後にあるモチベーションは、ホスト・エンディアンとは関係なく一度書かれたイニシエータのための汎用ペイロード・トランザクションを作るＣ＋＋コードを可能にすること、さらに一度の関数呼び出しで、ホスト・エンディアンにマッチするように変換されたトランザクションを持つことである。各変換関数は、既存の汎用ペイロード・トランザクションを直接修正する。変換関数は対になっており、to_hostendian 関数と from_hostendian 関数はいつも一緒に使われる。to_hostendian 関数はトランスポート・インタフェースを介してとトランザクションを送る前にイニシエータにより呼ばれる。from_hostendian 関数は受取った後に呼ばれる。

４つの関数ペアが提供されている。もっとも一般的で強力な_generic、制限された場合だけに使える_word、_aligned、_single である。_generic 関数による変換は計算が重いので、使えるところには他の関数を使うべきである。

変換関数は、算術モードとバイト・オーダー・モードの両方を含むような、よくある例も扱えるくらいの柔軟性がある。算術モードは、モデリングされているコンポーネントのエンディアンに関わらず、算術演算を実行するときにコンポーネントがデータ・ワードをホストのエンディアンフォーマットで格納する。バイト・オーダー・モードではコンポーネントはホストのエンディアンを無視して、アドレス順序で配列に格納する。シミュレーション速度のために、算術モードが推奨される。バイト・オーダー・モードは、データを汎用ペイロード間でコピーするバイト入れ替えで必要とされる。

変換関数はデータ・ワードコンセプトが使われている。これは TLM-2.0 汎用ペイロード・データ配列のソケット幅やワード幅のいずれとも独立である。データ・ワードは(モデリングされているコンポーネントのエンディアンに関係なく)コンポーネント・モデルでホスト・エンディアンの順序でバイト格納するレジスタを表します。データ・ワード幅がソケット幅と異なるとき、hostendian 関数はエンディアン変換を実行しなければいけない場合がある。データ・ワードが 1 バイトだけ広いなら、hostendian 関数はバイト・オーダー・モードと効果的に変換をする。

まとめると hostendian 変換関数でとられるアプローチは、イニシエータのコードをあたかもホストコンピュータのエンディアンがモデリングされたコンポーネントのエンディアンとマッチするかのように書く、その一方でそれぞれのデータ・ワードは実際のホスト・エンディアンを保つ。ホストマシンのワード長より広いデータ・ワードには、ホスト・エンディアン順序の配列を使う。そして、ホスト・エンディアンがモデルのエンディアンと異なるときは、単純に hostendian 変換関数をコールする。

7.19.2 クラス定義

```
namespace tlm {
  template<class DATAWORD>
  inline void tlm_to_hostendian_generic(tlm_generic_payload *, unsigned int );
  template<class DATAWORD>
```

```
  inline void tlm_from_hostendian_generic(tlm_generic_payload *, unsigned int );
  template<class DATAWORD>
  inline void tlm_to_hostendian_word(tlm_generic_payload *, unsigned int);
  template<class DATAWORD>
  inline void tlm_from_hostendian_word(tlm_generic_payload *, unsigned int);
  template<class DATAWORD>
  inline void tlm_to_hostendian_aligned(tlm_generic_payload *, unsigned int);
  template<class DATAWORD>
  inline void tlm_from_hostendian_aligned(tlm_generic_payload *, unsigned int);
  template<class DATAWORD>
  inline void tlm_to_hostendian_single(tlm_generic_payload *, unsigned int);
  template<class DATAWORD>
  inline void tlm_from_hostendian_single(tlm_generic_payload *, unsigned int);
  inline void tlm_from_hostendian(tlm_generic_payload *);
} // namespace tlm
```

7.19.3 ルール

a) to_hostendian を含む形の関数の第 1 引数は、トランスポート・インタフェースを通して送られたのであればバリッドである汎用ペイロードトランザクション・オブジェクトへのポインタでなければならない。その関数は、トランザクション・オブジェクトを構築し初期化した後でかつ、インタフェース・メソッド・コールへそれを送る前にだけ呼び出されなければならない。

b) from_hostendian を含む形の関数の第 1 引数は、to_hostendian に以前に送った汎用ペイロードトランザクション・オブジェクトへのポインタでなければならない。その関数は、イニシエータが与えられたトランザクションの応答を受け取るか、そのトランザクションが完了したときにだけ、呼び出されなければならない。その関数は、トランザクションとその配列を修正してよいので、トランザクション・オブジェクトのライフタイムの終わりにだけ呼び出される。

c) もし to_hostendian 関数が与えられたトランザクションのために呼び出されたら、対応する from_hostendian 関数もまた同じテンプレートと関数の引数で呼び出されなければならない。あるいは、tlm_from_hostendian(tlm_generic_payload*)関数が与えられたトランザクションのために呼び出されて良い。この関数は、テンプレートと関数の引数の値を取り出すために（無視可能拡張として）トランザクション・オブジェクトに蓄えられた追加コンテキスト情報を使う。この関数の実行は少し遅い。

d) hostendian 関数の第 2 引数は、バイトで表されたトランザクションが通過するローカルソケットの幅である。これは、ローカルソケットに関係した汎用ペイロードのデータ配列のワード長に等しい。これは、2 の累乗である。

e) hostendian 関数のテンプレートの引数は、エンディアン変換のための内部イニシエータのデータ・ワード型である。sizeof(DATAWORD)演算は、データ・ワードの幅をバイト数で調べるために使われ、DATAWORD 型の代入演算は、コピーにおいて使われる。sizeof(DATAWORD)は 2 の累乗である。

f) to_hostendian の実装は、コンテキスト情報を蓄えるために汎用ペイロードのトランザクション・オブジェクトに拡張を加える。これは、from_hostendian を呼ぶ前に to_hostendian は一度だけ呼び出すことができることを意味する。

g) 以下の制約は hostendian 関数のすべてのペアに共通である。整数倍の用語は、1x、2x、3x、… 等を意味する。

- ソケット幅は 2 の累乗でなければならない。
- データ・ワード幅は 2 の累乗でなければならない。
- ストリーミング幅・アトリビュートはデータ・ワード幅の整数倍でなければならない。
- データ長アトリビュートはストリーミング幅アトリビュートの整数倍でなければならない。

h) hostendian_generic 関数は、それ以上の特別な制約はない。特に、それらはバイト・イネーブル、ストリーミングと揃わないアドレスとデータ幅をサポートする。

i) 残りの関数のペア、すなわち hostendian_word、hostendian_aligned、hostendian_single はすべて以下の追加制約を受ける。

- データ・ワード幅は、ソケット幅より大きくてはならない。その結果、ソケット幅は、データ・ワード幅の 2 の累乗倍でなければならない。
- ストリーミング幅アトリビュートはデータ幅アトリビュートと等しくなければならない。つまり、ストリーミングはサポートされない。
- バイト・イネーブルの範囲は、データ・ワード幅より細かくてはならない。つまり、与えられたデータ・ワードのバイトは、すべてがイネーブルかディセーブルかのどちらかでなければならない。
- もしバイト・イネーブルがあるなら、バイト・イネーブル長アトリビュートはデータ長アトリビュートと等しくなければならない。

j) hostendian_aligned 関数だけが以下の追加制約を受ける。

- アドレス・アトリビュートは、ソケット幅の整数倍でなければならない。
- データ長アトリビュートは、ソケット幅の整数倍でなければならない。

k) hostendian_single 関数だけが以下の追加制約を受ける。

- データ長アトリビュートはデータ・ワード幅に等しくなければならない。
- データ配列はデータ・ワードの固まりをまたいではならない。その結果、データ配列はソケットの固まりをまたいではならない。

7.20 汎用ペイロードの拡張

7.20.1 イントロダクション

拡張メカニズムは汎用ペイロードに組み込まれていて、汎用ペイロード無しに使うことを意図していない。その目的は、汎用ペイロードにアトリビュートの追加を許すことである。

拡張は無視可能（ignorable）か無視不可能（no-ignorable）、必須（mandatory）か非必須（non-mandatory）のどちらかである。

- 無視可能拡張

  無視可能であることは、その拡張を加えられたコンポーネント以外のどのコンポーネントもその拡張がないかのように動作することが許されることである。結果として、無視可能拡張が追加されたコンポーネントは、その拡張の存在に反応する他のコンポーネントに依存することはできないし、無視可能拡張を受け取るコンポーネントはその拡張を認識するほかのコンポーネントに依存することはできない。この定義は、汎用ペイロード拡張と拡張されたフェーズに同様に適応される。コンポーネントは、無視可能拡張がないことでフェールしたりエラー応答をしたりしてはならない。この意味で無視可能拡張は非必須拡張でもある。コンポーネントは、無視可能拡張があることでフェールまたはエラー応答を返して良いし、その拡張を無視するかどうか選ぶことができる。一般に、無視可能拡張は、与えられた拡張がない場合にデフォルト値を使うことで内部接続のコンポーネントやターゲットを通常に動作させることができるような明白で安全なデフォルト値が存在すると考えて良い。1 つの例は、デフォルトは最も低いレベルである、トランザクションに関連付けられた特権レベルである。無視可能拡張は、補助の、サイドバンドの、シミュレーションに関連した情報やメタデータを送るために使うことができる。たとえば、特別なトランザクション識別子、トランザクションが作成されたときの wall-time、診断用ファイル名などである。無視可能拡張は、基本プロトコルで許されている。

- 無視不可能で必須の拡張

  無視不可能拡張は、そのトランザクションを受け取るすべてのコンポーネントが受け取ったら処理しなければならない拡張である。必須拡張は存在することを要求される拡張である。無視不可能で必須の拡張は、特別なプロトコルの詳細をモデル化するために汎用ペイロードを扱うときに使うことができる。無視不可能で必須の拡張は、新しいプロトコル・トレイツ･クラスの定義を必要とする。

7.20.2 原理

拡張メカニズムの背後にある原理は、2 つある。1 つめは、アダプタやブリッジを使わず直接相互に接続することを許すような、同じプロトコル・トレイツ・クラスで特化された類の汎用ペイロードのコア・アトリビュート・セットを運ぶ TLM-2.0 ソケットが許されていることである。2 つめは、同じ汎用ペイロートと拡張メカニズムに基づく異なるプロトコルの間での簡単なアダプタを使用することを許すことである。拡張メカニズムがない場合、汎用ペイロードに新しいアトリビュートを加えるためには複数のアダプタを義務付ける新しいトランザクション・クラスの定義が必要になる。拡張メカニズムは、汎用ペイロードに変化を与えることができ、異なるプロトコルを運ぶソケットを縦断するために必要になる多くのコーディングを減らすことができる。

7.20.3 拡張ポインタとオブジェクトとトランザクション・ブリッジ

拡張は tlm_extension クラスから派生した型のオブジェクトである。汎用ペイロードは拡張オブジェクトへのポインタ配列を含む。すべての汎用ペイロードのオブジェクトは拡張のすべての型の単体のインスタンスを運ぶことができる。

拡張を指し示すポインタ配列はすべての登録された拡張のために 1 つのスロットを持つ。set_extension メソッドは単純にポインタを上書きし、原則としてイニシエータか、インターコネ

クト・コンポーネントかターゲットから呼び出されることができる。これはとても自由度の高い低レベルのメカニズムを提供するが、使用間違いも起こしやすい。拡張オブジェクトの所有権と削除はユーザによって良く理解され注意深く考慮されなければならない。

2 つの別の汎用ペイロード・トランザクションの間にトランザクション・ブリッジを作るとき、受信するトランザクション・オブジェクトから送信するトランザクション・オブジェクトへすべての拡張を必要であればコピーし、送信するトランザクションとその拡張を所有し管理するのはブリッジの責任である。同じことがデータ配列とバイト・イネーブル配列にも言える。deep_copy_from と update_original_from_メソッドは、データ配列とバイト・イネーブル配列や拡張オブジェクトを含むトランザクション・オブジェクトのディープコピー(ポインタ先の中身までコピーすること)をトランザクション・ブリッジが行えるようにするために提供されている。もしそのブリッジが送信するトランザクションにさらなる拡張を加えるなら、それらの拡張はそのブリッジによって所有される。

拡張の管理は「7.5 汎用ペイロードのメモリ管理」に十分に記述されている。

7.20.4 ルール

a) 拡張はイニシエータかインターコネクトかターゲットのコンポーネントにより加えることができる。特に、拡張の作成はイニシエータには制限がない。

b) どんな数の拡張でも汎用ペイロードの個々のインスタンスに加えることができる。

c) 無視可能拡張の場合、（その拡張を加えたコンポーネントを除く）どのコンポーネントもその拡張を無視することができ、無視可能拡張は必須拡張ではない。無視可能拡張が無いためや無視可能拡張への応答がないために、コンポーネントがフェールしていては、相互運用性が無くなる。

d) 与えられた拡張の存在を強制する組み込みのメカニズムはなく、拡張の無視を強制するメカニズムもない。

e) 個々の拡張の意味はアプリケーションが定義する。あらかじめ定義された拡張はない。

f) 拡張は tlm_extension クラスからユーザ定義クラスを派生させて作り、ユーザ定義クラスそれ自身の名前を tlm_extension のテンプレート引数として渡してそのクラスのオブジェクトを作らなければならない。ユーザ定義クラスは汎用ペイロードの拡張されたアトリビュートを表すメンバを含んで良い。

g) tlm_extension_base クラスの free 仮想メソッドは、拡張オブジェクトを削除しなければならない。このメソッドは、拡張のユーザ定義のメモリ管理の実装を上書きするかもしれない。しかし、これは必須ではない。

h) tlm_extension クラスの clone 純粋仮想関数は、すべての拡張されたアトリビュートを含む拡張オブジェクトのクローンを作るためにユーザ定義の拡張クラスの中に定義されなければならない。この clone メソッドは、汎用ペイロードのメモリ管理と共に使うことを意図されている。それは、すべての拡張オブジェクトのコピーを作って、オリジナルのオブジェクトのデストラクションがあっても副作用なくそのコピーが残るようにしなければならない。

i) tlm_extension クラスの copy_from 純粋仮想関数は、別の拡張オブジェクトのアトリビュートをコピーすることによって自身のオブジェクトを修正するためにユーザ定義の拡張クラス

の中で定義されなければならない。

j) tlm_extension クラステンプレートをインスタンスすると ID 公開データ・メンバが初期化され、これは汎用ペイロード・オブジェクトを持つ与えられた拡張を再登録しその拡張にユニークな ID を割り当てる効果を持たなければならない。その ID はプログラム実行中のすべてに対してユニークでなければならない。だから、tlm_extension クラステンプレートの個々のインスタンスは個別の ID を持たなければならず、その一方で、1 つの与えられた型のすべての拡張のオブジェクトは同じ ID を共有しなければならない。

k) 汎用ペイロードは、可変サイズの配列に拡張のポインタを蓄え、拡張の ID はその配列の中への拡張ポインタのインデックスを与えるようにふるまわなければならない。拡張の汎用ペイロードへの登録は、その拡張のための配列インデックスを保持しなければならない。個々の汎用ペイロードのオブジェクトは現在実行中のプログラムの中で登録されるすべての拡張へのポインタを蓄える能力のある配列を含まなければならない。

l) 拡張配列の中のポインタはトランザクションが生成されたときには空でなければならない。

m) 個々の汎用ペイロードのオブジェクトはすべての与えられた拡張型のせいぜい 1 つのオブジェクトへのポインタしか蓄えることができない。（しかし、異なる拡張型のたくさんのオブジェクトを蓄える。）（汎用ペイロード・オブジェクトに同じ型のいくつかの拡張オブジェクトを参照できるようにする instance_specific_extension ユーティリティ・クラスがある。「12.4 インスタンス固有の拡張」を参照。

n) max_num_extensions 関数は、拡張配列の大きさである、拡張型の数を返さなければならない。

o) メソッド set_extension と set_auto_extension、get_extension、clear_extension、release_extension は複数の形態で提供され、関数テンプレートの使用や拡張ポインタ引数の使用、ID 引数の使用のような異なる方法でアクセスされてその拡張を認識する。ID 引数を受け取る関数は、汎用ペイロードのオブジェクトのクローンを作るときのような特別なプログラムタスクのために意図されており、アプリケーションで一般的に使うためのものではない。

p) set_extension(T*)メソッドは、ポインタの配列の中の T 型の拡張オブジェクトを示すポインタを引数の値で置き換えなければならない。その引数は登録された拡張へのポインタでなければならない。関数の戻り値は、この呼び出しで置き換えられた汎用ペイロードの中のポインタの前の値でなければならない。それは空（null）のポインタかもしれない。set_auto_extension(T*)メソッドは、その拡張が自動削除にマークされるのを除き、同様にふるまわなければならない。

q) set_extension(unsigned int, tlm_extension_base*)メソッドは、第 1 引数で与えられる配列インデックスの示す、配列のポインタの中の拡張オブジェクトへのポインタを第 2 引数の値で置き換えなければならない。与えられたインデックスは、拡張 ID として登録されていなければならない。そうでない場合の関数のふるまいは未定義である。関数の戻り値は、与えられた配列インデックスへのポインタの前の値でなければならない。それは空（null）のポインタかもしれない。set_auto_extension(unsigned int, tlm_extension_base*)メソッドは、その拡張が自動削除にマークされるのを除き、同様にふるまわなければならない。

r) メモリ・マネージャが存在するとき、拡張するための set_auto_extension の呼び出しは、set_extension の呼び出しに続いてすぐに同じ拡張の release_extension を呼び出すのに等しい。メモリ・マネージャがないときに set_auto_extension を呼び出すとランタイムエラーを起こす。

s) もし拡張が自動削除とマークされたら、与えられた拡張オブジェクトはユーザ定義のメモリ・マネージャの free メソッドの実装によって削除されるかプールされなければならない。free メソッドは、トランザクション・オブジェクトの参照数が 0 に到達したときに tlm_generic_payload クラスのメソッド解放によって呼び出される。拡張オブジェクトは tlm_generic_payload クラスの reset メソッドを呼び出すか拡張オブジェクトそれ自身の free メソッドを呼び出すことによって削除されることができる。

t) もし汎用ペイロードのオブジェクトがすでにその型の拡張への空（null）でないポインタを含んでいたら、古いポインタは上書きされる。

u) メソッド関数 get_extension(T*&)と T* get_extension()は、もしそれが存在するなら、与えられた型の拡張オブジェクトへのポインタを、存在しなければ空（null）のポインタを返さなければならない。T 型は tlm_extension から派生した型のオブジェクトへのポインタでなければならない。この関数テンプレートを使って存在しない拡張の検索を試すことはエラーではない。

v) get_extension(unsigned int)メソッドは、引数によって与えられた ID の拡張オブジェクトへのポインタを返さなければならない。与えられたインデックスは拡張の ID として登録されていなければならない。そうでない場合のこの関数のふるまいは未定義である。もし与えられたインデックスへのポインタが拡張オブジェクトへのポインタではなかった場合、この関数は空（null）のポインタを返さなければならない。

w) メソッド clear_extension(const T*)と clear_extension()は、汎用ペイロード・オブジェクトから与えられた拡張を取り除かなければならない。すなわち、拡張配列の中の対応するポインタを空（null）に設定しなければならない。拡張は、引数として拡張オブジェクトのポインタを渡すか、clear_extension<ext_type>()のように関数テンプレートのパラメタの型を使うことで特定される。もし存在するならば、その引数は tlm_extension から派生した型のオブジェクトへのポインタでなければならない。clear_extension メソッドはその拡張オブジェクトを削除してはならない。

x) メソッド release_extension(T*)と release_extension()は、もしトランザクション・オブジェクトがメモリ・マネージャを持つならば自動削除としてその拡張をマークしなければならない。そうでない場合、これらのメソッドは、拡張オブジェクトの free メソッドの呼び出しと拡張配列の中の対応するポインタに空（null）を設定することで与えられた拡張を削除しなければならない。拡張は、引数として拡張オブジェクトのポインタを渡すか、release_extension<ext_type>()のように関数テンプレートのパラメタの型を使うことで特定される。もし存在するならば、その引数は tlm_extension から派生した型のオブジェクトへのポインタでなければならない。

y) release_extension メソッドのふるまいは、トランザクション・オブジェクトがメモリ・マネージャを持つかどうかに依存することに注意せよ。メモリ・マネージャがある場合は、拡張はまれに自動削除にマークされ、アクセス可能な状態で続く。メモリ・マネージャがない場合は、拡張ポインタがクリアされるだけでなく拡張オブジェクトそれ自身も削除される。存在しない拡張オブジェクトを開放しないように気をつけなければならない。開放した場合にはランタイムエラーになる。

z) メソッド clear_extension と release_extension は、たとえば set_auto_extension でセットされた拡張のように自動削除としてマークされた拡張や release_extension ですでに開放された拡張の

ために呼び出してはならない。その場合、ランタイムエラーになるかもしれない。

aa) 個々の汎用ペイロードのトランザクションは、すべての登録された拡張へのポインタを蓄えられる十分な領域を割り当てられなければならない。これは次の 2 つの方法のどちらかで達成できる。1 つは C++の静的な初期化の後にトランザクション・オブジェクトを構築するとであり、もう 1 つは静的な初期化の後で最初にトランザクション・オブジェクトを使う前に resize_extensions メソッドを呼び出すことである。前に述べたの方法において、拡張配列の大きさを設定するのは汎用ペイロードのコンストラクタの責任である。後に述べた方法において、最初に拡張にアクセスする前に resize_extensions メソッドを呼び出すのはアプリケーションの責任である。

bb) resize_extensions メソッドは、すべての登録された拡張に応じられるように汎用ペイロードの中の拡張配列のサイズを増加させなければならない。

例

```
// Showing an ignorable extension
// User-defined extension class
struct ID_extension: tlm::tlm_extension<ID_extension>
{
  ID_extension() : transaction_id(0) {}

  virtual tlm_extension_base* clone() const {          // Must override pure virtual clone method
    ID_extension* t = new ID_extension;
    t->transaction_id = this->transaction_id;
    return t;
  }
  // Must override pure virtual copy_from method
  virtual void copy_from(tlm_extension_base const &ext) {
    transaction_id = static_cast<ID_extension const &>(ext).transaction_id;
  }
unsigned int transaction_id;
};

// The initiator
struct Initiator: sc_module
{ ...
  void thread() {
  tlm::tlm_generic_payload trans;
  ...
  ID_extension* id_extension = new ID_extension;
  trans.set_extension( id_extension );                 // Add the extension to the transaction
```

```
   for (int i = 0; i < RUN_LENGTH; i += 4) {
      ...
      ++ id_extension->transaction_id;          // Increment the id for each new transaction
      ...
      socket->b_transport(trans, delay);
      ...

// The target
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
{ ...
   ID_extension* id_extension;
   trans.get_extension( id_extension );         // Retrieve the extension
   if (id_extension) {                          // Extension is not mandatory
      char txt[80];
      sprintf(txt, "Received transaction id %d", id_extension->transaction_id);
      SC_REPORT_INFO("TLM-2.0", txt);
  }
   ...

   // Showing a new protocol traits class with a mandatory extension
   struct cmd_extension: tlm::tlm_extension<cmd_extension>
   {                                            // User-defined mandatory extension class
      cmd_extension(): increment(false) {}
      virtual tlm_extension_base* clone() const {
      cmd_extension* t = new cmd_extension;
      t->increment = this->increment;
      return t;
   }
   virtual void copy_from(tlm_extension_base const &ext) {
      increment = static_cast<cmd_extension const &>(ext).increment;
    }
      bool increment;
};

struct my_protocol_types                        // User-defined protocol types class
{
   typedef tlm::tlm_generic_payload tlm_payload_type;
   typedef tlm::tlm_phase tlm_phase_type;
};

struct Initiator: sc_module
```

```
{
  tlm_utils::simple_initiator_socket<Initiator, 32, my_protocol_types> socket;
  ...
  void thread() {
    tlm::tlm_generic_payload trans;
    cmd_extension* extension = new cmd_extension;
    trans.set_extension( extension );                    // Add the extension to the transaction
    ...
    trans.set_command(tlm::TLM_WRITE_COMMAND); // Execute a write command
    socket->b_transport(trans, delay);
    ...
    trans.set_command(tlm::TLM_IGNORE_COMMAND);
    extension->increment = true;                         // Execute an increment command
    socket->b_transport(trans, delay);
    ...
    ...

  // The target
  tlm_utils::simple_target_socket<Memory, 32, my_protocol_types> socket;

  virtual void b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
  {
    tlm::tlm_command cmd = trans.get_command();
    ...
    cmd_extension* extension;
    trans.get_extension( extension );                    // Retrieve the command extension
    ...
  if (!extension) {                                      // Check the extension exists
    trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
    return;
 }
  if (extension->increment) {
    if (cmd != tlm::TLM_IGNORE_COMMAND) {   // Detect clash with read or write
      trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
      return;
   }
    ++ m_storage[adr];                                   // Execute an increment command
    memcpy(ptr, &m_storage[adr], len);
  }
```

8. 基本プロトコルとフェーズ

8.1 フェーズ

8.1.1 イントロダクション

- tlm_phase クラスは、ノンブロッキング・トランスポート・インタフェース・クラスのテンプレートと基本プロトコルによって使用されるデフォルトのフェーズ型である。
- tlm_phase オブジェクトは unsigned int 値によりフェーズを表す。
- tlm_phase クラスは、unsigned int 型と、4 つの基本プロトコル・フェーズ BEGIN_REQ、END_REQ、BEGIN_RESP、および END_RESP の列挙体と代入互換である。
- tlm_phase_enum によって提供される 4 つのフェーズの組は、DECLARE_EXTENDED_PHASE マクロを使用して拡張することができる。
- このマクロは対応するオブジェクトを返す get_phase メソッドにより、tlm_phase を継承するシングルトン・クラスを作成する。そのオブジェクトは新しいフェーズとして使用可能である。
- 各アプリケーションは、相互利用性を最大限にするために tlm_phase_enum の 4 つのフェーズのみを使用すべきである。もし、 それ以上のフェーズが特定のプロトコルの詳細部をモデル化するのに必要であるなら、DECLARE_EXTENDED_PHASE を使用すべきである。
- DECLARE_EXTENDED_PHASE は tlm_phase 型と代入互換である。
- 無視可能に対する無視不可能あるいは必須な拡張の原理は、汎用ペイロード拡張と同様にフェーズにも当てはまる。 つまり、無視可能なフェーズは基本プロトコルに包含される。
- 無視可能なフェーズは、受信側と送信側の両方から無視可能にする必要がある。受信側の場合、受信側が単にフェーズ遷移を受け取っていないかのように動作できるという意味で受信側が無視可能でなければならない。同様に送信側では、受信側からたとえ応答がなくても送信側が実行を継続できるという意味で無視可能でなければならない。
- このような意味でフェーズが無視可能でないなら、新しいプロトコル・トレイツ・クラスを定義するべきである。

8.1.2 クラス定義

```
namespace tlm {
  enum tlm_phase_enum {
    UNINITIALIZED_PHASE=0, BEGIN_REQ=1, END_REQ, BEGIN_RESP, END_RESP };
  class tlm_phase{
  public:
    tlm_phase();
    tlm_phase( unsigned int );
    tlm_phase( const tlm_phase_enum& );
    tlm_phase& operator= ( const tlm_phase_enum& );
```

```
    operator unsigned int() const;
  };
  inline std::ostream& operator<< ( std::ostream& , const tlm_phase& );
  #define DECLARE_EXTENDED_PHASE(name_arg) ¥
  class tlm_phase_##name_arg : public tlm::tlm_phase{ ¥
  public: ¥
    static const tlm_phase_##name_arg& get_phase(); ¥
    implementation-defined ¥
  }; ¥
  static const tlm_phase_##name_arg& name_arg=tlm_phase_##name_arg::get_phase()
} // namespace tlm
```

8.1.3 ルール

a) デフォルト・コンストラクタ tlm_phase はフェーズの値を 0 に設定するものとする。対応する列挙体は文字列 UNINITIALIZED_PHASE である。

b) メソッドの tlm_phase(unsigned int)、operator= と operator unsigned int は対応する unsigned int もしくは enum を使ってフェーズの値を得たり設定したりするものとする。

c) 関数 operator<< はフェーズの名前に対応する文字列を与えられた出力ストリームに書くものとする。たとえば 「BEGIN_REQ」である。

d) DECLARE_EXTENDED_PHASE(name_arg) マクロは tlm_phase を継承する tlm_phase_{name_arg}という名前の新しいシングルトン・クラスを生成する。tlm_phase にはパブリック・メソッドの get_phase があり生成された静的オブジェクトへの参照を返す。このマクロの引数は、対応するフェーズを示す operator<<によって書かれる文字列として使用される。

e) 上記の目的は、静的定数 name_arg によって示されるオブジェクトが拡張フェーズを表し、nb_transport のフェーズ引数として渡されてもよいようにするためである。

例

```
DECLARE_EXTENDED_PHASE(ignore_me);                  // Declare two extended phases
DECLARE_EXTENDED_PHASE(internal_ph);                // Only used within target
struct Initiator: sc_module
{ ...
  { ...
    phase = tlm::BEGIN_REQ;
    delay = sc_time(10, SC_NS);
    socket->nb_transport_fw( trans, phase, delay );  // Send phase BEGIN_REQ to target

    phase = ignore_me;                               // Set phase variable to the extended phase
    delay = sc_time(12, SC_NS);
```

```
    socket->nb_transport_fw( trans, phase, delay );          // Send the extended phase 2ns later
    ...

struct Target: sc_module
{
  ...
  SC_CTOR(Target)
  : m_peq("m_peq", this, &Target::peq_cb) {}                 // Register callback with PEQ

  virtual tlm::tlm_sync_enum nb_transport_fw( tlm::tlm_generic_payload& trans,
  tlm::tlm_phase& phase, sc_time& delay ) {
    cout << "Phase = " << phase << endl;                    // use overloaded operator<< to print phase
    m_peq.notify(trans, phase, delay);                       // Move transaction to internal queue
    return tlm::TLM_ACCEPTED;
  }
  void peq_cb(tlm::tlm_generic_payload& trans, const tlm::tlm_phase& phase)
  {                                                          // PEQ callback
    sc_time delay;
    tlm::tlm_phase phase_out;
    if (phase == tlm::BEGIN_REQ) {                           // Received BEGIN_REQ from initiator
    phase_out = tlm::END_REQ;
    delay = sc_time(10, SC_NS);
    socket->nb_transport_bw(trans, phase_out, delay);        // Send END_REQ back to initiator
    phase_out = internal_ph;                                 // Use extended phase to signal internal event
    delay = sc_time(15, SC_NS);
    m_peq.notify(trans, phase_out, delay);                   // Put internal event into PEQ
  }
  else if (phase == internal_ph)                             // Received internal event
  {
    phase_out = tlm::BEGIN_RESP;
    delay = sc_time(10, SC_NS);
    socket->nb_transport_bw(trans, phase_out, delay);        // Send BEGIN_RESP back to initiator
    }
  }                                                          // Ignore phase ignore_me from initiator
  tlm_utils::peq_with_cb_and_phase<Target, tlm::tlm_base_protocol_types> m_peq;
};
```

8.2　基本プロトコル

8.2.1　イントロダクション

基本プロトコルはメモリ・マップド・バスへのインタフェースを持つトランザクション・レベル・

モデルのコンポーネント間の最大の相互利用性を確保するためのルールセットから構成される。

基本プロトコルはここに記載されたTLM-2.0インタオペラビリティ・レイヤの各種クラスおよび、この項で定義されたルールの使用を必要とする。:

a) TLM-2 コアトランスポート、ダイレクト・メモリ、およびデバッグ・トランスポート・インタフェース。「4 TLM-2 コア・インタフェース」を参照。

ソケットクラスの tlm_initiator_socket と tlm_target_socket（または、これらから派生したクラス）。「0

8.3 ダイレクト・メモリ・インタフェース

8.3.1 イントロダクション

ダイレクト・メモリ･インタフェース（DMI）は、イニシエータがターゲット内のメモリ領域に直接アクセスできる手段を提供する。つまりトランスポート・インタフェースを使わずに直接ポインタを使ってメモリアクセスが可能である。DMI によって、イニシエータからインターコネクト･コンポーネントに接続されたターゲットへの複数の b_transport 関数や nb_transport 関数の呼び出しをバイパスできるため、イニシエータとターゲット間のメモリアクセスを高速化することが可能となる。

ダイレクト・メモリ・インタフェースには 2 つのインタフェースがあり、ひとつはイニシエータからターゲットへのフォワード・パス呼び出しともうひとつはターゲットからイニシエータへのバックワード･パス呼び出しである。フォワード･パスは、ある指定されたアドレスに対する DMI アクセス（読み込みまたは書き込み）の特定モードを要求するために利用され、DMI 領域の境界を含む tlm_dmi 型の DMI デスクリプタへのリファレンスを返す。バックワード・パスは、フォワード･パスによって確立されたＤＭＩポインタを無効にするためにターゲットから呼び出される。フォワード･パスとバックワード･パスはゼロ、ひとつ、または複数のインターコネクト・コンポーネントを通ってもよいが、同じソケットを通る対応するトランスポート呼び出しと同一であるべきである。

DMI ポインタは、フォワード・パスにそってトランザクションを通過することによって要求される。デフォルトの DMI トランザクション型は、tlm_generic_payload であり、トランザクション･オブジェクトのコマンドとアドレス・アトリビュートのみ使用する。DMI は、トランスポート・インタフェースの拡張と同じ仕組みである。つまり、DMI 要求には、無視可能または必須拡張があってもよいが、すべての無視不可能拡張のときには、新しいプロトコル・トレイツ・クラス定義する必要がある。（tlm_generic_payload の型定義が記載されている「7.2.2tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義」を参照）

DMI デスクリプタは、イニシエータで消費されるレイテンシ値を返し、その結果ルーズリー・タイムド・モデリングに十分なタイミング精度を確保することができる。

DMI ポインタはデバッグ用途にも使えるが、通常デバッグのトラフィックよりもメモリアクセスのトラフィックの方が支配的であるため、デバッグ用途では、DMI ではなくデバッグ・トランザクション・インタフェースで十分である。DMI ポインタをデバッグ用に使う場合、レイテンシ値は無視すべきである。

8.3.2 クラス定義

```
namespace tlm {

  class tlm_dmi
  {
  public:
    tlm_dmi() { init(); }
    void init();

    enum dmi_access_e {
      DMI_ACCESS_NONE = 0x00,
      DMI_ACCESS_READ = 0x01,
      DMI_ACCESS_WRITE = 0x02,
      DMI_ACCESS_READ_WRITE = DMI_ACCESS_READ | DMI_ACCESS_WRITE
    };
    unsigned char* get_dmi_ptr() const;
    sc_dt::uint64 get_start_address() const;
    sc_dt::uint64 get_end_address() const;
    sc_core::sc_time get_read_latency() const;
    sc_core::sc_time get_write_latency() const;
    dmi_access_e get_granted_access() const;
    bool is_none_allowed() const;
    bool is_read_allowed() const;
    bool is_write_allowed() const;
    bool is_read_write_allowed() const;

    void set_dmi_ptr(unsigned char* p);
    void set_start_address(sc_dt::uint64 addr);
    void set_end_address(sc_dt::uint64 addr);
    void set_read_latency(sc_core::sc_time t);
    void set_write_latency(sc_core::sc_time t);
    void set_granted_access(dmi_access_e t);
    void allow_none();
    void allow_read();
    void allow_write();
    void allow_read_write();
  };

  template <typename TRANS = tlm_generic_payload>
  class tlm_fw_direct_mem_if : public virtual sc_core::sc_interface
  {
```

```
  public:
    virtual bool get_direct_mem_ptr(TRANS& trans, tlm_dmi& dmi_data) = 0;
  };
  class tlm_bw_direct_mem_if : public virtual sc_core::sc_interface
  {
  public:
    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range) = 0;
  };

} // namespace tlm
```

8.3.3 get_direct_mem_ptrメソッド

p) get_direct_mem_ptr メソッドは、ターゲットではなく、イニシエータまたはインターコネクト・コンポーネントからのみ呼び出されるべきである。

q) trans 引数には、イニシエータで生成された DMI トランザクション・オブジェクトへの参照値を渡す。

r) dmi_data 引数には、イニシエータで生成された DMI デスクリプタへの参照値を渡す。

s) すべてのインターコネクト・コンポーネントでは、get_direct_mem_ptr 呼び出しをイニシエータからターゲットへのフォワード・パスに沿って通過させる必要がある。別の言い方でいうと、インターコネクト・コンポーネントのターゲット・ソケットで、get_direct_mem_ptr の実装は、イニシエータ・ソケットの get_direct_mem_ptr メソッドを呼び出してもよい。

t) それぞれの get_direct_mem_ptr 呼び出しは、イニシエータからターゲットへのトランスポート呼び出しの対応するセットと全く同一のパスを辿る。言い換えれば、それぞれの DMI 要求は、一つのイニシエータと一つのターゲット間の相互作用を含む。これら二つのコンポーネント（イニシエータとターゲット）もまた、トランスポート・インタフェースを通る一つのトランザクション・オブジェクトのイニシエータとターゲットとしての役割をする。DMI は、重要なビット幅変換器のような、二番めのトランザクション・オブジェクトを生成するようなコンポーネントを通るパス上では使うことができない。（もし DMI でシミュレーション速度が非常に要求される場合には、シミュレーション・モデルを再構成する必要があるかもしれない）

u) すべてのインターコネクト・コンポーネントは、フォワード方向での trans と dmi_data 引数を渡す必要がある。唯一許されている修正は、以下のように記述される DMI トランザクション・オブジェクトのアドレス・アトリビュート値である。トランザクションと DMI デスクリプタのアドレス・アトリビュートは両方とも get_direct_mem_ptr メソッドからのリターン時に修正されるかもしれない。つまり、ターゲットからイニシエータへ関数呼び出しが順にリターンされるときである。

v) もし、指定されたアドレスへの DMI アクセスをターゲットがサポートしていれば、以下に記述するように、DMI デスクリプタのメンバをセットし、関数の返り値として true をセットする必要がある。ターゲットが DMI アクセスを許可するときは、DMI デスクリプタは許可されたアクセスの詳細を示すために使用される。

w) もし、指定されたアドレスへの DMI アクセスをターゲットがサポートしていなければ、以下に記述するように、アドレス範囲と DMI デスクリプタの型メンバのみセットし、関数の返り値としてfalseをセットする必要がある。ターゲットがDMIアクセスを拒否するときは、DMIデスクリプタは拒否されたアクセスの詳細を示すために使用される。

x) ターゲットは、この節で与えられるルールに従って、メモリ領域（不連続な領域含む）へのDMIアクセスを許可または拒否してもよい。

y) ターゲットが DMI アクセスを許可し、関数の返り値に true をセットする場合、インターコネクト・コンポーネントは、get_direct_mem_ptr メソッドから返ってきたときに関数の返り値に false をセットすることによって DMI アクセスを拒否してもよい。その逆は許可されていない。ターゲットが DMI アクセスを拒否した場合、インターコネクト・コンポーネントはDMIアクセスを許可するべきではない。

z) get_direct_mem_ptr を複数回呼び出した場合、ターゲットは同じメモリ領域、同じ時刻に DMIアクセスを複数のイニシエータに許可するかもしれない。アプリケーションは同期とデータの一貫性を保証する必要がある。

aa) それぞれの get_direct_mem_ptr の呼び出しは隣り合うメモリ領域への単一 DMI ポインタのみを返すので、実際にはそれぞれの DMI 要求は一つのターゲットでのみ実行可能である。別の言い方をすれば、もしメモリ領域が複数のターゲットに分散している場合、例えアドレス範囲が隣りあっていても、それぞれのターゲットは分割された DMI リクエストを要求するだろう。

bb) あるメモリ領域への読み込み又は書き込みアクセスが、ターゲットに副作用（つまり、メモリの状態だけではなく、ターゲットの状態にまで変化を与えること）が発生する場合は、ターゲットはメモリ領域への与えられた型の DMI アクセスを許可すべきではない。しかし、例えば、書き込みによってのみターゲットに副作用が発生するなら、ターゲットは与えられた領域への読み込みの DMI アクセスは許可してもよい。

cc) get_direct_mem_ptr は、invalidate_direct_mem_ptr 呼び出しを実装してもよい。

dd) get_direct_mem_ptr は、直接的または間接的に wait 読み出しを実装してはいけない。

8.3.4 テンプレート引数とtlm_generic_payloadクラス

n) tlm_fw_direct_mem_if テンプレートは DMI トランザクション・クラスの型でパラメタライズされる。

o) トランザクション・オブジェクトはダイレクト・メモリ・アクセスが要求するアドレスを指し示すアトリビュート及び、要求されるアクセスの型、つまり指定されたアドレスに対する読み込みアクセスか、書き込みアクセスかを含む。基本プロトコルの場合、これらは汎用ペイロードのコマンドとアドレスのアトリビュートである。

p) TRANS テンプレートの引数のデフォルト値は tlm_generic_payload クラスである。

q) 相互利用性を最大化するために、DMI トランザクション・クラスは tlm_generic_payload にすべきである。無視不可能拡張または他のトランザクション型を使うことは相互利用性を制限することになる。

r) イニシエータは DMI トランザクション・オブジェクトを生成及び管理する必要があり、ま

た get_direct_mem_ptr への引数として渡される前に適切なアトリビュートを設定する必要がある。

s) トランザクション・オブジェクトのコマンド・アトリビュートは、要求されている DMI アクセスの種類を示すために、イニシエータによって設定され、インターコネクト・コンポーネントまたはターゲットによって修正されるべきでない。基本プロトコルでは、許される値は読み込みアクセスの TLM_READ_COMMAND と書き込みアクセスの TLM_WRITE_COMMAND である。

t) 基本プロトコルでは、コマンド・アトリビュートの値に TLM_IGNORE_COMMAND は禁止されている。しかし、この値は他のプロトコルでは使用してよい。

u) トランザクション・オブジェクトのアドレス・アトリビュートは、ダイレクト・メモリ・アクセスが要求しているアドレスを指し示すために、イニシエータによって設定される。

v) フォワード・パスに沿って DMI トランザクション・オブジェクトが通過するインターコネクト・コンポーネントは、同じソケットの対応するトランスポート・インタフェースにトランザクションのアドレス・アトリビュートを、デコード及び、必要に応じて修正すべきである。例えば、インターコネクト・コンポーネントによって、ターゲットのアドレス幅やシステム・メモリ・マップにおける場所に応じて、アドレスをマスク（上位ビットの数を削減）する必要がある。

w) インターコネクト・コンポーネントは,アドレスエラーを検知した場合、get_direct_mem_ptr 呼び出しを通過させる必要はない。

x) 基本プロトコルの場合、イニシエータは、コマンドとアドレスを除いて汎用ペイロードのアトリビュートを設定する必要はない。また、ターゲットとどんなインターコネクト・コンポーネントでも、他のすべてのアトリビュートを無視してもよい。特に、応答ステータスのアトリビュートや許可されている DMI アトリビュートは無視してもよい。許可されている DMI アトリビュートは、トランスポート・インタフェースとともに利用することだけを想定している。

y) イニシエータは、一つの DMI 呼び出しから次の DMI 呼び出し、さらにはトランスポート・インタフェース、デバッグ・トランスポート・インタフェースの呼び出しを含めて、トランザクション・オブジェクトを再利用しても良い。

z) もしアプリケーションが、ターゲットが DMI アクセスの種類を判別する目的で、DMI トランザクション・オブジェクトにさらにアトリビュートを追加する必要があるならば、無関係なトランザクション・クラスに置き換えるよりも、汎用ペイロードを拡張することを推奨する。例えば、DMI トランザクションは、CPU の種類に応じて異なった DMI 要求をターゲットが処理するために、CPU Id が必要かもしれない。そのような拡張が無視不可能な場合は、新しいプロトコル・トレイツ・クラスを定義する必要がある。

8.3.5 tlm_dmiクラス

ee) DMI デスクリプタは tlm_dmi クラスのオブジェクトである。DMI デスクリプタは、イニシエータで生成されるが、そのメンバはインターコネクト・コンポーネントまたはターゲットで設定される。

ff) DMI デスクリプタは、次のアトリビュートを持つ：DMI ポインタ・アトリビュート、許可

されたアクセス型アトリビュート、開始アドレス・アトリビュート、終了アドレス・アトリビュート、読み込みレイテンシ・アトリビュート、そして書き込みレイテンシ・アトリビュート。これらアトリビュートのデフォルト値：DMI pointer attribute = 0,access type = DMI_ACCESS_NONE, start address = 0, end address = the maximum value of type sc_dt::uint64, read latency = SC_ZERO_TIME, and write latency = SC_ZERO_TIME

gg) init メソッドは、DMI 拡張子のメンバをデフォルト値に初期化する。

hh) DMI 拡張子は、イニシエータによって get_direct_mem_ptr への引数として渡されるときには必ず、デフォルト状態でなければならない。もし DMI 拡張子オブジェクトが蓄積されると、イニシエータは get_direct_mem_ptr への引数として渡される前に、DMI 拡張子をそのデフォルト状態にリセットしなければならない。init メソッドはこの目的のために呼び出される。

ii) インターコネクト・コンポーネントは DMI デスクリプタをターゲット方向に渡される際に修正してはいけないので、DMI デスクリプタはターゲットで受け取ったときに初期状態にあるべき。

jj) set_dmi_ptr メソッドは DMI ポインタ・アトリビュートを引数で渡される値に設定される。get_dmi_ptr メソッドは DMI ポインタ・アトリビュートの現在の値を返す。

kk) DMI ポインタ・アトリビュートは開始アドレス・アトリビュートの値に対応するストレージ場所を示すためにターゲットによって設定される。これは get_direct_mem_ptr 呼び出しで要求されるアドレス以下でなければならない。初期値は 0 である。

ll) DMI 領域のストレージは unsigned char*の型で表現される。ストレージは汎用ペイロードのデータ配列と構成を持つ。もしターゲットがその構成のメモリ領域へのポインタを返すことが出来ない場合は、ターゲットは DMI をサポートできず、get_direct_mem_ptr は false 値を返すべきである。TLM-2.0 におけるメモリ構成とエンディアンの扱い方の詳細は、「7.17 エンディアン」を参照。

mm) インターコネクト・コンポーネントは、get_direct_mem_ptr 関数呼び出しからのリターン・パスにおいて DMI アクセスを許可する領域を制限するために、DMI ポインタ・アトリビュートを修正することが許されている。

nn) set_granted_access メソッドは、許可されたアクセス型アトリビュートに引数で渡された値を設定する。get_granted_access メソッドは、許可されたアクセス型アトリビュートの現在の値を返す。初期値は DMI_ACCESS_NONE でなければならない。

oo) allow_none、allow_read、allow_write そして allow_read_write メソッドは、許可されたアクセス型アトリビュートをそれぞれ DMI_ACCESS_NONE、DMI_ACCESS_READ、DMI_ACCESS_WRITE、または DMI_ACCESS_READ_WRITE の値に設定する。

pp) is_none_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_NONE の値の場合及びその場合のみ真の値を返す。is_read_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_READ または DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。is_write_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_WRITE または DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。is_read_write_allowed メソッドは、許可されたアクセス型アトリビュートが DMI_ACCESS_READ_WRITE の値の場合及びその場合のみ真の値を返す。

qq) ターゲットは、許可されたまたは拒否されたアクセス型アトリビュートに、許可されたアクセスの型を設定する。ターゲットは、読み込みまたは読み込み／書き込みアクセスが許可される（または拒否される）ことよって読み込みアクセスの要求に答えることができ、同様に書き込みまたは読み込み／書き込みアクセスが許可される（または拒否される）ことよって書き込みアクセスの要求に答えることができる。インターコネクト・コンポーネントは、get_direct_mem_ptr 関数呼び出しのリターン・パスにおいて、DMI_ACCESS_READ_WRITE の値を DMI_ACCESS_READ または DMI_ACCESS_WRITE で上書きすることによって許可されたアクセスを制限ことができる。

rr) DMI 領域への読み込み･書き込みアクセスを拒否したいターゲットは､DMI_ACCESS_NONE ではなく、DMI_ACESS_READ_WRITE にアクセス型を許可すべきである。

例）

```
bool get_direct_mem_ptr( TRANS& trans, tlm::tlm_dmi& dmi_data )
{
// Deny DMI access to entire memory region
dmi_data.allow_read_write();
dmi_data.set_start_address( 0x0 );
dmi_data.set_end_address( (sc_dt::uint64)-1 );
return false;
}
```

ss) ターゲットは､次の場合には DMI_ACCESS_NONE を許可されたアクセス型に設定すべきである。すなわち、イニシエータへの読み込みや書き込みまたは読み込み／書き込みアクセスが許可されていない（または拒否されていない）が、その一方で、DMI トランザクション・オブジェクトへの拡張によって要求されるようないくつか他の種類のアクセスを許可している（または拒否している）ことを示す場合である。この値は、DMI トランザクション・オブジェクトの拡張によって事前に定義されたアクセス型（読み込み、書き込みと読み込み／書き込み）が不要または意味がなくなる場合にのみ利用されるべきである。この値は基本プロトコルの場合に利用されるべきではない。

tt) イニシエータは、許可されたアクセス型アトリビュートを使ってターゲットによって許可されている（または基本プロトコルとは別に、汎用ペイロードの拡張やまたほかの DMI トランザクション型を使って許可されている）DMI アクセスのこれらのモード（インターコネクトによって修正される可能性はあるが）のみを利用する責任がある。

uu) set_start_address と set_end_address メソッドは、開始と終了アドレス・アトリビュートをそれぞれ引数で渡される値を設定する。get_start_address と get_end_address メソッドは、開始と終了アドレス・アトリビュートの現在の値をそれぞれ返す。

vv) 開始と終了アドレス・アトリビュートは DMI 領域における最初と最後のバイトのアドレスを示すためにターゲットによって設定（またはインターコネクトによって修正）される。DMI 領域が許可されているか拒否されているかは、get_direct_mem_ptr メソッド返り値（true または false）によって決まる。全メモリ領域へのアクセスを拒否したいターゲットは､start_address に 0 を end_address に sc_dt::uint64 型の最大値を設定してもよい。

ww) ターゲットは、それぞれの get_direct_mem_ptr 呼び出しに対して一つの隣接するメモリ領域

を許可または拒絶のみできる。ターゲットは、開始と終了アドレス・アトリビュートを同じにすることによってメモリ領域を一つのアドレスに設定することができ、DMI 領域を任意の大きさに設定することもできる。

xx) 指定された領域への指定された型の DMI アクセスが許可されているとき、イニシエータはその領域が無効になるまでその領域のどこでも指定された型のアクセスを実行するかもしれない。別の言い方でいうと、DMI 要求によって指定されたアドレスはアクセス制限されない。

yy) get_direct_mem_ptr 呼び出しが通過するいかなるインターコネクト・コンポーネントでも、アドレス引数を変換するように開始と終了アドレス・アトリビュートを変換する必要がある。DMI デスクリプタにおけるアドレスのいかなる変換も、デスクリプタが get_direct_mem_ptr 関数呼び出しのリターン・パスを通過するときに実行される。例えば、ターゲットは開始アドレス・アトリビュートをそのターゲットが知っているメモリマップ内での相対アドレスに設定する。その場合インターコネクト・コンポーネントは相対アドレスをシステム上のメモリマップにおける絶対アドレスに変換する必要がある。初期値はそれぞれ 0 と sc_dt::uint64 の取り得る最大値である。

zz) インターコネクト・コンポーネントは、DMI アクセスが許可されている領域を制限するため、または DMI アクセスが許可されていない領域を拡張するために、開始と終了アドレス・アトリビュートを修正することができる。

aaa) もし get_direct_mem_ptr が true を返す場合は、開始と終了アドレス・アトリビュートで示される DMI 領域が DMI アクセスできる領域である。もし get_direct_mem_ptr が false を返す場合は、DMI アクセスできない領域である。

bbb) ターゲットまたはインターコネクト・コンポーネントが二つまたはそれ以上の get_direct_mem_ptr 呼び出しを受け取る場合は、二つまたはそれ以上の重なり合う DMI アクセス許可領域、または二つまたはそれ以上の重なり合う DMI アクセス禁止領域を返すかもしれない。

ccc) ターゲットまたはインターコネクト・コンポーネントは、(最初の領域を無効にするために invalidate_direct_mem_ptr を途中で呼び出さない限り)、重なりあう領域に対し、一方が許可され、もう一方が禁止されるような DMI 領域は返さない。例えば、どちらも読み込み及び読み込み／書き込みの両方であるか、書き込み及び読み込み／書き込みの両方である。

ddd) 別の言葉で言えば、DMI 領域の定義は invalidate_direct_mem_ptr を途中呼び出しによって最初の領域が無効化されない限り、生成された順番に依存しない。特に禁止 DMI 領域の生成は、同じアクセス型ですでに許可された DMI 領域に穴をあけることができない。逆もまた真。

eee) ターゲットは DMI アクセスをすべてのアドレス領域（開始アドレスを 0、終了アドレスを最大値）で禁止してもよい。多分ターゲットは DMI アクセスを全くサポートしないからだろう。その場合、インターコネクト・コンポーネントはこの禁止領域をターゲットが占有しているメモリマップの領域にとどめるべきである。もしインターコネクト・コンポーネントがアドレス領域をとどめることに失敗したら、イニシエータはすべてのシステム上のアドレス空間のすべてが DMI 禁止されていると勘違いをするだろう。

fff) set_read_latency と set_write_latency メソッドは、それぞれ読み込み及び書き込みレイテンシ・

アトリビュートに引数で渡された値を設定する。get_read_latency と get_write_latency メソッドは、それぞれ読み込み及び書き込みレイテンシ・アトリビュートの現在の値を返す。

ggg) 読み込み及び書き込みレイテンシ・アトリビュートは、それぞれ読み込み及び書き込みのメモリ・トランザクションにかかるバイトあたりの平均レイテンシを設定する。言い換えれば、直接的なメモリ操作を実行するイニシエータが等価な transport トランザクションによって転送されるバイト数と DMI 拡張子からの読み込みまたは書き込みレイテンシとを掛け合わせることによって実際のレイテンシを計算しなければならない。初期値は SC_ZERO_TIME。インターコネクト・コンポーネントとターゲットの両方が読み込み、書き込みのどちらかのレイテンシの値を増やし、その結果 DMI デスクリプタが get_direct_mem_ptr メソッドからのリターン・パスにおいてイニシエータからターゲットへ戻るときにレイテンシが積算される。許可されたアクセス型アトリビュートの値によって、一つまたは両方のレイテンシが有効になる。

hhh) イニシエータはダイレクト・メモリ・ポインタを使ってメモリにアクセスするときはいつでも、レイテンシを考慮する必要がある。もしイニシエータがレイテンシを無視すれば、タイミングが不正確になる。

8.3.6 invalidate_direct_mem_ptrメソッド

m) invalidate_direct_mem_ptr メソッドはターゲットまたはインターコネクト・コンポーネントだけによって呼び出される。

n) ターゲットは存在する DMI 領域の有効性やアクセス型を修正する変更の前に invalidate_direct_mem_ptr メソッドを呼び出す必要がある。例えば、存在する DMI 領域のアドレス領域を制限する前や、アクセス型を読み込み／書き込みから読み込みに変更する前や、アドレス空間をリマップする前などである。

o) start_range と end_range の引数は、DMI 領域が無効化されるアドレス範囲の最初と最後のアドレスを示す。

p) イニシエータが invalidate_direct_mem_ptr を受け取ったらすぐに指定されたアドレス範囲と重なり合う DMI 領域（get_direct_mem_ptr で事前に獲得していたもの）を無効にして廃棄する。

q) 部分的に重なり合う場合、つまり存在する DMI 領域の一部分だけ無効な場合には、イニシエータは存在する領域の境界を調整するか、または全体の領域を無効化する。

r) 各 DMI 領域は、ターゲットが invalidate_direct_mem_ptr 呼び出しで明示的に無効化するまでは有効のままである。各イニシエータは、有効な DMI 領域の管理表を管理し、無効化されるまで各領域を利用し続ける。

s) すべてのインターコネクト･コンポーネントは、デコードや必要に応じて対応するトランスポート・インタフェースのためにアドレス引数を修正するときに、invalidate_direct_mem_ptr 呼び出しをターゲットからイニシエータへのバックワード・パスに沿って通過させる義務がある。トランスポート・インタフェースがフォワード・パスのアドレスとバックワード・パスの DMI を変換するので、トランスポートと DMI 変換は相互に逆であるべきである。

t) もし一つの invalidate_direct_mem_ptr をターゲットから呼び出すならば、インターコネクト・コンポーネントは複数の invalidate_direct_mem_ptr をイニシエータへ呼び出してもよい。複数

のイニシエータが存在しそれぞれが同じターゲットに対してダイレクト・メモリ・ポインタを得るかもしれないので、インターコネクト・コンポーネントがすべてのイニシエータに対して invalidate_direct_mem_ptr を呼び出すのが安全な実装方法である。

u) インターコネクト・コンポーネントは、start_range を 0、end_range を sc_dt::uint64 の最大値に設定することによって、イニシエータにあるすべてのダイレクト・メモリ・ポインタを無効化することができる。

v) TLM-2.0 コア・インタフェース・メソッドは、invalidate_direct_mem_ptr を呼び出す実装でもよい。

w) invalidate_direct_ptr は get_direct_mem_ptr を直接的または間接的に呼び出す実装ではない。

x) invalidate_direct_mem_ptr は wait を直接的または間接的に呼び出す実装ではない。

8.3.7 DMIとtransportの比較

d) 定義によれば、ダイレクト・メモリ・インタフェースによって、イニシエータとターゲット間を直接アクセスすることが可能である。このとき、インターコネクト・コンポーネントはバイパスされる。一方、トランスポート・インタフェースでは、インターコネクト・コンポーネントをバイパス不可能である。

e) インターコネクト・コンポーネントが状態を保持しているまたは副作用がある場合でも(例えばバッファ機能を有するインターコネクト・コンポーネントやキャッシュ・メモリをモデル化しているインターコネクトコネクト・コンポーネント)、正しく動作するように注意が必要である。ダイレクト・メモリ・インタフェースとは違い、トランスポート・インタフェースは、インターコネクト・コンポーネントの状態にアクセスしたり、アップデートしたりすることが可能である。ダイレクト・メモリ・インタフェースの場合は、インターコネクト・インタフェースをバイパスする。最も安全な方法は、このようなインターコネクト・コンポーネントの場合には、常に DMI アクセスを拒否することである。

f) イニシエータは、トランスポート・インタフェースを呼び出すことと、ダイレクト・メモリ・ポインタを使うことを、相互に切り替えることが可能である。あるイニシエータが、トランスポート・インタフェースを利用している間に、もう一方のイニシエータで DMI を使うことも可能である。トランスポート呼び出しがタイミング・アノテーションを転送している場合は、正しく振舞うように保証するように注意が必要である。これはアプリケーション側がケアする必要がある。例えば、あるターゲットが DMI とトランスポートを同時にサポートしている場合や、トランスポートが呼び出されている間にすべての DMI ポインタが無効化されている場合である。

8.3.8 DMIとテンポラル・デカップリング

g) ターゲットやインターコネクト・コンポーネントで invalidate_direct_mem_ptr を呼び出すことによってのみ、DMI 領域を無効化することができる。

h) イニシエータは、次の考慮すべき条件において DMI ポインタを使う前に DMI 領域がまだ有効かどうかをチェックする責任がある。

i) 一度イニシエータが起動されたら、そのイニシエータが制御を手放すまで、他の SystemC プロセスは実行できないことは、SystemC のコ・ルーチン・セマンティクスによって保証され

る。特に、他の SystemC プロセスは DMI ポインタを無効化できない（現プロセスは可能かもしれないが)。結果として、テンポラル・デカップリングされたイニシエータが、DMI ポインタを使う度に与えられ DMI 領域がまだ有効かどうかを繰り返し確認する必要は必ずしもない。

j) イニシエータに呼び出されるインタフェース・メソッド・コールによって、他のコンポーネントが invalidate_direct_mem_ptr を呼び出し、これによってそのイニシエータに利用されている DMI 領域を無効化する場合がある。これは、テンポラル・デカップリングされたイニシエータが制御を譲り渡さずに実行する場合に当てはまる。

k) イニシエータが、他のコンポーネントと通信を行わない、あるいは、制御を保持して実行される間は、有効な DMI 領域は有効なままである。

l) DMI を使うテンポラル・デカップリングされたイニシエータが制御をカーネルに渡した後、他のテンポラル・でカップリングされたイニシエータによって同じ DMI 領域が、現クォンタム時間内で無効化することがあり得る。これは、一般的にテンポラル・デカップリングに本来ある基本的な不正確さを表しているが、この節で与えているルールに違反するものではない。

8.3.9 DMIヒントを使った最適化

e) DMI ヒント、つまり DMI 許可アトリビュートは、DMI アクセスの繰り返しポーリングをなくすことによってシミュレーション・スピードを最適化するメカニズムである。DMI ポインタが使える状態か確認するために get_direct_mem_ptr を呼び出すかわりに、イニシエータがトランスポート・インタフェースを通過する通常のトランザクションの DMI 許可アトリビュートを確認することができる。

f) 汎用ペイロードによって DMI 許可アトリビュートは提供される。ユーザ定義のトランザクションによって同じメカニズムを実装できる。その場合ターゲットによって DMI 許可アトリビュートの値を適切に設定すべきである。

g) DMI 許可アトリビュートを使うかどうかは自由に選択できる。イニシエータは汎用ペイロードの DMI 許可アトリビュートを自由に無視できる。

h) イニシエータが DMI 許可アトリビュートを有効にするための推奨手順は下記に示す。

vi. イニシエータが有効な DMI 領域のキャッシュに対するアドレスを確認する。

vii. もし DM ポインタが存在しないなら、イニシエータはトランスポート・インタフェースを通して通常のトランザクションを実行する。

viii. その後、イニシエータはトランザクションの DMI 許可アトリビュートを確認する。

ix. もしアトリビュートが DMI を許可していることを示していれば、イニシエータは get_direct_mem_ptr を呼び出す。

x. イニシエータはその呼び出しからの戻り値により有効な DMI 領域のキャッシュを修正する。

8.4 デバッグ・トランスポート・インタフェース

8.4.1 イントロダクション

デバッグ・トランスポート・インタフェースは、ターゲットのストレージに対して読み込み及び書き込みを行うための手段を提供する。このとき、通常のトランスポート・インタフェースで使われるイニシエータからターゲットへのフォワード・パスを経由するが、遅延、wait、イベント通知、副作用は一切発生させない。言い換えるとデバッグ・トランスポート・インタフェースは邪魔にはならない。デバッグ・トランスポート・インタフェースは、トランスポート・インタフェースと同じパスを通るため、デバッグ・トランスポート・インタフェースの実装は通常のトランザクションと同じアドレス変換を実行する。

例えばデバッグ・トランスポート・インタフェースによって、ISS に接続されたソフトウェアデバッガが、シミュレーション上の CPU モデルからアクセス可能なシステム内のメモリデータを読み出したり、書き込んだりできる。またデバッグ・トランスポート・インタフェースによって、イニシエータがシステムメモリの内容をシミュレーション中に解析目的でスナップショットを取ることや、エラボレーション終了時にシステム内のメモリのある領域を初期化することも可能である。

デフォルトのデバッグ・トランザクション型は tlm_generic_payload であり、使われているトランザクション・オブジェクトのコマンド、アドレス、データ長及びデータ・ポインタ・アトリビュートのみを利用する。デバッグ・トランザクションは、トランスポート・インタフェースにおける拡張と同じアプローチを取る。つまり、デバッグ・トランザクションは無視可能拡張を含むかもしれないが、無視不可能あるいは必須拡張の場合は新しいプロトコル・トレイツ・クラスを定義する必要がある（tlm_generic_payload の型定義の内容を含む「7.2.2 tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義」を参照）。

8.4.2 クラス定義

```
namespace tlm {

  template <typename TRANS = tlm_generic_payload>
  class tlm_transport_dbg_if : public virtual sc_core::sc_interface
  {
  public:
    virtual unsigned int transport_dbg(TRANS& trans) = 0;
  };

} // namespace tlm
```

8.4.3 TRANテンプレート引数とtlm_generic_payloadクラス

f) tlm_transport_dbg_if テンプレートはデバッグ・トランザクション・クラスの型でパラメタライズされている。

g) デバッグ・トランザクション・クラスには、デバッグ・アクセスに必要なコマンド、アドレス、データ長及びデータ・ポインタをターゲットに教えるためのアトリビュートが含まれる。

基本プロトコルの場合、これらは汎用ペイロードの対応するアトリビュートである。

h) TRANS テンプレート引数のデフォルト値は tlm_generic_payload クラスである。

i) 相互利用性を最大化するために、デバッグ・トランザクション・クラスは tlm_generic_payload であるべき。無視不可能拡張や他のトランザクション型を使えば、相互利用性が制限される。

j) もしアプリケーションがデバッグ・トランザクションにさらにアトリビュートを追加する必要がある場合には、関連性のないトランザクション・クラスに置き換えるよりも汎用ペイロードを拡張する方法を推奨する。その拡張が無視不可能または必須の場合には、新しいプロトコル・トレイツ・クラスを定義する必要がある。

8.4.4 ルール

s) transport_dbg への呼び出しは、通常のトランザクションで利用されるトランスポート・インタフェースと同じフォワード･パスに従う。

t) trans 引数はデバッグ・トランザクション・オブジェクトへの参照渡しである。

u) イニシエータはデバッグ･トランザクション･オブジェクトを生成し、管理する必要がある。また transport_dbg への引数で渡す前にオブジェクトの適切なアトリビュートを設定する必要がある。

v) トランザクション・オブジェクトのコマンド・アトリビュートは要求されているデバッグ・アクセスの種類を示すためにイニシエータによって設定され、インターコネクト・コンポーネントやターゲットから修正されない。基本プロトコルで、許可されている値は、ターゲットへの読み込みアクセスの TLM_READ_COMMAND、ターゲットと TLM_IGNORE_COMMAND への書き込みアクセスの TLM_WRITE_COMMAND である。

w) TLM_IGNORE_COMMAND と等しいコマンド・アトリビュートをもったトランザクションを受け取ったときは、ターゲットは読み込みや書き込みを実行すべきではないが、拡張されたデバッグ・トランザクションを実行する際の拡張を含む汎用ペイロードにおいてアトリビュートの値を使ってもよい。

x) トランスポート・トランザクションの場合のように、デバッグ・トランスポート・インタフェースを持つ無視不可能または必須の汎用ペイロードの拡張を使うためには、新しいプロトコル・トレイツ・クラスを定義する必要がある。

y) アドレス・アトリビュートはイニシエータによって読み込みまたは書き込みされる領域の最初のアドレスに設定される。

z) フォワード・パスに沿ってデバッグ・トランザクション・オブジェクトを通るインターコネクト・コンポーネントで、同じソケットの対応するトランスポート・インタフェースに、トランザクション・オブジェクトのアドレス・アトリビュートをデコード及び必要に応じて修正されるべきである。例えば、インターコネクト・コンポーネントで、ターゲットアドレス幅とシステム・メモリマッ上の場所に従ってアドレスをマスク（上位ビットを削減）する必要がある。

aa) インターコネクト・コンポーネントでは、アドレッシング・エラーを検知した場合 transport_dbg 呼び出しを通過させる必要はない。

bb) アドレス・アトリビュートは、もしデバッグ・ペイロードが複数のインターコネクト・コン

ポーネントを通って転送されれば、複数回修正されているかもしれない。デバッグ・ペイロードがイニシエータに戻ってきた時には、アドレス・アトリビュートのもともとの値は上書きされているかもしれない。

cc) データ長アトリビュートはイニシエータによって読み込むまたは書き込むバイト数に設定され、インターコネクト・コンポーネンツやターゲットによって修正されない。データ長アトリビュートは、ターゲットがバイトを読み込みまたは書き込みしない場合は、0 でもよい。データ・ポインタ・アトリビュートは null でもよい。

dd) データ・ポインタ・アトリビュートはイニシエータによってターゲットにコピーされる値（書き込み時）またはターゲットからコピーされる値（読み込み時）の配列のアドレスに設定され、インターコネクト・コンポーネントやターゲットによって修正されない。この配列はイニシエータによって割り当てられ、transport_dbg から戻ってくる前に削除されない。配列サイズは少なくともデータ長アトリビュートの値と同じである。データ長アトリビュートが 0 なら、データ・ポインタ・アトリビュートは null ポインタでもよく、配列をアロケートする必要はない。

ee) もし可能であれば、ターゲットにおける transport_dbg の実装は、指定されたアドレスを使って（インターコネクトを通してアドレス変換後に）指定されたバイト数を読み込みまたは書き込むようにする。書き込みコマンドの場合、ターゲットはデータ配列のコンテンツを修正してはいけない。

ff) トランスポート・インタフェースと一緒に利用されたときに、データ配列は汎用ペイロードのデータ配列と同じ構成を持つ。transport_dbg の実装は、ターゲット内のローカル・データ・ストレージの構成と汎用ペイロードの構成間で変換する必要がある。

gg) 基本プロトコルの場合、イニシエータはコマンド、アドレス、データ長及びデータ・ポインタ以外の汎用ペイロードのアトリビュートを設定する必要はない。ターゲットとインターコネクト・コンポーネントは他のすべてのアトリビュートを無視してもよい。特に、応答ステータス・アトリビュートは無視される。

hh) イニシエータは一つの呼び出しから次の呼び出し、デバッグ・トランスポート・インタフェース、トランスポート・インタフェース及び DMI に渡る呼び出しでトランザクション・オブジェクトを再利用してもよい。

ii) transport_dbg は、実際に読み込みまたは書き込みしたバイト数の回数を返す。それはデータ長アトリビュートの値よりも少ない。もしターゲットが実行できない時には、0 の値を返す。

jj) 直接的または間接的に transport_dbg は、wait を呼んではいけない。デバッグ書き込みコマンドを実行後するに反映される場合を置いておいて、インターコネクト・コンポーネントやターゲットの状態を修正してはいけない。

9. グローバル・クォンタム

9.1 イントロダクション

テンポラル・デカップリング機能は、SystemC のプロセスに対して、クォンタムと呼ばれる時間を実際のシミュレーション時刻より先に進めるもので、ルーズリー・タイムドコーディング・スタイルと関連付けて用いられる。テンポラル・デカップリング機能を用いると、スレッドのコンテクスト・スイッチとイベントを減らすことにより非常に高速なシミュレーション速度が実現できる。

テンポラル・デカップリングを行うプロセス間で明示的な同期を行う上で、クォンタム時間の使用は、厳密には必須ではない。プロセスは次の同期ポイントまで先行的に実行可能であるが、クォンタム時間を必要とするプロセスは、グローバル・クォンタムを使用すべきである。

テンポラル・デカップリング機能を用いる時には、b_transport()および nb_transport()メソッドの引数によって渡される時間は、通常、sc_time_stamp()の戻り値である現在のシミュレーション時刻とクォンタム境界からの相対時間として定義される。グローバル・クォンタムは、連続するクォンタム境界間のデフォルト時間間隔である。グローバル・クォンタム時間の値は、唯一の tlm_global_quantum クラスによって管理される。各プロセスはグローバル・クォンタム時間を使うべきだが、1 つのプロセスは、自身のローカル・クォンタム時間の計算を行うことだけが許可されている。

テンポラル・デカップリングの詳細については、「3.3.2 ルーズリー・タイムド・コーディング・スタイルとテンポラル・デカップリング」も参照のこと。

タイミング・アノテーションに関しては、「4.1.3 トランスポート・インタフェースにおけるタイミング・アノテーション」も参照のこと。

tlm_quantumkeeper ユーティリティ・クラスには、クォンタム時間を管理するためのメソッドが用意されている。クォンタム・キーパーの使い方に関しては、「12.2 クォンタム・キーパー」を参照のこと。

9.2 ヘッダファイル

グローバル・クォンタムのクラス定義は、tlm.h ヘッダファイルにある。

9.3 クラス定義

```
namespace tlm {
  class tlm_global_quantum
  {
  public:
    static tlm_global_quantum& instance();
    virtual ~tlm_global_quantum();
    void set( const sc_core::sc_time& );
    const sc_core::sc_time& get() const;
    sc_core::sc_time compute_local_quantum();
```

```
  protected:
    tlm_global_quantum();
  };
} // namespace tlm
```

9.4 tlm_global_quantumクラス

g) tlm_global_quantum クラスによって単一のグローバルなクォンタム時間が定義される。これがクォンタム時間のデフォルトとなる。テンポラル・デカップリングをサポートしているイニシエータは、グローバル・クォンタム時間の整数倍、あるいは必要であればより頻繁に同期を図るために用意されている。

h) それぞれのイニシエータはそれぞれ別のクォンタム時間を持つことも可能であるが、一般的には全てのイニシエータはグローバルなクォンタム時間を持つ。同期の回数が少ないイニシエータは他よりクォンタム時間を長く設定できるわけではあるが、一般的には、クォンタム時間が短いとシミュレーション時間に悪い影響を与える。

i) instance()メソッドは、唯一のグローバル・クォンタム・オブジェクトの参照を返す。

j) set()メソッドは、引数で受け取った値をグローバル・クォンタム値として設定する。

k) get()メソッドは、グローバル・クォンタムの値を返す。

l) compute_local_quantum()メソッドは、唯一のグローバル・クォンタムを元に、ローカル・クォンタムの値を計算して返す。ローカル・クォンタムはグローバル・クォンタムの次の整数倍数から、sc_time_stamp の値を引くことにより計算する。compute_local_quantum()がグローバル・クォンタムの整数倍のシミュレーション時間でコールされる場合は、ローカル・クォンタムはグローバル・クォンタムと等しい。そうでない場合は、ローカル・クォンタムはグローバル・クォンタムよりも小さい。

10. 統合インタフェースとソケット

10.1 統合インタフェース

10.1.1 イントロダクション

フォワード・トランスポート・インタフェース 、バックワード・トランスポート・インタフェースは、TLM-2.0 コア・インタフェースをグループ化している。これは、イニシエータ/ターゲット・ソケットによりグループ化される。

統合インタフェースには、トランスポート、DMI、デバッグ・トランスポート・インタフェースを含むが、TLM-1 コア・インタフェースは含まない。

フォワード・インタフェースは、イニシエータ・ソケットからターゲット・ソケットへのフォワード・パスにおけるメソッド呼び出しを提供する。バックワード I/F は、ターゲット・ソケットからイニシエータ・ソケットへのバックワード・パスにおけるメソッド呼び出しを提供する。ブロッキング・トランスポート・インタフェースとデバッグ・トランスポート・インタフェースはバックワード呼び出しパスを必要としない。

標準のイニシエータ・ソケットやターゲット・ソケットに関連のない新しいソケットクラスを定義し、結合インタフェースを使ってこれらのテンプレートクラスをテンプレート引数としてインスタンス化することは、技術的に可能であるが、相互利用性の観点からは推奨しない。一方、標準のソケットクラスから新しいソケットクラスを派生させる方法は、推奨する。

結合インタフェースのテンプレートは、プロトコル型クラスでパラメタライズされる。そのプロトコル・トレイツ・クラスは、フォワード・インタフェース、バックワード・インタフェースで使われる型、すなわちペイロード型とフェーズ型を定義する。プロトコル・トレイツ・クラスは、特定のプロトコルに関連付けられる。デフォルトのプロトコル型は tlm_base_protocol_types である。基本プロトコルに関しては、「8.2 基本プロトコル」を参照。

10.1.2 クラス定義

```
namespace tlm {
  // The default protocol traits class:
  struct tlm_base_protocol_types
  {
  typedef tlm_generic_payload tlm_payload_type;
  typedef tlm_phase tlm_phase_type;
  };
  // The combined forward interface:
  template< typename TYPES = tlm_base_protocol_types >
  class tlm_fw_transport_if
  : public virtual tlm_fw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
  typename TYPES::tlm_phase_type>
  , public virtual tlm_blocking_transport_if<typename TYPES::tlm_payload_type>
```

```
  , public virtual tlm_fw_direct_mem_if<    typename TYPES::tlm_payload_type>
  , public virtual tlm_transport_dbg_if<    typename TYPES::tlm_payload_type>
  {};                                                    // The combined backward interface:
  template < typename TYPES = tlm_base_protocol_types >
  class tlm_bw_transport_if
  : public virtual tlm_bw_nonblocking_transport_if<typename TYPES::tlm_payload_type ,
                                                   typename TYPES::tlm_phase_type >
  , public virtual tlm_bw_direct_mem_if
  {};
} // namespace tlm
```

10.2 イニシエータとターゲット・ソケット

10.2.1 イントロダクション

ソケットはポートとエクスポートを組み合わせたものである。イニシエータ・ソケットはフォワード・パス用のポートとバックワード・パスのためのエクスポートがあり、ターゲット・ソケットにはフォワード・パス用のエクスポートとバックワード・パス用のポートがある。また、ソケットは対向するソケットのポートとエクスポートの両方をそれぞれエクスポートとポートに割り当てるために SystemC ポートのバインディング・オペレータをオーバーロードする。ソケットを親から子または子から親のように階層的にバインドする場合、バインドする順番を慎重に考慮することが重要である。

イニシエータとターゲット・ソケットの両方が、C++の継承階層を使用することでコード化される。最上位の派生クラスの tlm_initiator_socket と tlm_target_socket のみが通常、直接アプリケーションで使用される。これら 2 つのソケットは、フォワードとバックワード・インタフェースによって使用される型を定義するプロトコル・トレイツ・クラスでパラメタ化される。ソケットが同一のプロトコル型である場合だけ、それらを一緒に割り当てることができる。デフォルト・プロトコル型は tlm_base_protocol_types クラスである。アプリケーションが新しいプロトコルを定義する場合、新しいプロトコルが汎用ペイロードに基づいているか否かに関係なく、新しいプロトコル・トレイツ・クラスによる統合インタフェース・テンプレートをインスタンス化するべきである。

イニシエータとターゲット・ソケットは以下の利点を提供する:

e) フォワード・パスとバックワード・パス両方の為に、トランスポート、ダイレクト・メモリ、デバッグ・トランスポート・インターフェースを 1 つのオブジェクトにグループ化する。

f) フォワード・パスとバックワード・パス両方のポートとエクスポートを、 1 回の呼び出しでバインドするメソッドを提供する。

g) 互換性のないプロトコル型でパラメタライズされたソケットに対する、強力な型チェック機能を提供する。

h) トランザクションを解釈するために利用できるバス幅パラメタを含む。

tlm_initiator_socket と tlm_target_socket クラスは、TLM2.0 標準のインタオペラビリティ・レイヤに

属する。更に、ユーティリティ・ネームスペース内にひとまとめにして、便利ソケットとして知られる派生ソケットクラスが提供されている。

10.2.2 クラス定義

```
namespace tlm {
  // Abstract base class for initiator sockets
  template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>
  >
  class tlm_base_initiator_socket_b
  {
  public:
    virtual ~tlm_base_initiator_socket_b() {}
    virtual sc_core::sc_port_b<FW_IF>     & get_base_port() = 0;
    virtual                       BW_IF   & get_base_interface() = 0;
    virtual sc_core::sc_export<  BW_IF>   & get_base_export() = 0;
  };
  // Abstract base class for target sockets
  template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>
  >
  class tlm_base_target_socket_b
  {
  public:
    virtual ~tlm_base_target_socket_b();
    virtual sc_core::sc_port_b<BW_IF>     & get_base_port() = 0;
    virtual sc_core::sc_export<FW_IF>     & get_base_export() = 0;
    virtual                       FW_IF   & get_base_interface() = 0;
  };
  // Base class for initiator sockets, providing binding methods
  template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>,
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
  >
```

```
class tlm_base_initiator_socket : public tlm_base_initiator_socket_b<BUSWIDTH, FW_IF, BW_IF>,
                                  public sc_core::sc_port<FW_IF, N, POL>
{
public:
  typedef FW_IF      fw_interface_type;
  typedef BW_IF      bw_interface_type;
  typedef sc_core::sc_port<fw_interface_type, N, POL>      port_type;
  typedef sc_core::sc_export<bw_interface_type>      export_type;
  typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
  base_target_socket_type;
  typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
  base_type;
  tlm_base_initiator_socket();
  explicit tlm_base_initiator_socket(const char* name);
  virtual const char* kind() const;

  unsigned int get_bus_width() const;

  void bind(base_target_socket_type& s);
  void operator() (base_target_socket_type& s);
  void bind(base_type& s);
  void operator() (base_type& s);
  void bind(bw_interface_type& ifs);
  void operator() (bw_interface_type& s);

  // Implementation of pure virtual functions of base class
  virtual sc_core::sc_port_b<FW_IF> & get_base_port() { return *this; }
  virtual      BW_IF      & get_base_interface() { return m_export; }
  virtual sc_core::sc_export<BW_IF> & get_base_export() { return m_export; }
  protected:
  export_type m_export;
};

// Base class for target sockets, providing binding methods
template <
  unsigned int BUSWIDTH = 32,
  typename FW_IF = tlm_fw_transport_if<>,
  typename BW_IF = tlm_bw_transport_if<>,
  int N = 1,
  sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
```

```
class tlm_base_target_socket : public tlm_base_target_socket_b<BUSWIDTH, FW_IF, BW_IF>,
public sc_core::sc_export<FW_IF>
{
public:
   typedef FW_IF      fw_interface_type;
   typedef BW_IF      bw_interface_type;
   typedef sc_core::sc_port<bw_interface_type, N, POL>      port_type;
   typedef sc_core::sc_export<fw_interface_type>      export_type;
   typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
   base_initiator_socket_type;
   typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
   base_type;
   tlm_base_target_socket();
   explicit tlm_base_target_socket(const char* name);
   virtual const char* kind() const;

   unsigned int get_bus_width() const;

   void bind(base_initiator_socket_type& s);
   void operator() (base_initiator_socket_type& s);
   void bind(base_type& s);
   void operator() (base_type& s);
   void bind(fw_interface_type& ifs);
   void operator() (fw_interface_type& s) ;
   int size() const;
   bw_interface_type* operator-> ();
   bw_interface_type* operator[] (int i);                    // Implementation of pure virtual functions of base class
   virtual sc_core::sc_port_b<BW_IF> & get_base_port() { return m_port; }
   virtual      FW_IF      & get_base_interface() { return *this; }
   virtual sc_core::sc_export<FW_IF> & get_base_export() { return *this; }
protected:
   port_type m_port;
};

// Principle initiator socket, parameterized with protocol traits class
template <
   unsigned int BUSWIDTH = 32,
   typename TYPES = tlm_base_protocol_types,
   int N = 1,
   sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
```

```
  class tlm_initiator_socket : public tlm_base_initiator_socket <
    BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
  {
  public:
    tlm_initiator_socket();
    explicit tlm_initiator_socket(const char* name);
    virtual const char* kind() const;
  };

  // Principle target socket, parameterized with protocol traits class
  template <
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm_base_protocol_types,
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
  >
  class tlm_target_socket : public tlm_base_target_socket <
    BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
  {
  public:
    tlm_target_socket();
    explicit tlm_target_socket(const char* name);
    virtual const char* kind() const;
  };
} // namespace tlm
```

10.2.3 tlm_base_initiator_socket_bとtlm_base_target_socket_b

c) 抽象ベースクラス tlm_base_initiator_socket_b と tlm_base_target_socket_b は、ソケットと関連付けられたポート、エクスポート、およびインタフェース・オブジェクトを返す、すべての派生クラスで上書きされる純粋仮想関数を宣言する。

d) これらソケットは、通常アプリケーションから直接使われることはない。

10.2.4 tlm_base_initiator_socketとtlm_base_target_socket

u) tlm_initiator_socket クラスに関して、name 引数をとるコンストラクタはモジュール階層にインスタンス名の文字列を設定するため、sc_port ベースクラスに属するコンストラクタに文字列引数を渡して、また、バックワード・パスの sc_export に対して同じ文字列に”_export”を付加した名前を設定する。"_export"の付加と sc_gen_unique_name の呼び出しは名前の衝突を回避する。例えば、tlm_initiator_socket(“foo")の呼び出しは、ポート名を"foo"に、エクスポート名を"foo_export"に設定する。デフォルト・コンストラクタについては、名前は、ポートに対しては sc_gen_unique_name("tlm_base_initiator_socket")を呼び出し、エクスポートに対しては sc_gen_unique_name("tlm_base_initiator_socket_export")を呼び出すことによって生成される。

v) tlm_base_target_socket クラスに関して、name の引数をとるコンストラクタは、sc_export ベースクラスに属するコンストラクタに文字列引数を渡して、モジュール階層にインスタンス名の文字列を設定する。また、同じ文字列に”_port”を付加した名前をバックワード・パスの sc_port に対して設定する。"_port"の付加と sc_gen_unique_name の呼び出しは、名前の衝突を回避するために行う。例えば、tlm_target_socket(“foo")の呼び出しはエクスポート名を"foo"に、ポート名を"foo_port"に設定する。デフォルト・コンストラクタについては、名前の生成は、ポートに対しては sc_gen_unique_name("tlm_base_target_socket")を呼び出し、エクスポートに対しては sc_gen_unique_name("tlm_base_target_socket_port")を呼び出すことによって行われる。

w) kind メソッドは、クラス名を C 文字列として返す。すなわち、それぞれ、”tlm_base_initiator_socket”または、”tlm_base_target_socket”を返す。

x) get_bus_width メソッドは、BUSWIDTH テンプレート引数の値を返す。

y) テンプレート引数 BUSWIDTH は、ソケットを介して転送される、個々のデータ・ワードのワード長を決定するもので、ワード中のビット数で表される。バースト転送では、BUSWIDTH は各ビート中のビット数を決定する。この属性の厳密な解釈はトランザクション型に依存する。汎用ペイロードにおける BUSWIDTH の意味は、「7.11 データ長アトリビュート」を参照。

z) ソケットとソケットをバインドする場合、二つのソケットは同一の BUSWIDTH テンプレート引数値を持たなければならない。イニシエータまたはターゲットにおける実行可能コードは、BUSWIDTH を参照し、それに従い動作する。

aa) ソケットを引数とする bind、operator（）メソッドは、メソッドへの引数として渡されたソケット・インスタンスと、それぞれのメソッドが属するソケット・インスタンスをバインドするものとする。

bb) インタフェースを引数とする bind、operator（）メソッドは、メソッドへの引数として渡されたチャネル・インスタンスと、メソッドが属するソケット・インスタンスのエクスポートをバインドするものとする。（チャネルはインタフェースを実装するクラスを表す SystemC 用語。）

cc) イニシエータ・ソケットをターゲット・ソケットにバインドする場合、bind、operator()メソッドは、イニシエータ・ソケットの port をターゲット・ソケットの export にバインドする。また、ターゲット・ソケットの port とイニシエータ・ソケットの export をバインドする。これは、双方のソケットが同じ階層に置かれている場合に適用される。

dd) イニシエータ・ソケットまたはターゲット・ソケットの、どちらかのソケットのバインド・メソッドまたは operator()を呼ぶことによって、イニシエータ・ソケットをターゲット・ソケットに全く同じ効力でバインドすることができる。どちらの場合でも、フォワード・パスはイニシエータ・ソケットからターゲット・ソケットの向きとなる。

ee) イニシエータ・ソケットをイニシエータ・ソケットにバインドする場合、または、ターゲット・ソケットをターゲット・ソケットにバインドする場合、bind、operator()メソッドは、それぞれもう片方のソケットの port と port、export と export をバインドする。これは、階層をまたぐバインド、すなわち、子モジュール上のソケットを親モジュール上のソケットにバインドする場合、または、親モジュール上のソケットを子モジュールのソケットにバインドする場合のように、トランザクションがモジュール階層上下に受け渡される場合に適用される。

ff) 階層バインディングにおいては、正しい順序でソケットをバインドする必要がある。イニシエータ・ソケットをイニシエータ・ソケットにバインドする場合は、子ソケットが親ソケットに対してバインドされなければならない。ターゲット・ソケットをターゲット・ソケットにバインドする場合は、親ソケットが子ソケットに対してバインドされなければならない。このルールは、tlm_base_initiator_socket が sc_port から派生しており、tlm_base_target_socket が sc_export から派生していることと合致している。階層をさかのぼる場合は、port から port へ、トップ階層においては port から export へ、階層を下がる場合は export から export へバインドされなければならない。

gg) tlm_base_initiator_socket と tlm_base_target_socket の二つのソケットを一緒にバインドするために、フォワード・インタフェース型、バックワード・インタフェース型、バス幅は同じでなければならない。

hh) ターゲット・ソケットの size メソッドは、バックワード・パスのターゲット・ソケットのポートの size メソッドを呼び出し、そのポートの size で返される値を返す。

ii) ターゲット・ソケットの operator->メソッドは、バックワード・パスのターゲット・ソケットのポートの operator->メソッドを呼び出し、そのポートの operator->で返される値を返す。

jj) ターゲット・ソケットの operator[]メソッドは、バックワード・パスのターゲット・ソケットのポートの operator[]メソッドを同じ引数で呼び出し、そのポートの operator[]で返される値を返す。

kk) tlm_base_initiator_socket と class tlm_base_target_socket は、マルチ・ソケットとしてそれぞれ動作する。すなわち、一つのイニシエータ・ソケットは、複数のターゲット・ソケットとバインドすることができ、一つのターゲット・ソケットは、複数のイニシエータ・ソケットとバインドすることができる。2 つのクラステンプレートは、テンプレート・パラメタとしてバインディングの数を示すパラメタとポートバインディングポリシーを指定するパラメタを取る。それらは、関連する sc_port テンプレートをインスタンス化する際にパラメタ化する実装中で使用される。

ll) tlm_base_initiator_socket または tlm_base_target_socket が複数回バインドされた場合、operator[]メソッドは接続先の対応するオブジェクトを指定する目的に使用できる。インデックス値は bind または operator()メソッドがソケットをバインドするために呼び出された順序で決定される。しかしながら、そのようなソケットによって受信される着信インタフェース・メソッド呼び出しは、呼び出し側を特定するメカニズムを持たないという意味で匿名となる。一方、便利ソケットでは、そのようなメカニズムが提供される。「9.1.4 マルチ・ソケット」を参照。

mm) 例えば、一つのソケットが、二つの別々のターゲットにバインドされる場合を考える。socket[0]->nb_transport_fw(...)と socket[1]->nb_transport_fw()は、2 個のターゲットを扱うことができるが、これら二つのターゲットから入ってくる nb_transport_bw()で呼び出し側を特定する方法はない。

nn) 仮想メソッド get_base_port と get_base_export の実装は、ソケットの port と export のオブジェクトをそれぞれ返す。仮想メソッド get_base_interface の実装は、イニシエータ・ポートの場合は export オブジェクトを、ターゲット・ソケットの場合はソケット・オブジェクトそのものを返すものとする。

10.2.5 tlm_initiator_socketとtlm_target_socket

e) tlm_initiator_socket と tlm_target_socket は、プロトコル・トレイツ・クラスをテンプレート・パラメタとしてとる。アプリケーションは、通常ベース・ソケットよりこれらのソケット（またはこれらから派生する便利ソケット）を使用するべきである。

f) tlm_initiator_socket と tlm_target_socket クラスのコンストラクタは、対応するベースクラスのコンストラクタを呼ぶものとする。char*引数が存在する場合はそれを渡す。

g) tlm_initiator_socket と tlm_target_socket を相互にバインドする場合は、同じプロトコル・トレイツ・クラス（デフォルトは tlm_base_protocol_types）、および同じバス幅を持つ必要がある。新規プロトコル型に対しても、新しいプロトコル・トレイツ・クラスを定義することにより、ソケット間の強力な型チェックが可能である。これは汎用ペイロードに基づいたプロトコルでなくても可能である。

h) kind メソッドは、クラス名を C 文字列として返す。すなわち、それぞれ、”tlm_initiator_socket”または”tlm_target_socket”を返す。

例

```
#include <systemc>
#include "tlm.h"
using namespace sc_core;
using namespace std;
struct Initiator: sc_module, tlm::tlm_bw_transport_if<>          // Initiator implements the bw interface
{
  tlm::tlm_initiator_socket<32> init_socket;                     // Protocol types default to base protocol
  SC_CTOR(Initiator) : init_socket("init_socket") {
    SC_THREAD(thread);
    init_socket.bind( *this );                                   // Initiator socket bound to the initiator itself
  }
  void thread() {                                                // Process generates one dummy transaction
    tlm::tlm_generic_payload trans;
    sc_time delay = SC_ZERO_TIME;
    init_socket->b_transport(trans, delay);
  }
  virtual tlm::tlm_sync_enum nb_transport_bw(
    tlm::tlm_generic_payload& trans,
    tlm::tlm_phase& phase,
    sc_core::sc_time& t) {
    return tlm::TLM_COMPLETED;                                   // Dummy implementation
  }
  virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
   { }                                                           // Dummy implementation
```

```
};

struct Target: sc_module, tlm::tlm_fw_transport_if<>                    // Target implements the fw interface
{
  tlm::tlm_target_socket<32> targ_socket;                               // Protocol types default to base protocol
  SC_CTOR(Target) : targ_socket("targ_socket") {
    targ_socket.bind( *this );                                          // Target socket bound to the target itself
  }
  virtual tlm::tlm_sync_enum nb_transport_fw(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_core::sc_time& t) {
      return tlm::TLM_COMPLETED;                                        // Dummy implementation
    }
  virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
    { }                                                                 // Dummy implementation
  virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmi_data)
    { return false; }                                                   // Dummy implementation
  virtual unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
    { return 0; }                                                       // Dummy implementation
};

SC_MODULE(Top1)                         // Showing a simple non-hierarchical binding of initiator to target
{
  Initiator *init;
  Target *targ;
  SC_CTOR(Top1) {
    init = new Initiator("init");
    targ = new Target("targ");
    init->init_socket.bind(targ->targ_socket);                          // Bind initiator socket to target socket
  }
};

struct Parent_of_initiator: sc_module                                   // Showing hierarchical socket binding
{
  tlm::tlm_initiator_socket<32> init_socket;
  Initiator* initiator;
  SC_CTOR(Parent_of_initiator) : init_socket("init_socket") {
    initiator = new Initiator("initiator");
    initiator->init_socket.bind( init_socket );             // Bind initiator socket to parent initiator socket
  }
};
```

```
struct Parent_of_target: sc_module
{
   tlm::tlm_target_socket<32> targ_socket;
   Target* target;
   SC_CTOR(Parent_of_target) : targ_socket("targ_socket") {
      target = new Target("target");
      targ_socket.bind( target->targ_socket );          // Bind parent target socket to target socket
   }
};

SC_MODULE(Top2)
{
   Parent_of_initiator *init;
   Parent_of_target *targ;
   SC_CTOR(Top2) {
      init = new Parent_of_initiator("init");
      targ = new Parent_of_target("targ");
      init->init_socket.bind(targ->targ_socket);        // Bind initiator socket to target socket at top level
   }
};
```

11. 汎用ペイロード

11.1 イントロダクション

汎用ペイロードはコア・インタフェースを通過するトランザクション・オブジェクトのための TLM-2.0 が提供するクラス型である。汎用ペイロードは基本プロトコルと密接に関係する。基本プロトコル自身は汎用ペイロードを使ったときの相互利用性を保証するためのルールを定義している。「8.2 基本プロトコル」を参照。

汎用ペイロードはメモリ・マップド・バスモデルの相互利用性を向上することを目的としており、2 つのレベルでこれらを実現する。一つ目は、バス・プロトコルの正確な詳細が重要でないメモリ・マップド・バスの抽象モデルを作るとき、汎用ペイロードは当面の相互利用性を保証する既成の一般目的のペイロードを提供する。そして、同時に無視可能なアトリビュートのための拡張メカニズムも提供する。二つ目は、汎用ペイロードは特定のバス・プロトコルの詳細モデルのベースとなり得ることである。異なるプロトコル間のブリッジもしくはアダプタが必要なとき、ときにはブリッジを書くことが些細なことになるほど実装コスト削減での利点があり、シミュレーション速度を向上させる点での利点がある。

汎用ペイロードは特にメモリ・マップド・バスのモデリングを目的とする。これは、典型的なメモリ・マップド・バス・プロトコルのアトリビュートであるコマンド、アドレス、データ、バイト・イネーブル、シングルワード転送、バースト転送、ストリーミング、応答ステータスを持つ。また、汎用ペイロードはメモリ・マップド・バス以外のプロトコルのモデリングのベースとして使うこともできる。

汎用ペイロードは典型的なメモリ・マップド・バス・プロトコルの全てのアトリビュートを持つわけではない。しかし、拡張メカニズムを持つため、アプリケーションはそれ自身の特別のアトリビュートを追加することができる。

特殊なプロトコルがバスベースかどうかに関わらず、モデリングと相互利用性はプロトコルのオーナーが責任を持つもので、OSCI の範囲外である。それは、プロトコル・オーナーや、熟練者が量産しようとしているモデルの内容や、それらの特定のプロトコルの記述ガイドラインに期する問題である。しかしながら、汎用ペイロードはここでも利用可能である。なぜならば、汎用ペイロードは、モデル作成の共通のスターティング・ポイントを提供し、多くの場合、トランザクション・レベル・モデルで異なるプロトコル間のブリッジ開発のコストを削減する。

汎用ペイロードはイニシエータ・ソケット、ターゲット・ソケットとともに使うことを推奨する。ソケットは汎用ペイロードのデータ配列を解釈する際に使われるバス幅パラメタ、フォワード・パス、バックワード・パス、汎用ペイロードを使う使わないに関わらず異なるプロトコル間の強力な型チェッキングのメカニズムを提供する。

汎用ペイロードはブロッキング・トランスポート・インタフェースとノンブロッキング・トランスポート・インタフェースで使うことができる。また、汎用ペイロードはダイレクト・メモリ・トランスポート・インタフェース、デバッグ・トランスポート・インタフェースでも使うことができる。その場合は、制限されたアトリビュートだけが使われている。

11.2 拡張性と相互利用性

汎用ペイロードのゴールはメモリ・マップド・バス・モデル間の相互運用性を可能にすることであるが、すべてのバスが同様に作成されるというわけではない。2 つのトランザクション・レベル・モデルがあり、それらが異なるプロトコルを用い詳細なレベルでプロトコルがモデル化されているなら、通常の物理的システムで行なわれるように、2 つのモデル間のプロトコル変換を行い通信できるようにアダプタやブリッジが挿入されなければならない。これに対して、設計フローの初期段階で作られる多くのトランザクション・レベル・モデルは、特定のプロトコル規格の詳細を考慮することはない。そのようなモデルでは、特定のアドレスで始まる 1 ブロックのデータをコピーするので十分であり、それらのモデルにおいて、素晴らしい相互運用性を得るのに直接汎用ペイロードが使用できる。

汎用ペイロード拡張機能は、どんな型のいろいろな拡張もトランザクション・オブジェクトに定義して、追加することが可能である。各拡張は新しい属性の集合を表現し、トランザクション・オブジェクトと共に伝送される。イニシエータ、相互接続コンポーネント、およびターゲットは同様に拡張を作成し、追加し、書いて、読むことができる。拡張機能自体は少しの制限も課さない。もちろん、この拡張機能の規律のない使用が相互運用性を危うくするので、規律ある使用が強く奨励される。しかし、柔軟性を必要とするならそれは可能である!

拡張機能の使用には、ソケットをバインドする時のコード化の簡便さの増大とコンパイル時の型チェックが減少するというトレードオフがある。もし、汎用ペイロード拡張の規律のない使用が許されているとするなら、各アプリケーションは、各インターコネクト・コンポーネントおよびターゲット間の明示的なランタイム・チェックを行なうことにより拡張間のいかなる非互換性をも検出することが強いられる。しかし、特定の拡張の存在を強制するメカニズムが全くないであろう。TLM-2.0 規格は、これらの落とし穴を避けるために具体的なコード化ガイドラインを定める。

3 種類の、そして 3 種類のみの推奨される拡張手法がある。これらはブロッキングおよびノン・ブロッキング・トランスポート・インタフェースのトランザクション・テンプレート引数 TRANS、および統合インタフェースのテンプレート引数 TYPES に関する代替手段である。

d) 汎用ペイロードをそのまま、無視可能な拡張と共に使用する

e) tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義する

f) 新しいプロトコル・トレイツ・クラスと新しいトランザクション型を定義する

a→b→c の順に相互利用性が損なわれる。また、これらの手法を a, b, c の順に以下に説明する。

汎用ペイロードから新クラスを派生させる事は可能であるが、それが相互運用性のための推奨の手法でないということを強調しておかなければならない。

また、これらの 3 つのオプションが単一のシステム・モデルの中で混在するかもしれないということも強調しておかなければならない。特に、拡張機能が効率的な相互運用性を可能にするように設計されているので、最初の 2 つのオプションの混在は価値がある。

11.2.1 汎用ペイロードをそのまま、無視可能な拡張と共に使用

e) この場合、トランザクション型は tlm_generic_payload である。そして、フェーズ型は tlm_phase

である。そして、統合インタフェースへのプロトコル・トレイツ・クラスは tlm_base_protocol_types である。これらはそれぞれ、トランスポート・インタフェースに関する TRANS 引数と統合インタフェースに関する TYPES 引数に対するデフォルト値である。基本プロトコルの標準のイニシエータとターゲット・ソケットを使用するモデルは、それらのモデルが汎用ペイロードの意味論とベース・プロトコルを尊重する限り、そのようなモデルと共に相互利用できるだろう。「8.2 基本プロトコル」を参照。

f) この場合、汎用ペイロードの拡張や拡張されたフェーズは"無視可能"とする。"無視可能"とは拡張されたコンポーネント以外のコンポーネントが、この拡張が存在しないものとして動作することを許容することである。「7.20.1.1 無視可能拡張」を参照。

g) 拡張が無視可能であるなら、定義上、ターゲットでその拡張のサポートを強制するコンパイル時のチェックは必要とされない。そして、実際、無視可能拡張機能はコンパイル時のチェックをサポートしない。

h) 汎用ペイロードは、本質的にプロトコルの小さなバリエーションをサポートする。一般的な原則として、ターゲットは汎用ペイロードのあらゆる機能をサポートすることが推奨される。しかし、例えば、特定のコンポーネントはバイト・イネーブルをサポートするかもしれないし、しないかもしれない。汎用ペイロードの特定の機能をサポートできないターゲットは、標準のエラー応答を発生させなければならない。これは汎用ペイロードの仕様の一部であるとして考えられるべきである。

11.2.2 tlm_generic_payloadに対するtypedefを含む新しいプロトコル・トレイツ・クラスを定義

g) この場合、トランザクション型は tlm_generic_payload、フェーズ型は tlm_phase であるが、ソケットを特殊化したプロトコル・トレイツ･クラスは、デフォルトの tlm_base_protocol_types ではなく、新しいアプリケーションで定義されたクラスである。これは、拡張汎用ペイロードが異なった型として扱われるのを保証し、イニシエータとターゲット・ソケットがバインドされたとき、コンパイル時の型チェック機能を提供する。

h) 新しいプロトコル型は独自のルールを持つが、独自ルールは、汎用ペイロードのメモリ・マネージメント・ルール(7.5 汎用ペイロード・メモリ管理を参照)と属性の変更可能性のためのルール（7.7 デフォルト値と属性の更新性を参照）を含む、基本プロトコルのルールを拡張、またはそれと矛盾していてもよい。しかしながら、一貫性と相互運用性のために、基本プロトコルの規則とコーディング･スタイルにできるだけ従うことを推奨する。8.2 ベース・プロトコルを参照のこと。

i) 汎用ペイロード拡張メカニズムは、無視可能または無視不可能、必須拡張に制限なく利用される。すべての拡張の動作は新しいプロトコル・トレイツ・クラスを用いて完全にドキュメント化すべきである。

j) トランザクション型が tlm_generic_payload であるので、トランザクションは汎用ペイロード型を使用する相互接続コンポーネントあるいはターゲットを通して転送でき、すべての拡張を含んだ形で、全体として複製を作ることができる。これは再利用できるコンポーネントを作成したり、アダプタを作成したり、異なるプロトコル間のブリッジを作成するための良い出発点を提供するが、ユーザは非常に慎重に拡張汎用ペイロードのセマンティックスを検討しなければならない。

k) イニシエータからゼロまたは何個かの相互接続コンポーネントを通ってターゲットへのトランザクションが後に続く経路の全長に沿って全く同じプロトコル・トレイツ・クラスを使用することは普通である。しかしながら、アダプタやバス・ブリッジを、あるプロトコルでトランザクションを受け取り、他のプロトコルの送出されるトランザクションとして変換するインターコネクト・コンポーネントとしてモデル化することも可能である。到着するトランザクションに対してはターゲットとして動作し、送出されるトランザクションに対してはイニシエータとして動作するトランザクション・ブリッジを作成することも可能である。

l) 異なるプロトコル・トレイツ・クラスを利用するソケット間で汎用ペイロードを渡す場合、ユーザは拡張としてではなく、汎用ペイロードとして認識しているコンポーネントを通してトランザクションを転送できることを保証するために、非常に慎重にそれぞれの拡張のセマンティックスを検討しなければならない。一般的な規則はない。不運な事故なしで、拡張を認識しないコンポーネントを通してある種の拡張を転送できる場合もある。例えば、データのセキュリティー・レベルを指定する属性がある。他の拡張は、明示的な追加を必要とするか、または全くサポートできないであろう。例えば、インターコネクトがロックされること、と指定する属性がある。

11.2.3 新しいプロトコル・トレイツ・クラスと新しいトランザクション型を定義

e) この場合は、トランザクション型は汎用ペイロードとは関係がない。

f) 新しいプロトコル・トレイツ・クラスは統合インタフェースとソケットをパラメタライズするように定義する必要がある。

g) 新しいトランザクション型が汎用ペイロードとかなり異なるか、非常に特別なプロトコルを表すとき、この選択は正当化される。

h) 最大限の相互運用性のために汎用ペイロードを使用するつもりであるなら、お勧めの手法は、新しいクラスを定義して使用するより、前2つの節の1つで説明される汎用ペイロードを使用することである。

11.3 汎用ペイロード・アトリビュートとメソッド

汎用ペイロード・クラスはプライベートなアトリビュートとそれに値をセット/ゲットするパブリックなアクセス関数を持つ。それらのアクセス関数の正確な実装は、実装定義(実装依存)である。

アトリビュートの大部分の設定はイニシエータが行い、相互接続コンポーネントやターゲットで変更しないものとする。アドレスやDMI許可情報、レスポンス・ステータスと拡張属性はインターコネクトやターゲットが変更する場合がある。リード・コマンドの場合、ターゲットはデータ配列を変更する。

11.4 クラス定義

```
namespace tlm {
  class tlm_generic_payload;
  class tlm_mm_interface {
  public:
    virtual void free(tlm_generic_payload*) = 0;
    virtual ~tlm_mm_interface() {}
```

```
};
unsigned int max_num_extensions();    class tlm_extension_base
{
public:
   virtual tlm_extension_base* clone() const = 0;
   virtual void free() { delete this; }
   virtual void copy_from(tlm_extension_base const &) = 0;
protected:
   virtual ~tlm_extension_base() {}
};
template <typename T>
tlm_extension : public tlm_extension_base
{
public:
   virtual tlm_extension_base* clone() const = 0;
   virtual void copy_from(tlm_extension_base const &) = 0;
   virtual ~tlm_extension() {}
   const static unsigned int ID;
};
enum tlm_command {
   TLM_READ_COMMAND,
   TLM_WRITE_COMMAND,
   TLM_IGNORE_COMMAND
};
enum tlm_response_status {
   TLM_OK_RESPONSE = 1,
   TLM_INCOMPLETE_RESPONSE = 0,
   TLM_GENERIC_ERROR_RESPONSE = -1,
   TLM_ADDRESS_ERROR_RESPONSE = -2,
   TLM_COMMAND_ERROR_RESPONSE = -3,
   TLM_BURST_ERROR_RESPONSE = -4,
   TLM_BYTE_ENABLE_ERROR_RESPONSE = -5
};
#define TLM_BYTE_DISABLED 0x0
#define TLM_BYTE_ENABLED 0xff    class tlm_generic_payload {
public:
   // Constructors and destructor
   tlm_generic_payload();
   explicit tlm_generic_payload( tlm_mm_interface* );
   virtual ~tlm_generic_payload();
private:
```

```
    // Disable copy constructor and assignment operator
    tlm_generic_payload( const tlm_generic_payload& );
    tlm_generic_payload& operator= ( const tlm_generic_payload& );
  public:
    // Memory management
    void set_mm( tlm_mm_interface* );
    bool has_mm() const;
    void acquire();
    void release();
    int get_ref_count() const;
    void reset();
    void deep_copy_from( const tlm_generic_payload& ) const;
    void update_original_from( const tlm_generic_payload & , bool use_byte_enable_on_read = true );
    void update_extensions_from(const tlm_generic_payload & );
    void free_all_extensions();                                   // Access methods
    tlm_command get_command() const;
    void set_command( const tlm_command );
    bool is_read();
    void set_read();
    bool is_write();
    void set_write();
    sc_dt::uint64 get_address() const;
    void set_address( const sc_dt::uint64 );
    unsigned char* get_data_ptr() const;
    void set_data_ptr( unsigned char* );
    unsigned int get_data_length() const;
    void set_data_length( const unsigned int );
    unsigned int get_streaming_width() const;
    void set_streaming_width( const unsigned int );
    unsigned char* get_byte_enable_ptr() const;
    void set_byte_enable_ptr( unsigned char* );
    unsigned int get_byte_enable_length() const;
    void set_byte_enable_length( const unsigned int );            // DMI hint
    void set_dmi_allowed( bool );
    bool is_dmi_allowed() const;
    tlm_response_status get_response_status() const;
    void set_response_status( const tlm_response_status );
    std::string get_response_string();
    bool is_response_ok();
    bool is_response_error();                                     // Extension mechanism
    template <typename T> T* set_extension( T* );
```

```
    tlm_extension_base* set_extension( unsigned int , tlm_extension_base* );
    template <typename T> T* set_auto_extension( T* );
    tlm_extension_base* set_auto_extension( unsigned int index , tlm_extension_base* );
    template <typename T> void get_extension( T*& ) const;
    template <typename T> T* get_extension() const;
    tlm_extension_base* get_extension( unsigned int ) const;
    template <typename T> void clear_extension( const T* );
    template <typename T> void clear_extension();
    template <typename T> void release_extension(T* ext);
    template <typename T> void release_extension();
    void resize_extensions();
  };
} // namespace tlm
```

b) 」を参照。

c) 汎用ペイロード・クラス tlm_generic_payload。「7 汎用ペイロード」を参照。

d) フェーズクラス tlm_phase

基本プロトコルのルールは、汎用ペイロードおよびフェーズの拡張を許容するが、それは、拡張が無視可能な場合のみに限られる。 無視可能でない拡張は、新しいプロトコル・トレイツ・クラスの定義を必要とする。「7.2.1 汎用ペイロードをそのまま、無視可能な拡張と共に」を参照。

基本プロトコルは事前に定義された tlm_base_protocol_types クラスによって表される。しかしながらこのクラスでは、2 種類の型だけしか定義されていない。（ソケットへのテンプレート引数としての）このクラスを使用するすべてのコンポーネントは、基本プロトコルのルールを遵守する規約に従う必要がある。

新しいプロトコル・トレイツ・クラスを定義することが必要な場合（例えば、基本プロトコルの機能が特定のプロトコルをモデル化するためには不十分な場合など)、新しいプロトコル・トレイツ・クラスは、それに関連している基本プロトコルのルールに優先する。しかしながら、一貫性と相互利用性の観点から、新しいプロトコルの特性クラスに関連するルールとコーディング・スタイルは、できるだけ基本プロトコルに従うことを推奨する。「7.2.2 tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義」を参照。

本規格のこのセクションは、特に基本プロトコルについての記載ではあるが、他のプロトコルをモデル化するときのガイドとして使用してもかまわない。他のプロトコル・トレイツ・クラスで表現される特定のプロトコルは、追加のフェーズを含んだり、タイミング・アノテーション、トランザクション・オーダリングなどのために独自のルールを採用したりしてもかまわない。そのことにより、基本プロトコルとの互換性は失われるかもしれない。

11.4.1 クラス定義

```
namespace tlm {
  struct tlm_base_protocol_types
  {
    typedef tlm_generic_payload tlm_payload_type;
```

```
    typedef tlm_phase tlm_phase_type;
  };
} // namespace tlm
```

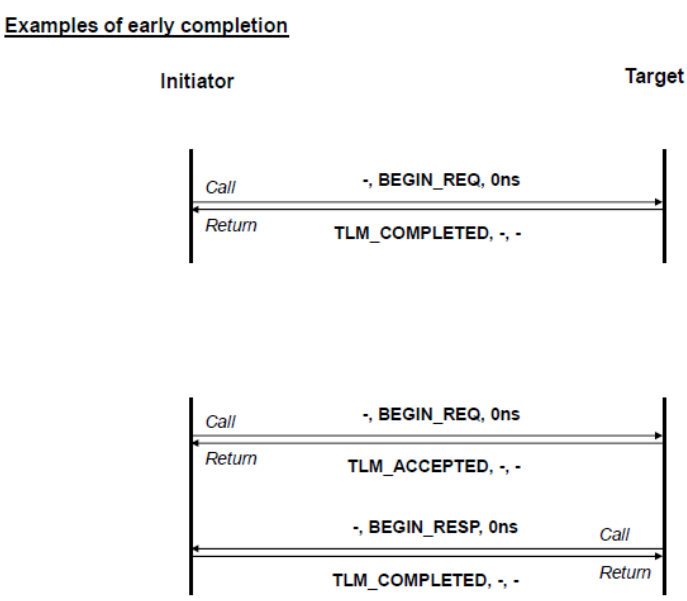
11.4.2 基本プロトコル・フェーズ・シーケンス

a) 基本プロトコルでは、ブロッキング・トランスポート・インタフェース、ノンブロッキング・トランスポート・インタフェース、または両方を同時に使用することができる。ブロッキング・トランスポート・インタフェースはフェーズ情報を運ばない。基本プロトコルとともに使用される場合、nb_transport を呼び出す順番を管理する制約は b_transport を呼び出す順番を管理する制約よりも強い。したがって、 nb_transport はアプロキシメイトリー・タイムド記述に適しており、b_transport はルーズリー・タイムド記述に適している。

b) それぞれのトランザクションに対するフェーズ遷移の完全なシーケンスは、以下の通りである。

BEGIN_REQ

Figure 13



g) フェーズ END_RESP への遷移はまた、そのホップに関してトランザクションの終わりを示すものとする、その場合、呼び出し先関数は TLM_COMPLETED の値を返す義務は無い。

h) BEGIN_REQ を受けとった後にイニシエータ側の方向に TLM_COMPLETED が戻ってきた場合、暗黙の END_REQ と暗黙の BEGIN_RESP も一緒に戻されている。したがって、イニシエータは、汎用ペイロードの応答ステータスを確認しなければならない。また、ただちに次のトランザクションである BEGIN_REQ を送信することができる。

i) BEGIN_REQ を受けとった後に返される TLM_COMPLETED は、それと一緒に暗黙の BEGIN_RESP も返されることになる。そのためレスポンス除外ルールにより、該当するソケットを経由して進行中のレスポンスが既に存在する場合、この状況(TLM_COMPLETE を返すこと)は禁じられている。この状況では、呼び出し先関数は、TLM_COMPLETED の代わりに TLM_ACCEPTED を返さなければならず、次のイニシエータ側へのレスポンスを送信する前に END_RESP を待たなければならない。

j) BEGIN_REQ を受けとった後に返される TLM_COMPLETED は、トランザクションの終わりを示すため、インターコネクト・コンポーネントまたはイニシエータは、次に、その同じトランザクションのために同じソケットを経由して END_RESP を送信する事は禁じられる。

k) BEGIN _RESP を受けとった後にあるコンポーネントが TLM_COMPLETED をターゲット側方向に返す場合、これは TLM_COMPLETED と一緒に暗黙の END_RESP を返すことになる。

l) もし、あるコンポーネントがターゲット側方向のコンポーネントから最初に END_REQ を受信することなく BEGIN_RESP を受信した場合、イニシエータは BEGIN_RESP の直前に暗黙の END_REQ を仮定するものとする。これは同じトランザクションの場合のみに当てはまる。BEGIN_RESP はそれ以外のトランザクションに対しては暗黙の END_REQ の受信を意味しない。また BEGIN_REQ を受信するターゲットは、前のトランザクションに対し暗黙の END_RESP を受信したと見なすことはできない。

m) 上記は、nb_transport のタイミング・アノテーション引数の値にかかわらず成り立つ。

n) フォワード・パスかバックワード・パスのいずれかのパスで TLM_COMPLETED を返す時、あるいはフォワード・パスかリターン・パスで END_RESP を渡す場合、基本プロトコルのトランザクションは、（その特定のホップに関して）完了する。

o) END_RESP がフォワード・パスで送信される場合、呼び出し先関数は TLM_ACCEPTED か TLM_COMPLETED を返すことができる。トランザクションはどちらの場合でも完了する。

p) それぞれのトランザクションは異なるホップ上の異なる時間に完了することができる。nb_transport に渡されたトランザクション・オブジェクトはメモリ・マネージャを所持しなければならず、汎用ペイロードの被参照数がゼロに達したとき、トランザクション・オブジェクトの寿命は終了する。汎用ペイロードのトランザクションの acquire メソッドを呼び出すすべてのコンポーネント・オブジェクトはトランザクションの終了時、または、終了前に release メソッドを呼び出さなければならない。「7.5 汎用ペイロードのメモリ管理」を参照。

q) コンポーネントが不正あるいは順不同なフェーズ遷移を受信した場合、これは送信側の誤りである。 受信側の振舞いは未定義である。 これは、実行時にエラーが発生するかもしれないことを意味している。

11.4.3 許可されたフェーズの遷移

前項のルールの全ての説明をまとめた、基本プロトコルでのそれぞれのホップ上の許容されるフェーズ遷移の集合は、以下の表のようになる。

前の状態	呼び出しパス	呼び出し時のフェーズ引数	戻り時のフェーズ引数	戻り時の状態	応答正当確認	生存期間完了	次の状態
//rsp	Forward	BEGIN_REQ	-	Accepted			req
//rsp	Forward	BEGIN_REQ	END_REQ	Updated			//req
//rsp	Forward	BEGIN_REQ	BEGIN_RESP	Updated	✓		rsp
//rsp	Forward	BEGIN_REQ	-	Completed	✓	✓	//rsp
req	Backward	END_REQ	-	Accepted			//req
req	Backward	BEGIN_RESP	-	Accepted	✓		rsp
req	Backward	BEGIN_RESP	END_RESP	Updated	✓	✓	//rsp
req	Backward	BEGIN_RESP	-	Completed	✓	✓	//rsp
//req	Backward	BEGIN_RESP	-	Accepted	✓		rsp
//req	Backward	BEGIN_RESP	END_RESP	Updated	✓	✓	//rsp
//req	Backward	BEGIN_RESP	-	Completed	✓	✓	//rsp
rsp	Forward	END_RESP	-	Accepted	✓	✓	//rsp
rsp	Forward	END_RESP	-	Completed	✓	✓	//rsp

a) req, //req, rsp, //rsp は、それぞれ、 BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP を示す。

b) これらのフェーズ状態変化は、nb_transport（タイミング・アノテーション）の sc_time 引数の値から独立している。言い換えると、nb_transport の呼び出しはタイミング・アノテーショ

ンの値にかかわらず表に示した状態変化を引き起こす。(タイミング・アノテーションには、トランザクションのその後の実行を遅らせるという効果があるかもしれない。)

c) 「前の状態」列は、nb_transport を呼び出す前のそれぞれのホップの状態を示す。

d) 「呼び出しパス」列は、対応するメソッドがフォワード・パス（nb_transport_fw）かバックワード・パス（nb_transport_bw）のどちらで呼び出されるかを示す。

e) 「呼び出し時のフェーズ引数」列は、nb_transport を呼び出すフェーズ引数の値を与える。これは呼び出し先関数に渡されるフェーズとなる。

f) 「戻り時のフェーズ引数」列は nb_transport からの戻り時のフェーズ引数の値を与える。フェーズ引数はメソッドが TLM_UPDATED を返す場合だけ有効である。

g) 「戻り時の状態」列は nb_transport メソッドの戻り値を与える。それらの値は、Accepted（TLM_ACCEPTED）、Updated（TLM_UPDATED）、または Completed（TLM_COMPLETED）である。

h) 「応答正当確認」列は、トランザクションの nb_transport メソッドからの戻り時の応答ステータス・アトリビュートが正当であるかどうかの確認の要否を示す。

i) 「生存期間完了」列は、トランザクションがそのホップに関して生存期間の終わりに達したかどうかを示す。すなわち、あるホップ上のあるトランザクションに対し、nb_transport 呼び出しがそれ以上許されないかどうかを示す。

j) 「次の状態」列は nb_transport からの戻り後のそれぞれのホップの状態を示す。

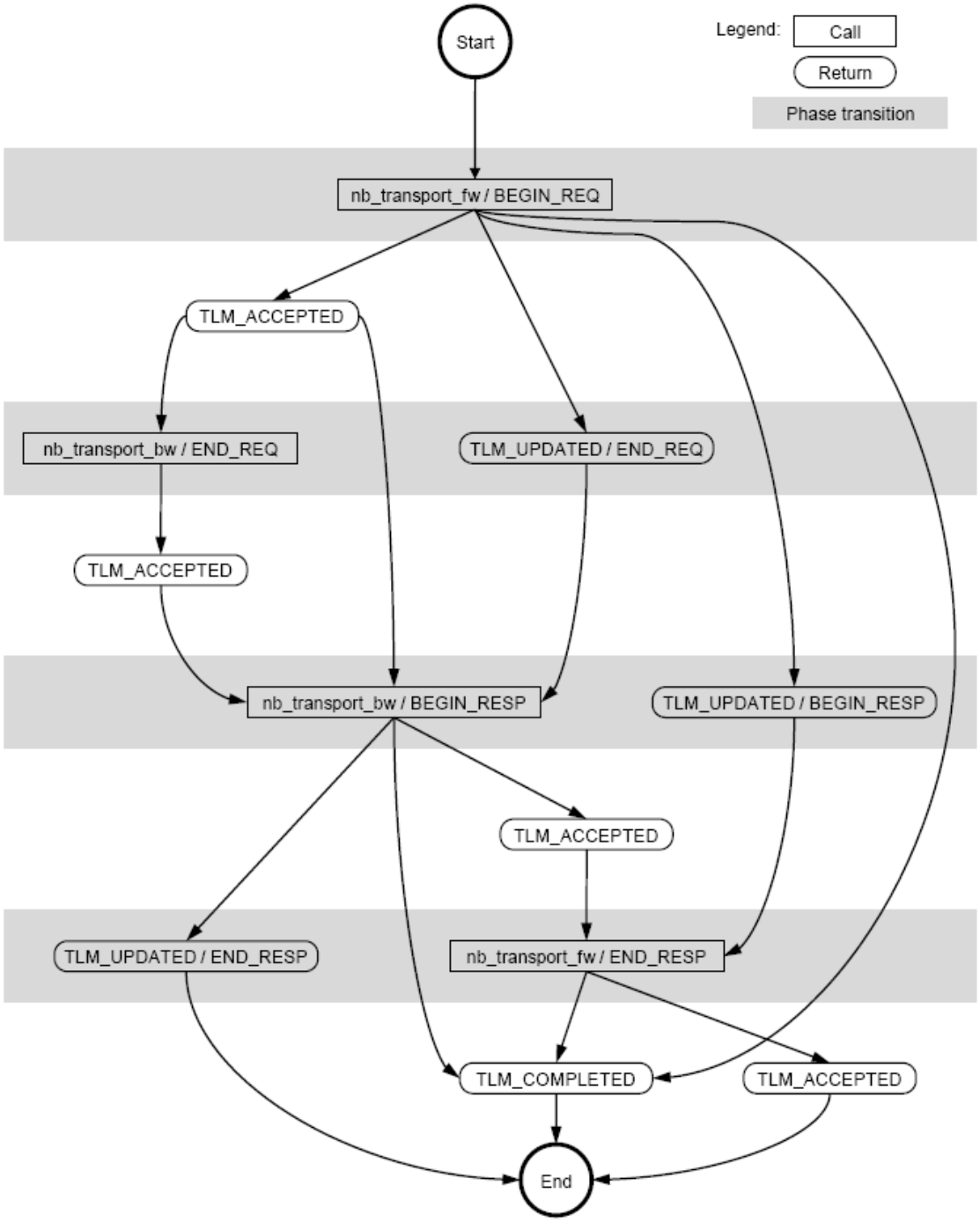
k) フェーズ遷移は、呼び出し元関数（「戻り時のフェーズ引数」列の「-」で示される）または、呼び出し先関数のいずれかによって引き起こされる。

l) 無視可能なフェーズ拡張は BEGIN_REQ と END_RESP の間の任意な点で挿入することができる。

m) 有効なレスポンスは、完了の成功を示さない。トランザクションは成功したかもしれないし、成功していないかもしれない。

n) 次のページのグラフ形式の図 14 は、それぞれのホップ上での基本プロトコルで許容される nb_transport の呼び出しシーケンスを示す。グラフの Start から End までの経路は単一のトランザクションに対する許容される呼び出しシーケンスを表す。長方形はフェーズ引数の値とともに nb_transport の呼び出しを表す。角丸長方形は戻り値と戻り時の適切なフェーズ引数の値を示す。大きな灰色の長方形の帯はフェーズ遷移を示す。

nb_transport Call Sequence for each Base Protocol Transaction Figure 14



11.4.4 無視可能なフェーズ

拡張フェーズはそれらが無視可能なフェーズであれば基本プロトコルとともに使用することができる。無視可能なフェーズは受信側に無視される可能性がある。

a) 一般的に、基本プロトコルの 4 つのフェーズに拡張フェーズを加える推奨の方法は、新しいプロトコル・トレイツ・クラスを定義することである。「7.2.2 tlm_generic_payload に対する typedef を含む新しいプロトコル・トレイツ・クラスを定義」を参照。無視可能なフェーズは、拡張フェーズの特別で制限された場合である。無視可能なフェーズの主な目的は、モデルのタイミング精度を増加させるために特別なタイミング・ポイントを基本プロトコルに追加することを許容することである。たとえば、無視可能なフェーズがイニシエータからターゲットへのデータ転送の開始時刻の目印を付けてもよい。

b) nb_transport の呼び出しに関する場合、もし呼び出し先関数がフェーズを無視するのであるならば、TLM_ACCEPTED の値を返すものとする。呼び出し先関数が TLM_UPDATED を返す場合、呼び出し元関数は、リターン・パスで渡されるフェーズを無視してもよいが、フェーズが無視されているのを示すための特定のアクションを取る必要はない。

c) nb_transport インタフェースは、nb_transport の呼び出し元関数に対し、呼び出し先関数がフェーズを無視している場合と、呼び出し先関数が後で反対のパスを経由して応答する場合とを区別するためのいかなる方法も提供しない。呼び出し先関数はどちらの場合でも TLM_ACCEPTED を返すものとする。

d) 無視可能なフェーズの存在は、基本プロトコルの BEGIN_REQ、END_REQ、BEGIN_RESP、および END_RESP の４つのフェーズの順序やセマンティックスを変えないものとする。また、無視可能なフェーズにより基本プロトコルのいかなるルールが破られることも許されない。

e) 無視可能なフェーズは、特定のソケット上の特定のトランザクションに対し、BEGIN_REQ の前または END_RESP の後には起こらないものとする。BEGIN_REQ の前または END_RESP の後の無視可能なフェーズの存在は、基本プロトコル違反であり誤りである。

f) 無視可能なフェーズの存在は、汎用ペイロード・アトリビュートの正当性に関するルール、あるいはそれらのアトリビュートを変更するためのルールを変えないものとする。例えば、無視可能なフェーズを受け取った後、インターコネクト・コンポーネントはアドレス・アトリビュートの変更のみを、DMI はアトリビュートおよび拡張の変更のみを許可される。「7.7 アトリビュートのデフォルト値と変更可否」を参照。

g) 後ほど定義されるトランスペアレントなコンポーネントにおける例外はあるが、無視可能なフェーズの受取側がそのフェーズを認識しないなら（すなわち、フェーズは無視されている）、受取側はフォワード、バックワードまたはリターン・パスでそのフェーズを伝播しないものとする。言い換えれば、そのフェーズのセマンティックスについて完全に理解する場合だけ、コンポーネントは引数のフェーズを nb_transport 呼び出しに渡すことが許される。

h) 無視可能なフェーズの受取側が、そのフェーズを認識し、また、基本プロトコルに違反していないなら、そのコンポーネントの動作は、基本プロトコルのスコープ外であり、基本プロトコルでは未定義となる。受取側はフェーズが属する拡張プロトコルのセマンティックスに

従わなければならない。

i) 定義により、無視可能なフェーズを送信するコンポーネントは、TLM_ACCEPTED の値を返す nb_transport の最小限の応答以外に、そのフェーズが送信されるコンポーネントに対しどのような種類の応答も要求できない。定義により、応答を要求するフェーズは無視可能ではない。そのような場合の推奨の方法は、基本プロトコルに拡張を使用するよりむしろ新しいプロトコル・トレイツ・クラスを定義することである。これは互換性のないプロトコルのソケットのバインドを防ぐ。

j) 他方では、到着する拡張フェーズを認識する基本プロトコルに準拠したコンポーネントは、あらかじめ合意された何らかの拡張プロトコルのルールに従って、逆のパスを経由してさらに別の拡張フェーズを送信することによって応答することができる。この可能性は、基本プロトコルのルールが破られていない限り、TLM-2.0 規格によって許容される。例えば、そのような拡張プロトコルは汎用ペイロード拡張を利用することができる。

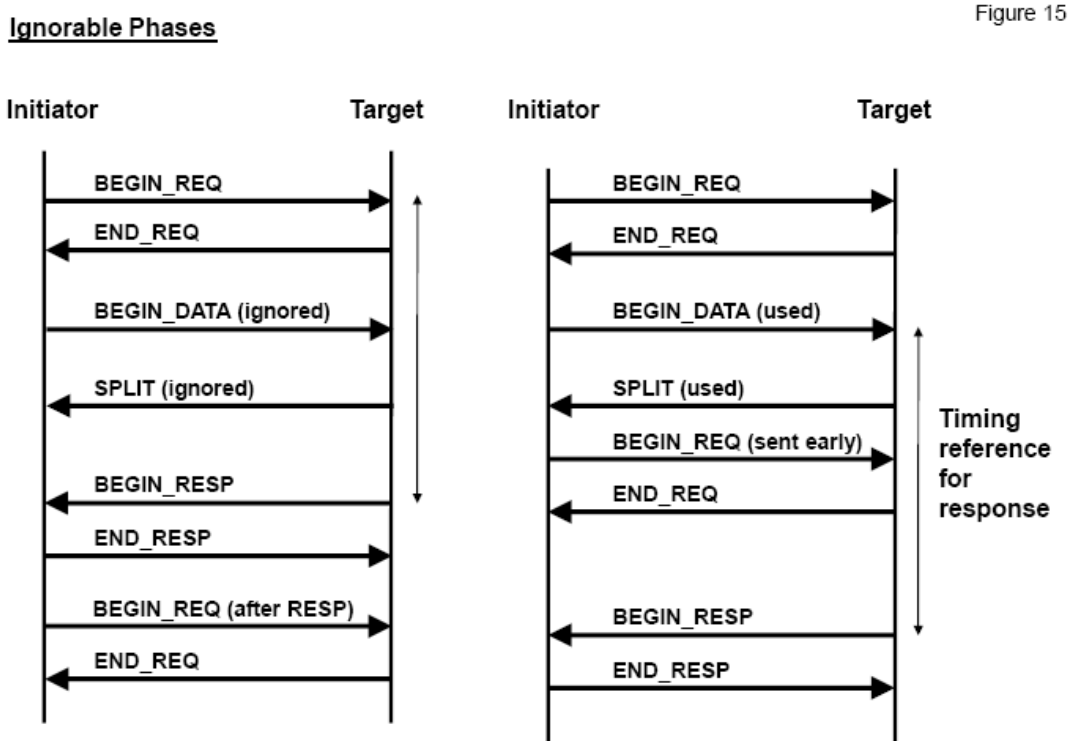
k) いわゆるトランスペアレント･インターコネクト・コンポーネント（同じコンポーネント中に含まれるターゲット・ソケットとイニシエータ・ソケット間のどのような TLM-2.0 のインタフェース・メソッドも、即座に直接的に通過させる）を作成することは可能である。この規格でトランスペアレントなコンポーネントを認める唯一の目的は、チェッカとモニタを考慮するためである。それらは通常 1 個のターゲット・ソケット、1 個のイニシエータ・ソケットがあり、すべてのトランザクションを変更することなく両方の方向に通過させる機能を持つ。

l) トランスペアレントなコンポーネント中では、すべての TLM-2.0 コア・インタフェース・メソッドの実装は、少しでもシミュレーション時間を消費したり、遅延を挿入したり、または wait の呼び出しをしてはならない。そして、ただちに、すべての引数をそのまま引き渡しながら、反対側のソケット（イニシエータ・ソケットならターゲット・ソケットを、ターゲット・ソケットならイニシエータ・ソケット）を経由して、同一のインタフェース・メソッド・コールを行なわなければならない。そのようなインタフェース・メソッドは、汎用ペイロード拡張を 1 つの例外として、それ以外のどのような引数（トランザクション･オブジェクト、フェーズ、および遅延）の値も変更しないものとする。そのようなトランスペアレントなコンポーネントを経由するルーティングは固定されており、トランザクション・アトリビュートやフェーズに依存しないものとする。

m) 上記のルールの帰結として、トランスペアレントなコンポーネントはすべての拡張フェーズや無視可能なフェーズをどちらの方向へも通過させる。

例

Figure 15



イニシエータから生成された無視可能なフェーズの例として、イニシエータからの書き込みコマンドに関する場合に、データ転送の最初のビートをマークするフェーズを挙げることが出来る。このフェーズを認識したインターコネクト・コンポーネント、またはターゲットは、コマンドとアドレスが利用可能になる時と、データ転送の始まりとを見分ける。このフェーズを無視したターゲットは、コマンド、アドレス、およびデータの取得のためのただ一つのタイミングリファレンスとして BEGIN_REQ フェーズを使用しなければならない。

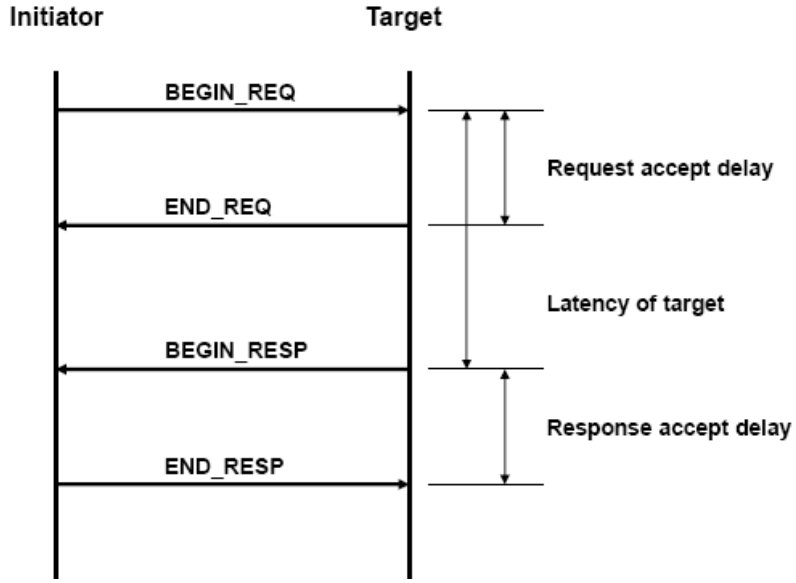
ターゲットで生成された無視可能なフェーズの例として、スプリット・トランザクションをマークするフェーズを挙げることが出来る。このフェーズを認識したイニシエータは、ターゲットがソプリット・トランザクションを処理する準備ができているのを理解し、スプリット・フェーズを受け次第、すぐに次のリクエストを送信することができる。スプリット・フェーズを無視したイニシエータは、2 番目のリクエストを送信する前に最初のリクエストに対するレスポンスを受信するまで待つかもしれない。

11.4.5 基本プロトコルのタイミングパラメーターとフロー制御

a) 4 つのフェーズにより、リクエスト受け付け遅延 （または連続トランザクションの送信時の最小イニシエーション・インターバル） 、 ターゲットのレイテンシ、 レスポンス受け付け遅延をモデル化可能である。この種の粒度のタイミング精度は AT コーディング・スタイルに適している。

Approximately-timed timing parameters

Figure 16



b) ライト・コマンドに対し、BEGIN_REQ フェーズは、インターコネクト・コンポーネントを経由してイニシエータからターゲットへの転送のためのデータが利用可能になる時間をマークする。概念的には、BEGIN_REQ フェーズへの遷移はデータ転送の最初のビートの始まりに対応する。ダウンストリーム・コンポーネントの責任は、転送時間を計算し、次の転送を受信する準備ができているとき END_REQ をアップストリームに返送することである。ダウンストリーム・コンポーネントは、データ転送の最終的なビートの終わりまで END_REQ を遅らせるのが自然であるが、そうする義務はない。

c) リード・コマンドに対し、BEGIN_RESP フェーズは、インターコネクト・コンポーネントを経由してターゲットからイニシエータへの転送のためのデータが利用可能になる時間をマークする。概念的には、BEGIN_RESP フェーズへの遷移はデータ転送の最初のビートの始まりに対応する。アップストリーム・コンポーネントの責任は、転送時間を計算し、次の転送を受信する準備ができているとき END_RESP をダウンストリームに返送することである。アップストリーム・コンポーネントがデータ転送の最終的なビートの終わりまで END_RESP を遅らせるのが自然であるが、そうする義務はない。

d) リード・コマンドの場合、もし、ダウンストリーム・コンポーネントが nb_transport_fw から TLM_COMPLETED を戻して早期にトランザクションを終了するなら、データ転送時間をなんらかの方法で計算することは、アップストリーム・コンポーネントの責任である。（END_RESP を送信することは許可されていないため）。

e) 基本プロトコルにおいて、ダウンストリーム・コンポーネントから直前のトランザクションについての END_REQ または BEGIN_RESP を受信するか、またはダウンストリーム・コンポーネントが nb_transport_fw から TLM_COMPLETED を返すことによって直前のトランザクションを完了するまで、イニシエータあるいはインターコネクト・コンポーネントは各ソケットを経由して BEGIN_REQ フェーズである新しいトランザクションを送信しないものとす

る。これはリクエスト除外ルールとして知られている。

f) 基本プロトコルにおいて、アップストリーム・コンポーネントから直前のトランザクションについての END_RESP を受信するか、またはアップストリーム・コンポーネントが TLM_COMPLETED でそのホップ上の直前のトランザクションを完了するまで、ターゲットあるいはインターコネクト・コンポーネントは各ソケットを経由して BEGIN_RESP フェーズである新しいトランザクションに応答しないものとする。これはレスポンス除外ルールとして知られている。

g) リクエストおよびレスポンス除外ルールを含むフェーズ遷移を管理するすべてのルールは、メソッド呼び出しの順番のみに基づくこととし、時間引数（タイミング・アノテーション）の値で影響を受けてはならないものとする。

h) それぞれのソケットを経由しノンブロッキング・トランスポート・インタフェースを使用して送信される連続したトランザクションは、パイプライン化することができる。END_REQ（または END_RESP）により、対になる各 BEGIN_REQ（または BEGIN_RESP）に応答することによって、インターコネクト・コンポーネントは、同時に、複数のトランザクション・オブジェクトが転送中であるのを可能にすることができる。すぐに END_REQ（または END_RESP）に応答しないことによって、インターコネクト・コンポーネントはイニシエータ（またはターゲット）から来るトランザクション・オブジェクトの流れに対しフロー制御を行なうことができる。

i) それぞれのソケットを経由する未処理の 2 つのリクエストまたはレスポンスが存在する可能性を除外するこのルールは、ノンブロッキング・トランスポート・インタフェースに当てはまるだけであり、b_transport の呼び出しには、直接の影響は全くないだろう。（b_transport 自身が nb_transport_fw を呼び出す場合、このルールは b_transport の呼び出しに間接的影響があるかもしれない。）

j) それぞれのトランザクションにおいて、BEGIN_REQ は常にイニシエータから開始され、ゼロ個あるいはそれ以上のインターコネクト・コンポーネントを経由して伝播され、最終的にターゲットがそれを受け取るものとする。それぞれのトランザクションにおいて、アップストリーム・コンポーネントから BEGIN_REQ を受信する前にインターコネクト・コンポーネントが BEGIN_REQ をダウンストリーム・コンポーネントに送信することは許されない。

k) それぞれのトランザクションにおいて、BEGIN_RESP は常にターゲットから開始され、ゼロ個あるいはそれ以上のインターコネクト・コンポーネントを経由して伝播され、最終的にイニシエータがそれを受け取るものとする。それぞれのトランザクションにおいて、ダウンストリーム・コンポーネントから BEGIN_RESP を受信する前にインターコネクト・コンポーネントが BEGIN_RESP をアップストリーム・コンポーネントに送信することは許可されない。これは、BEGIN_RESP が明示的であるか TLM_COMPLETED により暗黙的に送信されるかに関係なく適用される。

l) それぞれのトランザクションにおいて、インターコネクト・コンポーネントは、ダウンストリーム・コンポーネントから END_REQ を受信する前に、END_REQ をアップストリーム・コンポーネントに送信することができる。同様に、インターコネクト・コンポーネントはアップストリーム・コンポーネントから END_RESP を受信する前に、END_RESP をダウンストリーム・コンポーネントに送信することができる。END_REQ と END_RESP が明示的であるか、暗黙であるかにかかわらずこれは適用される。

m) END_REQ と END_RESP は主として隣接しているコンポーネント間のフロー制御のためにある。これらの二つのフェーズは標準の汎用ペイロード・アトリビュートの正当性を示すものではない。これらの二つのフェーズはエンド・ツー・エンドでの因果関係としては伝播されないため、それらをイニシエータからターゲット間あるいはターゲットからイニシエータ間の拡張の正当性を示すことを信頼できるものとして使用できないが、2 つの隣接しているコンポーネント間の拡張の正当性を示すことには使用できる。

n) 対応するフェーズを受信する前にインターコネクト・コンポーネントが拡張フェーズを送信することが許可されているかどうかは、該当の拡張フェーズに付随するルールによる。

Figure 17

Causality with b_transport

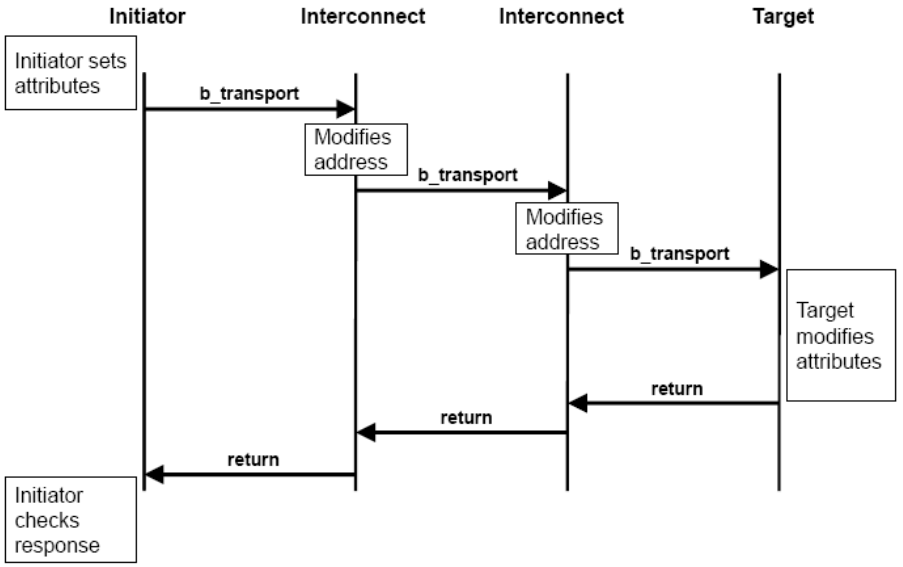
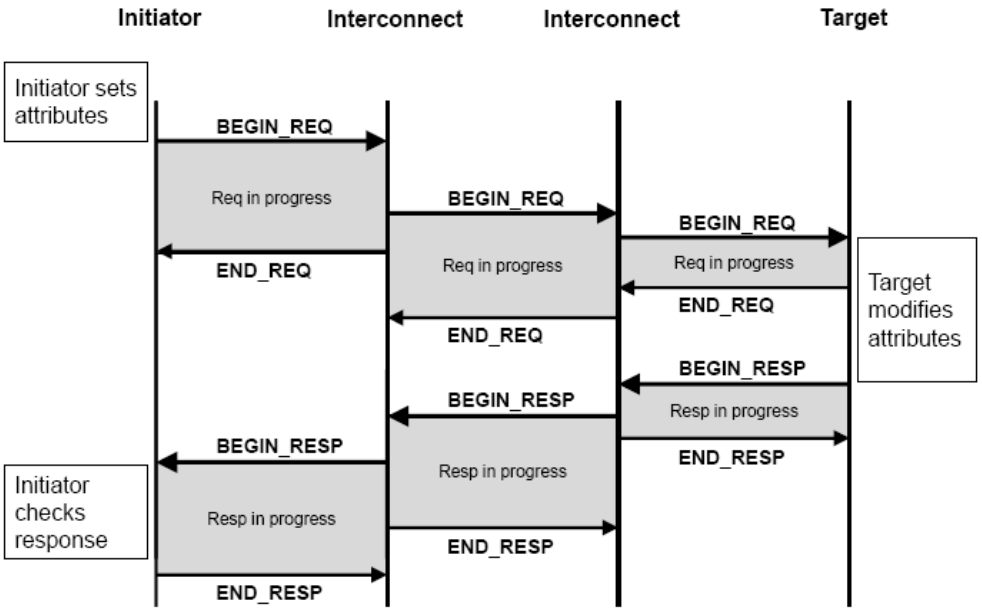


Figure 18

Causality with nb_transport



例

以下の擬似コードはタイミング・アノテーションと、リクエストおよびレスポンス除外ルールの間の相互作用を例示する:

```
void initiator_1_thread_process()
{

   // The initiator sends a request to be executed at +1000ns
   phase = BEGIN_REQ;
   delay = sc_time(1000, SC_NS);
   status = socket->nb_transport_fw (T1, phase, delay);
   assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(1010, SC_NS) );
   // END_REQ is returned immediately to be executed at +1010ns

   // Note that this is not a recommended coding style
   // With loosely-timed, the initiator would have called b_transport
   // With approximately-timed, the downstream component would have returned TLM_ACCEPTED
   // in order to synchronize, and the initiator would have been forced to wait for END_REQ

   // The initiator is allowed to send the next request immediately, to be executed at +1050ns
   phase = BEGIN_REQ;
   delay = sc_time(1050, SC_NS);
   status = socket->nb_transport_fw (T2, phase, delay);
   assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(1060, SC_NS) );

   // The initiator is technically allowed to send the next request at an earlier local time of +500ns,
   // although the decreased timing annotation is not a recommended coding style
   phase = BEGIN_REQ;
   delay = sc_time(500, SC_NS);
   status = socket->nb_transport_fw (T3, phase, delay);
   assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(510, SC_NS) );

   // The initiator now yields control, allowing other initiators to resume and simulation time to advance
   wait(…);
}

void initiator_2_thread_process()
{
   // Assume the calls below are appended to the transaction stream sent from the first initiator above

   // The second initiator sends a request to be executed at +10ns
```

```
  // The timing annotation as seen downstream has decreased from +510ns to +10ns
  // This is typical behavior for loosely-timed initiators
  phase = BEGIN_REQ; delay = sc_time(10, SC_NS);
  status = socket->nb_transport_fw (T4, phase, delay);
  assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(30, SC_NS) );
  // END_REQ is returned immediately to be executed at +30ns

  // The initiator sends the next request to be executed at +20ns, which overlaps with the previous request
  // This is technically allowed because the current phase of the hop is END_REQ, but is not recommended
  phase = BEGIN_REQ; delay = sc_time(20, SC_NS);
  status = socket->nb_transport_fw (T5, phase, delay);
  assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(40, SC_NS) );
  // END_REQ is returned immediately to be executed at +40ns

  // The initiator sends the next request to be executed at +0ns, which is before the previous two requests
  // This is technically allowed because the current phase of the hop is END_REQ, but is not recommended
  phase = BEGIN_REQ;
  delay = sc_time(0, SC_NS);
  status = socket->nb_transport_fw (T6, phase, delay);
  assert( status == TLM_UPDATED && phase == END_REQ && delay == sc_time(60, SC_NS) );
  // END_REQ is returned immediately to be executed at +60ns, overlapping the two previous requests
  // This is technically allowed, but is not recommended
  wait(…);
}
```

11.4.6 タイミング・アノテーション関連の基本プロトコル・ルール

a) この項のルールは 4.1.3 トランスポート・インタフェースにおけるタイミング・アノテーションと合わせて読む必要がある。

b) b_transport と nb_transport の実装において、時間引数 t を変更するために有効なローカル時間 sc_time_stamp() + t が関数呼び出しとその戻りの間で非減少でなければならないという制約がある。「4.1.3.1 sc_time 引数」を参照。

c) 個別のトランザクションの個別のソケットを経由する nb_transport の連続した呼び出しと戻りに対し、有効なローカル時間のシーケンスは非減少でなければならない。有効なローカル時間は sc_time_stamp() + t という式で与えられる。ここでは、t が nb_transport への時間引数である。このために、関数の呼び出しと関数からの戻りの両方がただ一つのシーケンスの一部であるとみなされるものとする。これは、フォワードおよびバックワードの両方のパスで同じように適用される。この意味は、それぞれのトランザクションに対し時間が逆行してはならないということである。

d) 上記のルールはまた b_transport の呼び出しと戻りの間にも適用される。「4.1.3.1 sc_time 引数」を参照。

e) さらに、個別のトランザクション・オブジェクトに関して、リクエストがイニシエータからターゲットに向かって伝播されて、レスポンスがターゲットからイニシエータに向かって伝播されるのに従って、各連続したトランザクション・メソッドの呼び出しと戻りにおいてそれぞれの実効的なローカル時間のシーケンスも非減少となる。この意味で、リクエストの伝播は b_transport と BEGIN_REQ フェーズの呼び出しの両方を含んでいる。レスポンスの伝播は b_transport からの戻り、すなわち BEGIN_RESP フェーズ、および TLM_COMPLETED を含んでいる。

f) 実効的なローカル時間は、タイミングアノテーション（時間引数）の値を増加させるか、あるいは SystemC のシミュレーション時間（b_transport のみ）を進めるか、またはその両方によって進めることができる。

g) 異なるトランザクション・オブジェクトについては、b_transport と nb_transport の呼び出しでの実効的なローカル時間は非減少になるべきという義務は全くない。そうではあるが、各イニシエータ・プロセスは一般に、非減少となる実効的なローカル時間で b_transport や nb_transport を呼び出すことが推奨される。そうでなければ、ダウンストリーム・コンポーネントは、順不同なトランザクションが別々のイニシエータから発せられたと推論して、それらの特定のトランザクションが実行される順序を選ぶことが、自由になる。しかしながら、異なった LT イニシエータからのトランザクションの流れがどこかで合流するなら、順不同である実効的なローカル時間を持つトランザクションが起こりうることである。

h) 個別のソケットについて、イニシエータは、ブロッキング、ノンブロッキング・トランスポート・インタフェース、ダイレクト・メモリ・インタフェース、およびトランスポート・デバッグ・インタフェースを経由して、異なる時間に同じトランザクション・オブジェクトを渡すことができる。また、イニシエータは、汎用ペイロードのメモリ管理ルールに従うすべての異なるトランザクション・インスタンスに対し、同じトランザクション・オブジェクトを再利用することが許される。「7.5 汎用ペイロードのメモリ管理」を参照。

11.4.7 b_transport関連の基本プロトコル・ルール

a) b_transport 呼び出しはリエントラントである。b_transport の実装は wait を呼び出すことができ、その間、異なるイニシエータは、タイミング・アノテーションの制約なく異なるトランザクション・オブジェクトを用い、別の b_transport を呼び出すことができる。

b) 同じイニシエータ・モジュールの中に複数のプロセスがある場合、各プロセスはトランザクション順序ルールを遵守し、別々のイニシエータであると見なされるものとする。特に、同じモジュールの別々のスレッドから実行される b_transport 呼び出しの順序に制約はない。またそれらの呼び出しが同じイニシエータ・ソケットからか異なるソケットからかどうかは関係ない。

c) インターコネクトまたはターゲットは、常に、異なるイニシエータ・スレッドからの複数の並列の b_transport 呼び出しが発生し、異なるプロセスから任意の並列の nb_transport_fw に対する呼び出しが発生していると考えることが出来る。従って、それらの呼び出しの互いの順序に制約はないと推論できる。b_transport により、1 つのリクエストが別のリクエストを追いこすことが許容される。

d) 同じソケットを経由し同じトランザクション・オブジェクトに対し、b_transport をリエントラントに呼び出すことは禁じられる。

例

```
The following pseudo-code fragments show a re-entrant b_transport call:
// Two initiator thread processes
void thread1()
{
   socket->b_transport( T1, sc_time(100, SC_NS) );
}

void thread2() {
   wait( 10, SC_NS );
   socket->b_transport( T2, sc_time(50, SC_NS) );                    // T2 overtakes T1
}
// Implementation of b_transport in the target
void b_transport( TRANS& trans, sc_time& t )
{
   wait( t );
   execute( trans );                                                 // T1 executed at 100ns, T2 executed at 60ns
   t = SC_ZERO_TIME;
}
```

11.4.8 リクエストとレスポンスの順序に関する基本プロトコル・ルール

以下のルールの目的は、イニシエータが一連のパイプライン化されたリクエストを特定のターゲットに送信する場合に、送信された順番どおりに、ターゲットでそれらのリクエストを実行することを保証することである。汎用ペイロード・トランザクションはイニシエータの同一性識別もターゲットの同一性識別も格納しないので、トランザクションの到着するソケットの同一性からイニシエータの同一性を推定できるだけである。また、ターゲットもアドレスとコマンド・アトリビュートの値から推定できるだけである。重複していないアドレスに送信されたリクエストの実行順序は保証されない。

a) 基本プロトコルは、異なったソケットを経由して到着するリクエスト、または、レスポンスが順不同に互いに遅延、インターリーブ、または実行されることを許可する。例えば、インターコネクト・コンポーネントは、特定のターゲット・ソケットを経由して到着するリクエストや特定のデータの長さを持っているリクエストに対しにより高い優先度を割り当て、、低い優先度のリクエストを追い越すさせることができる。別の例として、インターコネクト・コンポーネントは、対応するリクエストが元々受信された順序に合わせて、異なるイニシエータ・ソケットを経由して返って来るレスポンスを並べ替えることができる。

b) リクエストの経路決定は、決定論的であり、トランザクション・オブジェクトのアドレスとコマンド･アトリビュートだけに依存するものとする。(これらだけがトランスポート、DMI、およびデバッグ・トランスポート・インタフェースへの共通のアトリビュートである。) 進行中のトランザクションがある間、アドレスマップは変更しないものとする。

c) イニシエータまたはインターコネクト・コンポーネントが重複するアドレスに複数の並行リクエストをフォワード・パスに送信するなら、それらのリクエストは同じイニシエータ・ソ

ケットを経由して送信されるものとする。複数の並行リクエストとは、対応するレスポンスがターゲットからまだ受信されていないリクエストを意味する。重複するアドレスとは、トランザクション・オブジェクトのデータ配列における少なくとも 1 バイトが同じアドレスを共有することを意味する。同じアドレスへのリードとライトのリクエストは、それらが並行発生でないなら、異なったソケットを経由して送信することができる。

d) 同じターゲット・ソケットを経由して nb_transport_fw への到着した呼び出しによる重複アドレスがある状態で、インターコネクト・コンポーネント（またはターゲット）が複数の並行リクエストを受信したなら、それらを受け取ったのと同じ順番でそれらのリクエストを前方に送信する（または、それぞれ実行）するものとする。同じ順番とは、同じインタフェース・メソッドを呼び出す順序を意味する。（もし、インタフェース・メソッド・コール順序と有効なローカル時間の順序のセットのトランザクションが異なることがあるなら、それらのトランザクションを受信するどんなコンポーネントも、順不同にそれらを実行することが許可されることに注意すること。また、問題のリクエストが異なったイニシエータから発せられた場合もこのルールが成立することに注意すること。）

e) 上のルールは、到着する b_transport 呼び出しには当てはまらない。そのため、複数の並行リクエストの順序に制約はない。他方で、受信した nb_transport_fw 呼び出しを送信 b_transport 呼び出しに変換する場合、b_transport 呼び出しは、nb_transport 呼び出しの順序についてのルールに従うように直列化しなければならない。

f) 他方では、インターコネクト・コンポーネント、またはターゲットは、異なるターゲット・ソケットを経由して受信した場合、あるいは異なるイニシエータ・ソケットを経由して送信した場合、あるいはアドレスが重複しない場合、あるいは受信する b_transport 呼び出しの場合は、複数の並行リクエストを並べ替えることが許されている

g) レスポンスは並べ替えることができる。リクエストが送信された後、対応するレスポンスがイニシエータに同じ順番で返って来るという保証はない。

h) 基本プロトコルにおいて、インターコネクト・コンポーネントに対し複数の並行リクエストを並べ替えることを許すような無視可能な拡張を使用することは、技術的に可能であることに注意すること。その場合、拡張を付加したイニシエータは、ターゲットによる順不同な実行を許容できなければならない。他方では、リクエストを送信したのと同じ順序でのレスポンスを強制する拡張は無視可能な拡張でない。したがって、それは基本プロトコルでは、受入れられない。

11.4.9 b_transportとnb_transportの切り替えについての基本プロトコル・ルール

a) イニシエータまたはインターコネクト・コンポーネント中の各スレッドは、異なるトランザクション・オブジェクトに対し b_transport と nb_transport_fw との呼び出しを切り換えることが許される。その目的は、イニシエータが LT および AT コーディング・スタイル間をその時々において切り替えることを許すためである。b_transport と nb_transport_fw を交互に呼び出すイニシエータは、タイミング精度に関して期待してはならない。

b) 全てのインターコネクト・コンポーネントとターゲットはブロッキングおよびノン・ブロッキング・トランスポート・インタフェースの両方をサポートしなければならない。そして、両方のインタフェースからアクセスしやすいように内部状態情報を維持しなければならない。これは、同じソケット、または異なるソケットを経由して受信され到着するインタフェ

ース・メソッド・コールに適用される。

c) イニシエータまたはインターコネクト・コンポーネント内のスレッドは、同一のトランザクション・インスタンスに対し b_transport と nb_transport_fw の両方を呼び出してはならないものとする。スレッドは、その時々でトランザクション・インスタンスが異なる限りにおいて、同じトランザクション・オブジェクトに対し b_transport と nb_transport_fw の両方を呼び出すことができることに注意すること。

d) イニシエータまたはインターコネクト・コンポーネント中のスレッドは、その同じスレッドから先行する nb_transport_fw 呼び出しにより進行中のトランザクションがまだある場合、b_transport を呼び出してはならないことを推奨する。すなわち、被参照数が 0 でない先行するトランザクションがある場合である。さもなければ、ダウンストリーム・コンポーネントは、2 つのトランザクションが別々のイニシエータから到着したと誤って推論するかもしれない。

e) 便利ソケット simple_target_socket は、基本プロトコル・ターゲットが、b_transport と nb_transport_fw のどちらか一方だけを実装するだけで、どうやってブロッキングおよびノンブロッキング・トランスポートインタフェースの両方をサポートできるかを示す例にもなっている。「12.1.2 シンプル・ソケット」を参照。

11.4.10その他の基本プロトコル・ルール

a) 複数の並行なソケットを経由して、または、同時に複数の並行な経路に沿ってそれぞれのトランザクション・オブジェクトを送信してはならないものとする。それぞれのトランザクション・インスタンスは、トランスポート、ダイレクト・メモリ、およびデバッグ・トランスポートの各インタフェースに共通の、コンポーネントとソケットの組を経由し、各トランザクション・インスタンスの生存期間中固定化される、ユニークで明確な経路を取るものとする。もちろん、個別のソケットを経由して送信される異なるトランザクションは、異なる経路を取ることができる。すなわち、それらは異なる経路で届けられる。また、コンポーネントは、インターコネクト・コンポーネントとして、あるいは、ターゲットとして機能するかどうかを動的に選択してもよい点に注意すること。

b) アップストリーム・コンポーネントは、それがインターコネクト・コンポーネント、または、直接ターゲットに関連づけられるかどうかを知るべきでなく、知る必要もない。同様に、ダウンストリーム・コンポーネントは、それがインターコネクト・コンポーネント、または、直接イニシエータに関連づけられるかどうかを知るべきでなく、知る必要もない。

c) ライト・トランザクション（TLM_WRITE_COMMAND）において、応答ステータスの TLM_OK_RESPONSE は、ライト・コマンドがターゲットで正常に完了したことを示す。ターゲットは b_transport から戻る前か、あるいはバックワードまたはリターン・パスに沿って BEGIN_RESP を送信する前か、TLM_COMPLETED を返す前に応答ステータスを設定しなければならない。言い換えれば、インターコネクト・コンポーネントは、ターゲットから正常な完了を確認することなく、ライト・トランザクションの完了を伝達することは許されない。このルールの目的は、ターゲット・シミュレーション・モデルの中でデータの一貫性を保証することである。

d) リード・トランザクション（TLM_READ_COMMAND）において、応答ステータスの TLM_OK_RESPONSE は、リード・コマンドが完了し、汎用ペイロード・データ配列がター

ゲットによって変更されたことを示す。ターゲットは、b_transport から戻る前か、あるいはバックワードまたはリターン・パスに沿って BEGIN_RESP を送信する前か、TLM_COMPLETED を返す前に応答ステータスを設定しなければならない。

11.4.11 トランザクションの順序に関する基本プロトコル・ルールのまとめ

以下の表は基本プロトコルでの、トランザクションの順序に関するルールのまとめを示す。ルールの完全な解説については、上の項を参照すること。

基本プロトコルの順番ルールは 3 つのカテゴリのルールの和集合である。これらのルールとは、タイミング・アノテーションに関するコア・トランスポート・インタフェースのルール、因果関係とフェーズに関する基本プロトコルに特有のルール、そして、パイプライン化されたリクエストが、イニシエータにより期待される順序でターゲットにおいて実行されることを保証するという基本プロトコルに特有のルールである。

状況	順序ルール
インタフェース・メソッド・コール順序と異なる実効的なローカル時間の順序	受信側は、順不同なトランザクションを実行、または送信することができる。他のすべてのルールより優先する。
同一ソケットを経由する同一トランザクションについての連続したトランスポート・メソッド呼び出しと戻り	実効的なローカル時間は非減少の順序でなければならない。
同一イニシエータ・プロセスからの連続したトランスポート・メソッド呼び出し	実効的なローカル時間の順序は非減少であることを推奨する。
同一イニシエータ・プロセスからの連続したトランスポート・メソッド呼び出し	先行の nb_transport トランザクションがまだ生存しているなら b_transport を呼び出さないことを推奨する。
同一ソケットを経由する異なるトランザクションについての連続したトランスポート・メソッド呼び出し	実効的なローカル時間の順序の義務は無いが、もし到着するトランザクションの流れが非減少であるなら、非減少を推奨する。
nb_transport についてのみ。同一ソケットを経由する未処理の 2 つのリクエストまたは 2 つのレスポンス	禁止する
異なるソケットを経由し到着するトランザクション	実行、または送信する順序の義務は無い。
重複したアドレスを持つ複数の並行リクエスト	もし、転送するなら、同一ソケットを経由して送信しなければならない。
nb_transport による同一ソケットを経由して到着する重複したアドレスを持つ複数の並行リクエスト	それらを受信したのと同じ順番で実行、または転送しなければならない。
b_transport により到着する複数の並行リクエスト	実行、または転送する順序の義務は無い。
複数の並行レスポンス	実行、または送り戻する順序の義務は無い。

11.4.12基本プロトコルに準拠したコンポーネント作成のガイドライン

この項は基本プロトコルコンポーネント作成のためのマニュアルを含む。これは本書のほかの場所により詳しく書かれたいくつかのルールの簡潔な再掲であり、利便性のために提供する。

11.4.12.1 基本プロトコル・イニシエータ作成のガイドライン

a) メモリ・マップド・バスとの各接続に対し、それぞれ1個のイニシエータ・ソケットをクラス tlm_initiator_socket（または派生クラス）からインスタンス化すること。

b) tlm_initiator_socket がテンプレート引数 TYPES にデフォルト値 tlm_base_protocol_types を取るようにすること。

c) メソッド nb_transport_bw と invalidate_direct_mem_ptr を実装すること。（イニシエータは便利ソケット simple_initiator_socket をインスタンス化することによって、これらのメソッドを明示的に実装する必要性を回避することができる。）

d) b_transport あるいは nb_transport_fw に汎用ペイロードを引数として渡す前に、汎用ペイロード・トランザクション・オブジェクトのそれぞれのアトリビュートを設定すること。特に呼び出しの前に忘れずに応答ステータスと DMI ヒント・アトリビュートをリセットすることを忘れてはならない。（バイト・イネーブル長アトリビュートは、バイト・イネーブル・ポインタ・アトリビュートが0である場合設定される必要はなく、拡張も使用する必要はない。）

e) 汎用ペイロードの拡張機構を使用するときには、必ずすべての拡張がターゲットとインターコネクト・コンポーネントで無視可能になるようにすること。

f) フェーズ・シーケンス、フロー制御、タイミング・アノテーション、およびトランザクション順序に関する基本プロトコルのルールに従うこと。

g) トランザクションの完了時（または BEGIN_RESP を受信した後）に、応答ステータス・アトリビュートの値を確認すること。

11.4.12.2 nb_transportを呼び出すイニシエータ作成のガイドライン

a) トランザクションを引数として nb_transport_fw に渡す前に、トランザクション・オブジェクトに対しメモリ・マネージャを設定し、トランザクションの acquire メソッドを呼び出すこと。トランザクションの完了の際には、release メソッドを呼び出すこと。

b) nb_transport_fw を呼び出すときには、トランザクションの状態に従って、BEGIN_REQ または END_RESP に対し、フェーズ引数を設定すること。先行するトランザクションから END_REQ を受信する（または、推論する）前に、BEGIN_REQ を送信しないこと。

c) それぞれのトランザクションに対し、一連の nb_transport_fw の呼び出しを行なうときには、有効なローカル時間（シミュレーション時間+タイミング・アノテーション）の値が必ず非減少であるシーケンスを形成するようにすること。

d) フェーズ値 END_REQ または BEGIN_RESP が到着した場合、それらをバックワード・パス（nb_transport_bw の呼び出し）、あるいはリターン・パス（nb_transport_fw から戻った TLM_UPDATED）、あるいは暗黙に（例えば、nb_transport_fw から戻った TLM_COMPLETED）受信したか否かに関わらず、適切に応答すること。BEGIN_REQ あるいは END_RESP であるフェーズ値の到着は不正である。他のすべてのフェーズ値の到着は無視可能であるとして扱

うこと。

11.4.12.3 基本プロトコル・ターゲット作成のガイドライン

a) メモリ・マップド・バスとの各接続に対し、それぞれ 1 個のターゲット・ソケットをクラス tlm_target_socket（または派生クラス）からインスタンス化すること。

b) tlm_target_socket がテンプレート引数 TYPES にデフォルト値 tlm_base_protocol_types を取るようにすること。

c) メソッド b_transport、nb_transport_fw、get_direct_mem_ptr および transport_dbg を実装すること。（ターゲットは便利ソケット simple_target_socket を使用することによって、これらのメソッドを明示的に実装する必要性を軽減することができる。）

d) b_transport と nb_transport_fw メソッドの実装においては、応答ステータス、DMI ヒント、およびすべての拡張といった例外はあるが、汎用ペイロードのあらゆるアトリビュートの値を確認し、それらの値に応じ動作すること。汎用ペイロードの完全な機能を実装するよりむしろ、ターゲットは、エラー応答を発生させることによってそれぞれのアトリビュートに応答することを選ぶほうがよい。トランザクションの成否を示すように応答ステータス・アトリビュートの値を設定すること。

e) フェーズ・シーケンス、フロー制御、タイミング・アノテーション、およびトランザクション順序に関する基本プロトコルのルールに従うこと。

f) get_direct_mem_ptr の実装においては、値 false を返すか、あるいはコマンドの値と汎用ペイロードのアドレス・アトリビュートを点検し動作すること。また、戻り値および DMI デスクリプタ（tlm_dmi クラス）のすべてのアトリビュートを適切に設定すること。

g) transport_dbg の実装においては、値 0 を返すか、汎用ペイロードの、コマンド、アドレス、データ長およびデータ・ポインタといった各アトリビュートの値を確認し、それに従い動作すること。

h) 各インタフェースに関しては、ターゲットは、汎用ペイロード内のすべての無視可能な拡張を確認し、それに従い動作することもできるが、それは強制ではない。

11.4.12.4 nb_transport を呼び出すターゲット作成のガイドライン

a) nb_transport_bw を呼び出すときには、トランザクションの状態に従って、END_REQ または BEGIN_RESP をフェーズ引数に設定すること。先行するトランザクションから END_RESP を受信する（または、推論する）前に、BEGIN_RESP を送信しないこと。

b) それぞれのトランザクションに対し、一連の nb_transport_bw の呼び出しを行なうときには、有効なローカル時間（シミュレーション時間+タイミング・アノテーション）の値が必ず非減少であるシーケンスを形成するようにすること。

c) フェーズ値 BEGIN_REQ または END_RESP が到着した場合、それらをフォワード・パス（nb_transport_fw の呼び出し）、あるいはリターン・パス（nb_transport_bw から戻った TLM_UPDATED）、あるいは暗黙に（例えば、nb_transport_bw から戻った TLM_COMPLETED）受信したか否かに関わらず、適切に応答すること。フェーズ値 END_REQ あるいは BEGIN_RESP の到着は不正である。他のすべてのフェーズ値の到着は無視可能であるとして扱うこと。

d) nb_transport_fw の実装においては、メソッドから戻った後もトランザクション・オブジェクトに対するポインタまたは参照を保持する必要がある場合には、トランザクションの acquire メソッドを呼び出すこと。トランザクション・オブジェクトが終了したら、release メソッドを呼び出すこと。

11.4.12.5 基本プロトコル・インターコネクト・コンポーネント作成のガイドライン

a) メモリ・マップド・バスとの各接続に対し、それぞれ 1 個のイニシエータあるいはターゲット・ソケットを、tlm_initiator_socket クラスあるいは tlm_target_socket クラス（または派生クラス）からインスタンス化すること。

b) 各ソケットがテンプレート引数 TYPES にデフォルト値 tlm_base_protocol_types を取るようにすること。

c) 各イニシエータ・ソケットに対し、nb_transport_bw と invalidate_direct_mem_ptr メソッドを実装し、各ターゲット・ソケットに b_transport、nb_transport_fw、get_direct_mem_ptr、および transport_dbg メソッドを実装すること。（便利ソケットを使用することによって、すべてのメソッドを明示的に実装する必要性を軽減することができる。）

d) フォワードおよびバックワード・パスの両方に到着するインタフェース・メソッド・コールを、リクエストとレスポンス除外ルール、およびトランザクション順序ルール、および TLM_COMPLETED 受信後はそれ以上の呼び出しは許されないといった各ルールを遵守しながら転送すること。無視可能なフェーズは転送しないこと。get_direct_mem_ptr と transport_dbg メソッドの実装は、トランザクション・オブジェクトを転送することなく、それぞれ値 false と 0 を返すことができる。

e) トランスポート・インタフェースの実装においては、インターコネクト・コンポーネントで変更できる汎用ペイロード・アトリビュートは、アドレス、DMI ヒント、および拡張だけである。他のアトリビュートは変更してはならない。その他のアトリビュートの変更を必要とするコンポーネントは、新しいトランザクション・オブジェクトを組み立てなければならず、その結果、そのコンポーネント自身がイニシエータになる。

f) フォワード・パスの汎用ペイロード・アドレス・アトリビュートを解読すること。そして、システム・メモリ・マップ上でのターゲットの配置に従って、必要なら、アドレス・アトリビュートを変更すること。これは、トランスポート、およびダイレクト・メモリ、そしてデバッグ・トランスポート・インタフェースに適用される。

g) トランスポート・インタフェースの実装においては、フェーズ・シーケンス、フロー制御、タイミング・アノテーション、およびトランザクション順序に関する基本プロトコルのルールに従うこと。

h) get_direct_mem_ptr の実装においては、フォワード・パスの DMI デスクリプタのアトリビュートを変更してはならない。リターン・パスの DMI ポインタ、DMI 開始アドレス、終了アドレス、および DMI アクセス・アトリビュートを適切に変更すること。

i) invalidate_direct_mem_ptr の実装においては、バックワード・パスに従って呼び出しを転送する前に、アドレス・レンジ引数を変更すること。

j) nb_transport_fw の実装においては、関数から戻った後も、トランザクション・オブジェクトに対するポインタまたは参照を保持する必要がある場合には、トランザクションの acquire

メソッドを呼び出すこと。トランザクション・オブジェクトが終了したら、release メソッドを呼び出すこと。

k) 各インタフェースに関して、インターコネクト・コンポーネントは、汎用ペイロード中のどの無視可能な拡張も確認し、それに従い実行することができる。ただし、それは強制ではない。トランザクションが、さらに拡張される必要があるなら、すべての拡張が確実に他のコンポーネントで無視可能になるようにすること。拡張のための汎用ペイロード・メモリ管理ルールを尊重すること。

12. ユーティリティ

ユーティリティは、一貫性のあるコーディング・スタイルを保つために提供される一連のクラス定義である。ユーティリティは、インタオペラビリティ・レイヤには属していないため、相互利用性を保つために必須なものではない。

12.1 便利ソケット

12.1.1 イントロダクション

コンポーネント・モデルをより簡単に書き易くするため、追加機能を実装した便利ソケット・ファミリーがある。 便利ソケットは、tlm_initiator_socket クラスと tlm_target_socket クラスから派生したクラスである。便利ソケットは、TLM-2.0 のインタオペラビリティ・レイヤの一部ではないため、(tlm ではなく) ネームスペース tlm_utils の中にて定義されている。

12.1.1.1 標準ソケットと便利ソケット概要

便利ソケットは以下のテーブルにて要約される。

- コールバック登録

  ソケットを、対応するインタフェース関数を実装しているオブジェクトに対してバインドする(通常の)方法ではなく、インタフェース・メソッド・コールに対するコールバック関数を登録するメソッドを提供する。

- マルチポート

  ソケットクラスのテンプレート引数として、バインド可能な数と、バインドポリシーを与えることができる。これにより、単一イニシエータ・ソケットを複数のターゲット・ソケットにバインドすることや、その逆ができる。

- b-nb 変換

  ターゲット・ソケットに対して呼び出された b_transport 関数を nb_transport_fw 関数呼出しに変換したり、その逆を行うことができる。(表中の)– は、イニシエータ・ソケットであることを示す。

- タグ付け

  どのソケットに対してインタフェース・メソッド・コールが呼ばれたかを識別するためのタグ情報を持つ。

クラス名	コールバック登録?	マルチポート?	b/nb 変換?	タグ付け?
tlm_initiator_socket	no	yes	-	no
tlm_target_socket	no	yes	no	no
simple_initiator_socket	yes	no	-	no
simple_initiator_socket_tagged	yes	no	-	yes

simple_target_socket	yes	no	yes	no
simple_target_socket_tagged	yes	no	yes	yes
passthrough_target_socket	yes	no	no	no
passthrough_target_socket_tagged	yes	no	no	yes
multi_passthrough_initiator_socket	yes	yes	-	yes
multi_passthrough_target_socket	yes	yes	no	yes

12.1.1.2 ソケット・バインディング表

標準ソケットと便利ソケット間にて可能なバインディングの組み合わせは以下の表にて要約される。From と To をソケット型とすると、バインディングは、From（To） 又は From.bind（To）の形式で行う。

To From	tlm-init	simple-init	multi-init	tlm-targ	simple-targ	multi-targ
tlm-init	1			1	1	N :1
simple-init	1			1	1	N :1
multi-init			1	1 :M	1 :M	N :M

tlm-targ	1*	1*		1	1	
simple-targ	1*	1*				
multi-targ						1

上記のテーブルは、次の４つに分類されている。

子階層から親階層へのバインディング	イニシエータからターゲットへのバインディング
逆方向のバインディング	親階層から子階層へのバインディング

Key	
tlm-init	tlm_initiator_socket
simple-init	simple_initiator_socket or passthrough_initiator_socket
multi-init	multi_passthrough_initiator_socket
tlm-targ	tlm_target_socket
simple-targ	simple_target_socket or simple_target_socket_tagged or passthrough_target_socket or passthrough_target_socket_tagged
multi-targ	multi_passthrough_target_socket
1*	Target.bind（Initiator）のように逆方向に呼んだとしても、実際にはイニシエータからターゲットへのバインディングとなる。

12.1.2 シンプル・ソケット

12.1.2.1 イントロダクション

シンプル・ソケットは、使用するのが簡単なことから”シンプル・ソケット”と呼ばれる。 シンプル・ソケットは、インタオペラビリティ・レイヤのソケットである tlm_initiator_socket と tlm_target_socket から派生したクラスであるため、それらの型のソケットに直接バインドできる。

シンプル・ソケットは、対応するインタフェースを実装するオブジェクトに対してソケットをバインドする代わりに、コールバック関数を登録する方法を提供している。コールバック関数は、インタフェース・メソッド・コールが来る毎に呼び出される。コールバック・メソッドは、ソケットによってサポートされている各インタフェースに対して登録できる。

シンプル・ソケットを使用する場合、すべてのインタフェース・メソッドに対してコールバックを登録してもいいが、必須ではない。特に、シンプル・ターゲット・ソケットの場合、**b_transport** または、**nb_transport_fw** のどちらか片方のみを登録するだけでよい。この場合、登録されていない方のインタフェース・メソッドが呼び出された場合には、登録されている方のメソッドに自動変換される。この変換手法は単純なものではない。イニシエータとターゲットの双方の基本プロトコルの規則に準拠して変換が行われる。passthrough_target_socket は、simple_target_socket の一種であるが、ブロッキングとノンブロッキング呼び出しの間の変換機能はサポートしていない。

現在、シンプル・ソケットを実装するにあたり、、ダイナミックプロセスを利用している。したがって、OSCI（proof-of-concept）SystemC シミュレータの現在のリリースで、シンプル・ソケットを使用しているアプリケーションをコンパイルする際には、SystemC のヘッダファイルをインクルードする前に、SC_INCLUDE_DYNAMIC_PROCESSES マクロを定義する必要がある。

12.1.2.2 クラス定義

```
namespace tlm_utils {
  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class simple_initiator_socket : public tlm::tlm_initiator_socket<BUSWIDTH, TYPES> {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;

    simple_initiator_socket();
    explicit simple_initiator_socket( const char* n );

    void register_nb_transport_bw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));
```

```
    void register_invalidate_direct_mem_ptr(
      MODULE* mod,
      void (MODULE::*cb)(sc_dt::uint64, sc_dt::uint64));
};

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
>
class simple_target_socket : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;

    simple_target_socket();
    explicit simple_target_socket( const char* n );

    tlm::tlm_bw_transport_if<TYPES> * operator ->();

    void register_nb_transport_fw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));

    void register_b_transport(
      MODULE* mod,
      void (MODULE::*cb)(transaction_type&, sc_core::sc_time&));

    void register_transport_dbg(
      MODULE* mod,
      unsigned int (MODULE::*cb)(transaction_type&));

    void register_get_direct_mem_ptr(
      MODULE* mod,
      bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
};

template <
```

```
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
  >
  class passthrough_target_socket : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;

    passthrough_target_socket();
    explicit passthrough_target_socket( const char* n );

  void register_nb_transport_fw(
    MODULE* mod,
    sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));

  void register_b_transport(
    MODULE* mod,
    void (MODULE::*cb)(transaction_type&, sc_core::sc_time&));

  void register_transport_dbg(
    MODULE* mod,
    unsigned int (MODULE::*cb)(transaction_type&));

  void register_get_direct_mem_ptr(
    MODULE* mod,
    bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
  };
} // namespace tlm_utils
```

12.1.2.3 ヘッダファイル

シンプル・ソケットはそれぞれ、tlm_utils/simple_initiator_socket.h, tlm_utils/simple_target_socket.h, tlm_utils/passthrough_target_socket.h ヘッダファイルの中に定義される。

12.1.2.4 規則

a) コンストラクタに char *型の引数が指定された場合には、その引数を対応するベースクラスのコンストラクタに引き渡す。(引数が無い) デフォルトのコンストラクタが使用される場合には、sc_gen_unique_name("simple_initiator_socket"), sc_gen_unique_name("simple_target_socket"), sc_gen_unique_name("simple_passthrough_socket")

の戻り値を、それぞれベースクラスのコンストラクタの引数として指定する。

b) simple_initiator_socket は、simple_target_socket 又は、passthrough_target_socket に対して、それぞれの bind メソッド又は operator()によってバインドすることができる。なおこの二つの効果は同じである。いずれにしても、フォワード・パスは常にイニシエータからターゲットの方向になる。

c) simple_initiator_socket は tlm_target_socket に、tlm_initiator_socket は simple_target_socket 又は、passthrough_target_socket にそれぞれバインドできる。

d) （通常の）simple_initiator_socket、simple_target_socket、passthrough_target_socket は、コールバック関数をそれぞれ登録することにより、インタフェース・メソッド・コールを実装することができる(子階層の他のソケットに対して階層的にバインドする方法は採らない)。これに対して、子階層の simple_initiator_socket は、親階層の tlm_initiator_socket に対して階層的にバインドでき、親階層の tlm_target_socket は、子階層の simple_target_socket 又は、passthrough_target_socket に対して階層的にバインドできる。

e) nb_transport_fw コールバックが登録されているシンプル・ターゲット・ソケットでは、b_transport コールバック関数の登録は必須ではない。受信した b_transport 呼び出しは、自動的に登録されている nb_transport_fw 呼び出しに変換される。（「12.1.2.5 シンプル・ターゲット・ソケット b/nb 変換」を参照）

f) b_transport コールバックが登録されているシンプル・ターゲット・ソケットでは、nb_transport_fw コールバック関数の登録は必須ではない。nb_transport_fw が呼び出された場合、b_transport に登録されているコールバック関数を呼び出し、続いてバックワード・パスの nb_transpiort_bw を呼び出す。

g) ターゲットのシンプル・ターゲット・ソケットに b_transport コールバックも nb_transport_fw コールバックも登録されていない場合、実行時エラーになる。（コンパイルエラーにはならない）

h) ターゲットのパススルー・ターゲット・ソケットに対しては、**b_transport** と **nb_transport_fw** コールバック関数の両方を登録しなければならない。登録しなかった場合には、実行時エラーとなる。

i) ターゲットのシンプル・ターゲット・ソケット、または、パススルー・ターゲット・ソケットに対しては、必ずしも transport_dbg 関数を登録する必要は無い。登録しなかった場合に、**transport_dbg** が呼ばれた場合には、0 をリターンする。

j) ターゲットのシンプル・ターゲット・ソケット、または、パススルー・ターゲット・ソケットに対しては、必ずしも **get_direct_mem_ptr** 関数を登録する必要は無い。登録しなかった場合に、get_direct_mem_ptr が呼ばれた場合には、false 値をリターンする。

k) イニシエータのシンプル・イニシエータ・ソケットに対しては、**nb_transport_bw** コールバック関数を必ず登録しなければならない。登録しなかった場合には、**nb_transport_bw** メソッドの応答時に、実行時エラーとなる。

l) イニシエータのシンプル・イニシエータ・ソケットに対しては、必ずしも **invalidate_direct__mem_ptr** 関数を登録する必要はない。登録しなかった場合に、**invalidate_direct_mem_ptr** が呼ばれた場合には、単に無視される。

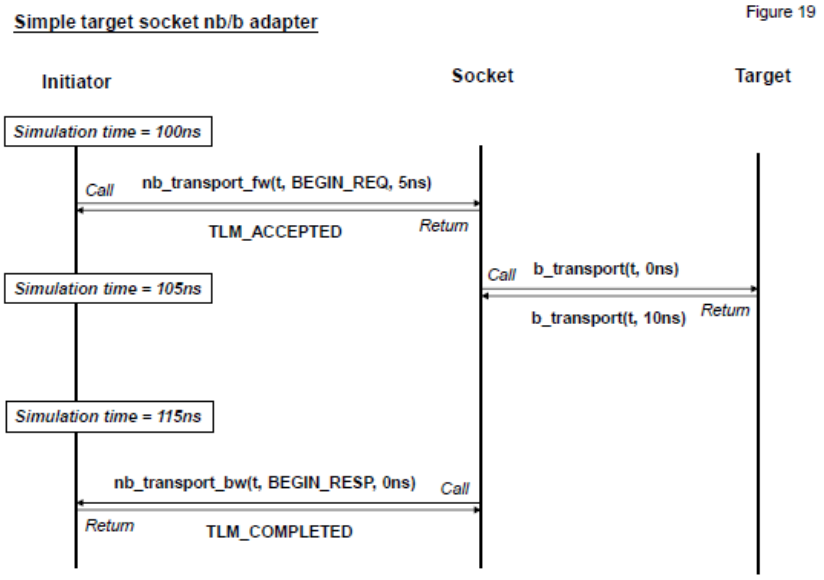
12.1.2.5 シンプル・ターゲット・ソケットb/nb変換

a) b_transport か nb_transport_fw メソッドのどちらかが simple_target_socket クラスのソケットを通して呼ばれたにも関わらず、対応するコールバック関数が登録されていない場合には、シンプル・ターゲット・ソケットは、2 つのインタフェースを結ぶアダプタのように振舞う。

b) シンプル・ターゲット・ソケットがアダプタとして機能する時、イニシエータの観点から、また、ターゲットにある b_transport か nb_transport_fw メソッドを実装する観点から、基本プロトコルの規則に準拠して動作する。

c) ソケットは、(自動変換する際に) トランザクション・オブジェクトに対して、一切変更を加えずにそのまま渡される。また、新しいトランザクション・オブジェクトを新規に作ることもしない。

d) ターゲットに登録されているのが、nb_transport_fw コールバック関数だけの場合、イニシエータからコールされた b_transport メソッドの実行中は、イニシエータが、nb_transport_fw を呼ぶことはできない。これはシンプル・ターゲット・ソケットの現在のインプリメンテーションの制限である。



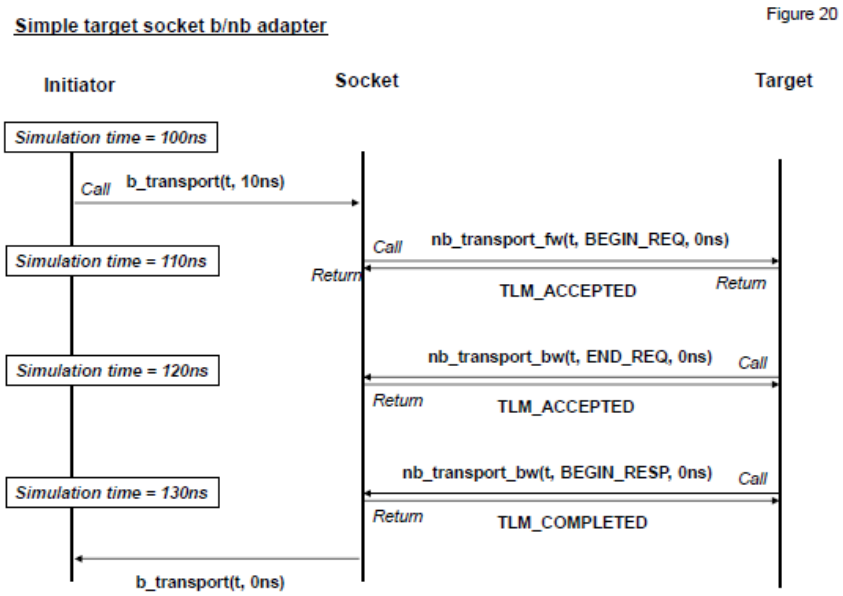
e) 図 19 は、イニシエータが nb_transport_fw メソッドをコールする様子を示している。イニシエータは nb_transport_fw メソッドをコールするが、ターゲットは、シンプル・ターゲット・ソケットに b_transport コールバック関数が登録されているだけである。イニシエータは BEGIN_REQ を送る。そして、ソケットは TLM_ACCEPTED をイニシエータにリターンする。ソケットは次に、b_transport をコールし、リターン後に BEGIN_RESP をイニシエータにリターンする。最後に、イニシエータは TLM_COMPLETED をソケットにリターンする。SystemC では、ノンブロッキング・メソッドから直接ブロッキング・メソッドを呼ぶことは許されていない。したがって、ソケットは、nb_transport_fw メソッドから直接 b_transport 関数をコールするのではなく、別のプロセスから b_transport をコールする。

f) 図 19 は 1 つの可能なシナリオを示しているにすぎない。他の例として、図の最後の遷移で、イニシエータは TLM_COMPLETED ではなく、TLM_ACCEPTED をリターンすることもで

きる。その場合、ソケットはイニシエータからこの後に END_RESP を受け取ることを期待する。また、ターゲットは、b_transport が呼ばれた場合、wait を発行する場合もある。

g) 図 20 は、イニシエータが、b_transport メソッドをコールする様子を示している。イニシエータは、b_transport メソッドをコールするが、ターゲットでは、シンプル・ターゲット・ソケットに nb_transport_fw コールバック関数しか登録されていない。イニシエータが、b_transport メソッドをコールすると、ソケットとターゲットは、基本プロトコルの規定に従って、nb_transport_fw を使いハンドシェークする。ターゲットは、END_REQ を送るかもしれない。あるいは、直接 BEGIN_RESP フェーズまでジャンプするかもしれない。ソケットは、コールされた nb_transport_bw が BEGIN_RESP であることを確認し TLM_COMPLETED をリターンする。

Figure 20



例：

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"          // Header files from utilities
#include "tlm_utils/simple_target_socket.h"

struct Initiator: sc_module
{
  tlm_utils::simple_initiator_socket<Initiator, 32, tlm::tlm_base_protocol_types> socket;

  SC_CTOR(Initiator)
  : socket("socket")                                    // Construct and name simple socket
  {                                                     // Register callbacks with simple socket
    socket.register_nb_transport_bw( this, &Initiator::nb_transport_bw );
```

```
    socket.register_invalidate_direct_mem_ptr( this, &Initiator::invalidate_direct_mem_ptr );
  }

  virtual tlm::tlm_sync_enum nb_transport_bw(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay ) {
    return tlm::TLM_COMPLETED;                          // Dummy implementation
  }
  virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
  { }                                                   // Dummy implementation
};

struct Target: sc_module                                // Target component
{
  tlm_utils::simple_target_socket<Target, 32, tlm::tlm_base_protocol_types> socket;

  SC_CTOR(Target)
  : socket("socket")                                    // Construct and name simple socket
  {                                                     // Register callbacks with simple socket
    socket.register_nb_transport_fw( this, &Target::nb_transport_fw );
    socket.register_b_transport( this, &Target::b_transport );
    socket.register_get_direct_mem_ptr( this, &Target::get_direct_mem_ptr );
    socket.register_transport_dbg( this, &Target::transport_dbg );
  }

  virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
  { }                                                   // Dummy implementation

  virtual tlm::tlm_sync_enum nb_transport_fw(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay ) {
    return tlm::TLM_ACCEPTED;                           // Dummy implementation
  }

  virtual bool get_direct_mem_ptr( tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmi_data)
  { return false; }                                     // Dummy implementation

 virtual unsigned int transport_dbg(tlm::tlm_generic_payload& r)
 { return 0; }                                          // Dummy implementation };

  SC_MODULE(Top)
  {
    Initiator *initiator;
```

```
    Target *target;
    SC_CTOR(Top) {
    initiator = new Initiator("initiator");
    target = new Target("target");
    initiator->socket.bind( target->socket );                    // Bind initiator socket to target socket
  }
};
```

12.1.3 タグ付きシンプル・ソケット

12.1.3.1 イントロダクション

タグ付きシンプル・ソケットは、シンプル・ソケットの一種である。タグ付きシンプル・ソケットは、どのソケットを通じてコールバック関数が呼ばれたかを特定できる整数 id をタグとして持っている。これは、複数のイニシエータ・ソケットやターゲット・ソケットがあり、それぞれのソケットに対して同じコールバック・メソッドを登録したい時に役に立つ。コールバック関数が呼ばれるとき、そのコールバック関数を登録したときに使用した id 番号が、第一引数として追加される。

12.1.3.2 ヘッダファイル

タグ付きシンプル・ソケットのクラス定義は、対応するシンプル・ソケットと同じヘッダファイルに格納される。すなわち、tlm_utils/simple_initiator_socket.h, tlm_utils/simple_target_socket.h, tlm_utils/passthrough_target_socket.h の中に定義される。

12.1.3.3 クラス定義

```
namespace tlm_utils {

  template <
  typename MODULE,
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm::tlm_base_protocol_types
  >

  class simple_initiator_socket_tagged : public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;

    simple_initiator_socket_tagged();
    explicit simple_initiator_socket_tagged( const char* n );
```

```
  void register_nb_transport_bw(
    MODULE* mod,
    sync_enum_type (MODULE::*cb)(int, transaction_type&, phase_type&, sc_core::sc_time&),
    int id);

  void register_invalidate_direct_mem_ptr(
    MODULE* mod,
    void (MODULE::*cb)(int, sc_dt::uint64, sc_dt::uint64),
    int id);
  };

  template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
   >

  class simple_target_socket_tagged : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    typedef tlm::tlm_fw_transport_if<TYPES> fw_interface_type;
    typedef tlm::tlm_bw_transport_if<TYPES> bw_interface_type;
    typedef tlm::tlm_target_socket<BUSWIDTH, TYPES> base_type;

    simple_target_socket_tagged();
    explicit simple_target_socket_tagged( const char* n );

    tlm::tlm_bw_transport_if<TYPES> * operator ->();

  void register_nb_transport_fw(
    MODULE* mod,
    sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
    int id);

  void register_b_transport(
    MODULE* mod,
    void (MODULE::*cb)(int id, transaction_type&, sc_core::sc_time&),
    int id);
```

```
  void register_transport_dbg(
     MODULE* mod,
     unsigned int (MODULE::*cb)(int id, transaction_type&),
     int id);

  void register_get_direct_mem_ptr( MODULE* mod,
     bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
     int id);
  };

  template <
     typename MODULE,
     unsigned int BUSWIDTH = 32,
     typename TYPES = tlm::tlm_base_protocol_types
  >

  class passthrough_target_socket_tagged : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;

    passthrough_target_socket_tagged();
    explicit passthrough_target_socket_tagged( const char* n );

    void register_nb_transport_fw(
       MODULE* mod,
       sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
       int id);

  void register_b_transport(
       MODULE* mod,
       void (MODULE::*cb)(int id, transaction_type&, sc_core::sc_time&),
       int id);

  void register_transport_dbg(
       MODULE* mod,
       unsigned int (MODULE::*cb)(int id, transaction_type&),
       int id);
```

```
  void register_get_direct_mem_ptr(
      MODULE* mod,
      bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
      int id);
  };
} // namespace tlm_utils
```

12.1.3.4 規則

a) コンストラクタに char *型の引数が指定された場合には、その引数を対応するベースクラスのコンストラクタに引き渡す。（引数が無い）デフォルトのコンストラクタが使用される場合には、sc_gen_unique_name(“simple_initiator_socket_tagged”), sc_gen_unique_name(“simple_target_socket_tagged”), sc_gen_unique_name(“simple_passthrough_socket_tagged”) の戻り値を、をそれぞれベースクラスのコンストラクタの引数として指定する。

b) int id タグが引数として追加される以外は、タグ付きシンプル・ソケットは、タグ付けをされていないシンプル・ソケットと全く同じ振る舞いをする。

c) 任意のコールバック・メソッドを、異なったタグを使用することにより、複数のソケットに登録できる。これがタグ付きソケットの目的である。

d) タグ (int id) は、コールバック関数登録時のインタフェース・メソッドの最後の引数になる。しかし、ソケットのコールバック関数実装の本体には、最初の引数としてこのタグが付けられる。

e) タグ付きシンプル・ソケットは、マルチ・ソケットではない。タグ付きシンプル・ソケットは、他のコンポーネントの複数のマルチ・ソケットにバインドすることができない。（「12.1.4 マルチ・ソケット」を参照）

12.1.4 マルチ・ソケット

12.1.4.1 イントロダクション

マルチ・ソケットは、タグ付きシンプル・ソケットの一種である。マルチ・ソケットは、他のコンポーネントの複数のソケットにバインドすることができる。タグ付きシンプル・ソケットは、どのソケットを通して呼び出されたのかを特定可能であるが、マルチ・ソケットのコールバックも、マルチポートのインデックス番号をタグとして、インタフェース・メソッド・コールが別のコンポーネントのどのソケットから呼び出されたかを特定できる。また、他の便利ソケットと異なり、マルチ・ソケットは、イニシエータ、ターゲット双方で、子ソケットから親ソケットへの階層バインドをサポートする。

12.1.4.2 ヘッダファイル

マルチ・ソケットのクラス定義は、それぞれ、tlm_utils/multi_passthrough_initiator_socket.h, tlm_utils/multi_passthrough_target_socket.h ヘッダファイルの中に定義される。

12.1.4.3 クラス定義

```
namespace tlm_utils
{
  template < typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
 >
  class multi_passthrough_initiator_socket : public multi_init_base< BUSWIDTH, TYPES, N, POL>
  {
  public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    typedef multi_init_base<BUSWIDTH, TYPES, N, POL> base_type;
    typedef typename base_type::base_target_socket_type base_target_socket_type;
    multi_passthrough_initiator_socket();
    multi_passthrough_initiator_socket(const char* name);
    ~multi_passthrough_initiator_socket();

    void register_nb_transport_bw(
      MODULE* mod,
      sync_enum_type (MODULE::*cb)(int, transaction_type&, phase_type&, sc_core::sc_time&));

    void register_invalidate_direct_mem_ptr(
      MODULE* mod,
      void (MODULE::*cb)(int, sc_dt::uint64, sc_dt::uint64));

    // Override virtual functions of the tlm_initiator_socket:
    virtual tlm::tlm_bw_transport_if<TYPES>& get_base_interface();
    virtual sc_core::sc_export<tlm::tlm_bw_transport_if<TYPES> >& get_base_export();
    void bind(base_target_socket_type& s);
    void operator() (base_target_socket_type& s);

    // SystemC standard callback
    // multi_passthrough_initiator_socket::before_end_of_elaboration must be called from
    // any derived class
    void before_end_of_elaboration();

    // Bind multi initiator socket to multi initiator socket (hierarchical bind)
    void bind(base_type& s);
```

```
    void operator() (base_type& s);

    tlm::tlm_fw_transport_if<TYPES>* operator[](int i);
    unsigned int size();
};

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0, sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>

class multi_passthrough_target_socket : public multi_target_base< BUSWIDTH, TYPES, N, POL> {
public:
    typedef typename TYPES::tlm_payload_type transaction_type;
    typedef typename TYPES::tlm_phase_type phase_type;
    typedef tlm::tlm_sync_enum sync_enum_type;
    typedef sync_enum_type
                        (MODULE::*nb_cb)(int, transaction_type&, phase_type&, sc_core::sc_time&);
    typedef void        (MODULE::*b_cb)(int, transaction_type&, sc_core::sc_time&);
    typedef unsigned int (MODULE::*dbg_cb)(int, transaction_type& txn);
    typedef bool        (MODULE::*dmi_cb)(int, transaction_type& txn, tlm::tlm_dmi& dmi);

    typedef multi_target_base<BUSWIDTH, TYPES, N, POL> base_type;
    typedef typename base_type::base_initiator_socket_type base_initiator_socket_type;
    typedef typename base_type::initiator_socket_type initiator_socket_type;

    multi_passthrough_target_socket();
    multi_passthrough_target_socket(const char* name);
    ~multi_passthrough_target_socket();

    void register_nb_transport_fw (MODULE* mod, nb_cb cb);
    void register_b_transport (MODULE* mod, b_cb cb);
    void register_transport_dbg (MODULE* mod, dbg_cb cb);
    void register_get_direct_mem_ptr (MODULE* mod, dmi_cb cb);

    // Override virtual functions of the tlm_target_socket:
    virtual tlm::tlm_fw_transport_if<TYPES>& get_base_interface();
    virtual sc_core::sc_export<tlm::tlm_fw_transport_if<TYPES> >& get_base_export();
    // SystemC standard callback
```

```
    // multi_passthrough_target_socket::end_of_elaboration must be called from any derived class
    void end_of_elaboration();
    void bind(base_type& s);
    void operator() (base_type& s); tlm::tlm_bw_transport_if<TYPES>* operator[] (int i); unsigned int size();
  };
} // namespace tlm_utils
```

12.1.4.4 規則

a) ベースクラス multi_init_base と multi_target_base はインプリメンテーション依存であるので、直接アプリケーションから使用してはいけない。

b) コンストラクタに char *型の引数が指定された場合には、その引数を対応するベースクラスのコンストラクタに引き渡す。(引数が無い) デフォルトのコンストラクタが使用される場合には、sc_gen_unique_name("multi_passthrough_initiator_socket"), sc_gen_unique_name("multi_passthrough_target_socket")の戻り値を、それぞれベースクラスのコンストラクタの引数として使用する。

c) クラス multi_passthrough_initiator_socket と、multi_passthrough_target_socket は、マルチ・ソケットとして振る舞う。つまり、一つのイニシエータ・ソケットは複数のターゲット・ソケットとバインドでき、一つのターゲット・ソケットは複数のイニシエータ・ソケットとバインドできる。この 2 つのクラステンプレートは、バインド数とポート・バインド・ポリシーを定義するテンプレート・パラメタを持ち、sc_port のテンプレート・インスタンスをパラメタライズするために、クラス・インプリメンテーションの中で使われている。

d) 一つの multi_passthrough_initiator_socket は、複数の tlm_target_socket、複数の simple_target_socket、複数の passthrough_target_socket、複数の multi_passthrough_target_sockets. とバインドできる。 複数の tlm_initiator_sockets、複数の simple_initiator_socket、複数の multi_passthrough_initiators_sockets は、一つの multi_passthrough_target_socket にバインドできる。

e) 一つの multi_passthrough_initiator_socket は、ただ一つの multi_passthrough_initiator_socket に階層バインドできる。 multi_passthrough_target_socket は、ただ一つの multi_passthrough_target__socket と階層バインドできる。これら二つのケースを除いて、マルチ・ソケットは他のソケットと階層バインドすることはできない。マルチ・バインドの機能は階層バインドには適用できないが、一つ又は複数のイニシエータ・ソケットが、一つ又は複数のターゲット・ソケットにバインドするときのみ許される。

f) バインディング・オペレータは、イニシエータ・ソケットからターゲット・ソケットの方向のみで使用できる。言い換えれば、tlm_target_socket や simple_target_socket クラスと異なり、multi_passthrough_target_socket にはターゲット・ソケットをイニシエータ・ソケットにバインドする (逆方向の) 演算子がない。

g) 階層バインドを使用する場合には、子階層のイニシエータ・マルチ・ソケットは親階層のイニシエータ・マルチ・ソケットに、また、親階層のターゲット・マルチ・ソケットは、子階層のターゲット・マルチ・ソケットにバインドしなければならない。これは、上記のイニシエータからターゲットへのバインド方向のルールと一貫性がある。

h) クラス multi_passthrough_initiator_socket または、multi_passthrough_target_socket が複数回バインドされていた場合、メソッド operator []により、 ソケットがバインドしているオブジェクトを区別することができる。インデックス値は、ソケットをバインドする為に bind メソッドか operator()が呼ばれた順番で決まる。これと同じインデックス値は、コールバックで id タグを決める際に使われる。

i) 例えば、multi_passthrough_initiator_socket が 2 つの異なるターゲットにバインドされている場合を考えてみる。この場合、これらは以下のように区別する。

socket[0]->nb_transport_fw(...)

socket[1]->nb_transport_fw(...)

2 つこれら 2 つのターゲットからの nb_transport_bw()メソッド呼び出しは、それぞれ 0 と 1 のタグを持つ。

j) メソッド size は、現在接続されているマルチ・ソケットのインスタンス数をリターンする。SystemC のマルチポートのように、サイズがエラボレーション中すなわち、end_of_elaboration コールバックの前に呼ばれると、その値は、インプリメンテーション依存の値をリターンする（つまり値が保証されない）。なぜなら、どの時点でポートがバインドされるかについては、インプリメンテーション依存だからである。

k) 子階層のマルチ・ソケットへの階層バインディングが無い場合には、ターゲットは、マルチ・ソケットに対して **b_transport** と **nb_transport_fw** コールバックを登録しなくてはいけない。登録されない場合は、該当するメソッドが呼ばれたときにのみ、実行時エラーが発生する。

l) 子階層のマルチ・ソケットへの階層バインディングが無い場合には、ターゲットは、マルチ・ソケットに対して、**transport_dbg** コールバックを登録しなくても良い。この場合に **transport_dbg** が呼ばれると 0 を返す。

m) 子階層のマルチ・ソケットへの階層バインディングが無い場合には、ターゲットは、マルチ・ソケットに対して、**get_direct_mem_ptr** コールバックを登録しなくても良い。この場合に **get_direct_mem_ptr** が呼ばれると false を返す。

n) 子階層のマルチ・ソケットへの階層バインディングが無い場合には、イニシエータは、マルチ・ソケットに対して **nb_transport_bw** コールバックを登録しなくてはいけない。登録されない場合は、**nb_transport_bw** が呼ばれたときにのみ、実行時エラーが発生する。

o) 子階層のマルチ・ソケットへの階層バインディングが無い場合には、イニシエータは、マルチ・ソケットに対して、**invalidate_direct_mem_ptr** コールバックを登録しなくても良い。この場合に **invalidate_direct_mem_ptr** が呼ばれると無視される。

例

```
// Initiator component with a multi-socket
struct Initiator: sc_module
{
  tlm_utils::multi_passthrough_initiator_socket<Initiator> socket;
  SC_CTOR(Initiator) : socket("socket") {
```

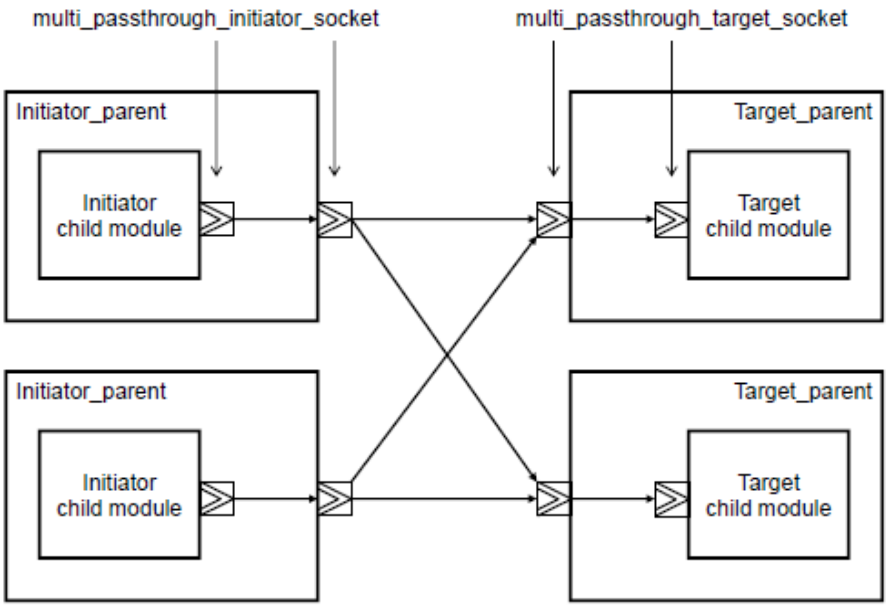
```
    // Register callback methods with socket
        socket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);
        socket.register_invalidate_direct_mem_ptr(this, &Initiator::invalidate_direct_mem_ptr); ...
};

struct Initiator_parent: sc_module
    {
    tlm_utils::multi_passthrough_initiator_socket<Initiator_parent> socket;
    Initiator *initiator;

    SC_CTOR(Initiator_parent) : socket("socket") {
        initiator = new Initiator("initiator");
        // Hierarchical binding of initiator socket on child to initiator socket on parent
        initiator->socket.bind( socket );
    }
};
```

Figure 21

Hierarchical Binding of Multi-sockets




```
struct Target: sc_module
{
    tlm_utils::multi_passthrough_target_socket<Target> socket;

    SC_CTOR(Target) : socket("socket") {
        // Register callback methods with socket
        socket.register_nb_transport_fw( this, &Target::nb_transport_fw);
```

```
        socket.register_b_transport( this, &Target::b_transport);
        socket.register_get_direct_mem_ptr(this, &Target::get_direct_mem_ptr);
        socket.register_transport_dbg( this, &Target::transport_dbg); ...
};

// Target component with a multi-socket
struct Target_parent: sc_module
{
   tlm_utils::multi_passthrough_target_socket<Target_parent> socket;
   Target *target;

   SC_CTOR(Target_parent) : socket("socket") {
      target = new Target("target");
      // Hierarchical binding of target socket on parent to target socket on child
      socket.bind( target->socket );
   }
};

SC_MODULE(Top)
{
 Initiator_parent *initiator1;
 Initiator_parent *initiator2;
 Target_parent *target1;
 Target_parent *target2;

 SC_CTOR(Top)
 {
   // Instantiate two initiator and two target components
   initiator1 = new Initiator_parent("initiator1");
   initiator2 = new Initiator_parent("initiator2");
   target1 = new Target_parent ("target1");
   target2 = new Target_parent ("target2");

   // Bind two initiator multi-sockets to two target multi-sockets
   initiator1->socket.bind( target1->socket );
   initiator1->socket.bind( target2->socket );
   initiator2->socket.bind( target1->socket );
   initiator2->socket.bind( target2->socket );
 }
};
```

12.2 クォンタム・キーパー

12.2.1 イントロダクション

テンポラル・デカップリング機能は、SystemC のプロセスに対して、クォンタムと呼ばれる時間を実際のシミュレーション時刻より先に進めるものである。詳細は「5 グローバル・クォンタム」を参照のこと。

ユーティリティ・クラス tlm_quantumkeeper には、クォンタム時間を管理・制御するためのメソッドが用意されている。テンポラル・デカップリング機能を用いる時には、一定のコーディング・スタイルを保つために、クォンタム・キーパーを用いることを推奨する。しかしながら、SystemC 記述の中に直接この機能を実装するということも容易である。tlm_quantumkeeper クラスを使用するかどうかに関わらず、テンポラル・デカップリング機能を用いる全てのモデルは、tlm_global_quantum クラスによって管理されるグローバル・クォンタムを参照すべきである。

tlm_quantumkeeper クラスのネームスペースは tlm_utils である。

テンポラル・デカップリングの詳細については、「3.3.2 ルーズリー・タイムド・コーディング・スタイルとテンポラル・デカップリング」も参照のこと。

タイミング・アノテーションに関しては、「4.1.3 トランスポート・インタフェースにおけるタイミング・アノテーション」も参照のこと。

12.2.2 ヘッダファイル

クォンタム・キーパーのクラス定義は、tlm_utils/tlm_quantumkeeper.h ヘッダファイルにある。

12.2.3 クラス定義

```
namespace tlm_utils {
class tlm_quantumkeeper
{
public:
  static void set_global_quantum( const sc_core::sc_time& );
  static const sc_core::sc_time& get_global_quantum();
  tlm_quantumkeeper(); virtual ~tlm_quantumkeeper();
  virtual void inc( const sc_core::sc_time& );
  virtual void set( const sc_core::sc_time& );
  virtual sc_core::sc_time get_current_time() const;
  virtual sc_core::sc_time get_local_time();
  virtual bool need_sync() const;
  virtual void sync();
  void set_and_sync(const sc_core::sc_time& t)
  {
    set(t);
    if (need_sync()) sync();
  }
```

```
   virtual void reset();
protected:
   virtual sc_core::sc_time compute_local_quantum();
};
} // namespace tlm_utils
```

12.2.4 テンポラル・デカップリングを使用するプロセスに関する一般ルール

a) シミュレーション速度を最大にするためには、全てのイニシエータ・モデルがテンポラル・デカップリングを用い、他の SystemC プロセスを使用しないか、または、必要最小限に抑える必要がある。

b) 理想的には、テンポラル・デカップリングを行うイニシエータ・モデルのみが、SystemC プロセスを持ち、それぞれの SystemC プロセスにおいては、時間を先に進めておき、クォンタム量を超えた時のみシミュレーション・カーネルに制御を譲るようにするとよい。

c) テンポラル・デカップリングを持つイニシエータは、他のプロセスとの通信において、明示的に同期をとる方法が定義されていれば、クォンタム時間を用いる必要はない。クォンタム時間を用いる場合は、イニシエータ間の一般的な通信間隔よりも小さくなるよう設定すべきである。そうしないと重要なプロセス間通信が失われ、モデルが正常に動作しないかもしれない。

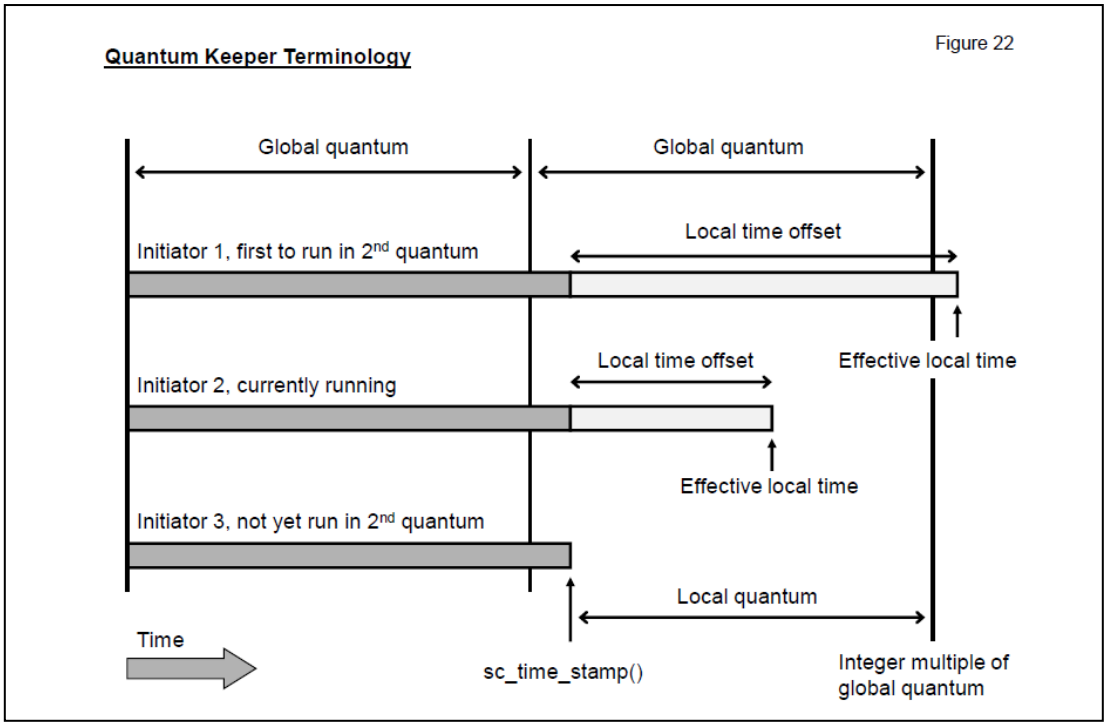
d) Yield とは、SC_THREAD または SC_CTHREAD の場合には、wait()を実行すること、SC_METHOD の場合には、関数から return することを意味する。

e) テンポラル・デカップリングは、あくまでも標準的な SystemC カーネル上で動作するものであるので、イベントはスケジュールできるし、プロセスは停止、再開することもでき、ルーズリー・タイムド・モデルは他のコーディング・スタイルのモデルと混在できる。

f) すべてのイニシエータがテンポラル・デカップリングを使用しなければいけないというわけではない。テンポラル・デカップリングを使用しているプロセスと使用していないプロセスは混在可能である。しかし、テンポラル・デカップリングを使用していないプロセスがシミュレーション・スピードのボトルネックとなる場合が多い。

g) テンポラル・デカップリングを行う各イニシエータは、本章で説明するローカル・タイム・オフセット変数を用いて、ローカルな処理時間と通信時間を計算する。クォンタム・キーパーがローカル・タイム・オフセットを管理することを推奨する。

h) sc_time_stamp()メソッドは、現在のクォンタムが開始した実シミュレーション時間を返す。



i) ローカル・タイム・オフセットは、SystemC スケジューラからは知ることができない。トランスポート・インタフェースを使用する場合は、b_transport()あるいは nb_transport()関数の引数にローカル・タイム・オフセットを渡すべきである。

j) nb_transport メソッドでテンポラル・デカップリングやクォンタム・キーパーを使用することは、ルール上禁止されているわけではない。しかし、アプロキシメイトリー・タイムド・コーディング・スタイル固有のプロセス間通信のオーバーヘッドが大きく、テンポラル・デカップリングによる速度の利点が失われる可能性があるため、常に有利に働くとは限らない。

k) クォンタム内でのプロセスの実行順序は SystemC スケジューラの制御の下にあり、SystemC の仕様上不確定である。このため明示的な同期機構がない場合、あるプロセスで書き換えられた変数が別のプロセスで読まれた時、その値は不確定である。クォンタム長と比較して変数値の変化の頻度は低いと仮定すると、変数の新しい値は現在のクォンタムで有効になる場合もあれば、次のクォンタムで有効になる場合もある。このため、アプリケーションは新しい値が有効になるタイミングの正確さを求めてはならない。そうでない場合には、アプリケーションは適切な同期機構を用いて変数へのアクセスをガードすべきである。

l) テンポラル・デカップリングを行うプロセスから変数やオブジェクトの値を読み出す場合には、現在のプロセス、あるいは現クォンタムですでに実行済のテンポラル・デカップリングを行う別のプロセスで変更されていない限り、現在のクォンタムの開始した時点における値を返す。とくに、テンポラル・デカップリングを行うプロセスから sc_signal にアクセスした場合、その値はクォンタム開始時点の値となる。これは、sc_time_stamp()で得られる SystemC のシミュレーション時間が、クォンタム内では進まないためである。

12.2.5 tlm_quantumkeeperクラス

a) コンストラクタはローカル・タイム・オフセットを SC_TIME_ZERO に設定するが、仮想関

数の compute_local_quantum()はコールしない。コンストラクタはローカル・クォンタムを計算しないため、アプリケーションはクォンタム・キーパー・オブジェクトを生成直後に、reset()メソッドを呼びださなければならない。

b) tlm_quantumkeeper クラスの実装では、sc_time クラスの静的オブジェクトは生成してはならないが、コンストラクタは sc_time クラスのオブジェクトを生成してもよい。これは、アプリケーションが最初のクォンタム・キーパー・オブジェクトを生成する前に、sc_core::sc_set_time_resolution()関数をコールしてもよいことを意味している。

c) set_global_quantum()メソッドは、引数で渡された値をグローバル・クォンタム時間として設定するが、ローカルのクォンタム時間は変更しない。get_global_quantum()メソッドはグローバル・クォンタム時間を返す。set_global_quantum()を呼んだ後には、reset()メソッドを呼び出し、ローカル・クォンタム時間を再計算させることが望ましい。

d) get_local_time()メソッドはローカル・タイム・オフセットを返す。

e) get_current_time()メソッドは、実効ローカル・タイム、すなわち、sc_time_stamp()+ローカル・タイム・オフセットを返す。

f) inc()メソッドは、引数で渡した値をローカル・タイム・オフセットに加算する。

g) set()メソッドは、引数で渡した値をローカル・タイム・オフセットに設定する。

h) need_sync()メソッドは、現在のローカル・タイム・オフセットがローカル・クォンタムより大きい時に true を返す。

i) sync()メソッドは、wait(ローカル・タイム・オフセット)を呼び出し、シミュレーション時間が実効ローカル・タイムと一致するまで当該プロセスを停止させ、その後 reset()メソッドを呼び出す。

j) set_and_sync()メソッドは、連続的に set()メソッド、need_sync()メソッド、sync()メソッドを呼び出す便利なメソッドである。このメソッドを書き換えてはならない。

k) reset()メソッドは、compute_local_quantum()を呼び出し、ローカル・タイム・オフセットを SC_ZERO_TIME に戻す。

l) tlm_quantumkeeper クラスの compute_local_quantum()メソッドは、tlm_global_quantum クラスの compute_local_quantum メソッドを呼び出す。しかし、より小さいローカル・クォンタムを計算するために、書き換えられる場合がある。

m) tlm_quantumkeeper クラスはクォンタム・キーパーのデフォルトの実装という位置づけである。tlm_quantumkeeper クラスを継承したクラスを作り、compute_local_quantum メソッドを書き換えることが可能ではあるが、通常は行わない。

n) ローカル・タイム・オフセットがローカル・クォンタム以上になった場合には、プロセスをカーネルに制御を譲る必要がある。このとき wait()を直接呼び出すのではなく、sync()メソッドを使うことを強く推奨する。

o) クォンタム時間を超えた場合に強制的に同期を実行するような仕組みはない。したがって、イニシエータは自分で need_sync メソッドを呼び出し、必要に応じて sync()を呼び出す必要がある。

p) b_transport()メソッドは、その呼び出し前後で sc_time_stamp の値(グローバル時間)が変化して

いた場合に、自身でカーネルに制御を渡してもよい。ローカル・タイム・オフセットやタイミング・アノテーションの値は、常に sc_time_stamp で表される現在時刻に対する相対値で表現される。b_transport()あるいは nb_transport_fw()から復帰した時、イニシエータは set()メソッドにより、クォンタム・キーパーのローカル・タイム・オフセットを設定し、need_sync()メソッドにより同期チェックを行う必要がある。

q) イニシエータがクォンタム時間より前に同期を行う必要がある場合、すなわち、実行を中断してシミュレーション時間をローカル時間まで進める必要が生じた場合、sync()を呼び出すか、あるいは明示的に別のイベントを wait させることができる。これにより、他のプロセスが実行可能となる。この方法を、 sync-on-demand（要求に応じた同期）と呼ぶ。

r) sync()を頻繁にコールし過ぎると、テンポラル・デカップリングの効果が減ってしまう。

例

```
struct Initiator: sc_module                                    // Loosely-timed initiator
{
  tlm_utils::simple_initiator_socket<Initiator> init_socket;
  tlm_utils::tlm_quantumkeeper m_qk;                           // The quantum keeper
  SC_CTOR(Initiator) : init_socket("init_socket") {
    SC_THREAD(thread);                                         // The initiator process
    ...
    m_qk.set_global_quantum( sc_time(1, SC_US) );              // Replace the global quantum
    m_qk.reset();                                              // Re-calculate the local quantum
  }
  void thread() {
    tlm::tlm_generic_payload trans;
    sc_time delay;
    trans.set_command(tlm::TLM_WRITE_COMMAND);
    trans.set_data_length(4);
    for (int i = 0; i < RUN_LENGTH; i += 4) {
      int word = i;
      trans.set_address(i);
      trans.set_data_ptr( (unsigned char*)(&word) );
      delay = m_qk.get_local_time();                           // Annotate b_transport with local time
      init_socket->b_transport(trans, delay );
      qk.set( delay );                                         // Update qk with time consumed by target
      m_qk.inc( sc_time(100, SC_NS) );                         // Further time consumed by initiator
      if ( m_qk.need_sync() ) m_qk.sync();                     // Check local time against quantum
      }
    }
  ...
};
```

12.3 ペイロード・イベント・キュー

12.3.1 イントロダクション

ペイロード・イベント・キュー（PEQ）は、SystemC のイベント通知用のキューであり、各通知は関連付けられたトランザクション・オブジェクトを持つ。それぞれのトランザクションは遅延と共に PEQ 内に格納され、現在のシミュレーション時間と設定された遅延時間の合計時間になった時点で PEQ から出力される。

2 つのペイロード・イベント・キューがユーティリティとして提供されている。PEQ は便利なだけではなく、アプロキシメイトリー・タイムド・コーディング・スタイルにおけるタイミング・アノテーションの仕組みを理解する上でも概念的に関連がある。しかし、ここで説明するペイロード・イベント・キューを使わなくても、アプロキシメイトリー・タイムド・モデルの実装は可能である。アプロキシメイトリー・タイムド・モデルでは、nb_transport()によって遅延時間を持つトランザクションを受け取った時、それを遅延時間とともに PEQ へ格納するのが適している。PEQ は、nb_transport()コールに関するタイミング・ポイントを正確なシミュレーション時刻に発生するようにスケジューリングする。

PEQ の notify()メソッドを、引数に遅延を指定してコールすることにより、PEQ へトランザクションを格納する。即刻通知をスケジュールする notify()メソッドもある。遅延時間は現在のシミュレーション時間（sc_time_stamp）と加算され、トランザクションが PEQ の最後から出力する時間となる。イベントのスケジューリングは、内部的に sc_event クラスを利用した SystemC の時間イベント通知によって管理される。待ち状態のイベント通知がある時に notify()メソッドがコールされた場合は、他の通知がキャンセルまでの間、最も早いシミュレーション時間における通知が残ったままとなる。

PEQ には 2 種類があって、それぞれトランザクションの出てきかたが異なる。peq_with_get の場合は、get_event()メソッドをコールすると、トランザクションが取り出し可能な状態であればいつでも通知されるイベントを返す。get_next_transaction()メソッドは、有効になっているトランザクションを一度に 1 つを取り出すために、繰り返しコールされるべきである。

peq_with_cb_and_phase の場合は、コールバック・メソッドをコンストラクタ引数で登録し、各トランザクションが生じた際にそのメソッドがコールされる。この特定の PEQ は、通知ごとにトランザクション・オブジェクトとフェーズ・オブジェクトの両方を持ち、両方がコールバック関数の引数として渡される。

例題については「8.1 フェーズ」を参照のこと。

peq_with_cb_and_phase の現在の実装ではダイナミックプロセスを使用する。そのため、OSCI がリリースしているリファレンスシミュレータを用いて peq_with_cb_and_pahse を使うアプリケーションをコンパイルする時には、SystemC ヘッダファイルをインクルードする前に SC_INCLUDE_DYNAMIC_PROCESSES マクロを定義する必要がある。

12.3.2 ヘッダファイル

2 つのペイロード・イベント・キューのクラス定義は、tlm_utils/peq_with_get.h および tlm_utils/peq_with_cb_and_phase.h ヘッダファイルになければならない。

12.3.3 クラス定義

```
namespace tlm_utils {
  template <class PAYLOAD>
  class peq_with_get : public sc_core::sc_object
  {
  public:
    typedef PAYLOAD transaction_type;

    peq_with_get(const char* name);

    void notify(transaction_type& trans, sc_core::sc_time& t);
    void notify(transaction_type& trans);

    transaction_type* get_next_transaction();
    sc_core::sc_event& get_event();
    void cancel_all();
  };

  template<typename OWNER, typename TYPES=tlm::tlm_base_protocol_types>
  class peq_with_cb_and_phase : public sc_core::sc_object
  {
  public:
    typedef typename TYPES::tlm_payload_type tlm_payload_type;
    typedef typename TYPES::tlm_phase_type tlm_phase_type;
    typedef void (OWNER::*cb)(tlm_payload_type&, const tlm_phase_type&);

    peq_with_cb_and_phase(OWNER* , cb );
    peq_with_cb_and_phase(const char* , OWNER* , cb);
    ~peq_with_cb_and_phase();

    void notify (tlm_payload_type& , tlm_phase_type& , const sc_core::sc_time& );
    void notify (tlm_payload_type& , tlm_phase_type& );
    void cancel_all();
  };
} // namespace tlm_utils
```

12.3.4 ルール

a) notify メソッドは PEQ にトランザクションを挿入しなけらばならない。そのトランザクションは t1+t2 の時間で PEQ から取り出されなければならない。ここで t1 は notify メソッドがコールされた時点での sc_time_stamp()が返す値である。t2 は notify メソッドの引数に指定した sc_time 変数の値である。即時通知の場合には、トランザクションは SystemC スケジューラ

の現在の評価フェーズで取り出されなければならない。

b) 前のルールによって、トランザクションはどのような順番で挿入され、取り出されてもよい。トランザクションを挿入した順番に取り出す必要はない。

c) どんな時でも、PEQ に挿入可能なトランザクション数に制限はない。

d) 同一の時間に取り出されるトランザクションが複数挿入された場合、挿入された順番で同一の評価フェーズ（同一のデルタサイクル）で全トランザクションが取り出されなければならない。

e) cancel_all メソッドは PEQ から全トランザクションを即座に削除しなけらばならない。実質的には PEQ を初期状態に戻す。これが PEQ から全トランザクションを削除する唯一の方法である。

f) peq_with_get クラスの PAYLOAD テンプレート引数は、PEQ で用いるトランザクション型名でなければならない。

g) get_event メソッドは PEQ から次のトランザクションを取り出し可能になった時に通知されるイベントへのリファレンスを返さなければならない。同一の評価フェーズ（同一のデルタサイクル）で複数のトランザクションが取り出せる場合には、イベントは一度しか通知されない。

h) get_next_transaction メソッドは PEQ から取り出せるトランザクション・オブジェクトへのポインタを返し、PEQ からそのトランザクション・オブジェクトを削除しなけらばならない。イベント通知が発生した評価フェーズで PEQ からトランザクションを取り出さない場合には、その時刻もしくはその後の時刻での get_next_taransaction()コールでそのトランザクションは取り出せる。

i) 現在の評価フェーズにおいて取り出せるトランザクションが１つもない場合は、get_next_transaction メソッドは NULL ポインタを返さなければならない。

j) peq_with_cb_and_phase クラスの TYPES テンプレート引数には、PEQ で用いるトランザクションとフェーズ型を含むプロトコル・トレイツ・クラス名でなければならない。

k) peq_with_cb_and_phase クラスの OWNER テンプレート引数は、PEQ のコールバック関数をメンバとして持つクラス型でなければならない。通常は PEQ インスタンスを持つ親モジュールである。

l) peq_with_cb_and_phase クラスのコンストラクタ引数である OWNER*は、PEQ のコールバック関数をメンバとして持つオブジェクトへのポインタでなけらばならない。通常は PEQ インスタンスを持つ親モジュールである。

m) peq_with_cb_and_phase クラスのコンストラクタ引数である cb は、PEQ のコールバック関数名でなければならない。それはメンバ関数でなければならない。

n) peq_with_cb_and_phase クラスの実装は、PEQ からトランザクション・オブジェクトの取り出しが可能になったときはいつでも、PEQ のコールバック関数をコールしなければならない。コールバック関数の第一引数はトランザクション・オブジェクトへの参照で、第二引数はフェーズ・オブジェクトへの参照であり、これらは対応した notify メソッドへ渡されたものである。

o) peq_with_cb_and_phase クラスの実装は、SystemC の SC_METHOD プロセスから PEQ のコールバック関数を呼び出さなければならない。したがって、コールバック関数はノンブロッキングでなければならない。

p) peq_with_cb_and_phase クラスの実装は、一つのトランザクションにつき一回だけ PEQ のコールバック関数を呼び出すことになっている。コールバック関数を呼び出した後で、実装は PEQ からトランザクション・オブジェクトを削除しなければならない。PEQ のコールバック関数は、同一の評価フェーズで複数回呼び出される可能性がある。

12.4 インスタンス固有の拡張

12.4.1 イントロダクション

汎用ペイロードは、個々のトランザクション・オブジェクトが個々の拡張型のインスタンスをせいぜい一つ含むことができるように、拡張オブジェクトへのポインタの配列を持つ。このメカニズム単体で、与えられたトランザクション・オブジェクトに同じ拡張の複数のインスタンスを直接加えることを許さない。この節では、インスタンス固有の拡張を提供するユーティリティ一式、すなわち、同じ型の複数拡張を単体のトランザクション・オブジェクトに加えることについて述べる。

インスタンス固有の拡張型は、tlm_extension クラスと同様な使い方で instance_specific_extension クラステンプレートを使うことで作られる。tlm_extension と違って、アプリケーションは仮想メソッド clone と copy_from の実装を要求されないし許されない。そのアクセスメソッドは set_extension、get_extension、clear_extension、resize_extension に制限される。インスタンス固有拡張の自動削除はサポートされない。したがって、set_extension を呼ぶコンポーネントは clear_extension も呼び出すべきである。tlm_extension クラスと同様に、resize_extensions メソッドはトランザクション・オブジェクトが静的初期化の間に作られたときだけ呼び出される必要がある。

インスタンス固有の拡張は、instance_specific_extension_accessor クラスのオブジェクトを使ってアクセスされる。このクラスはただ１つの operator()メソッドを提供し、そのメソッドはアクセスメソッドを呼び出すことができる代理オブジェクトを返す。instance_specific_extension_accessor クラスの個々のオブジェクトは、同じトランザクション・オブジェクトに使われているときでさえ、拡張オブジェクトの別個のセットへのアクセスを与える。

以下のクラス定義においてイタリックの単語は実装において定義される名前で、アプリケーションによって直接使われてはならない。

12.4.2 ヘッダファイル

インスタンス固有の拡張のクラス定義は、tlm_utils/instanc_specific_extensions.h ヘッダファイルの中になければならない。

12.4.3 クラス定義

```
namespace tlm_utils {

  template <typename T>
  class instance_specific_extension : public implementation-defined {
```

```
  public:
    virtual ~instance_specific_extension();
  };

  template<typename U>
  class proxy {
  public:
    template <typename T> T* set_extension(T*);
    template <typename T> void get_extension(T*&) const;
    template <typename T> void clear_extension(const T*);
    void resize_extensions();
  };

  class instance_specific_extension_accessor {
  public:
    instance_specific_extension_accessor();

    template<typename T> proxy< implementation-defined >& operator() ( T& );
  };

} // namespace tlm_utils
```

例

```
struct my_extn : tlm_utils::instance_specific_extension<my_extn> {
  int num;                                          // User-defined extension attribute
};

struct Interconnect: sc_module
{
  tlm_utils::simple_target_socket<Interconnect> targ_socket;
  tlm_utils::simple_initiator_socket<Interconnect> init_socket;
  ...
  tlm_utils::instance_specific_extension_accessor accessor;
  static int count;

  virtual tlm::tlm_sync_enum nb_transport_fw(
    tlm::tlm_generic_payload& trans, tlm::tlm_phase& phase, sc_time& delay )
 {
    my_extn* extn;
    accessor(trans).get_extension(extn);            // Get existing extension
    if (extn) {
```

```
        accessor(trans).clear_extension(extn);          // Delete existing extension
      } else {
        extn = new my_extn;
        extn->num = count++;
        accessor(trans).set_extension(extn);            // Add new extension
    }
    return init_socket->nb_transport_fw( trans, phase, delay );
  }
    ...
};

... SC_CTOR(Top) {
    // Transaction object passes through two instances of Interconnect
    interconnect1 = new Interconnect("interconnect1");
    interconnect2 = new Interconnect("interconnect2");
    interconnect1->init_socket.bind( interconnect2->targ_socket );
    ...
```

13. TLM-1 とアナリシス・ポート

以下の TLM-1 のコア・インタフェース及び tlm_fifo チャネル、アナリシス・インタフェース、アナリシス・ポートは依然として OSCI 標準であり、TLM-2.0 ソフトウェアの配布物に入れられている。しかし、TLM-2.0 標準の主体からは切り離されており、ここでは詳しく触れない。

13.1 TLM-1 コア・インタフェース

シグネチャ transport(const REQ& , RSP&)付きのトランスポート・メソッド はTLM-1 に含まれていなかったが、TLM 2.0 において付け加えられた。

```
namespace tlm {

// Bidirectional blocking interfaces
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_core::sc_interface
{
public:
   virtual RSP transport( const REQ& ) = 0;
   virtual void transport( const REQ& req , RSP& rsp ) { rsp = transport( req ); } };

// Uni-directional blocking interfaces
template < typename T > class tlm_blocking_get_if : public virtual sc_core::sc_interface
{
public:
   virtual T get( tlm_tag<T> *t = 0 ) = 0;
   virtual void get( T &t ) { t = get(); }
};

template < typename T >
class tlm_blocking_put_if : public virtual sc_core::sc_interface
{
public:
   virtual void put( const T &t ) = 0;
};

// Uni-directional non blocking interfaces
template < typename T >
class tlm_nonblocking_get_if : public virtual sc_core::sc_interface
{
public:
   virtual bool nb_get( T &t ) = 0;
```

```
    virtual bool nb_can_get( tlm_tag<T> *t = 0 ) const = 0;
    virtual const sc_core::sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_nonblocking_put_if : public virtual sc_core::sc_interface
{
public:
    virtual bool nb_put( const T &t ) = 0;
    virtual bool nb_can_put( tlm_tag<T> *t = 0 ) const = 0;
    virtual const sc_core::sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const = 0;
};

// Combined uni-directional blocking and non blocking
template < typename T >
class tlm_get_if :
    public virtual tlm_blocking_get_if< T > ,
    public virtual tlm_nonblocking_get_if< T > {};

template < typename T >
class tlm_put_if :
    public virtual tlm_blocking_put_if< T > ,
    public virtual tlm_nonblocking_put_if< T > {};

// Peek interfaces template < typename T >
class tlm_blocking_peek_if : public virtual sc_core::sc_interface
{
public:
    virtual T peek( tlm_tag<T> *t = 0 ) const = 0;
    virtual void peek( T &t ) const { t = peek(); }
};

template < typename T >
class tlm_nonblocking_peek_if : public virtual sc_core::sc_interface
{
    public: virtual bool nb_peek( T &t ) const = 0;
    virtual bool nb_can_peek( tlm_tag<T> *t = 0 ) const = 0;
    virtual const sc_core::sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
```

```
class tlm_peek_if :
    public virtual tlm_blocking_peek_if< T > ,
    public virtual tlm_nonblocking_peek_if< T > {};

// Get_peek interfaces template < typename T >
class tlm_blocking_get_peek_if :
    public virtual tlm_blocking_get_if<T> ,
    public virtual tlm_blocking_peek_if<T> {};

template < typename T >
    class tlm_nonblocking_get_peek_if :
    public virtual tlm_nonblocking_get_if<T> ,
    public virtual tlm_nonblocking_peek_if<T> {};

template < typename T >
class tlm_get_peek_if :
    public virtual tlm_get_if<T> ,
    public virtual tlm_peek_if<T> ,
    public virtual tlm_blocking_get_peek_if<T> ,
    public virtual tlm_nonblocking_get_peek_if<T> {};
} // namespace tlm
```

13.2 TLM-1 fifoインタフェース

```
namespace tlm {
    // Fifo debug interface
    template< typename T >
    class tlm_fifo_debug_if : public virtual sc_core::sc_interface
    {
    public:
        virtual int used() const = 0;
        virtual int size() const = 0;
        virtual void debug() const = 0;

    // non blocking peek and poke - no notification. n is index of data :
    // 0 <= n < size(), where 0 is most recently written, and size() – 1 is oldest ie the one about to be read.
        virtual bool nb_peek( T & , int n ) const = 0;
        virtual bool nb_poke( const T & , int n = 0 ) = 0;
    };

    // Fifo interfaces
    template < typename T >
```

```
    class tlm_fifo_put_if :
        public virtual tlm_put_if<T> ,
        public virtual tlm_fifo_debug_if<T> {};

    template < typename T >
    class tlm_fifo_get_if :
        public virtual tlm_get_peek_if<T> ,
        public virtual tlm_fifo_debug_if<T> {};
} // namespace tlm
```

13.3 tlm_fifo

```
namespace tlm {
    template <typename T>
    class tlm_fifo :
        public virtual tlm_fifo_get_if<T>,
        public virtual tlm_fifo_put_if<T>,
        public sc_core::sc_prim_channel
    {
    public:
        explicit tlm_fifo( int size_ = 1 );
        explicit tlm_fifo( const char* name_, int size_ = 1 );
        virtual ~tlm_fifo();

        T get( tlm_tag<T> *t = 0 );
        bool nb_get( T& );
        bool nb_can_get( tlm_tag<T> *t = 0 )
        const; const sc_core::sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const;

        T peek( tlm_tag<T> *t = 0 ) const;
        bool nb_peek( T& ) const;
        bool nb_can_peek( tlm_tag<T> *t = 0 ) const;
        const sc_core::sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const;

        void put( const T& );
        bool nb_put( const T& );
        bool nb_can_put( tlm_tag<T> *t = 0 ) const;
        const sc_core::sc_event& ok_to_put( tlm_tag<T> *t = 0 ) const;

        void nb_expand( unsigned int n = 1 );
        void nb_unbound( unsigned int n = 16 );
        bool nb_reduce( unsigned int n = 1 );
```

```
    bool nb_bound( unsigned int n );
    bool nb_peek( T & , int n ) const;
    bool nb_poke( const T & , int n = 0 );

    int used() const;
    int size() const;
    void debug() const;

    static const char* const kind_string;
    const char* kind() const;
};
```

13.4 アナリシス・インタフェースとアナリシス・ポート

アナリシス・ポートは、解析、、機能の正確性をチェックするとか機能カバレージ情報を取得するなど意味あるタスクのため、複数のコンポーネントにトランザクションを配布することをサポートするのも目的としている。アナリシス・ポートのもっとも大切な点は、一つのアナリシス・ポートが複数のチャネル（サブスクライバー）にバインド可能であることである。アナリシス・ポートそれ自身は、インタフェース・メソッド write 呼び出しを各サブスクライバーに対して複製する。アナリシス・ポートは 0 個以上のサブスクライバーに対してもしくは他の複数のアナリシス・ポート・バインドすることができ、また、バインドしなくても構わない。

それぞれのサブスクライバーは、tlm_analysis_if の write()メソッドを実装する。このメソッドはトランザクションへの const リファレンスを渡し、サブスクライバーはただちに処理を実行することが許される。そうでなければ、サブスクライバーがトランザクションのライフタイムを延長しておきたいような場合には、自分でトランザクション・オブジェクトに対する deep copy を行う義務がある。その時点でそのサブスクライバーは実質上新しいトランザクションのイニシエータとなり、コピーのメモリ管理の責任を持つことになる。

アナリシス・ポートはモデルの動作そのものに関わるようなことに対して用いるべきではなく、解析のためデータを除いたり脇に渡したりするためにのみ使う。 tlm_analysis_if インタフェースは、tlm_write_if を継承したクラスである。この tlm_write_if は、アナリシスの目的だけではなく、他の目的にも使用が許されている。例えば、「12.3 ペイロード・イベント・キュー」を参照のこと。

tlm_analysis_fifo は、トランザクションを fifo へ書き込む tlm_analysis_if を実装した無限サイズの tlm_fifo である。また、tlm_fifo は tlm_analysis_triple をサポートする。これはトランザクション、開始時間、終了時間から成る。

13.4.1 クラス定義

```
namespace tlm {
  // Write interface
  template <typename T>
  class tlm_write_if : public virtual sc_core::sc_interface {
  public:
    virtual void write( const T& ) = 0;
```

```
};

template <typename T>
class tlm_delayed_write_if : public virtual sc_core::sc_interface {
public:
   virtual void write( const T& , const sc_core::sc_time& ) = 0;
};

// Analysis interface
template < typename T >
class tlm_analysis_if : public virtual tlm_write_if<T>
{
};

template < typename T >
class tlm_delayed_analysis_if : public virtual tlm_delayed_write_if<T>
{
};

// Analysis port
template < typename T>
class tlm_analysis_port : public sc_core::sc_object , public virtual tlm_analysis_if< T >
{
public:
   tlm_analysis_port();
   tlm_analysis_port( const char * );
   // bind and () work for both interfaces and analysis ports, since analysis ports implement the analysis interface
   void bind( tlm_analysis_if<T> & );
   void operator() ( tlm_analysis_if<T> & );
   bool unbind( tlm_analysis_if<T> & );

   void write( const T & );
};

// Analysis triple
template< typename T>
struct tlm_analysis_triple {
   sc_core::sc_time start_time;
   T transaction;
   sc_core::sc_time end_time;
   // Constructors
```

```
    tlm_analysis_triple();
    tlm_analysis_triple( const tlm_analysis_triple &triple );
    tlm_analysis_triple( const T &t );

    operator T() { return transaction; }
    operator const T& () const { return transaction; }
  };

  // Analysis fifo - an unbounded tlm_fifo
  template< typename T >
  class tlm_analysis_fifo :
  public tlm_fifo< T > ,
  public virtual tlm_analysis_if< T > ,
  public virtual tlm_analysis_if< tlm_analysis_triple< T > > {
  public:
    tlm_analysis_fifo( const char *nm ) : tlm_fifo<T>( nm, -16 ) {}
    tlm_analysis_fifo() : tlm_fifo<T>( -16 ) {}
    void write( const tlm_analysis_triple<T> &t ) { nb_put( t ); }
    void write( const T &t ) { nb_put( t ); }
  };
} // namespace tlm
```

13.4.2 ルール

a) tlm_write_if と tlm_analysis_if（およびこれらの delayed 付き版）は、単方向、ネゴシエーション無し、ノンブロッキングのインタフェースであり、引数で渡されるトランザクションに対して即座に応答しなければならない。

b) tlm_analysis_port のコンストラクタは、ポートのインスタンス名の文字列を、モジュール階層のインスタンス名を設定するために、ベースクラスの sc_object のコンストラクタの引数として渡さなければならない。

c) bind メソッドは、引数とした渡されたサブスクライバーをアナリシス・ポートのインスタンスに登録しなけらばならない。その結果、write メソッドの呼び出しは登録されたサブスクライバーへ渡されなければならない。複数のサブスクライバーを単一のアナリシスポートインスタンスに登録することも許される。

d) operator() は、bind()メソッドと同一でなけらばならない。

e) サブスクライバーの数が 0 であっても構わない。その場合には、write()メソッドは伝達されない。

f) unbind()メソッドは、bind()の逆の効果を持たなければならない。つまり、引数として渡されたサブスクライバーはアナリシス・ポートのインスタンスのサブスクライバーのリストから削除されなければならない。

g) tlm_analysys_port の write()メソッドが呼ばれた場合、そのポートに登録されたすべてのサブ

スクライバーの write()メソッドを呼び出し、引数を const リファレンスとして渡さなければならない。

h) write()メソッドはノンブロッキングであり、その実装が wait()を呼んではいけない。

i) write()メソッドは、const リファレンスとして渡されたトランザクション・オブジェクトを変更してはいけない。また、トランザクション・オブジェクトに関連するすべてのデータ（汎用ペイロードのデータ a、バイト・イネーブル配列）を書き換えてはいけない。

j) サブスクライバーの write メソッドの実装が呼び出し側へ return する前にそのトランザクションに対しての処理を完了することができない場合には、そのサブスクライバーはトランザクション・オブジェクトの deep copy とそれに伴うメモリ管理の責任を負わなければならない。

k) tlm_analysys_fifo クラスのコンストラクタは、無限 tlm_fifo を作らなければならない。

l) tlm_analysys_fifo クラスの write()メソッドは tlm_fifo ベースクラスの nb_put()を同一の引数とともに呼び出さなければならない。

例

```
struct Trans                                          // Analysis transaction class
{
  int i;
};

struct Subscriber: sc_object, tlm::tlm_analysis_if<Trans>
{
  Subscriber(const char* n) : sc_object(n) {}
  virtual void write(const Trans& t)
  {
    cout << "Hello, got " << t.i << "¥n";             // Implementation of the write method
  }
};

SC_MODULE(Child)
{
  tlm::tlm_analysis_port<Trans> ap;
  SC_CTOR(Child) : ap("ap")
  {
    SC_THREAD(thread);
  }
  void thread()
  {
    Trans t = {999};
    ap.write(t);                                      // Interface method call to the write method of the analysis port
```

```
  }
};

SC_MODULE(Parent)
{
  tlm::tlm_analysis_port<Trans> ap;
  Child* child;
  SC_CTOR(Parent) : ap("ap")
  {
    child = new Child("child");
    child->ap.bind(ap);                    // Bind analysis port of child to analysis port of parent
  }
};

SC_MODULE(Top)
{
  Parent* parent;
  Subscriber* subscriber1;
  Subscriber* subscriber2;
  SC_CTOR(Top)
  {
    parent = new Parent("parent");
    subscriber1 = new Subscriber("subscriber1");
    subscriber2 = new Subscriber("subscriber2");
    parent->ap.bind( *subscriber1 );       // Bind analysis port to two separate subscribers
    parent->ap.bind( *subscriber2 );       // This is the key feature of analysis ports
  }
};
```

14. Glossary（用語集）

注：青字は SystemC LRM にて定義・使用されている用語

Words	Descriptions	訳語	説明
adaptor	A module that connects a transaction level interface to a pin level interface (in the general sense of the word interface) or that connects together two transaction level interfaces, often at different abstraction levels. An adapter may be used to convert between two sockets specialized with different protocol types. See *bridge, transactor*.	アダプタ	トランザクション・レベル・インタフェースとピン・レベル・インタフェース間、もしくは、2 つのトランザクション・レベル・インタフェース(しばしば異なる抽象度のインタフェース)を接続するモジュール。異なるプロトコル型の 2 つのソケットを変換するために用いられる。「ブリッジ」、「トランザクタ」を参照。
approximately timed (AT)	A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. See *cycle approximate*.	アプロキシメイトリー・タイムド	外部から観測可能な状態と対応した詳細リファレンス・モデルの状態に一対一対応が付き、その対応が状態遷移順序を保つ（正確なタイミングは保たない）ようなモデリング・スタイル。タイミング精度は定義されない。 「サイクル近似」を参照。
attribute (of a transaction)	Data that is part of and carried with the transaction and is implemented as a member of the transaction object. These may include attributes inherent in the bus or protocol being modeled, and attributes that are artefacts of the simulation model (a timestamp, for example).	アトリビュート	トランザクションの一部で、トランザクションで運ばれるもので、トランザクション・オブジェクトのメンバとして実装されるデータ。モデル化する BUS もしくはプロトコルが本来持つアトリビュート、タイムスタンプ等のシミュレーション・モデルのアトリビュートも含む。
automatic deletion	A generic payload extension marked for automatic deletion will be deleted at the end of the transaction lifetime, that is, when the transaction reference count reaches 0.	自動削除	自動削除マークされた汎用ペイロード拡張は、トランザクション・ライフタイムの終わり、すなわちトランザクション参照カウントが 0 に達すると、削除される。
backward path	The calling path by which a target or interconnect component makes interface method calls back in the direction of another interconnect component or the initiator.	バックワード・パス	ターゲット、インターコネクト・コンポーネントが、別のインターコネクト・コンポーネント、イニシエータの向きにインタフェース・メソッド・コールバックする際に使用するパス。

base protocol	A protocol traits class consisting of the generic payload and tlm_phase types, together with an associated set of protocol rules which together ensure maximal interoperability between transaction-level models	基本プロトコル	汎用ペイロードと tlm_phase 型からなるプロトコルのトレイツ（traits）クラスであり、トランザクション・レベル・モデルの間の最大相互利用性を保証するためのプロトコル規則を持つ。
bidirectional interface	A TLM-1 transaction level interface in which a pair of transaction objects, the request and the response, are passed in opposite directions, each being passed according to the rules of the unidirectional interface. For each transaction object, the transaction attributes are strictly readonly in the period between the first timing point and the end of the transaction lifetime.	双方向インタフェース	TLM-1 のトランザクション・レベル・インタフェース。トランザクション・オブジェクトのペア、リクエストとレスポンスが、それぞれ反対方向に、単方向インタフェースのルールに従って送信される。各トランザクション・オブジェクトのトランザクション・アトリビュートは最初のタイミング・ポイントとトランザクション・ライフタイムの終わりの間は厳密に読み出しのみ。（値を書き換えてはいけない。）
blocking	Permitted to call the wait method. A blocking function may consume simulation time or perform a context switch, and therefore shall not be called from a method process. A blocking interface defines only blocking functions.	ブロッキング	wait 関数を呼び出すことが許されている。ブロッキング関数はシミュレーション時間を消費する。また、コンテクスト・スイッチする。したがってメソッド・プロセスから呼び出してはいけない。ブロッキング・インタフェースはブロッキング関数のみ定義する。
blocking transport interface	A blocking interface of the TLM-2.0 standard which contains a single method **b_transport**. Beware that there still exists a blocking transport method named **transport**, part of TLM-1.	ブロッキング・トランスポート・インタフェース	TLM-2.0 標準におけるブロッキング・インフェースであり、ただ一つのメソッド b_transport からなる。TLM-1 標準の一部である transport という名のブロッキング・トランスポート・メソッドは依然として存在する事に注意されたい。
bridge	A component connecting two segments of a communication network together. A bus bridge is a device that connects two similar or dissimilar memory-mapped buses together. See *adapter, transaction bridge, transactor*.	ブリッジ	通信ネットワークの 2 つのセグメントを接続するコンポーネント。バス・ブリッジは類似のもしくは異なる 2 つのメモリ・マップド・バスを接続するデバイス。「アダプタ」、「トランサクション・ブリッジ」、「トランザクタ」を参照。

caller	In a function call, the sequence of statements from which the given function is called. The referent of the term may be a function, a process, or a module. This term is used in preference to *initiator* to refer to the caller of a function as opposed to the initiator of a transaction.	呼び出し元関数	関数呼び出しにおいて、ある関数を呼び出している一連のステートメントで、関数、プロセス、モジュール。この用語は、トランザクションのイニシエータとしてではなく、関数の呼び出し元としてのイニシエータを指すときに使われる。
callee	In a function call, the function that is called by the caller. This term is used in preference to *target* to refer to the function body as opposed to the target of a transaction.	呼び出し先関数	関数呼び出しにおいて、呼び出し元関数から呼び出される関数。この用語は、トランザクションのターゲットとしてではなく、呼び出される関数本体としてのターゲットを指すときに使われる。
channel	A class that implements one or more interfaces or an instance of such a class. A channel may be a hierarchical channel or a primitive channel or, if neither of these, it is strongly recommended that a channel at least be derived from class **sc_object**. Channels serve to encapsulate the definition of a communication mechanism or protocol. (SystemC term)	チャネル	1 つ以上のインタフェースを実装したクラス、もしくは、そのクラスのインスタンス。チャネルは階層チャネル、プリミティブ・チャネル、そうでなければ、sc_object クラスを継承したチャネルが強く推奨される。チャネルはコミュニケーション機構もしくはプロトコルをの定義を隠蔽するのに使われる。(SystemC 用語)
child	An instance that is within a given module. Module A is a *child* of module B if module A is *within* module B. (SystemC Term)	子	あるモジュールの中にインスタンスされたもの。モジュール B の中にモジュール A があるなら、モジュール A はモジュール B の子である。(SystemC 用語)
combined interfaces	Pre-defined groups of core interfaces used to parameterize the socket classes. There are four combined interfaces: the blocking and non-blocking forward and backward interfaces.	統合インタフェース	ソケットクラスをパラメタライズするためあらかじめ定義されたコア・インタフェースのグループ。次の 4 つの統合インタフェースがある: ブロッキング・フォワード、ブロッキング・バックワード、ノンブロッキング・フォワード、ノンブロッキング・バックワード・インタフェース。
component	An instance of a SystemC module. This standard recognizes three kinds of component; the initiator, interconnect component, and target.	コンポーネント	SystemC モジュールのインスタンス。この標準ではイニシエータ、インターコネクト・コンポーネント、ターゲットの 3 種類のコンポーネントがある。

convenience socket	A socket class, derived from **tlm_initiator_socket** or **tlm_target_socket**, that implements some additional functionality and is provided for convenience. Several convenience sockets are provided as utilities.	便利ソケット	tlm_initiator_socket か tlm_target_socket から派生したソケットクラスであり、何らかの追加機能を実装して、利便性のため提供される。ユーティリティとして数個の便利ソケットが提供される。
core interface	One of the specific transaction level interfaces defined in this standard, including the blocking and non-blocking transport interface, the direct memory interface, and the debug transport interface. Each core interface is an *interface proper*. The core interfaces are distinct from the generic payload API.	コア･インタフェース	この標準で定義されるトランザクション・レベル・インタフェースの一つで、ブロッキングとノンブロッキング・トランスポート・インターフェース、ダイレクト・メモリ、インタフェース、デバッグ・トランスポート・インタフェースを含む。各コア・インタフェースはインタフェース・プロパである。コア・インタフェースは汎用ペイロード API とは別物である。
cycle accurate	A modeling style in which it is possible to predict the state of the model in any given cycle at the external boundary of the model and thus to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model in each cycle, but which is not required to explicitly reevaluate the state of the entire model in every cycle or to explicitly represent the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles.	サイクル精度	モデルの外部境界で、あるサイクルでのモデルの状態の予測ができる、従って、モデルの状態とそれに対応する RTL モデルの外部観測可能な状態との対応付けが毎サイクルできるモデリング・スタイル。ただし、毎サイクルでモデル全体の状態の再評価、または、全ての外部ピン、内部レジスタの状態を表すことは要求されない。この用語は、サイクルの概念を持つモデルにのみ使われる。
cycle approximate	A model for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding cycle accurate model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. This term is only applicable to models that have a notion of cycles.	サイクル近似	モデルの外部観測可能な状態とそのモデルに対応するサイクル精度モデルの状態間に一対一の対応付けが可能なモデル。その対応付けは、状態遷移を保持するが、正確なタイミングは保持しない。タイミングの精度は定義されない。この用語は、サイクルの概念を持つモデルにのみ使われる。

cycle count accurate, cycle count accurate at transaction boundaries	A modeling style in which it is possible to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model as sampled at the timing points marking the boundaries of a transaction. A cycle count accurate model is not required to be cycle accurate in every cycle, but is required to accurately predict both the functional state and the number of cycles at certain key timing points as defined by the boundaries of the transactions through which the model communicates with other models.	サイクル数精度、トランザクション境界でのサイクル数精度	モデルの状態とそのモデルに対応するRTLモデルの外部観測可能な状態との間で、トランザクション境界を表すタイミング・ポイントで状態をサンプルすることで一対一の対応付けが可能なモデリング・スタイル。サイクル数精度モデルは全てのサイクルでサイクル精度である必要はないが、モデル間のコミュニケーションに使われるトランザクションの境界で定義されるキーとなるタイミング・ポイントで機能的状態とサイクル数が正確に予測されることが求められる。
declaration	A C++ language construct that introduces a name into a C++ program and specifies how the C++ compiler is to interpret that name. Not all declarations are definitions. For example, a class declaration specifies the name of the class but not the class members, while a function declaration specifies the function parameters but not the function body. (See *definition*.) (C++ term)	宣言	C++プログラムにある名前を導入し、C++コンパイラがその名前をいかに解釈すべきかを指定する C++の構成概念。宣言は定義とは限らない。例えば、クラス宣言はクラス名を規定するが、クラス・メンバーは規定しない。一方、関数宣言は、関数のパラメタを規定するが、関数本体は規定しない。(「定義」を参照) (C++用語)
definition	The complete specification of a variable, function, type, or template. For example, a class definition specifies the class name and the class members, and a function definition specifies the function parameters and the function body. (See *declaration*.) (C++ term)	定義	変数、関数、型、テンプレートの完全な規定。例えば、クラス定義はクラス名とクラス・メンバーを規定する。関数定義は、関数のパラメタと関数本体を規定する。(「宣言」参照) (C++用語)
effective local time	The current time within a temporally decoupled initiator. effective_local_time = sc_time_stamp() + local_time_offset	実効ローカル・タイム	テンポラル・デカップリングでのイニシエータの現在の時間 effective_local_time = sc_time_stamp() + local_time_offset
exclusion rule	A rule of the base protocol that prevents a request or a response being sent through a socket if there is already a request or a response (respectively) in progress through that socket. The base protocol has two exclusion rules, the request exclusion rule and the response exclusion rule, which act independently of one another.	排他規則	既にソケット上にリクエストあるいはレスポンスがある場合、そのソケットでリクエストあるいはレスポンスが送信されないようにする基本プロトコルの規則。基本プロトコルにはリクエストの排他規則とレスポンスの排他規則の 2 つの排他規則があり、それらは互いに独立に動作する。

extension	A user-defined object added to and carried around with a generic payload transaction object, or a user-defined class that extends the set of values that are assignment compatible with the tlm_phase type. An ignorable extension may be used with the base protocol, but a non-ignorable or mandatory extension requires the definition of a new protocol traits class.	拡張	汎用ペイロード・トランザクション・オブジェクトに付け加えられ、かつ一緒に転送されるユーザ定義オブジェクト、あるいは tlm_phase 型に代入コンパチブルな値のセットを拡張するユーザ定義クラス。無視可能拡張は基本プロトコルと一緒に使用してもよいが、無視できない拡張、必須拡張は新しいプロトコルのトレイツ（traits）クラスの定義を必要とする。
forward path	The calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target.	フォワード･パス	イニシエータかインターコネクト・コンポーネントが別のインターコネクト・コンポーネントかターゲットの向きにインタフェース・メソッド・コールする際に使用するパス。
generic payload	A specific set of transaction attributes and their semantics together defining a transaction protocol which may be used to achieve a degree of interoperability between loosely timed and approximately timed models for components communicating over a memory-mapped bus. The same transaction class is used for all modeling styles.	汎用ペイロード	トランザクション・アトリビュートのセットとそれらのセマンティクスで、メモリ･マップド･バスを介して通信するコンポーネントのルーズリー･タイムド、アプロキシメイトリー・タイムド・モデル間のある程度の相互利用を達成するために使われるトランザクション・プロトコルを定義する。同じトランザクション・クラスが全てのモデリング・スタイルで使われる。
global quantum	The default time quantum used by every quantum keeper and temporally decoupled initiator. The intent is that all temporally decoupled initiators should typically synchronize on integer multiples of the global quantum, or more frequently on demand.	グローバル･クォンタム	すべてのクォンタム・キーパーとテンポラル・デカップリングされたイニシエータが使用する、デフォルトのタイム・クォンタム。 その意味は、すべてのテンポラル・デカップリングされたイニシエータが通常はグローバル・クォンタムの整数倍、または、要求に応じてより頻繁に同期を取るべきであるということ。
hierarchical binding	Binding a socket on a child module to a socket on a parent module, or a socket on a parent module to a socket on a child module, passing transactions up or down the module hierarchy.	階層バインディング	子モジュールのソケットを親モジュールのソケットへバインドする、あるいは親モジュールのソケットを子モジュールのソケットにバインドする事。モジュール階層をまたがってトランザクションを受け渡す事ができる。

hop	The interface method call path between two adjacent components en route from initiator to target. A hop consists of one initiator socket bound to one target socket. In order to be transported from initiator to target, a transaction may need to pass over multiple hops. The number of hops between an initiator and a target is always one greater than the number of interconnect components.	ホップ	イニシエータからターゲットへの経路の途中にある 2 つの隣接するコンポーネント間のインタフェース・メソッド・コールのパス。ホップは 1 つのターゲット・ソケットにバインドされた 1 つのイニシエータ・ソケットからなる。イニシエータからターゲットに転送するためにトランザクションは複数のホップの通過が必要となる場合がある。イニシエータとターゲット間のホップの数はインターコンポーネントの数より常に 1 つ多い。
ignorable extension	A generic payload extension that may be ignored by any component other than the component that set the extension. An ignorable extension is not required to be present. Ignorable extensions are permitted by the base protocol.	無視可能な拡張	拡張を設定したコンポーネント以外のコンポーネントからは無視される汎用ペイロード拡張。無視可能な拡張は必須ではない。無視可能な拡張は基本プロトコルで許されている。
ignorable phase	A phase, created by the macro DECLARE_EXTENDED PHASE, that may be ignored by any component that receives the phase and that cannot demand a response of any kind. Ignorable phases are permitted by the base protocol.	無視可能なフェーズ	DECLARE_EXTENDED PHASE マクロによって作られるフェーズで、フェーズを受け取る側のコンポーネントから無視され、、いかなるレスポンスも要求できない。無視可能なフェーズは基本プロトコルで許されている。
initiator	A module that can initiate transactions. The initiator is responsible for initializing the state of the transaction object, and for deleting or reusing the transaction object at the end of the transaction‘s lifetime. An initiator is usually a master and a master is usually an initiator, but the term *initiator* means that a component can initiate transactions, whereas the term *master* means that a component can take control of a bus. In the case of the TLM-1 interfaces, the term *initiator* as defined here may not be strictly applicable, so the terms *caller* and *callee* may be used instead for clarity.	イニシエータ	トランザクションを開始できるモジュール。イニシエータは、トランザクション・オブジェクトの状態を初期化すること、トランザクション・ライフタイムの最後でトランザクション・オブジェクトを削除するか、または再利用する責任がある。イニシエータは通常マスタであり、マスタは通常イニシエータである。ただし、イニシエータと言う用語はトランザクションを開始することのできるコンポーネントを意味し、マスタと言う用語はバスの制御を取ることのできるコンポーネントを意味する。TLM-1 インタフェースにおいては、イニシエータと言う用語は厳密には適切でないため、代わりに呼び出し元関数、呼び出し先関数といった用語を使って明確にするのがよい。

initiator socket	A class containing a port for interface method calls on the forward path and an export for interface method calls on the backward path. A socket overloads the SystemC binding operators to bind both port and export	イニシエータ・ソケット	フォワード・パス上のインタフェース・メソッド・コールのためのポートとバックワード・パス上のインタフェース・メソッド・コールのためのエクスポートからなるクラス。このソケットはポートとエクスポートの両方をバインドするために SystemC のバインド演算子をオーバーロードする。
interconnect component	A module that accesses a transaction object, but does act as an initiator or a target with respect to that transaction. An interconnect component may or may not be permitted to modify the attributes of the transaction object, depending on the rules of the payload. An arbiter or a router would typically be modeled as an interconnect component, the alternative being to model it as a target for one transaction and an initiator for a separate transaction.	インターコネクト・コンポーネント	トランザクション・オブジェクトにアクセスするが、そのトランザクションに関してイニシエータあるいはターゲットとして動作するモジュール。インターコネクト・コンポーネントは、ペイロードのルールに従って、トランザクション・オブジェクトのアトリビュートを変更することを許可または禁止される。アービタとラウターは典型的にインターコネクト・コンポーネントとしてモデル化される。他の方法は、あるトランザクションに対してはターゲット、別のトランザクションに対してはイニシエータとしてモデル化することである。
interface	A class derived from class **sc_interface**. An interface proper is an interface, and in the object-oriented sense a channel is also an interface. However, a channel is not an interface proper. (SystemC term)	インタフェース	sc_interface を継承したクラス。インタフェース・プロパはインタフェース。オブジェクト指向の意味では、チャネルもインタフェース。ただし、チャネルはインタフェース・プロパではない。（SystemC 用語）
Interface Method Call (IMC)	A call to an interface method. An interface method is a member function declared within an interface. The IMC paradigm provides a level of indirection between a method call and the implementation of the method within a channel such that one channel can be substituted with another without affecting the caller. (SystemC term)	インタフェース・メソッド・コール	インタフェース・メソッドの呼び出し。インタフェース・メソッドとはインタフェース内で宣言されたメンバ関数。IMC の枠組みは、メソッド呼び出しとメソッドの実装の分離を提供する。例えば、呼び出し元に影響を与えることなく、チャネルの置き換えが可能となる。（SystemC 用語）

interface proper	An abstract class derived from class **sc_interface** but not derived from class sc_object. An interface proper declares the set of methods to be implemented within a channel and to be called through a port. An interface proper contains pure virtual function declarations, but typically contains no function definitions and no data members. (SystemC term)	インタフェース・プロパ	sc_interface を継承するが sc_object は継承しない抽象クラス。インタフェース・プロパはチャネル内で実装されるメソッドを宣言し、それらはポートを介して呼び出される。インタフェース・プロパは純粋バーチャル関数の宣言を持つが、通常は、関数定義、データ・メンバは持たない。（SystemC 用語）
interoperability	The ability of two or more transaction level models from diverse sources to exchange information using the interfaces defined in this standard. The intent is that models that implement common memory-mapped bus protocols in the programmers view use case should be interoperable without the need for explicit adaptors. Furthermore, the intent is to reduce the amount of engineering effort needed to achieve interoperability for models of divergent protocols or use cases, although it is expected that adaptors will be required in general.	相互利用性	この標準で定義するインタフェースを使うことにより、異なった出所のトランザクション・レベル・モデルで情報を交換できるようになること。この意味は、PV ユースケースで共通のメモリ・マップド・バスプロトコルを実装したモデルはアダプタ無しで相互利用できるはずだということ。さらに、異なったプロトコル、ユースケースのモデルについても、一般にはアダプタが要求されるが、相互利用のための開発作業の削減を意味する。
interoperability layer	The subset of classes in this standard that are necessary for interoperability. The interoperability layer comprises the TLM-2.0 core interfaces, the initiator and target sockets, the generic payload, **tlm_global_quantum** and **tlm_phase**. Closely related to the base protocol.	インタオペラビリティ・レイヤ	相互利用のために必要なこの標準のクラスのサブセット。インタオペラビリティ・レイヤは TLM-2.0 のコア・インタフェース、イニシエータ・ソケットとターゲット・ソケット、汎用ペイロード、tlm_global_quantum と tlm_phase からなる。 基本プロトコルと密接に関係する。
lifetime (of an object)	The lifetime of an object starts when storage is allocated and the constructor call has completed, if any. The lifetime of an object ends when storage is released or immediately before the destructor is called, if any. (C++ term)	（オブジェクトの）ライフタイム	オブジェクトのライフタイムは、記憶領域が割り当てられ、コンストラクタがあれば、コンストラクタの呼び出しが完了した時に開始され、割り当てられて記憶領域が開放された時、、もしくはデストラクタがあれば、デストラクタが呼び出される直前に終了。する（C++用語）

lifetime (of a transaction)	The period of time that starts when the transaction becomes valid and ends when the transaction becomes invalid. Because it is possible to pool or re-use transaction objects, the lifetime of a transaction object may be longer than the lifetime of the corresponding transaction. For example, a transaction object could be a stack variable passed as an argument to multiple put calls of the TLM-1 interface.	（トランザクションの）ライフタイム	トランザクションが有効になった時に始まり、無効になったときに終了する時間間隔。トランザクション・オブジェクトはプールまたは再利用される可能性があるため、そのトランザクション・オブジェクトのライフタイムは、対応するトランザクション自身のライフタイムより長い場合がある。例えば、トランザクション・オブジェクトは、TLM-1 インタフェースの複数呼び出し時の引数として渡されるスタック変数であることもある。
local quantum	The amount of simulation time remaining before the initiator is required to synchronize. Typically, the local quantum equals the current simulation time subtracted from the next largest integer multiple of the global quantum, but this calculation can be overridden for a given quantum keeper.	ローカル・クォンタム	イニシエータが同期を要求されるまでに残っているシミュレーション時間の量。通常、ローカル・クォンタムはグローバル・クォンタムの次の最も大きい整数倍から現在のシミュレーション時刻を差し引いたものと等しいが、クォンタム・キーパーを与えることで、この計算をオーバーライドすることができる。
local time offset	Time as measured relative to the most recent quantum boundary in a temporally decoupled initiator. The timing annotation arguments to the **b_transport** and *nb_transport* methods are local time offsets. effective_local_time = sc_time_stamp() + local_time_offset	ローカル・タイム・オフセット	テンポラル・デカップリングされたイニシエータにおいて、最新のクォンタム境界（時刻）から経過した時間。 b_transport 及び nb_transport メソッドのタイミング・アノテーションの引数はローカル・タイム・オフセットである。 effective_local_time = sc_time_stamp() + local_time_offset

loosely timed	A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely timed model may include timer models and a notional arbitration interval or execution slot length. Some users adopt the practice of inserting random delays into loosely timed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.	ルーズリー・タイムド	オペレーティング・システムのブートと複数スレッド間の明示的な同期管理が無い場合に、複数スレッドを管理するために必要とされる機能をサポートするのに十分なだけのタイミング情報を表現するモデリング・スタイル。ルーズリー・タイムド・モデルには、タイマー・モジュールと概念的な調停間隔や、実行スロット長が含まれる場合がある。ユーザによっては、それらプロトコルの頑健性をテストするにあたり、ルーズリー・タイムドの記述にランダム遅延を入れる場合があるが、これによってモデリング・スタイルの基本的な性質を変える事はない。
master	This term has no precise technical definition in this standard, but is used to mean a module or port that can take control of a memory-mapped bus in order to initiate bus traffic, or a component that can execute an autonomous software thread and thus initiate other system activity. Generally, a bus master would be an initiator.	マスタ	この標準では、この用語には正確な技術定義は無いが、バス・トラフィックを開始する為にメモリ・マップド・バスを制御できるモジュール、ポート、また自律的なソフトウェア・スレッドを実行するもしくは他のシステム動作を起動するコンポーネントを指すのに使われる。一般的には、バス・マスタはイニシエータである。
memory manager	A user-defined class that performs memory management for a generic payload transaction object. A memory manager must provide a **free** method, called when the reference count of the transaction reaches 0.	メモリ・マネージャ	汎用ペイロードのトランザクション・オブジェクトについてメモリ管理をするユーザ定義のクラス。メモリ・マネージャはトランザクションの参照カウントが 0 になった時に呼ばれる解放メソッドを提供しなければならない。
method	A function that implements the behavior of a class. This term is synonymous with the C++ term *member function*. In SystemC, the term *method* is used in the context of an *interface method call*. Throughout this standard, the term *member function* is used when defining C++ classes (for conformance to the C++ standard), and the term *method* is used in more informal contexts and when discussing interface method calls. (SystemC term)	メソッド	クラスの動作を実装した関数。この用語は C++用語のメンバ関数と同意義。SystemC でメソッドとは、インタフェース・メソッド・コールの文脈の中で用いられる。この標準を通じて、メンバ関数は C++クラスを定義する時に使用され（C++標準に一致させるため）メソッドはより略式的な文脈か、またはインタフェース・メソッド・コールを論じている時に使用される。（SystemC 用語）

multi-socket	One of a family of convenience sockets that can be bound to multiple sockets belonging to other components. An initiator multi-socket can be bound to more than one target socket, and more than one initiator socket can be bound to a single multi-target socket. When calling interface methods through multi-sockets, the destinations are distinguished using the subscript operator.	マルチ・ソケット	便利ソケットの一つで、他のコンポーネントに属する複数のソケットにバインドすることができる。イニシエータ・マルチ・ソケットに一つ以上のターゲット・ソケットをバインドすることができ、また、単一のマルチ・ターゲット・ソケットに一つ以上のイニシエータ・ソケットをバインドすることができる。マルチ・ソケットを通してインタフェース・メソッドを呼び出す際、ディスティネーションは、添字演算子を使用して識別される。
nb_transport	The **nb_transport_fw** and **nb_transport_bw** methods. In this document, the italicized term *nb_transport* is used to describe both methods in situations where there is no need to distinguish between them.	nb_transport	nb_transport_fw と nb_transport_bw メソッド。本書では、nb_transport というイタリック体で示した用語は、それらを区別する必要がない状況にいて両方のメソッドを指すのに使用されている。
non-blocking	Not permitted to call the wait method. A non-blocking function is guaranteed to return without consuming simulation time or performing a context switch, and therefore may be called from a thread process or from a method process. A non-blocking interface defines only non-blocking functions.	ノンブロッキング	wait の呼び出しが許されない。ノンブロッキング関数はシミュレーション時間の消費、コンテクスト・スイッチ無しでリターンすることが保証されている。そのため、ノンブロッキング関数はスレッド・プロセスもしくはメソッド・プロセスから呼び出される。ノンブロッキング・インタフェースはノンブロッキング関数のみ定義する。
non-blocking transport interface	A non-blocking interface of the TLM-2.0 standard. There a two such interfaces, containing methods named **nb_transport_fw** and **nb_transport_bw**.	ノンブロッキング・トランスポート・インタフェース	TLM-2.0 標準におけるノンブロッキング・インタフェース。nb_transport_fw と nb_transport_bw と命名されたメソッドを含んだ、2 種類のインタフェースがある。
object	A region of storage. Every object has a type and a lifetime. An object created by a definition has a name, whereas an object created by a new expression is anonymous. (C++ term)	オブジェクト	ある記憶領域。あらゆるオブジェクトは型とライフタイムを持つ。定義によって作成されたオブジェクトは名前を持つが、new 構文によって作成されたオブジェクトは匿名である。(C++用語)

opposite path	The path in the opposite direction to a given path. For the forward path, the opposite path is the forward return path or the backward path. For the backward path, the opposite path is the forward path or the backward return path.	逆方向パス	逆方向のパス。 フォワード・パスでは逆方向パスはフォワードのリターン・パスあるいはバックワード・パスである。バックワード・パスではフォワード・パスあるいはバックワードのリターン・パスである。
parent	The inverse relationship to *child*. Module A is the *parent* of module B if module B is a *child* of module A. (SystemC term)	親	子とは逆の関係。もしモジュールBがモジュールAの子であるならば、A は B の親である。（SystemC 用語）
payload event queue (PEQ)	A class that maintains a queue of SystemC event notifications, where each notification carries an associated transaction object. Transactions are put into the queue annotated with a delay, and each transaction pops out of the back of queue at the time it was put in plus the given delay. Useful when combining the non-blocking interface with the approximately-timed coding style.	ペイロード･イベント･キュー	SystemC イベント通知のキューを保持するクラスであり、各通知は関連トランザクション・オブジェクトを持つ。トランザクションは遅延時間といっしょにキューに積まれる。そして、各トランザクションはそれらが積まれた時刻＋与えられた遅延時間後にキューから取り出される。アプロキシメイトリー・タイムド・コーディング・スタイルとノンブロッキング・インタフェースを組み合わせるときに有用である。
phase	A period in the lifetime of a transaction. The phase is passed as an argument to the non-blocking transport method. Each phase transition is associated with a timing point. The timing point may be delayed by an amount given by the time argument to *nb_transport*.	フェーズ	トランザクションのライフタイムのある期間。 フェーズはノンブロッキング・トランスポート・メソッドの引数として渡される。 それぞれのフェーズ・トランジッションはタイミング・ポイントと関係する。 タイミング・ポイントは nb_transport の時間引数分遅れる。
phase transition	A change to the value of the phase argument of the non-blocking transport method as marked by each call to *nb_transport* and each return from *nb_transport* with a value of TLM_UPDATED.	フェーズ･トランジション	ノンブロッキング・トランスポート・メソッドのフェーズ引数の値の変化であり、nb_transport 呼び出しとして示され、nb_transport からのリターンは TLM_UPDATED。

programmers view (PV)	The use case of the software programmer who requires a functionally accurate, loosely timed model of the hardware platform for booting an operating system and running application software.	プログラマーズ・ビュー（PV）	ソフトウェア・プログラマのユースケースで、オペレーティング・システムのブートとアプリケーション・ソフトウェアの実行のため、ハードウェア・プロットフォームの機能の正確な、ルーズリー・タイムドのモデルが要求される。
protocol traits class	A class containing a **typedef** for the type of the transaction object and the phase type, which is used to parameterize the combined interfaces, and effectively defines a unique type for a protocol.	プロトコル･トレイツ･クラス	トランザクション・オブジェクトとフェーズ型の typedef を含むクラス。統合したインタフェースをパラメタライズするのに使用され、また、プロトコルのためにユニークな型を効率良く定義することができる。
quantum	In temporal decoupling, the amount a process is permitted to run ahead of the current simulation time.	クォンタム	テンポラル・デカップリングにおいて、現在のシミュレーション時刻より先行して実行が許される時間量。
quantum keeper	A utility class used to store the local time offset from the current simulation time, which it checks against a local quantum.	クォンタム･キーパー	現在のシミュレーション時刻からのローカル時間へのオフセットを保持するためのユーティリティ・クラス。その値は、ローカル・クォンタムに対してチェックされる。
request	For the base protocol, the stage during the lifetime of a transaction when information is passed from the initiator to the target. In effect, the request transports generic payload attributes from the initiator to the target, including the command, the address, and for a write command, the data array. (The transaction is actually passed by reference and the data array by pointer.)	リクエスト	基本プロトコルにおいて、イニシエータからターゲットへ情報が送られる、トランザクション・ライフタイムのステージ。実質的には、リクエストはコマンド、アドレス、ライト・コマンド時のデータ配列などの汎用ペイロードのアトリビュートをイニシエータからターゲットへ転送する。(トランザクションは実際には参照渡し、データ配列はポインタ渡しにより行われる。)

response	For the base protocol, the stage during the lifetime of a transaction when information is passed from the target back to the initiator. In effect, the response transports generic payload attributes from the target back to the initiator, including the response status, and for a read command, the data array. (The transaction is actually passed by reference and the data array by pointer.)	レスポンス	基本プロトコルにおいて、ターゲットからイニシエータへ情報が送られる、トランザクション・ライフタイムのステージ。実質的には、レスポンスはステータス、リード・コマンド時のデータ配列などの汎用ペイロードのアトリビュートをターゲットからイニシエータへ返送する。(トランザクションは実際には参照渡し、データ配列はポインタ渡しにより行われる。)
return path	The control path by which the call stack of a set of interface method calls is unwound along either the forward path or the backward path. The return path for the forward path can carry information from target to initiator, and the return path for the backward path can carry information from initiator to target.	リターン・パス	インタフェース・メソッド・コールの呼び出しスタックが、フォワード・パスもしくはバックワード・パスを戻るために呼ぶコントロールパス。フォワード・パスのリターン・パスはターゲットからイニシエータへ情報を運び、バックワード・パスのリターン・パスはイニシエータからターゲットまで情報を運ぶこと。
simple socket	One of a family of convenience sockets that are simple to use because they allows callback methods to be registered directly with the socket object rather than the socket having to be bound to another object that implements the required interfaces. The simple target socket avoids the need for a target to implement both blocking and non-blocking transport interfaces by providing automatic conversion between the two.	シンプル・ソケット	便利ソケットの一つで、ソケットを必要なインタフェースを実装する別のオブジェクトへバインドする代わりにコールバック・メソッドを直接にソケット・オブジェクトに登録することを許すので、使い方が単純。シンプル・ターゲット・ソケットは、ブロッキングとノンブロッキング・トランスポート・インタフェース間の自動変換を提供することによって両方のトランスポート・インタフェースをターゲットに実装する必要性をなくしている。
slave	This term has no precise technical definition in this standard, but is used to mean a reactive module or port on a memory-mapped bus that is able to respond to commands from bus masters, but is not able itself to initiate bus traffic. Generally, a slave would be modeled as a target.	スレーブ	この標準では、この用語に正確な技術定義はないが、バス・マスタからのコマンドに応答する事が出来るが、バス・トラフィックを起こすことができないメモリ・マップド・バス上のモジュールまたはポートを指すのに使われる。一般的にスレーブは、ターゲットとしてモデリングされる。

socket	See initiator socket and target socket	ソケット	「イニシエータ・ソケット」と「ターゲット・ソケット」を参照のこと。
standard error response	The behavior prescribed by this standard for a generic payload target that is unable to execute a transaction successfully. A target should either a) execute the transaction successfully or b) set the response status attribute to an error response or c) call the SystemC report handler.	標準エラー応答	汎用ペイロードのターゲットがトランザクションをうまく完了できない時の、この標準において定義された動作。ターゲットは、次のいずれかを行う。 a) トランザクションを完了させる。 b) レスポンス・ステータス・アトリビュートに、エラーレスポンスをセットする。 c) SystemC のリポートハンドラを呼ぶ。
sticky extension	A generic payload extension object that is not deleted (either automatically or explicitly) at the end of life of the transaction object, and thus remains with the transaction object when it is pooled. Sticky extensions are not deleted by the memory manager.	スティッキー拡張	トランザクション・オブジェクトのライフの終了時に（自動的または明示的にも）削除されない汎用ペイロードの拡張オブジェクト。また、プールされたときトランザクション・オブジェクトと共に残される。スティッキー拡張はメモリ・マネージャによって削除されない。
synchronize	To yield such that other processes may run, or when using temporal decoupling, to yield and wait until the end of the current time quantum.	同期	他のプロセスを動作させるために実行を譲ること。あるいは、テンポラル・デカップリングを使用している際、現在のタイム・クォンタムの終わりまで実行を譲るか待つこと。
synchronization -on-demand	The action of a temporally decoupled process when it yields control back to the SystemC scheduler so that simulation time may advance and other processes run in addition to the synchronization points that may occur routinely at the end of each quantum.	オンデマンド同期	テンポラル・デカップリングされたプロセスが SystemC スケジューラへコントロールを戻す動作であり、これにより、各クォンタムの終わりに生じる同期ポイントに加え、シミュレーション時間は進み、他のプロセスも実行される。
tagged socket	One of a family of convenience sockets that add an int id tag to every incoming interface method call in order to identify the socket (or element of a multi-socket) through which the transaction arrived.	タグ付きソケット	便利ソケットの一つで、トランザクションが到着したソケット（または、マルチ・ソケットの要素）を特定するために、受信したあらゆるインタフェース・メソッド・コールに整数型の id タグを加えたもの。

target	A module that represents the final destination of a transaction, able to respond to transactions generated by an initiator, but not itself able to initiate new transactions. For a write operation, data is copied from the initiator to one or more targets. For a read operation, data is copied from one target to the initiator. A target may read or modify the state of the transaction object. In the case of the TLM-1 interfaces, the term *target* as defined here may not be strictly applicable, so the terms *caller* and *callee* may be used instead for clarity.	ターゲット	トランザクションの最終的な到達先を表すモジュールで、イニシエータによって生成されるトランザクションに応答する事が出来る。しかしそれ自身は新しいトランザクションを開始する事は出来ない。ライト操作においては、データがイニシエータから 1 個以上のターゲットへコピーされる。リード操作においては、データはターゲットからイニシエータへコピーされる。ターゲットは、トランザクション・オブジェクトの状態を読む事や、変更する事が可能である。TLM-1 インタフェースにおいては、ターゲットと言う用語は厳密には適切でないため、代わりに呼び出し元関数、呼び出し先関数といった用語を使って明確にするのがよい。
target socket	A class containing a port for interface method calls on the backward path and an export for interface method calls on the forward path. A socket also overloads the SystemC binding operators to bind both port and export.	ターゲット・ソケット	バックワード・パス上のインタフェース・メソッド・コールのためのポートとフォワード・パス上のインタフェース・メソッド・コールのためのエクスポートからなるクラス。 また、このソケットはポートとエクスポートの両方をバインドするために SystemC のバインド演算子をオーバーロードする。
temporal decoupling	The ability to allow one or more initiators to run ahead of the current simulation time in order to reduce context switching and thus increase simulation speed.	テンポラル・デカップリング	コンテクスト・スイッチを抑えてシミュレーション速度を向上させる目的で、シミュレーション時間に先行して 1 つ以上のイニシエータが動作することを許す機能。
timing annotation	The **sc_time** argument to the **b_transport** and *nb_transport* methods. A timing annotation is a local time offset. The recipient of a transaction is required to behave as if it had received the transaction at effective_local_time = sc_time_stamp() + local_time_offset.	タイミング・アノテーション	b_transport と nb_transport メソッドの sc_time 引数。 タイミング・アノテーションはローカル・タイム・オフセットである。トランザクションの受信側はトランザクションが effective_local_time = sc_time_stamp() + local_time_offset で受けたように振る舞うことを要求される。

timing point	A significant time within the lifetime of a transaction. A loosely-timed transaction has two timing points corresponding to the call to and return from **b_transport**. An approximately-timed base protocol transaction has four timing points, each corresponding to a phase transition.	タイミング・ポイント	トランザクションのライフタイム内の重要な時刻。ルーズリータイムド・トランザクションには b_transport の呼び出しとリターンに対応した 2 つのタイミング・ポイントがある。アプロキシメイトリー・タイムドの基本プロトコル・トランザクションはフェーズ・トランジションに対応する 4 つのタイミング・ポイントがある。
TLM-1	The first major version of the OSCI Transaction Level Modeling standard. TLM-1 was released in 2005.	TLM-1	OSCI ＴＬM標準の最初のメジャーバージョン。TLM-1 は 2005 年にリリースされた。
TLM-2.0	The second major version of the OSCI Transaction Level Modeling standard. This document describes TLM-2.0.	TLM-2.0	OSCI ＴＬM標準の 2 つ目のメジャーバージョン。本ドキュメントはTLM -2.0 について記載している。
traits class	In C++ programming, a class that contains definitions such as typedefs that are used to specialize the behavior of a primary class, typically by having the traits class passed as a template argument to the primary class. The default template parameter provides the default traits for the primary class.	トレイツ・クラス	C++プログラミングにおいて、プライマリクラスの振舞いを特化する際に使用する typedefs などの定義を含むクラスであり、通常はプライマリクラスのテンプレート引数としてトレイツ・クラスを持たせる。デフォルト・テンプレート・パラメータはプライマリクラスのデフォルト・トレイツを提供している。
transaction	An abstraction for an interaction or communication between two or more concurrent processes. A transaction carries a set of attributes and is bounded in time, meaning that the attributes are only valid within a specific time window. The timing associated with the transaction is limited to a specific set of timing points, depending on the type of the transaction. Processes may be permitted to read or modify attributes of the transaction, depending on the protocol.	トランザクション	二つ以上の並列プロセス間の相互作用や通信の抽象化。トランザクションはアトリビュートのセットを伝え、時間と結びついている。これは、アトリビュートは特定の時間範囲内でのみ有効である事を意味する。トランザクションに結び付けられた時間は、トランザクションの型に依存した特定のタイミング・ポイントのセットに限定される。プロセスは、プロトコルに依存するが、トランザクションのアトリビュートのリードもしくは変更を許可される。

transaction bridge	A component that acts as the target for an incoming transaction and as the initiator for an outgoing transaction, usually for the purpose of modeling a bus bridge. See *bridge*	トランザクション･ブリッジ	通常、バス・ブリッジのモデリングを目的としたコンポーネントで、トランザクションを受け取る際にはターゲットとして、トランザクションを発行する際にはイニシエータとして振舞う。「ブリッジ」を参照。
transaction instance	A unique instance of a transaction. A transaction instance is represented by one transaction object, but the same transaction object may be re-used for several transaction instances.	トランザクション･インスタンス	トランザクションのユニークなインスタンス。トランザクション・インスタンスは１つのトランザクション・オブジェクトで表わされるが、同じトランザクション・オブジェクトは複数のトランザクション・インスタンスに再利用されるかもしれない。
transaction object	"The object that stores the attributes associated with a transaction. The type of the transaction object is passed as a template argument to the core interfaces."	トランザクション･オブジェクト	トランザクションが持つアトリビュートを保持するオブジェクト。トランザクション・オブジェクトはコア・インタフェースへテンプレート引数として渡される。
transaction level (TL)	The abstraction level at which communication between concurrent processes is abstracted away from pin wiggling to transactions. This term does not imply any particular level of granularity with respect to the abstraction of time, structure, or behavior.	トランザクション・レベル（TL）	並列プロセス間通信で、ピンレベルの信号動作からトランザクションへ抽象化された抽象度。この用語は、時間、構造、動作の抽象度に関する詳細な粒度については言及しない。
transaction level model, transaction level modeling (TLM)	A model at the transaction level and the act of creating such a model, respectively. Transaction level models typically communicate using function calls, as opposed to the style of setting events on individual pins or nets as used by RTL models.	トランザクション・レベル・モデル､トランザクション･レベル･モデリング（TLM）	トランザクション・レベルのモデルと、そのようなモデルを作成することを表す。トランザクション・レベル・モデルは一般的に、RTL モデルによって使用されるような各ピンやネット上にイベントを設定するようなスタイルとは異なり、関数呼び出しを使用して通信を行う。

transactor	A module that connects a transaction level interface to a pin level interface (in the general sense of the word interface) or that connects together two or more transaction level interfaces, often at different abstraction levels. In the typical case, the first transaction level interface represents a memory mapped bus or other protocol, the second interface represents the implementation of that protocol at a lower abstraction level. However, a single transactor may have multiple transaction level or pin level interfaces. See *adaptor*, *bridge*.	トランザクタ	トランザクション・レベル・インタフェースからピン・レベル・インタフェース（一般的な語句インタフェースの意味）への接続や、二つ以上のトランザクション・レベル・インタフェースの接続(しばしば異なる抽象度間で)に使用されるモジュール。一般的には、初めのトランザクション・レベル・インタフェースはメモリ・マップド・バスやその他プロトコルを表し、次のレベルのインタフェースは、低抽象度のプロトコルの実装を表す。しかし、一つのトランザクタが、複数のトランザクション・レベルや、ピン・レベル・インタフェースを持つ場合もある。「アダプタ・ブリッジ」を参照。
transparent component	A interconnect component with the property that all incoming interface method calls are propagated immediately through the component without delay and without modification to the arguments or to the transaction object (extensions excepted). The intent of a transparent component is to allow checkers and monitors to pass ignorable phases.	トランスペアレント・コンポーネント	受け取ったすべてのインタフェース・メソッド・コールが引数あるいはトランサクション・オブジェクト（拡張を除く）の変更なし、遅延なしで直ちに伝播させるという性質を持ったインターコネクト・コンポーネント。トランスペアレント・コンポーネントの目的は、チェッカやモニタが無視可能なフェーズをそのまま通過させるためである。
transport interface	The one and only bidirectional core interface in TLM-1. The transport interface passes a request transaction object from caller to callee, and returns a response transaction object from callee to caller. TLM-2.0 adds separate blocking and non-blocking transport interfaces.	トランスポート・インタフェース	TLM-1 では一つしかない双方向のコア・インタフェース。トランスポート・インタフェースは呼び出し元関数から呼び出し先関数へリクエスト・トランザクション・オブジェクトを送り、呼び出し先関数から呼び出し元関数へレスポンス・トランザクション・オブジェクトを返す。 TLM-2.0 ではブロッキング・トランスポート・インタフェースとノンブロッキング・トランスポート・インタフェースが追加されている。

unidirectional interface	A TLM-1.0 transaction level interface in which the attributes of the transaction object are strictly readonly in the period between the first timing point and the end of the transaction lifetime. Effectively, the information represented by the transaction object is strictly passed in one direction either from caller to callee or from callee to caller. In the case of **void put(const T& t)**, the first timing point is marked by the function call. In the case of **void get(T& t)**, the first timing point is marked by the return from the function. In the case of **T get()**, strictly speaking there are two separate transaction objects, and the return from the function marks the degenerate end-of-life of the first object and the first timing point of the second.	単方向インタフェース	TLM-1.0 のトランザクション・レベル・インタフェースで、トランザクション・オブジェクトのアトリビュートが最初のタイミング・ポイントからトランザクション・ライフタイムの最後までの期間で厳密に読み出しのみに限定されている。トランザクション・オブジェクトで表される情報は、呼び出し元関数から呼び出し先関数、または呼び出し先関数から呼び出し元関数へ、厳密に一方向に送られる。void put(const T&t)の場合では、最初のタイミング・ポイントは関数呼び出し時である。T valid get(T& t)の場合では、最初のタイミング・ポイントはこの関数からのリターン時である。T get()の場合、厳密に言うと二つのトランザクション・オブジェクトが関わっている。この関数からのリターンは 1 つ目のオブジェクトの終わりと 2 つ目のオブジェクトの最初のタイミング・ポイントを表す。
untimed	A modeling style in which there is no explicit mention of time or cycles, but which includes concurrency and sequencing of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes and blocking FIFOs. Some users adopt the practice of inserting random delays into untimed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.	アンタイムド	時間やサイクルを明示的に表現しないモデリング・スタイル。しかしオペレーションの並列性や順序は含まれる。明示的な時間概念は無いが、複数並列スレッド間のオペレーションの順序性は、イベント、ミューテックス、ブロッキング FIFO 等の同期プリミティブを使って達成されなければならない。ユーザによっては、プロトコルの頑健性をテストするため、アンタイムドな記述にランダム遅延を入れる場合があるが、このやり方によってモデリング・スタイルの基本的な性質が変わることはない。
utilities	A set of classes of the TLM-2.0 standard that are provided for convenience only, and are not strictly necessary to achieve interoperability between transaction-level models.	ユーティリティ	TLM-2.0 標準のクラスセットで、利便性のためにのみ提供されており、必ずしもトランザクション・レベル・モデルの間の相互利用性を達成するのには必要ではない。

valid	The state of an object returned from a function by pointer or by reference, during any period in which the object is not deleted and its value or behavior remains accessible to the application. (SystemC term)	バリッド	ポインタまたは参照で関数からリターンされるオブジェクトの状態で、そのオブジェクトが削除されておらず、その値や動作をアプリケーションからアクセスが可能である期間の状態。（SystemC 用語）
within	The relationship that exists between an instance and a module if the constructor of the instance is called from the constructor of the module, and also provided that the instance is not within a nested module. (SystemC term)	包含	インスタンスのコンストラクタがモジュールのコンストラクタから呼ばれる場合で、また、そのインスタンスが入れ子になったモジュールに包含されないと仮定したとき、インスタンスとモジュールの間に存在する関係。（SystemC 用語）
yield	Return control to the SystemC scheduler. For a thread process, to yield is to call wait. For a method process, to yield is to return from the function.	（制御を）譲る	SystemC スケジューラに制御を返すこと。スレッド・プロセスにおいて、制御を譲るには wait 文を呼ぶ。 メソッド・プロセスにおいては、制御を譲るには、関数からリターンする。

付録 ： TLM 2.0.1 キットに含まれる例題の説明

TLM 2.0.1 のアーカイブには詳細な例題とその説明資料（パワーポイント）が含まれているので、ここでその概要を説明します。

大きく分けて、examples ディレクトリに含まれる総合的な example と、unit_test ディレクトリに含まれる単体の機能をチェックするための unit_test の二つがあります。

しかし、examples ディレクトリにあるデザインの中では、unit_test の中にある AT のバスモデル（SimpleBusAT.h）と LT のバスモデル（SimpleBusLT.h）が流用されているため、注意が必要です。

1. 方法

examples と、unit_test の両方とも、Makefile が、unix 用と VisualC++用に用意されていますので、OSCI の SystemC 2.2.0 又は、SystemC 2.1.v1 の環境があれば簡単にデザインを走らせる事ができます。

動かすための詳細な方法については、README.txt ファイルにかかれていますが、Linux の場合には次の方法によって動かせます。

```
setenv TARGET_ARCH linux
# 64bit の場合には、linux64

setenv TLM_HOME /home/demo/TLM-2009-07-15
setenv SYSTEMC_HOME /home/tools/systemc-RE4-32-gcc343/systemc-2.2.0

cd /home/demo/TLM-2009-07-15/unit_test/tlm/build-unix
make check
make run

cd   /home/demo/TLM-2009-07-15/examples/tlm/build-unix
make check
make run
```

make run の場合には、実行結果を画面にそのまま出力するようになっています。

make check の場合には、あらかじめ含まれているリファレンスのログファイルと、実際に走らせた結果のログファイルを比較して同じである場合には、OK というメッセージを出すようになっています。

つまり、unit_test も、examples も、様々な実行環境におけるレグレッションテストの目的で使用できるようになっています。

なお､64bit 環境において 32bit でコンパイルされた SystemC シミュレータを動作させる場合には、-m32 オプションを Makefile.config の FLAGS と LDFLAGS に以下のように追加してから、make clean; make check 等を再度実行する必要があります。

```
FLAGS      = -m32 -g -Wall -pedantic -Wno-long-long $(FLAG_WERROR) ¥
               -DSC_INCLUDE_DYNAMIC_PROCESSES $(PFLAGS) ¥
               -I$(SYSTEMC_INC_DIR) -I$(TLM_INC_DIR)
LDFLAGS   = -m32 -L$(SYSTEMC_LIB_DIR) -lsystemc $(PLDFLAGS)
```

2. ドキュメント

それぞれのディレクトリに README.txt が含まれている他、example の個別のディレクトリの中には、パワーポイントにて、そのデザインの全体ブロック図、出力結果の説明、内部モデル構造図、タイミングチャートの解説等が含まれている。

unit_test に関しては、README.txt 以外の資料は無い。

3. unit_test

ユニット・テストの中には、それぞれの個別の機能をチェックするための Example が含まれています。

個別の機能チェックだけなので、モデルとしては一部の機能を省略していたりする場合があるので注意が必要です。

3.1. models ディレクトリ

SimpleBusLT	LT のみインプリメントしたバスモデル（AT は、b->nb 自動変換にて対応） タグ付きソケットを使用
SimpleBusAT	LT,AT の両方をインプリメントしたバスモデル。タグ付きソケットを使用

bus の unit_test では、SimpleBusAT が使用されている。

SimpleLTInitiator1	標準ソケットを使用した基本的な LT イニシエータ。
SimpleLTInitiator1_DMI	SimpleLTInitiator1 に DMI を追加したもの。
SimpleLTTarget1	標準ソケットを使用した基本的な LT ターゲット。DMI もサポートしています。
SimpleLTInitiator2	シンプル・ソケットを使用した LT イニシエータ
SimpleLTInitiator2_DMI	SimpleLTInitiator2 に DMI を追加したもの
SimpleLTTarget2	シンプル・ソケットを使用した LT ターゲット。DMI もサポートしています。

SimpleATInitaitor1	シンプル・ソケットを使用した AT イニシエータ
SimpleATInitaitor2	シンプル・ソケットを使用した AT イニシエータ タイミング・アノテーション使用　フェーズ省略
SimpleATTarget1	シンプル・ソケットを使用した AT ターゲット
SimpleATTarget2	シンプル・ソケットを使用した AT ターゲット　タイミング・アノテーション使用、フェーズ省略

CoreDecouplingLTInitiator	テンポラル・デカップリングを使用した LT イニシエータ
ExplicitgLTTarget.h	直接 wait()を呼び出している LT ターゲットモデル
ExplicitgATTarget.h	AT のみをインプリメントしている AT ターゲット。(LT は、シンプル・ソケットの自動変換にて対応)

3.2. bus

AT のバスモデルに、4 つの LT イニシエータ、3 つの AT イニシエータ、一つのターゲットモデルを接続したバスモデル検証用基本モデル。

3.3. bus_DMI

bus に対して、DMI を使用したも

3.4. endian_conv

Little Endian、Big Endian のコンバージョンをテストするもの

Linux マシンだけでなく Sparc マシン等の異なる endian の環境でもテストする必要があります。

3.5. multi_socket

AT のバスとしてマルチ・ソケットを使用した例

instance spefic extension も使用しています。

3.6. cancel_all

Payload Event Queue の cancel_all を使用した例

3.7. update_original

Generic Payload の update_original_from メソッドを使用した例

3.8. nb2b_adaptor

AT->LT アダプタを実現する例

3.9. p2p

このディレクトリには、Bus モデルを使用せず、イニシエータとターゲットをポイントツーポイント（p2p）で接続して個別のテストを行っている、サブディレクトリが含まれています。

3.9.1. p2p/BaseSocketLT

LT イニシエータと LT ターゲットを直接接続したモデル。もっとも基本的な例

3.9.2. p2p/CoreDecoupling

テンポラル・デカップリングを使用した例

3.9.3. p2p/HierarchicalSocket

階層バインディングを使用した例

3.9.4. p2p/RegisterSOcketProcessLT

シンプル・ソケットのコールバック登録機能を用いた例

3.9.5. p2p/SimpleAT

AT モデルとして TLM_ACCEPTED を返す例

3.9.6. p2p/SimpleAT_TA

AT モデルとして TLM_UPDATED を返す例

3.10. static_extensions

Generic payload の拡張機能を使用するものが 3 例含まれています。

4. example

4.1. lt

SimpleBuSLT モデルに、LT イニシエータと LT ターゲットを接続した例

4.2. lt_dmi

lt_example をベースに、DMI を使用したもの

4.3. lt_temporal_decouple

lt をベースに、テンポラル・デカップリングを使用したもの

4.4. lt_extension_mandatory

Generic Payload に無視できない必須の拡張機能を使った例 base_protocol からはずれるために、ソケットの 3 つ目の引数に特定クラスを指定しています。

4.5. lt_mixed_endian

BigEndian のイニシエータと LittleEndian のイニシエータが混在している例

4.6. at_1_phase

AT ターゲットモデルにて、すぐに TLM_COMPLETED を返す 1 フェーズモデル

（2 タイミング・ポイント）

4.7. at_2_phase

AT ターゲットモデルにて、TLM_UPDATED にて、END_REQ を返すことにより 2 フェーズにしたモデル。

（3 タイミング・ポイント）

4.8. at_4_phase

At ターゲットモデルにて、すべてのフェーズを使用したモデル

名称と異なり、実際には、4 フェーズではなく、3 フェーズです。

（4 タイミング・ポイント）

4.9. at_extension_optional

Generic Payload に無視可能な拡張を行った例。lt_extension_mandatory と異なり標準のソケットを使用。

4.10. at_mixed_target

at_1_phase,at_2_phase,at_4_phase の 3 種のターゲットを混在させた例

4.11. at_ooo

アウト・オブ・オーダー（ooo）トランザクションを実現している例

4.3.3 SystemC Draft LRM差分解説

JEITA EDA技術専門委員会

標準化小委員会

SystemCワーキンググループ

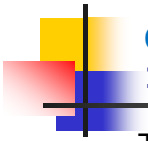
© Copyright 2011 JEITA, All rights reserved JEITA

CLARIFICATION

1.1 Scope

- This standard defines SystemC®1 as an ANSI standard C++ class library for system and hardware design and defines TLM-2.0 as an ANSI standard C++ class library for transaction-level modeling.

JEITA



CLARIFICATION

1.2 Purpose

- The specific purpose of this standard is to provide a precise and complete definition of the SystemC and TLM-2.0 class libraries so that a SystemC implementation can be developed with reference to this standard alone. This standard is not intended to serve as a users' guide or to provide an introduction to SystemC or TLM-2.0, but does contain useful information for end users.

JEITA



CLARIFICATION

1.3 Subsets

- It is anticipated that tool vendors will create implementations that support only a subset of this standard or that impose further constraints on the use of this standard. Such implementations are not fully compliant with this standard but may nevertheless claim partial compliance with this standard and may use the name SystemC. See also 10.1 for a description of TLM-2.0-compliance.

JEITA

CLARIFICATION

1.5 Guidance for readers

- ……..
- Readers should bear in mind that the primary obligation of a tool vendor is to implement the abstract semantics defined in Clause 5, using the framework and constraints provided by the class definitions starting in Clause 6.
- CLARIFICATION: The clauses from Clause 10 onward define the public interface to the TLM-2.0 class library, including the classes of the interoperability layer and the utilities.
  (10章より前の公開インターフェースをTLM-2.0クラス・ライブラリと定義)
- Annex A is intended to aid the reader in the understanding of the structure and intent of the SystemC class library.
- ……..

JEITA

ADDITION

2. References

- This standard shall be used in conjunction with the following publications:
- ISO/IEC 14882:2003, Programming Languages—C++.3
- IEEE Std 1364™-2001, IEEE Standard Verilog® Hardware Description Language.4, 5
- IEEE Std 1666-2005, SystemC Language Reference Manual
- Requirements Specification for TLM 2.0, Version 1.1, September 16, 2007

JEITA

ADDITION

3. Bibliography

- The following books may provide useful background information:
- Transaction-Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems, edited by Frank Ghenassia, published by Springer 2005, ISBN 10 0 387-26232-6(HB), ISBN 13 978-0-387-26232-1(HB)
- Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms, by Tim Kogel, Rainer Leupers, and Heinrich Meyr, published by Springer 2006, ISBN 10 1-4020-4825-4(HB), ISBN 13978-1-4020-4825-4(HB)
- ESL Design and Verification, by Brian Bailey, Grant Martin and Andrew Piziali, published by Morgan Kaufmann/Elsevier 2007, ISBN 10 0 12 373551-3, ISBN 13 978 0 12 373551-5

JEITA

CLARIFICATION

4.1.2 Implementation, application

- The word *implementation is used to mean any specific implementation of the full CLARIFICATION:* SystemC and TLM-2.0 class libraries as defined in this standard, only the public interface of which need be exposed to the application.
- The word *application is used to mean a C++ program, written by an end user, that uses the* CLARIFICATION: SystemC and TLM-2.0 class libraries, that is, uses classes, functions, or macros defined in this standard.

JEITA

CLARIFICATION
4.3.4 Namespaces and internal naming

- CLARIFICATION: An implementation shall place every declaration and definition specified by this standard, with the one exception of **sc_main**, within one of the four namespaces **sc_core**, **sc_dt**, **tlm**, and **tlm_utils**. The core language and predefined channels shall be placed in the namespace **sc_core**. The SystemC data types proper shall be placed in the namespace **sc_dt**. CLARIFICATION: The SystemC utilities are divided between the two namespaces **sc_core** and **sc_dt**.

- 実装はこの規格によって指定されたあらゆる宣言と定義を**sc_main**という1つの例外と4つの名前空間**sc_core**、**sc_dt**、**tlm**、および**tlm_utils**の1つの中で置くものとする。コア言語と事前に定義されたチャンネルは名前空間**sc_core**に置かれるものとする。SystemCデータ型自体は名前空間**sc_dt**に置かれるものとする。SystemCユーティリティは2つの名前空間**sc_core**と**sc_dt**の間に分割される。

JEITA

CLARIFICATION
5.1.1 Instantiation

- CLARIFICATION: Instances of class **sc_module** and class **sc_prim_channel** may only be created within a module or from function **sc_main** (or a function called from **sc_main**), or in the absence of an **sc_main** (see 5.3.5), in the form of one or more top-level modules. Instances of class **sc_port** and class **sc_export** can only be created within a module. It shall be an error to instantiate a module or primitive channel other than as described above, or to instantiate a port or export other than within a module.

- クラス**sc_module**とクラス**sc_prim_channel**のインスタンスは、モジュール内か、function **sc_main**から(または、**sc_main**から呼ばれたfunction)、または、**sc_main**がないとき(5.3.5を参照)、1つ以上のトップレベルモジュールの形で作成されるだけである。クラス**sc_port**とクラス**sc_export**のインスタンスはモジュールの中でのみ作成できる。それは、上で説明されたものを除いたモジュールかプリミティブのチャンネルをインスタンスするか、またはモジュールを除いたportまたはexportをインスタンスするとエラーになる。

JEITA

CLARIFICATION
5.2.1.2 Evaluation phase

- CLARIFICATION: From the set of runnable processes, select a process instance, remove it from the set, and only then trigger or resume its execution. Run the process instance immediately and without interruption up to the point where it either returns or calls the function **wait.**
- 実行可能なプロセスのセットから、プロセス・インスタンスを選択、セットからそれを削除、その次にトリガーまたは実行を再開する。プロセス・インスタンスをすぐに、そしてreturnsまたはwaitファンクションの呼ばれるポイントまで中断なしに実行する。
- A process may execute an immediate notification, in which case determine which process instances are currently sensitive to the notified event and add all such process instances to the set of runnable processes. CLARIFICATION: Such process instances shall be executed in the current evaluation phase. The currently executing process instance may itself be one such process, in which case it will be executed more than once in the same evaluation phase.
- プロセスはどのケースが決定されるか、どのプロセス・インスタンスが現在通知されたイベントにセンシティブであるかで即座に通知を実効する。そのようなプロセス・インスタンスは現在の評価段階で実行されるものとする。現在執行中のプロセス・インスタンスはそれ自体がその過程の一つであり同じ評価フェーズの中で1回より多く実行される。
- CLARIFICATION: NOTE 4—When an immediate notification occurs, only processes that are currently sensitive to the notified event shall be made runnable. This excludes processes that are only made dynamically sensitive to the notified event later in the same evaluation phase, unless those processes happen to be statically sensitive to the given event.
- 即座の通知が起きるとき、現在通知された出来事にセンシティブなプロセスだけをrunnableにするものとする。これは後で同じ評価フェーズでダイナミックに通知されたイベントにセンシティブにされるだけであるプロセスを除く。それらのプロセスがたまたま静的に特定のイベントにセンシティブでないなら、

JEITA

CLARIFICATION
5.3.4.1 Function sc_main

- CLARIFICATION: An application shall provide a function **sc_main** in the global namespace with the following declaration. The order and types of the arguments and the return type shall be as shown here:

  int **sc_main**(int argc, char* argv[]);

JEITA

CLARIFICATION
5.3.4.2 Function sc_start

- CLARIFICATION: The implementation shall provide a function **sc_start** in the namespace **sc_core**, overloaded with the following signatures:
  - void **sc_start();**
  - void **sc_start(const sc_time&);**
  - void **sc_start(double, sc_time_unit);**

JEITA

ADDITION and CLARIFICATION
5.4.1 before_end_of_elaboration

- The following actions may be performed directly or indirectly from the member function before_end_of_elaboration.
  - a) The instantiation of objects of class **sc_module, sc_port, sc_export, sc_prim_channel**
  - b) The instantiation of objects of other classes derived from class **sc_object**
  - c) Port binding
  - d) Export binding
  - e) The macros SC_METHOD, SC_THREAD, SC_CTHREAD, SC_HAS_PROCESS
  - f) ADDITION: The member **sensitive** and member functions **dont_initialize, set_stack_size, reset_signal_is, and async_reset_signal_is**, and of the class **sc_module**
  - g) Calls to event finder functions
  - h) Calls to function **sc_spawn** to create static spawned processes
  - i) CLARIFICATION: Calls to member function **set_sensitivity** of class **sc_spawn_options** to set the sensitivity of a spawned process using an event, an interface, or an event finder function. A port cannot be used because port binding may have been deferred.
    イベント、インタフェース、またはイベントファインダーを使用することでspawnedプロセスのsensitivityを設定するクラスsc_spawn_optionsのメンバー機能set_sensitivityを呼び出す。ポートバインドが保留されるかも知れないため、ポートは使用できない。
  - j) ADDITION: Calls to member functions **reset_signal_is** and **async_reset_signal_is** of the class **sc_spawn_options**
    クラスsc_spawn_optionsのメンバー機能のreset_signal_isとasync_reset_signal_isを呼ぶ。
- k) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **initialize** of class **sc_inout)**

JEITA

ADDITION

5.4.2 end_of_elaboration

- The following constructs shall not be used directly or indirectly within callback **end_of_elaboration:**
  a) The instantiation of objects of class **sc_module, sc_port, sc_export, sc_prim_channel**
  b) Port binding
  c) Export binding
  d) The macros SC_CTOR, SC_CTHREAD
  e) ADDITION: The member functions **reset_signal_is and async_reset_signal_is of the class sc_module**
  f) Calls to event finder functions
  g) Calls to member function **notify of the class sc_event**

JEITA

ADDITION

5.4.3 start_of_simulation

- The following constructs shall not be used directly or indirectly within callback **start_of_simulation:**
  a) The instantiation of objects of class **sc_module, sc_port, sc_export, sc_prim_channel**
  b) Port binding
  c) Export binding
  d) The macros SC_CTOR, SC_METHOD, SC_THREAD, SC_CTHREAD, SC_HAS_PROCESS
  e) ADDITION: The member **sensitive and member functions dont_initialize, set_stack_size, reset_signal_is, and async_reset_signal_is of the class sc_module**
  f) Calls to event finder functions
  g) Calls to member function **notify of the class sc_event**

JEITA

ADDITION

5.5.1 Function declarations

```
namespace sc_core {
    enum sc_stop_mode
    {
      SC_STOP_FINISH_DELTA ,
      SC_STOP_IMMEDIATE
    };
    extern void sc_set_stop_mode( sc_stop_mode mode );
    extern sc_stop_mode sc_get_stop_mode();
    void sc_stop();
    const sc_time& sc_time_stamp();
    ADDITION: const sc_time& sc_max_time();
    const sc_dt::uint64 sc_delta_count();
    bool sc_is_running();
    CORRECTION: bool sc_pending_activity_at_current_time();
}
```

追加？

JEITA

CHANGE and CLARIFICATION

5.5.2 Function sc_stop, sc_set_stop_mode, and sc_get_stop_mode

- The function sc_set_stop_mode shall set the current stop mode to the value passed as an argument. The function sc_get_stop_mode shall return the current stop mode. CHANGE: Function sc_set_stop_mode may be called during elaboration or from one of the callbacks before_end_of_elaboration, end_of_elaboration, or start_of_simulation. It shall be an error to call function sc_set_stop_mode from the scheduler other than from one of these three callbacks.
- sc_set_stop_modeは引数に値をパスするように現在のストップ・モードをセットするものとする。sc_get_stop_modeは現在のストップ・モードを返すものとする。sc_set_stop_modeはエラボレーション中か、コールバックのbefore_end_of_elaboration、end_of_elaboration、またはstart_of_simulationの1つから呼ばれる。それは、これら3つのエラボレーションの1つ以外のスケジューラからsc_set_stop_modeと呼ばれるとエラーになる。

追加

- CLARIFICATION: The default stop mode shall be SC_STOP_FINISH_DELTA.
- デフォルトのストップ・モードはSC_STOP_FINISH_DELTAになる。

JEITA

ADDITION
5.5.4 ADDITION: Function sc_max_time

- The implementation shall provide a function **sc_max_time** with the following declaration:
  const sc_time& **sc_max_time();**
- The function **sc_max_time** shall return the maximum value of type **sc_time**, calculated after taking into account the time resolution. Since function **sc_max_time** necessarily returns an object of type **sc_time** that represents a non-zero time value, the time resolution cannot be modified after a call to **sc_max_time**. Every call to **sc_max_time** during a given simulation run shall return an object having the same value and representing the maximum simulation time. The actual value is implementation-defined. Whether each call to **sc_max_time** returns the same object or a different object is implementation-defined.
- sc_max_timeは時間解像度を考慮に入れた後に計算されたsc_timeの最大値を返すものとする。sc_max_timeが必ず非ゼロ時間値を表すsc_timeのオブジェクトを返すので、sc_max_timeへの呼び出しの後に時間解像度を変更できない。特定のシミュレーション実行の間のsc_max_timeへのあらゆる呼び出しが同じ値を持って、最大シミュレーション時間を表すオブジェクトを返すものとする。実際の値は実装で定義されている。sc_max_timeへの各呼び出しは同じオブジェクトか異なったオブジェクトを返すかどうかが、実装で定義されている。

JEITA

CLARIFICATION
5.5.5 Function sc_delta_count

- The function **sc_delta_count** shall return an integer value that is incremented exactly once in each delta cycle, and thus returns a count of the absolute number of delta cycles that have occurred during simulation, starting from zero. CLARIFICATION: The delta count shall be 0 during elaboration, and shall be 0 during the first evaluation phase. The delta count shall be incremented between the evaluation phase and the update phase of each delta cycle.
- sc_delta_countは正確にそれぞれのデルタ・サイクルで一度増加される整数値を返すものとする。そして、その結果、0から始めて、シミュレーションの間に起こっているデルタサイクルの絶対数のカウントを返す。デルタ・カウントは、エラボレーション間は0であり、最初の評価フェーズの間も0になる。デルタ・カウントはそれぞれのデルタ・サイクルの評価フェーズとアップデート・フェーズの間で増加されるものとする。

JEITA

CORRECTION

5.5.7 CORRECTION: Function sc_pending_activity_at_current_time

追加

- The implementation shall provide a function **sc_pending_activity_at_current_time** with the following declaration:
  bool **sc_pending_activity_at_current_time**();
- The function **sc_pending_activity_at_current_time** shall return the value true if and only if the set of runnable processes is not empty or the set of update requests is not empty or the set of delta notifications and time-outs is not empty. By implication, if **sc_pending_activity_at_current_time** were to return the value false, the next action of the scheduler would be to execute the timed notification phase. This function may be called at any time during or following elaboration or simulation.

- sc_pending_activity_at_current_timeは、実行可能なプロセスのセットが空でないか、更新要求のセットが空でないか、デルタ通知とタイムアウトのセットが空でないならtrueを返すものとする。その結果、sc_pending_activity_at_current_timeが値のfalseを返すなら、スケジューラの次の動作はタイムド通知フェーズを実行することである。この機能はいつでもエラボレーションまたはシミュレーションの最中または後に呼ばれるかもしれない。

JEITA

CLARIFICATION

6.1 Class header files

- To use the SystemC class library features, an application shall include either of the C++ header files specified in this subclause at appropriate positions in the source code as required by the scope and linkage rules of C++. CLARIFICATION: The TLM-2.0 classes and TLM-2.0 utilities are in separate header files (see 11.8).

JEITA

ADDITION
6.2.2 Class definition

protected:

sc_module(const sc_module_name&);

sc_module();

ADDITION:

void **reset_signal_is(const sc_in<bool>& , bool);**

void **reset_signal_is(const sc_inout<bool>& , bool);**

void **reset_signal_is(const sc_out<bool>& , bool);**

void **reset_signal_is(const sc_signal_in_if<bool>& , bool);**

void **async_reset_signal_is(const sc_in<bool>& , bool);**

void **async_reset_signal_is(const sc_inout<bool>& , bool);**

void **async_reset_signal_is(const sc_out<bool>& , bool);**

void **async_reset_signal_is(const sc_signal_in_if<bool>& , bool);**

sc_sensitive† sensitive;

JEITA

CHANGE: void next_trigger(sc_event_or_list const &);

CHANGE: void next_trigger(sc_event_and_list const &);

void next_trigger(const sc_time&);

void next_trigger(double , sc_time_unit);

void next_trigger(const sc_time& , const sc_event&);

void next_trigger(double , sc_time_unit , const sc_event&);

CHANGE: void next_trigger(const sc_time& , sc_event_or_list const &);

CHANGE: void next_trigger(double , sc_time_unit , sc_event_or_list const &);

CHANGE: void next_trigger(const sc_time& , sc_event_and_list const &);

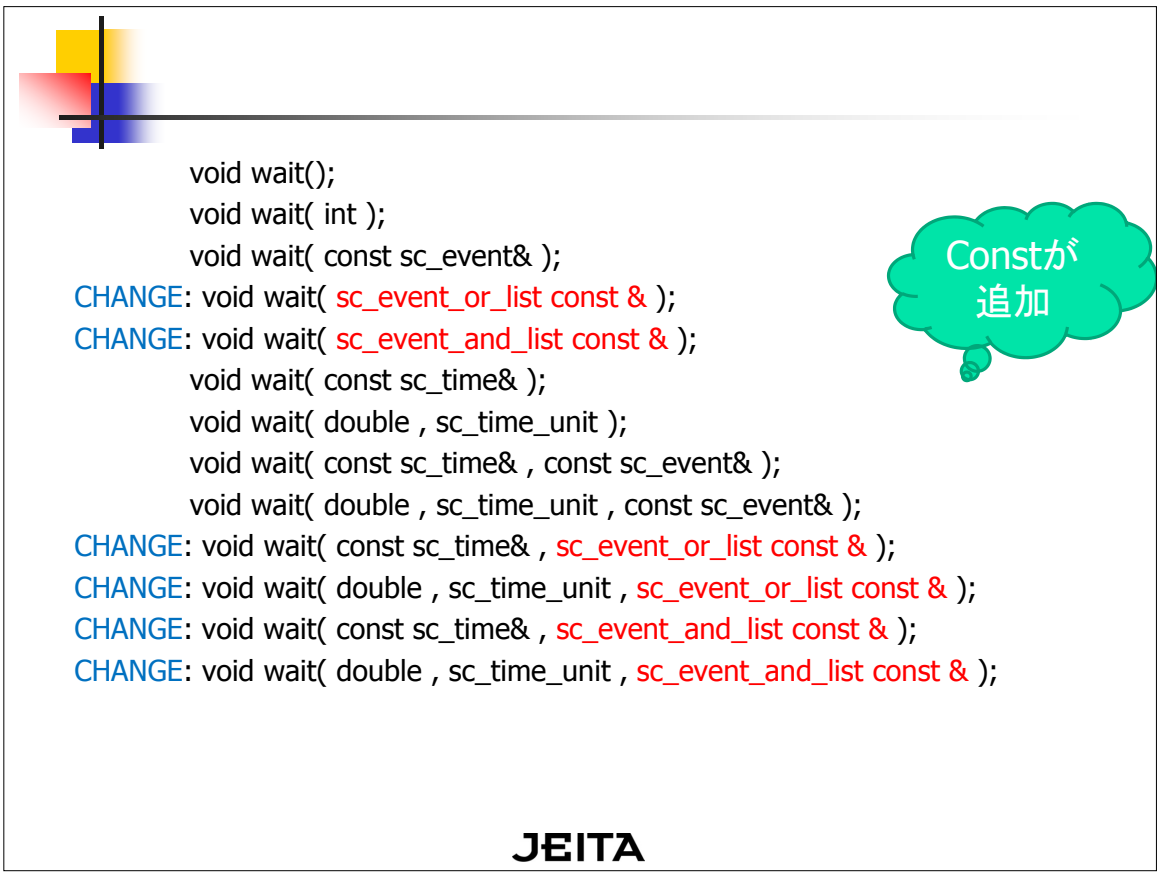
CHANGE: void next_trigger(double , sc_time_unit , sc_event_and_list const &);

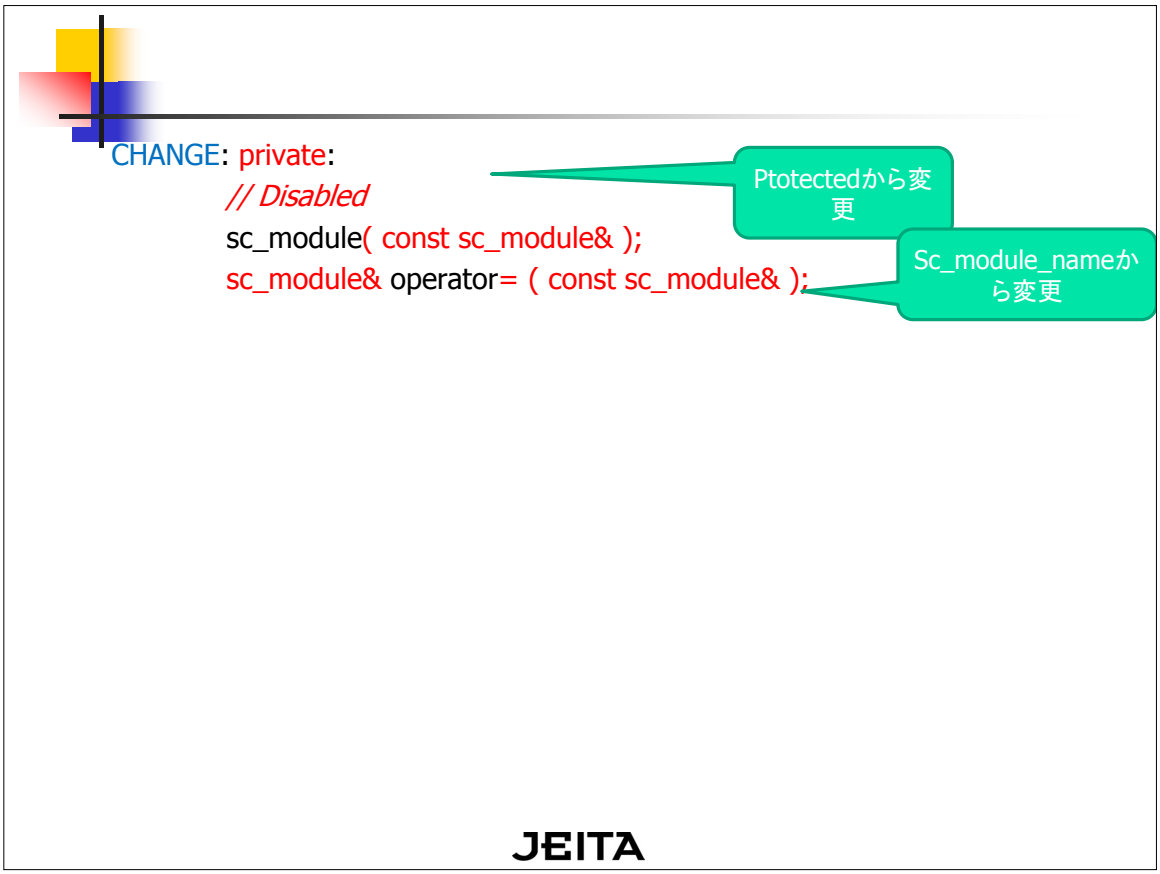
sc_event_or_list†&

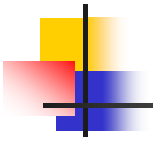
sc_event_and_list†&

Constが追加

JEITA


void wait();
void wait(int);
void wait(const sc_event&);
CHANGE: void wait(sc_event_or_list const &);
CHANGE: void wait(sc_event_and_list const &);
void wait(const sc_time&);
void wait(double , sc_time_unit);
void wait(const sc_time& , const sc_event&);
void wait(double , sc_time_unit , const sc_event&);
CHANGE: void wait(const sc_time& , sc_event_or_list const &);
CHANGE: void wait(double , sc_time_unit , sc_event_or_list const &);
CHANGE: void wait(const sc_time& , sc_event_and_list const &);
CHANGE: void wait(double , sc_time_unit , sc_event_and_list const &);
Constが追加
JEITA

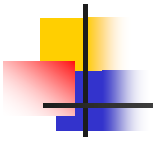

CHANGE: private:
// Disabled
sc_module(const sc_module&);
sc_module& operator= (const sc_module&);
Ptotectedから変更
Sc_module_nameから変更
JEITA



void next_trigger();
void next_trigger(const sc_event&);
CHANGE: void next_trigger(sc_event_or_list const &);
CHANGE: void next_trigger(sc_event_and_list const &);
void next_trigger(const sc_time&);
void next_trigger(double , sc_time_unit);
void next_trigger(const sc_time& , const sc_event&);
void next_trigger(double , sc_time_unit , const sc_event&);
CHANGE: void next_trigger(const sc_time& , sc_event_or_list const &);
CHANGE: void next_trigger(double , sc_time_unit , sc_event_or_list const &);
CHANGE: void next_trigger(const sc_time& , sc_event_and_list const &);
CHANGE: void next_trigger(double , sc_time_unit , sc_event_and_list const &);

Constが追加

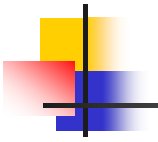
JEITA



void wait();
void wait(int);
void wait(const sc_event&);
CHANGE: void wait(sc_event_or_list const &);
CHANGE: void wait(sc_event_and_list const &);

Constが追加

JEITA



void wait(const sc_time&);
void wait(double , sc_time_unit);
void wait(const sc_time& , const sc_event&);
void wait(double , sc_time_unit , const sc_event&);
CHANGE: void wait(const sc_time& , sc_event_or_list const &);
CHANGE: void wait(double , sc_time_unit , sc_event_or_list const &);
CHANGE: void wait(const sc_time& , sc_event_and_list const &);
CHANGE: void wait(double , sc_time_unit , sc_event_and_list const &);



JEITA



CLARIFICATION

6.2.8 SC_HAS_PROCESS

- Macro SC_HAS_PROCESS shall only be used within the class definition, constructor body, or member function body of a module. The name of the module class being constructed shall be passed as the argument to the macro. CLARIFICATION: The macro invocation shall be terminated with a semicolon.

- マクロの実施はセミコロンで終えるものとする。

JEITA

CLARIFICATION and ADDITION

6.2.9 SC_METHOD, SC_THREAD, SC_CTHREAD

- The second argument passed to the macro SC_CTHREAD shall be an expression of the type **sc_event_finder.**
  CLARIFICATION: In each case, the macro invocation shall be terminated with a semicolon.
- その都度、マクロの実施はセミコロンで終えるものとする。

- ADDITION: A process instance can be suspended, resumed, disabled, enabled, killed, or reset using the member functions of class **sc_process_handle.**
- プロセス・インスタンスはsc_process_handleのメンバー機能を使用することでsuspended, resumed, disabled, enabled, killed, resetすることが出来る。

JEITA

CLARIFICATION and CHANGE

6.2.10 Method process

- A method process is said to be triggered when the kernel calls the function associated with the process instance. When a method process is triggered, the associated function executes from beginning to end, then returns control to the kernel. CHANGE: A method process can only be terminated by calling the kill method of a process handle associated with that process.
- メソッドプロセスはそのプロセスに関連するプロセスハンドルのkillメソッドの呼び出しでのみ終了できる。

- CLARIFICATION: Member function kind of the implementation-defined class associated with a method process instance shall return the string "**sc_method_process**".
- メソッドプロセスインスタンスに関連している実装・定義されたクラスのメンバーの機能の種類は文字列"sc_method_process"を返すものとする。

JEITA

CLARIFICATION

6.2.11 Thread and clocked thread processes

- CLARIFICATION: Member function kind of the implementation-defined class associated with a thread process instance shall return the string "**sc_thread_process**".
- スレッドプロセスインスタンスに関連している実装・定義されたクラスのメンバーの機能の種類は文字列"**sc_thread_process**"を返すものとする。

- CLARIFICATION: Member function kind of the implementation-defined class associated with a clocked thread process instance shall return the string "**sc_cthread_process**".
- クロックドスレッドプロセスインスタンスに関連している実装・定義されたクラスのメンバーの機能の種類は文字列"**sc_cthread_process**"を返すものとする。

JEITA

CHANGE and ADDITION

6.2.12 CHANGE: Clocked thread processes

- ADDITION: A clocked thread process may have any number of synchronous and asynchronous reset signals specified using reset_signal_is and async_reset_signal_is respectively.
- クロックドスレッドプロセスで、それぞれreset_signal_isとasync_reset_signal_isを使用して、いくつでも同期・非同期リセット信号を指定することができる。
- Any of the process control member functions of class sc_process_handle may be called for a clocked thread process, although certain caveats apply to the use of member functions suspend, resume, reset, and throw_it due to the fact that they can be called asynchronously with respect to the clock. These caveats are described below.
- sc_process_handleのプロセスコントロールメンバー機能のいくつかは、クロックドスレッドプロセスのために呼ばれるかもしれない。ある警告はクロックに関してそれらを非同期に呼ぶことができるという事実のため、suspend、resume、reset、およびthrow_itをメンバー機能の使用に適用する。これらの警告は下で説明される。

JEITA

- It is recommended to call disable and enable rather than suspend and resume for clocked thread processes. In the case that resume is called for a clocked thread process, it is the responsibility of the caller to ensure that resume is called during the evaluation phase that immediately follows the delta notification or timed notification phase in which the clock event notification occurs. If a clocked thread is resumed at any other time, that is, if the call to resume is not synchronized with the clock, the behavior shall be implementation defined.
- クロックドスレッドプロセスのためにsuspendやresumeするよりもむしろプロセスをdisableやenableにすることが勧められる。この場合、resumeはクロックドスレッドプロセスのために呼ばれると、すぐにクロックイベント通知が発生するデルタ通知かタイムド通知フェーズに続くのは、コーラーが、resumeを評価フェーズの間に呼ばれるのを保証する責任がある。クロックスレッドが他の時間ならいつでも再開されると、すなわち、resumeへの呼び出しがクロックと同期しないなら、振舞いは定義された実装通りになるだろう。

JEITA

- The process control member functions kill, reset, and throw_it may be called asynchronously with respect to the clock and their effect is immediate, even in the case of a clocked thread process. Similarly, the effect of an asynchronous reset attaining its active value is immediate for a clocked thread process.
- プロセスコントロールメンバー関数のkill, reset, throw_it はクロックに関して非同期に呼ばれるかもしれない。そして、それらの効果はクロックスレッドプロセスの場合でさえ即時である。同様に、非同期リセットがアクティブな値を得るという効果はクロックスレッドプロセスでも即時である。

- In summary, a clocked thread process shall have exactly one clock signal and may have any number of synchronous and asynchronous reset signals as well as being reset by calling the process control member function of class sc_process_handle. As a consequence, the body of the associated function should normally consist of reset behavior followed by a loop statement that contains a call to wait followed by the behavior to be executed in each clock cycle (see the example below).
- まとめると、クロックスレッドプロセスは、確実に1つのクロック信号を持って、sc_process_handleのプロセスコントロールメンバー機能を呼ぶことによってリセットされることと同様にいくつかの同期・非同期リセット信号を持っているかもしれない。結果として、関連する機能のボディーは通常リセットの振舞いがあって続いて各クロック周期で実行されるwaitの呼び出しを含むループ文があとに続く構成に成るはずである(下の例を参照)。

JEITA

Example:

```
sc_in<bool> clock;
sc_in<bool> reset;
sc_in<bool> async_reset;
SC_CTOR (M)
 {
SC_CTHREAD (CT1, clock.pos () ) ;
reset_signal_is (reset, true) ;
SC_CTHREAD (CT2, clock.pos () ) ;
async_reset_signal_is (async_reset, true) ;
}
void CT1 ()
 {
if  (reset)
 {
... // Reset actions
}
while (true)
 {
wait (1) ; // Wait for 1 clock cycle
... // Clocked actions
}
}
```

```
void CT2 ()
 {
... // Reset actions
while (true)
 {
try  {
while  (true)
 {
wait () ; // Wait for 1 clock cycle
... // Regular behavior
}
}
catch  (...)  // Some exception has been thrown
 {
... // Handle exception and go back to waiting for clock
}
}
}
```

JEITA

ADDITION

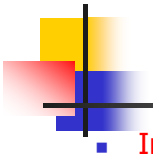
6.2.13 ADDITION: reset_signal_is and async_reset_signal_is

```
void reset_signal_is( const sc_in<bool>& , bool );
void reset_signal_is( const sc_inout<bool>& , bool );
void reset_signal_is( const sc_out<bool>& , bool );
void reset_signal_is( const sc_signal_in_if<bool>& , bool );
void async_reset_signal_is( const sc_in<bool>& , bool );
void async_reset_signal_is( const sc_inout<bool>& , bool );
void async_reset_signal_is( const sc_out<bool>& , bool );
void async_reset_signal_is( const sc_signal_in_if<bool>& , bool );
```

JEITA

- Member functions reset_signal_is and async_reset_signal_is of class sc_module shall determine the synchronous and asynchronous reset signals, respectively, of a thread, clocked thread or method process. These two member functions shall only be called in the body of the constructor, in the before_end_of_elaboration callback of a module, or in a member function called from the constructor or callback, and only after having created a process instance within that same constructor or callback.
- sc_moduleのメンバーファンクションのreset_signal_isとasync_reset_signal_isはスレッド、クロックドスレッドまたはメソッドプロセスについてそれぞれ同期・非同期リセット信号を決定するものとする。これらの2つのメンバーファンクションは、コンストラクタのボディーで、または、モジュールのbefore_end_of_elaborationコールバックで、または、コンストラクタかコールバックから呼ばれるメンバーファンクションで、そして、その同じコンストラクタかコールバックの中でプロセスインスタンスを作成した後にだけ呼ばれる。
- The order of execution of the statements within the body of the constructor or the before_end_of_elaboration callback shall be used to associate the call to reset_signal_is or async_reset_signal_is with a particular process instance; the call is associated with the most recently created process instance. If a module is instantiated within the constructor or callback between the process being created and function reset_signal_is or async_reset_signal_is being called, the effect of calling reset_signal_is or async_reset_signal_is shall be undefined.
- コンストラクタまたはbefore_end_of_elaborationコールバックのボディーの中のステートメントに関する実行順序は、reset_signal_isかasync_reset_signal_isへの呼び出しを特定のプロセスインスタンスに関連づけるのに使用されるものとする;呼び出しは直前に書かれたプロセスインスタンスに関連している。もし、モジュールは、プロセスが作成されるのとreset_signal_isかasync_reset_signal_isが呼び出されるのとの間にコンストラクタかコールバックの中にインスタンスされると、reset_signal_isかasync_reset_signal_isを呼ぶという効果は未定義になるだろう。

- The first argument passed to function reset_signal_is or async_reset_signal_is shall identify the signal instance to be used as the reset. The signal may be identified indirectly by passing a port instance that is bound to the reset signal. The second argument shall be the active level of the reset signal, meaning that the process is to be reset only when the value of the reset signal is equal to the value of this second argument.
- reset_signal_isかasync_reset_signal_isに渡された最初の引数は、リセットとして使用されるべき信号インスタンスであると認識するものとする。信号は、リセット信号にバウンドされるポートインスタンスをパスして、間接的に確認されるかもしれない。2番目の引数はリセット信号のアクティブなレベルになる。つまり、プロセスがリセット信号の値がこの2番目の引数の値と同じときだけ、リセットされることを意味する。
- When the reset signal (as specified by the first argument) attains its active value (as specified by the second argument), the process instance shall enter the synchronous reset state. While in the synchronous reset state, a process instance shall be reset each time it is resumed, whether due to an event notification or to a timeout. Being reset in this sense shall have the same effect as would a call to the reset method of a process handle associated with the given process instance. The synchronous reset state is described more fully in 6.6.6.2.
- リセット信号(最初の引数によって指定されるように)がアクティブな値を得るとき(2番目の引数によって指定されるように)、プロセスインスタンスは同期リセット状態に入るものとする。同期リセット状態にある間、イベント通知かタイムアウトにかかわらずそれが再開されるたびにプロセスインスタンスはリセットされるものとする。この意味で、リセットは特定のプロセスインスタンスに関連しているプロセスハンドルのリセットメソッドへの呼び出しと同じ効果があるものとする。同期リセット状態は6.6.6.2において、より詳しく説明される。



- In the case of async_reset_signal_is only, the process shall also be reset (in the sense described in the previous paragraph) whenever the reset signal attains its active value. In other words, since the reset signal is itself a primitive channel which can only change value in the update phase, the process instance would be reset by effectively calling the reset method of an associated process handle at some time during the evaluation phase immediately following the reset signal attaining its active value and in an order determined by the kernel.
- また、async_reset_signal_isの場合だけは、リセット信号がアクティブな価値を得るときはいつも(前のパラグラフで説明したように)プロセスはリセットされるものとする。言い換えれば、リセット信号がそれ自体でアップデートフェーズにおける値を変えることができるプリミティブなチャンネルであるので、事実上、プロセスインスタンスはカーネルで決定している順序にアクティブな値を得るリセット信号に従う評価フェーズの直前にいくつかの関連プロセスハンドルのリセットメソッドを呼ぶことによってリセットされるだろう。

JEITA



- A given process instance may have any number of synchronous and asynchronous reset signals. When all such reset signals attain the negation of their active value, the process instance shall leave the synchronous reset state unless a call to sync_reset_on is in force for the given process instance (see 6.6.6.2).
- 特定のプロセスインスタンスには、複数の同期・非同期リセット信号があるかもしれない。そのようなすべてのリセット信号がそれらの非アクティブな値に達するとき、特定のプロセスインスタンスには、sync_reset_onへの呼び出しが有効でない場合(6.6.6.2を参照)、プロセスインスタンスは同期リセットの状態から出るものとする。
- reset_signal_is and async_reset_signal_is are permitted for method processes, although the only effect of resetting a method process as a result of such a call is to cancel any dynamic sensitivity and to restore the static sensitivity of that process instance. In any case, a method process cannot enter the synchronous reset state. An implementation may generate a warning if reset_signal_is or async_reset_signal_is are used for method processes.
- reset_signal_isとasync_reset_signal_isはメソッドプロセスのために、容認されるけれども、そのような呼び出しの結果、メソッドプロセスをリセットするという唯一の効果がどんなダイナミックなセンシティビティもキャンセルして、そのプロセスインスタンスの静的なセンシティビティを回復する。どのような場合でも、メソッドプロセスは同期リセット状態に入ることができない。メソッドプロセスのためにreset_signal_isかasync_reset_signal_isが使用されるなら、実装は警告を発生させるかもしれない。

JEITA

CLARIFICATION
6.2.17 next_trigger

- CLARIFICATION: When called with one or more arguments, the function **next_trigger** shall set the dynamic sensitivity of the method process instance from which it is called for the very next occasion on which that process instance is triggered, and for that occasion only. The dynamic sensitivity is determined by the arguments passed to function **next_trigger**.
- 1つ以上の引数で呼ばれると、next_triggerはそのプロセスインスタンスが引き起こされる次のまさしくその時、およびその時だけに呼ばれるメソッドプロセスインスタンスのダイナミックセンシティビティを設定するものとする。ダイナミックセンシティビティはnext_triggerに渡された引数で決定する。

- void **next_trigger();**

  The process shall be triggered on the static sensitivity. In the absence of static sensitivity for this particular process instance, the process shall not be triggered again during the current simulation. CLARIFICATION: Calling **next_trigger** with an empty argument list shall cancel the effect of any earlier calls to **next_trigger** during the current execution of the method process instance, and thus in effect shall be equivalent to not calling **next_trigger** at all during a given process execution.

  プロセスは静的なセンシティビティで引き起こされるものとする。この特定のプロセスインスタンスのための静的なセンシティビティがないとき、再び現在のシミュレーションの間、プロセスを引き起こさないものとする。空のアーギュメントリストでnext_triggerと呼ぶのは、メソッドプロセスインスタンスの現在の実行の間、next_triggerへのどんな早目の呼び出しの効果もキャンセルして、その結果、事実上、特定プロセス実行の間、next_triggerと呼ばないのと全く同等になるだろう。

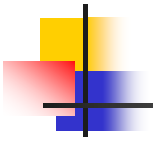
JEITA

- CHANGE: void **next_trigger**(sc_event_or_list const &);

  The argument shall take the form of a list of events separated by the operator| of class sc_event or sc_event_or_list. The process shall be triggered when any one of the given events is notified. The occurrence or non-occurrence of the other events in the list shall have no effect on that particular triggering of the process. If a particular event appears more than once in the list, the behavior shall be the same as if that event had appeared only once. It shall be an error to pass an empty event list object as an argument to **next_trigger** (see 6.8 and 6.9).

  引数はsc_event_or_listかsc_eventのオペレータ“｜”によって切り離されたイベントのリストの型を取るものとする。特定のイベントのどれかが通知されるとき、プロセスは引き起こされるものとする。リストの他のイベントの発生または非発生は、プロセスのその特定の引き金となるときに効き目がないものとする。リストの中の特定のイベントが一度より多く現れると、まるでそのイベントが一度だけ現れた時と同じ振舞いになるだろう。それは、next_triggerの引数に空のイベントリストオブジェクトを渡すとエラーになるだろう(6.8と6.9を参照) 。

JEITA



- CHANGE: void **next_trigger**(sc_event_and_list const &);

  The argument shall take the form of a list of events separated by the operator**&** of class sc_event or sc_event_and_list. In order for the process to be triggered, every single one of the given events shall be notified, with no explicit constraints on the time or order of those notifications. The process is triggered when the last such event is notified, last in the sense of being at the latest point in simulation time, not last in the list. An event in the list may be notified more than once before the last event is notified. If a particular event appears more than once in the list, the behavior shall be the same as if that event had appeared only once. It shall be an error to pass an empty event list object as an argument to **next_trigger** (see 6.8 and 6.9).

  引数はsc_event_and_listかsc_eventのオペレータに“＆”よって切り離されたイベントのリストの型を取るものとする。プロセスが引き起こされるために、特定のイベントのすべてがそれらの通知の時間または順序のときに明白な制約なしで通知されるものとする。プロセスはlastのようなイベントが通知されたとき、シミュレーション時間の最新のポイントにあるという意味での最終であるとき、またはリストのlastでないときにプロセスは引き起こされる。リストにおけるイベントはlastのイベントが通知される前に、一度よりも多く通知されるかもしれない。リストの中の特定のイベントが一度より多く現れると、まるでそのイベントが一度だけ現れた時と同じ振舞いになるだろう。それは、next_triggerの引数に空のイベントリストオブジェクトを渡すとエラーになるだろう(6.8と6.9を参照) 。

JEITA



- void **next_trigger**(const sc_time&);

  The process shall be triggered after the time given as an argument has elapsed. The time shall be taken to be relative to the time at which function **next_trigger** is called. When a process is triggered in this way, a time-out is said to have occurred. CLARIFICATION: In this method call and in those that follow below, the argument that specifies the time-out shall not be negative.

  プロセスは引数で与えられた時間が経過した後に引き起こされるものとする。時間は、next_triggerが呼ばれる時間に関連してかかるものとする。プロセスがこのように引き起こされるとき、タイムアウトが起こったと言われる。このメソッドの呼び出しと以下に続くものでは、タイムアウトを指定する引数は、負の値にしてはいけない。

JEITA

void next_trigger(double , sc_time_unit , const sc_event&);
CHANGE: void next_trigger(const sc_time& , sc_event_or_list const &);
CHANGE: void next_trigger(double , sc_time_unit , sc_event_or_list const &);
CHANGE: void next_trigger(const sc_time& , sc_event_and_list const &);
CHANGE: void next_trigger(double , sc_time_unit , sc_event_and_list const &);



JEITA

CHANGE
6.2.18 wait

- CHANGE: void wait(sc_event_or_list const &);
- The argument shall take the form of a list of events separated by the operator| of classes sc_event and sc_event_or_list. The process shall be resumed when any one of the given events is notified. The occurrence or non-occurrence of the other events in the list shall have no effect on the resumption of that particular process. If a particular event appears more than once in the list, the behavior shall be the same as if it appeared only once. It shall be an error to pass an empty event list object as an argument to wait (see 6.8 and 6.9).
- 引数はsc_event_or_listかsc_eventのオペレータ“｜”によって切り離されたイベントのリストの型を取るものとする。特定のイベントのどれかが通知されるとき、プロセスは再開されるものとする。リストの他のイベントの発生または非発生は、その特定のプロセスの再開のときに効き目がないものとする。リストの中の特定のイベントが一度より多く現れると、まるでそのイベントが一度だけ現れた時と同じ振舞いになるだろう。それは、waitの引数に空のイベントリストオブジェクトを渡すとエラーになるだろう(6.8と6.9を参照) 。

JEITA



- CHANGE: void **wait**(sc_event_and_list const &);

  The argument shall take the form of a list of events separated by the operator& of classes sc_event and sc_event_and_list. In order for the process to be resumed, every single one of the given events shall be notified, with no explicit constraints on the time or order of those notifications. The process is resumed when the last such event is notified, last in the sense of being at the latest point in simulation time, not last in the list. An event in the list may be notified more than once before the last event is notified. If a particular event appears more than once in the list, the behavior shall be the same as if it appeared only once. It shall be an error to pass an empty event list object as an argument to wait (see 6.8 and 6.9).

  引数はsc_event_and_listかsc_eventのオペレータ“＆”によって切り離されたイベントのリストの型を取るものとする。プロセスが再開されるために、特定のイベントのすべてがそれらの通知の時間または順序のときに明白な制約なしで通知されるものとする。プロセスはlastのようなイベントが通知されたとき、シミュレーション時間の最新のポイントにあるという意味での最終であるとき、またはリストのlastでないときにプロセスは再開される。リストにおけるイベントはlastのイベントが通知される前に、一度よりも多く通知されるかもしれない。リストの中の特定のイベントが一度より多く現れると、まるでそのイベントが一度だけ現れた時と同じ振舞いになるだろう。それは、waitの引数に空のイベントリストオブジェクトを渡すとエラーになるだろう(6.8と6.9を参照) 。

JEITA



- void **wait**(const sc_time&);

  The process shall be resumed after the time given as an argument has elapsed. The time shall be taken to be relative to the time at which function wait is called. When a process is resumed in this way, a time-out is said to have occurred. CLARIFICATION: In this method call and in those that follow below, the argument that specifies the time-out shall not be negative.

  プロセスは引数で与えられた時間が経過した後に再開されるものとする。時間は、waitが呼ばれる時間に関連してかかるものとする。プロセスがこのように再開されるとき、タイムアウトが起こったと言われる。このメソッドの呼び出しと以下に続くものでは、タイムアウトを指定する引数は、負の値にしてはいけない。

JEITA

void wait(double , sc_time_unit , const sc_event&);
CHANGE: void wait(const sc_time& , sc_event_or_list const &);
CHANGE: void wait(double , sc_time_unit , sc_event_or_list const &);
CHANGE: void wait(const sc_time& , sc_event_and_list const &);
CHANGE: void wait(double , sc_time_unit , sc_event_and_list const &);



JEITA

6.3.4 Module Hierarchy

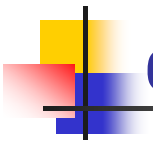
To keep track of the module hierarchy during elaboration, the implementation may maintain an internal stack of pointers to objects of class sc_module_name, referred to below as *the stack*. *ADDITION: For the* purpose of building hierarchical names, when objects of class sc_module, sc_port, sc_export, sc_prim_channel, or sc_event are constructed or when spawned or unspawned processes instances are created, they are assumed to exist within the module identified by the sc_module_name object on the top of the stack. In other words, each instance in the module hierarchy is named as if it were a child of the module identified by the item on the top of the stack at the point when the instance is created.

階層名をビルドする目的でsc_module, sc_port, sc_export, sc_prim_channel以外にsc_eventを追加しています。

ADDITION: NOTE 3—An object of type sc_event may have a hierarchical name and may have a parent in the object hierarchy, but in any case sc_event is not derived from sc_object and events are not themselves part of the object hierarchy.

sc_eventタイプのオブジェクトは、階層名を持っている可能性があり、オブジェクトの階層内の親があるかもしれませんが、どのような場合でもsc_eventがsc_objectからの派生でなく、イベント自体はオブジェクト階層の一部ではありません。

JEITA



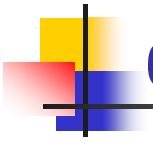
6.5.2 Class definition

sc_spawn_optionsクラスに以下のようにreset関数を新規に追加されています。

ADDITION:
```
void reset_signal_is ( const sc_in<bool>& , bool );
void reset_signal_is ( const sc_inout<bool>& , bool );
void reset_signal_is ( const sc_out<bool>& , bool );
void reset_signal_is ( const sc_signal_in_if<bool>& , bool );
void async_reset_signal_is ( const sc_in<bool>& , bool );
void async_reset_signal_is ( const sc_inout<bool>& , bool );
void async_reset_signal_is ( const sc_out<bool>& , bool );
void async_reset_signal_is ( const sc_signal_in_if<bool>& , bool );
```

・bool型　:x及びzの信号は使えない。
・マルチポートのサポート

JEITA



6.5.2 Class definition

ADDITION:
```
namespace sc_unnamed {
implementation-defined _1;
implementation-defined _2;
implementation-defined _3;
implementation-defined _4;
implementation-defined _5;
implementation-defined _6;
implementation-defined _7;
implementation-defined _8;
implementation-defined _9;
} // namespace sc_unnamed
```

実装依存のnamespace sc_unnamedの定義を追加

JEITA

6.5.5 Member functions

メンバー関数の追加によりその説明の追加しています。

ADDITION:
```
void reset_signal_is ( const sc_in<bool>& , bool );
void reset_signal_is ( const sc_inout<bool>& , bool );
void reset_signal_is ( const sc_out<bool>& , bool );
void reset_signal_is ( const sc_signal_in_if<bool>& , bool );
void async_reset_signal_is ( const sc_in<bool>& , bool );
void async_reset_signal_is ( const sc_inout<bool>& , bool );
void async_reset_signal_is ( const sc_out<bool>& , bool );
void async_reset_signal_is ( const sc_signal_in_if<bool>& , bool );
```

- reset_signal_isとasync_reset_signal_isはそれぞれ生成されたプロセス・インスタンスの同期または非同期リセット信号の引数として渡されたオブジェクトを追加するにはspawnオプションのプロパティを設定が必要。
- メソッドを呼び出すたびに、指定されたspawnオプション・オブジェクトにリセット信号を追加する。
- シグナルは、リセット信号にバインドされているポート・インスタンスを渡すことによって間接的に識別できる。
- スパンされたプロセス・インスタンスは、同期および非同期リセット信号をいくつか持ってもよい。

JEITA

6.5.6 sc_spawn

CLARIFICATION: It is the responsibility of the application to ensure that the memory location is still valid when the spawned function returns. For example, the memory location could be a data member of an enclosing sc_module, but should not be a stack variable that would have been deallocated by the time the spawned function returns.

アプリケーションがメモリの管理の責任をもつ
sc_spawnのT::result_type*型の引数は、プロセス・インスタンスに関連付けられている関数から返される値としてメモリ・ロケーションのポインタを渡すが、そのメモリ管理については生成された関数から戻るときにメモリ・ロケーションがまだ有効であることを保証するのはアプリケーションの責任である。
たとえば、メモリ・ロケーションは、閉じたsc_moduleのデータメンバであってもよいが、生成された関数がリターン(戻り)ときに解放されるスタック変数を使用してはいけない。

JEITA

6.5.6 sc_spawn（続き）

ADDITION: The only purpose of namespace **sc_unnamed is to allow the implementation to provide a set of** argument placeholders for use with **sc_bind. _1, _2, _3 ... shall be provided by the implementation to give access to the placeholders of the same names from the Boost libraries. These placeholders can be passed as arguments to sc_bind in order to defer the binding of function arguments until the call to the function object returned from sc_bind. Again, see the Boost documentation for details.**

6．5．2のネームスペースsc_unameはBoostライブラリから同じ名前のプレースフォルダへのアクセスを可能にするための実装について

・ ネームスペースsc_unnamedの唯一の目的は、実装がsc_bindで使用するための引数のプレースホルダのセットを提供できるようにする。
・_1、_2、_3．．．はBoostライブラリから同じ名前のプレースホルダへのアクセスを与えるための実装によって提供される。
・これらのプレースホルダは、sc_bindから返された関数オブジェクトに対する呼び出しまでに関数の引数のバインディング（結合）を延期するためにsc_bindへ引数として渡すことができる。

JEITA

6.5.6 sc_spawn（続き）

CLARIFICATION: Function sc_spawn shall return a valid process handle to the spawned process instance.

ADDITION: A process instance can be suspended, resumed, disabled, enabled, killed, or reset using the member functions of class sc_process_handle.

sc_spawnの戻り値は有効なプロセス・ハンドルとした定義より、
sc_process_handleの利用によりプロセスをコントロール可能

・sc_spawn関数は、スパンされたプロセス・インスタンスへ有効なプロセのスハンドルを返さなければならない。

・sc_process_handleクラスのメンバ関数を利用することでプロセス・インスタンスは、サスペンド、レジューム、ディスイネーブル、イネーブル、キル、あるいはリセットができる。

JEITA

6.5.7 SC_FORK and SC_JOIN

The text between SC_FORK and SC_JOIN shall consist of a series of one or more calls to function sc_spawn separated by commas. CLARIFICATION: The value returned from the function call may be discarded or the function call may be the only expression on the right-hand-side of an assignment to a variable, where the variable will be set to the process handle returned from sc_spawn.
The comma after the final call to sc_spawn and immediately before SC_JOIN shall be optional. There shall be no other characters other than white space separating SC_FORK, the function calls CLARIFICATION: or variable assignments, the commas, and SC_JOIN. If an application violates these rules, the effect shall be undefined.

ここは次ページのExampleにある実装について説明を追加
- SC_FORKとSC_JOINの間の記述は、カンマで区切られたsc_spawn機能に1つまたは複数の一連の呼び出しで構成
- CLARIFICATION：関数呼び出しから返された値は破棄されることがある。あるいは関数の呼び出しは、変数へ代入した右辺の単なる式でよい。その変数はsc_spawnから返されたハンドルをプロセスに設定される変数である。
- sc_spawnへの最終的な呼び出しの後にコンマとSC_JOIN前のコンマは任意。
- 関数コール、 CLARIFICATION: あるいは変数の割り当て、コンマ、SC_JOIN、SC_FORKを分離する他の空白文字以外があってはならない。

JEITA

6.5.7 SC_FORK and SC_JOIN

CLARIFICATION:Example 2:

```
sc_process_handle h1, h2, h3;

SC_FORK
h1 = sc_spawn ( arguments ) ,
h2 = sc_spawn ( arguments ) ,
h3 = sc_spawn ( arguments )
SC_JOIN
```

プロセス・ハンドルの記述例

JEITA

6.6 sc_process_handle

6.6.1 Description

CLARIFICATION: An invalid process handle may be associated with a single underlying process instance provided that process instance has terminated. An empty process handle is an invalid handle that is not associated with any underlying process instance. In other words, an invalid process handle is either an empty handle or is associated with a terminated process instance.

sc_process_handleの無効なプロセス・ハンドルについて

・　無効なプロセス・ハンドルはターミネートしたプロセスや空プロセスと定義
プロセス・インスタンスが終了すれば、無効なプロセス・ハンドルが１つの基本プロセス・インスタンスに関連付けられているかもしれない。

・空のハンドル・プロセスは、基本のプロセス・インスタンスに関連付けられていない無効なハンドルである。

JEITA

6.6 sc_process_handle（つづき）

ADDITION: The provision of operator<, which supplies a strict weak ordering relation on the underlying process instances, allows process handles to be stored in a standard C++ container such as std::map.

プロセスの関係についてoperator<を定義し、プロセス依存関係をみるために機能を追加

JEITA

6.6.2 Class definition

6.6.2 Class definition
ADDITION:
```
enum sc_descendant_inclusion_info
{
    SC_NO_DESCENDANTS,
    SC_INCLUDE_DESCENDANTS
};

class sc_unwind_exception: public std::exception
{
public:
   virtual const char* what () const throw ();
   virtual bool is_reset () const;
protected:
   sc_unwind_exception ();
   sc_unwind_exception ( const sc_unwind_exception& );
   virtual ~sc_unwind_exception ();
};
```

新規にenumや処理をプロセスハンドルクラスに追加

JEITA

6.6.2 Class definition(つづき)

```
class sc_process_handle
{
省略
ADDITION: bool operator< ( const sc_process_handle& ) const;
ADDITION: void swap ( sc_process_handle& );
省略
ADDITION:
        void suspend ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void resume ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void enable ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void kill ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void reset ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        bool is_unwinding () const;
        const sc_event& reset_event () const;
        void sync_reset_on
                ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void sync_reset_off
                ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        template <typename T>
                void throw_it ( const T& user_defined_exception,
                        sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );

ADDITION: bool sc_is_unwinding ();
};
```

新規に処理をプロセスハンドルクラスに追加

JEITA

6.6.4 Constructors

sc_process_handle ();

CLARIFICATION: The default constructor shall create an empty process handle. An empty process handle shall be an invalid process handle.

sc_process_handle (const sc_process_handle&);
The copy constructor shall duplicate the process handle passed as an argument. CLARIFICATION: The result will be two handles to the same underlying process instance or two empty handles.

explicit sc_process_handle (sc_object*);
If the argument is a pointer to a process instance, this constructor shall create a valid process handle to the given process instance. CLARIFICATION:Otherwise, this constructor shall create an empty process handle.

空プロセスの定義を追加。プロセス・ハンドル作成時や無効なプロセス・ハンドルなどが空プロセス・ハンドルとなる

JEITA

6.6.5 Member functions

sc_process_handle& operator= (const sc_process_handle&);
The assignment operator shall duplicate the process handle passed as an argument.
CLARIFICATION: The result will be two handles to the same underlying process instance or two empty handles.

Operator=でプロセス・ハンドルのコピーでは空プロセス・ハンドルとなる場合もあると定義。

ADDITION: bool operator< (const sc_process_handle&) const;
The less-than operator shall define a strict weak ordering relation on the underlying process instances in the following sense. Given three process handles H1, H2 and H3:
strict means that H1 < H1 shall return false (whether H1 is valid, invalid, or empty) weak means that if H1 and H2 are both handles (valid or invalid) associated with the same underlying process instance or both are empty handles, then H1 < H2 shall return false and H2 < H1 shall return false. Otherwise, if H1 and H2 are handles associated with different underlying process instances or if only one is an empty handle, then exactly one of H1 < H2 and H2 < H1 shall return true. ordering relation means that if H1 < H2 and H2 < H3 then H1 < H3 (whether H1, H2, or H3 are valid or invalid).

プロセスの順序関係のために新規にメンバ関数Operator<の説明追加。

JEITA

6.6.5 Member functions(つづき)

bool operator< (const sc_process_handle&) const; についてハンドルプロセス処理のコード例を追加されている。

```
Example:
sc_process_handle a, b; // Two empty handles

sc_assert ( !a.valid () && !b.valid () ); // Both are invalid
sc_assert ( a != b );
sc_assert ( !(a < b) && !(b < a) );
a = sc_spawn (...);
b = sc_spawn (...);
sc_assert ( a != b );
sc_assert ( (a < b) || (b < a) ); // Two handles to different processes
sc_process_handle c = b;
sc_assert ( b == c );
sc_assert ( !(b < c) && !(c < b) ); // Two handles to the same process
wait ( a.terminated_event () & b.terminated_event () );
sc_assert ( (a < b) || (b < a) ); // Same ordering whether handles are valid or not
if ( b.valid () ) // Handles may or may not have been invalidated
    sc_assert ( b == c );
else
    sc_assert ( b != c );
sc_assert ( b.valid () == c.valid () ); // Invalidation is consistent
sc_assert ( !(b < c) && !(c < b) ); // Two handles to the same process, whether valid or not
sc_assert ( c.terminated () );
```

JEITA

6.6.5 Member functions(つづき)

```
ADDITION: void swap ( sc_process_handle& );
Member function swap shall exchange the process instances underlying the process handles *this
and the sc_process_handle argument. If H1 < H2 prior to the call H1.swap (H2), then H2 < H1 after
the call. Either handle may be invalid.
Example:
sc_process_handle a, b = sc_get_current_process_handle ();
sc_assert ( b.valid () );
a.swap ( b );
sc_assert ( a == sc_get_current_process_handle () );
sc_assert ( !b.valid () );
```

ハンドリングするプロセス・インスタンスを切り替えるためにswapのメンバー関数を新規に追加

const char* name () const;
Member function name shall return the hierarchical name of the underlying process instance.
CLARIFICATION: If the process handle is invalid, member function name shall return an empty string, that is, a pointer to the string "". The implementation is only obliged to keep the returned string valid while the process handle is valid.

空の文字列""のポインタの明記

JEITA

6.6.5 Member functions(つづき)

const std::vector<sc_object*>& get_child_objects () const;
Member function **get_child_objects shall return a std::vector containing a pointer to every instance** of class **sc_object that is a child of the underlying process instance.**
CLARIFICATION: This shall include every dynamic process instance that has been spawned during the execution of the underlying process instance and has not yet been deleted, and any other application-defined objects derived from class sc_object created during the execution of the underlying process instance that have not yet been deleted.

get_child_object ()は実行中のプロセス・インスタンスの子へのポインターを含むstd::vectorを戻しますが、それには全インスタンスへ削除されていないあるいは実行中のダイナミック・プロセスも含むと明記。

JEITA

6.6.5 Member functions(つづき)

ADDITION: const std::vector<sc_event*>& **get_child_events () const;**
Member function **get_child_events shall return a std::vector containing a pointer to every object of type sc_event that is a hierarchically named event and whose parent is the current process instance.** If the process handle is invalid, member function **get_child_events shall return an empty std::vector.**
This same function shall be overridden in any implementation-defined classes derived from **sc_object and associated with spawned and unspawned process instances.**

新規にメンバー関数get_child_events( )が追加。
- 親が現プロセス・インスタンスとなる階層的に命名されたイベント(sc_event)のあらゆるオブジェクトへのポインタを含むstd:vectorを返す。
- プロセス・ハンドルが無効ならば、空のstd::vectorを返す。
- 同じ関数であればどんな実装においてもオーバーライドされるべきである。

JEITA

6.6.5 Member functions(つづき)

sc_object* get_parent_object () const;

Member function **get_parent_object shall return a pointer to the module instance or process** instance from which the underlying process instance was spawned. If the process handle is invalid, member function **get_parent_object shall return the null pointer.** CLARIFICATION: **If the parent** object is a process instance and that process has terminated, **get_parent_object shall return a** pointer to that process instance. A process instance shall not be deleted (nor any associated process handles invalidated) while the process has surviving children, but may be deleted once all its child objects have been deleted.

親プロセスがターミネートした場合、そのプロセスが子供を残している間は get_parent_objectはそのプロセス・インスタンスへのポインタを戻し、削除されないが、すべての子のオブジェクトが削除されると削除されるかもしれない。

JEITA

6.6.5 Member functions(つづき)

sc_object* **get_process_object () const;**
CLARIFICATION: If the process handle is valid, member function **get_process_object shall return** a pointer to the process instance associated with the process handle. If the process handle is invalid, member function **get_process_object shall return the null pointer. An application should test for a** null pointer before dereferencing the pointer. Moreover, an application should assume that the pointer remains valid only until the calling process suspends.

get_process_object () はnullを返さない場合の定義として「プロセス・ハンドルが有効ならば」という条件を明記

JEITA

6.6.5 Member functions(つづき)

bool terminated () const;
Member function **terminated shall return true if and only if the underlying process instance has***terminated. A thread or clocked thread process is terminated after the point when control is returned* from the associated function.
CHANGE: A process can also be *terminated by calling the **kill method*** of a process handle associated with that process. This is the only way in which a method process can be terminated. If the process handle is invalid, member function **terminated shall return the value**

・プロセスはKillメソッドでターミネートされ、メソッド・プロセスがターミネートされる唯一の方法
・プロセス・ハンドルが無効ならば、ターミネートされたメンバー関数は値(ポイント)を返す

6.6.5 Member functions(つづき)

When the underlying process instance terminates, CLARIFICATION: if the process instance has no surviving children, an implementation may choose to invalidate any associated process handles, but is not obliged to do so.
CLARIFICATION: An implementation shall not invalidate a process handle while the process instance has child objects. However, if an implementation chooses to invalidate a process handle, it shall invalidate every process handle associated with the underlying process instance at that time. In other words, it shall not be possible to have a valid and an invalid handle to the same underlying process instance in existence at the same time

・プロセス・インスタンスに生き残っている子がいないなら基本的なプロセス・インスタンスが終わるとき、実装では、どんな関連プロセスハンドルも無効にすることを選ぶかもしれませんが、そうするのは義務ではありません。
・実装ではプロセス・インスタンスには子のオブジェクトがある間、プロセス・ハンドルを無効にしてはならない。しかしながら、実装でプロセス・ハンドルを無効にしたならば、それは、その時、基本的なプロセス・インスタンスに関連しているあらゆるプロセスハンドルを無効にするべきである。

JEITA

6.6.5 Member functions(つづき)

If the process handle is valid, function **terminated will return true, or if invalid, terminated will return false.**
CLARIFICATION: The process instance shall continue to exist as long as the process handle is valid. Once all the handles associated with a given process instance have been invalidated, an implementation is free to delete the process instance, but is not obliged to do so. With respect to the ordering relation that defines the behavior of **operator< and method swap, in effect an invalid process handle remains** associated with a process instance even after that process instance has been deleted.

プロセスハンドルが有効であるなら、ターミネートしたファンクションは、Trueを返すか、または無効であるなら、falseを返す。
プロセス・ハンドルが有効である限り、プロセス・インスタンスは存続するものとします。 特定のプロセス・インスタンスに関連しているすべてのハンドルが一旦無効にされると、実装では、プロセス・インスタンスを削除することは自由ですが、そうすることは義務ではない。オペレータ<の振舞いとメソッドのswapを定義する順序関係に関してそのプロセス・インスタンスが削除された後にさえ、事実上無効なプロセス・ハンドルはプロセス・インスタンスに関連したままです。

JEITA

6.6.6 ADDITION: Member functions for process control

プロセス・コントロールのメンバ関数を定義

The member functions of class **sc_process_handle described in this clause and its subclauses are concerned** with process control. Examples of process control include suspending, resuming, killing or resetting a process instance. Process control involves a *calling process controlling a target process. In each case, the calling process calls a member function of a process handle associated with the target process.*
The calling process and the target process may be distinct process instances or may be the same process instance. In the latter case, certain process control member functions are meaningless, such as having a process instance attempt to resume itself.
Several of the process control member functions are organized as complementary pairs:
— **suspend and resume**
— **disable and enable**
— **sync_reset_on and sync_reset_off**
— **kill**
— **reset**
— **throw_it**
Each of the above member functions may be called where the underlying process instance is a method process, a thread process, or a clocked thread process. There are certain caveats concerning the use of **suspend, resume, reset, and throw_it with clocked thread processes (see 6.2.12).**

プロセスを制御するためのメンバ関数について定義
suspend, resume, disable, enable, sync_reset_on, sync_reset_off,
kill, reset, throw_it

JEITA

6.6.6 ADDITION: Member functions for process control（つづき）

プロセスコントロールのメンバ関数を定義

The distinction between **suspend/resume and disable/enable lies in the sensitivity of the target process** during the period while it is suspended or disabled. With **suspend, the kernel keeps track of the sensitivity of** the target process while it is suspended such that a relevant event notification or time-out while suspended would cause the process to become runnable immediately **resume is called. With disable, the sensitivity of** the target process is nullified while it is suspended such that the process is not made runnable by the call to **enable, but only on the next relevant event notification or time-out subsequent to the call to enable. In other** words, with **suspend/resume the kernel keeps a record of whether the target process would have awoken** while in fact being suspended, whereas with **disable/enable, the kernel entirely ignores the sensitivity of the** target process while disabled. Also see 6.2.12 regarding clocked thread processes.

Example: *コードについては本文参照してください。*

suspend/resume, disable/enableはセシビリティの違いがある。
・サスペンド/レジューム：カーネルはサスペンドされている間ターゲット・プロセスが起きているかどうかを記録の保持をする。
・ディスイネーブル/イネーブル：カーネルはディスイネーブルの間ターゲット・プロセスのセンシビリティを完全に無視する。

JEITA

6.6.6 ADDITION: Member functions for process control（つづき）

sync_reset_on and sync_reset_off provide a procedural way of putting a process instance into and out of the *synchronous reset state (see 6.6.6.3). This is equivalent in effect to having specified a reset signal using* the member function **reset_signal_is of class sc_module or sc_spawn_options and that reset signal** attaining its active value or the negation of its active value, respectively.

Example: *コードについては本文参照してください。*

sync_reset_onとsync_reset_off：
プロセス・インスタンスを同期リセット状態にするあるいはしない手続きを提供する。

JEITA

6.6.6 ADDITION: Member functions for process control（つづき）

kill interrupts and irrevocably terminates the target process. reset interrupts the target process and, in the case of a thread process, calls the associated function again from the top. **kill and reset both cause an** exception to be thrown within the target process, and that exception may be caught and handled by the application. Both have immediate semantics such that the target process is killed or reset before return from the function call.

Example: ***コードについては本文参照してください。***

A process may be reset in several ways: by a synchronous or asynchronous reset signal specified using **reset_signal_is or async_reset_signal_is respectively (see 6.2.13), by calling sync_reset_on, or by calling reset. Whichever alternative is used, the reset action itself is ultimately equivalent to a call to reset.**

•Kill：ターゲット・プロセスに割り込み、取消ができないターミネートする
•Reset：ターゲット・プロセスを割り込み、スレッド・プロセスの場合、
トップから再び関連する関数を呼び出す。
•共通：ターゲット・プロセス内に例外を引き起こし、その例外はアプリケーションによりキャチあるいはハンドルしてよい。
リセットは以下の呼び出しとなる。
reset_signal_is、async_reset_signal_isasync_reset_signal_is、reset

JEITA

6.6.6 ADDITION: Member functions for process control（つづき）

throw_it permits a user-defined exception to be thrown within the target process.

For each of the nine member functions for process control described in this clause and its subclauses, if the process handle is invalid, the implementation shall generate a warning and the member function shall return immediately without having any other effect.

throw_it：ターゲット・プロセス内にユーザ定義の例外を投げてよい
プロセス・ハンドルが無効であれば実装でワーニングを発生するものとする。

JEITA

6.6.6 ADDITION: Member functions for process control（つづき）

In this clause, the phrase *during elaboration or before the process has first executed shall encompass the* following cases:
— At any time during elaboration
— From one of the callbacks **before_end_of_elaboration, end_of_elaboration, or start_of_simulation**
— During simulation but before the given process instance has executed for the first time

In each case, if the **include_descendants argument has the value SC_INCLUDE_DESCENDANTS, the** member function shall be applied to the target process instance and recursively in bottom-up order to all its children in the object hierarchy that are also process instances. In other words, the member function shall be applied to the children, grandchildren, great grandchildren, and so forth, of the target process instance, starting with the deepest descendant. If the **include_descendants argument has the value SC_NO_DESCENDANTS, the member function shall only be applied to the target process instance. There** are no restrictions concerning how this argument may be used across pairs of calls to **suspend-resume, disable-enable, or sync_reset_on-sync_reset_off. For example, it is permitted to suspend a process and all** of its descendants but only resume the process itself.

The member functions for process control shall not be called during the update phase.

Beware that the rules given in the following subclauses are to be understood alongside the rules given in 6.6.6.11 concerning the interaction between the member functions for process control, which shall take precedence where appropriate.

JEITA

6.6.6 ADDITION: Member functions for process control（つづき）

この節では、エラボレーション間あるいはプロセスが初期実行される前のフェーズ
- エラボレーションの間
- before_end_of_elaboration, end_of_elaboration,あるいはstart_of_simulationの1つからコールバック
-特定のプロセス・インスタンスが最初に実行される前のシミュレーションの間
このケースでinclude_descendantsの引数がSC_INCLIDE_DESENDANTSを持つならば、メンバ関数は、ターゲット・プロセス・インスタンスのもっとも深い子孫から開始して子、孫、ひ孫などに適用される。
include_descendantsの引数がSC_NO_DESCENDANTSであれば、そのメンバ関数はターゲット・プロセス・インスタンスに単に適用される。
プロセス・コントロールに関するメンバ関数はアップデートのフェーズの間呼ばれない。

JEITA

ADDITION
6.6.6.1 Member functions to suspend and resume processes

void suspend (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

- プロセスの実行を中断する。
- resumeが呼ばれて再開されるまで実行可能にならない。
- プロセスが動作可能プロセス・セットに入っている場合は、そのプロセスはそのセットからただちに削除されなければならない。そのプロセスは現在の評価フェーズでは実行されない。
- 中断中にイベント通知、タイムアウトが発生した時、resumeが呼ばれた次の評価フェーズで動作可能にならなければならない。
- 中断しているプロセスに対してsuspendを呼んでも、そのプロセスに影響があってはならない。(子プロセスが中断される場合あり。)
- 中断しているプロセスの再開には、ただ一つのresume呼び出しが要求される。
- メソッドが中断されるとき、そのメソッドに関連した関数は、制御がカーネルに戻る前に、終わりまで実行されなければならない。
- スレッドが中断されるとき、suspendからその関連した関数へ制御が戻ることなく、プロセスは直ちに中断されなければならない。
- すでに終了しているプロセスに対してsuspendを呼んでも、そのプロセスに影響があってはならない。(終了していない子プロセスが中断される場合あり。)
- エラボレーション中もしくはプロセスが初めて実行される前のsuspend呼び出しは許されている。dont_initializeが有効のとき、実装は事実上event_and_list operator&を使ってセンシティビティに暗黙のイベントを追加しなければならない。また、resumeが呼ばれた時、このイベントに対して暗黙の通知を生成しなければならない。そうでなければ、実装はresumeが呼ばれたとき、プロセスを直ちに実行可能にしなければならない。事実上、プロセスが再開されるまで、初期化は延期されることになる。

JEITA

ADDITION
6.6.6.1 Member functions to suspend and resume processes

void resume (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

- suspendの効果を取り除く。
- 中断中にそのプロセスのセンシティビティがそれを動作可能にするときにのみ、現在の評価フェーズで実行可能にならなければならない。
- プロセスが中断されたときに動作可能プロセス・セットに入っていた場合、resumeはプロセスを現在の評価フェーズで動作可能にしなければならない。
- resumeの前にリセットがある場合、リセット前に起きたイベント通知、タイムアウトは、resumeの効果を決める時、無視されなければならない。
- スレッドはsuspend呼び出しの次の実行可能ステートメントから再開されなければならない。
- メソッドは関連する関数を呼び出すことにより(？)再開されなければならない。
- 中断していないプロセスに対してresumeを呼んでも、そのプロセスに影響があってはならない。(子プロセスが再開される場合あり。)
- 終了したプロセスに対してresumeを呼んでも、そのプロセスに影響があってはならない。(終了していない子プロセスが再開される場合あり。)
- エラボレーション中もしくはプロセスが初めて実行される前のresume呼び出しは許されている。その際、上記のルールが適用されなければならない。
- プロセスと関連ある関数はsuspend、resumeを呼び出すことができる。プロセスは同じプロセスによって中断と再開されねばならないということはない。

JEITA

ADDITION
6.6.6.1 Member functions to suspend and resume processes

void disable (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

- プロセス・インスタンスを動作不可にする。
- enableが呼ばれて動作可能になるまで実行可能にならない。
- 動作不可にされたプロセスが実行可能プロセス・セットに入っている場合、そのセットから取り除かれないで、現在の評価フェーズで実行されねばならない。
- 動作不可中に、このプロセスのセンシティブなイベント通知またはタイムアウトは無視されなければならない。
- 動作不可プロセスに対してdisableを呼んでも、そのプロセスに効果があってはならない。(子プロセスが動作不可になる場合あり。)
- 動作不可プロセスを動作可能にするには、ただ一回のenableの呼び出しのみが要求される。
- プロセスがそれ自身を動作不可にした場合、メソッド、スレッドにかかわらず、それに関連した関数はwait呼び出しポイント、または、カーネルに制御が戻る前の関数の終了ポイントまで実行されなければならない。
- 終了したプロセスに対してdisableを呼んでも、そのプロセスに効果があってはならない。(終了していない子プロセスが動作不可になる場合あり。)
- エラボレーション中もしくはプロセスが初めて実行される前にdisableを呼ぶことは許されている。disableが呼ばれたそのときにプロセスが既に実行可能プロセス・セットに入っている場合、たとえば初期化中、そのプロセスの実行は許可されなければならない。

ADDITION
6.6.6.1 Member functions to suspend and resume processes

void enable (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

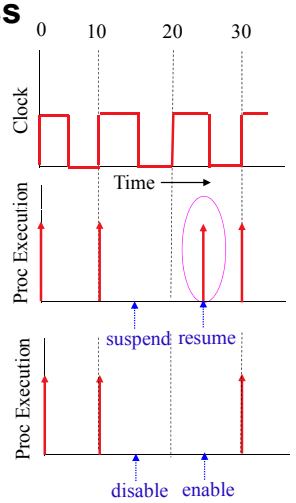
- disableの効果を取り除く。
- そのプロセスはenable後のイベント通知またはタイムアウトによって実行可能になる。resumeとは異なり、動作不可中に発生したイベント通知もしくはタイムアウトにより実行可能になることがあってはならない。
- 動作不可中にタイムアウトが発生し、そのプロセスがタイムアウト以外のイベントにセンシティブでない場合、そのプロセス・インスタンスは二度と実行されない(そのイベントがリセットでない限り)。実装はwarningを出してもよい。
- 動作可能になったプロセスが次に実行されるとき、それがスレッドならば、動作不可になったwait呼び出しの次の命令文から実行されなければならない。それがメソッドならば、その関連した関数(？)を呼び出すことにより実行されなければならない。
- 動作不可でないプロセスに対してenableを呼んでも、そのプロセスに影響があってはならない。(動作不可の子プロセスが動作可能になる場合あり。)
- 終了したプロセスに対してenableを呼んでも、そのプロセスに影響があってはならない。(終了していない子プロセスが動作可能になる場合あり。)
- エラボレーション中もしくはプロセスが初めて実行される前にenableを呼び出すことは許されている。その際、上記のルールが適用されなければならない。
- プロセスと関連ある関数はdisableもしくはenableを呼び出すことができる。同じプロセスによってプロセスは動作不可もしくは動作可能にされなければならないというわけではない。

JEITA

supend/resumeとdisable/enableの違い

Disabling and Enabling a Process

- `proc_handle.disable()` disables process
  - does not run again till process is enabled
  - {new effective sensitivity} = NULL
  - does not remember effective sensitivity triggering while disabled
- `proc_handle.enable()` lifts any previous disables
  - has no effect if process was not disabled
  - restores {old effective sensitivity}
- If effective sensitivity triggered while process was disabled, `enable()` will NOT execute process



cadence

NASCUG, Feb 21, 2007 Cadenceのプレゼン資料より JEITA

supend/resumeとdisable/enableの違い

	動作	効果が表れるポイント（呼び出し時点を起点）	動作の違い
suspend	プロセスの実行を中断 thread(自分自身):即時中断 method(自分自身):関数の最後まで実行	現在の評価フェーズ	runnable setから直ちに削除
disable	プロセスを実行不可にする 自分自身:wait、カーネルに制御が戻る関数の最後まで実行	次の評価フェーズ	runnable setに入っているときは現在の評価フェーズでは実行
resume	suspendされたプロセスを再開 thread:suspend呼び出しの次の文から再開 method:その関数呼び出しから再開(？)	現在の評価フェーズ	中断中にイベント通知、タイムアウトあったとき resumeされたときの評価フェーズで実行 中断時にrunnable setに入っていた時は、resumeされたときの評価フェーズで実行
enable	disableの効果を取り除く thread:wait呼び出しの次の文から実行 method:その関数呼び出しから実行(？)	次の評価フェーズ	disable中のイベント通知、タイムアウトは無視される enable後のイベント通知、タイムアウトでrunnableになる

JEITA

ADDITION
6.6.6.2 Member functions to kill and reset processes

- プロセスは3つの方法のひとつでリセットされる。
  - プロセス・ハンドルのメンバー関数resetを呼び出す。このとき、プロセスは直ちにリセットされなければならない。
  - 同期リセット状態にあるプロセスを再開する。
  - async_reset_signal_isにより指定されたリセット信号がリセット値になる。その時点でresetが呼ばれる。
- プロセス、関連したプロセス・ハンドルのメンバー関数resetとasync_reset_onの呼び出しターゲットでもあることにくわえて、reset_signal_isとasync_reset_signal_isを使って複数個のリセットを持つことができる。
- プロセスは3つの方法のひとつで同期リセット状態になれる。
  - プロセス・ハンドルのメンバー関数sync_reset_onを呼び出す。
  - プロセスのreset_signal_isで指定されたどれかのリセット信号が有効な値になる。
  - プロセスのasync_reset_signal_isで指定されたどれかのリセット信号が有効な値になる。
- プロセス・インスタンスは以下のすべての条件が満たされる時、同期リセット状態から出ることができる。
  - プロセス・ハンドルのメンバー関数sync_reset_offが呼ばれる、もしくは、メンバー関数sync_reset_onがまだ呼ばれていない。
  - プロセスのreset_signal_isで指定されたすべてのリセット信号が有効な値の否定になっている。
  - プロセスのasync_reset_signal_isで指定されたすべてのリセット信号が有効な値の否定になっている。

JEITA

ADDITION
6.6.6.2 Member functions to kill and reset processes

void kill (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

- プロセスを強制終了させる。
- 強制終了されたプロセスに関してsc_unwind_exceptionが投げられなければならない。
- 強制終了されたプロセスは現在のシミュレーション中に再び実行可能になってはいけない。強制終了されたプロセスは終了しなければならない。
- waitが呼ばれてスレッドが中断している場合、もしくは、スレッドもしくはメソッドが自分自身を強制終了している場合、sc_unwind_exceptionは関連した関数のコール・スタックの巻き戻しと局所オブジェクトのデストラクタ呼び出しを引き起こさなければならない。メソッドの場合は、プロセスが自分自身を強制終了しない限り、関連した関数は巻き戻すべきコール・スタックを持たない。しかし、killの呼び出しはメソッドの終了を引き起こさなければならない。
- 強制終了の例外が投げられた時、sc_unwind_exceptionクラスのメンバー関数is_resetは値falseを返さなければならない。
- killは直ちに効果を生じなければならない。強制終了されたプロセスは実行可能プロセス・セットから取り除かれなければならない。コール・スタックは巻き戻されなければならない。プロセスを終了状態にしなければならない。kill関数から戻る前に終了イベントを通知しなければならない。プロセスが他のプロセスを強制終了したとき、制御はkillを呼んだ関数に戻らなければならない。プロセスが自分自身を強制終了したとき、kill呼び出し以降の命令文は現在のシミュレーション中は実行されることはない。制御はカーネルに戻らなければならない。
- プロセスが強制終了されているとき、コール・スタックの巻き戻しは局所オブジェクトのデストラクタに呼び出しを引き起こすかもしれない。このようなデストラクタは直接間接にかかわらず、強制終了されているプロセスのwaitもしくはnext_triggerを呼び出してはいけない。しかし、他のプロセス、初めにkillを呼び出したプロセスも含む、を強制終了もしくは中断してもよい。言い換えると、killの呼び出しは副作用を持つかもしれないし、killの呼び出しがネストするかもしれない。このようなデストラクタがwaitもしくはnext_triggerを呼び出すとエラーとならねばならない。(SystemCレポート・ハンドらのエラーに対するデフォルト動作はSC_THROW動作を含む。それはデフォルトで例外を投げる。実装はsc_report_handlerが他の例外を投げる前にsc_unwind_exceptionを補足しなければならない。)
- 終了したプロセスに対してkillを呼び出しても、そのプロセスに影響があってはならない。(終了していない子プロセスが強制終了される場合あり。)
- エラボレーション中もしくはプロセスが初めて実行される前にkillを呼び出すことは許されている。この場合、このプロセスは現在のシミュレーション中に動作してはいけない。

JEITA

ADDITION
6.6.6.2 Member functions to kill and reset processes

void reset (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

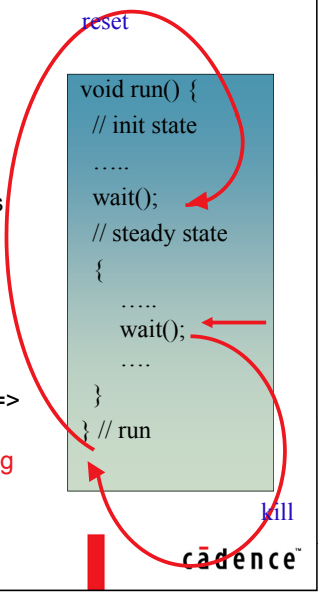
- プロセスをリセットする。
- そのプロセスは実行可能プロセス・セットとから取り除かれなければならない。
- そのプロセスに対してsc_unwind_exceptionが投げられなければならない。
- そのプロセスの動的センシティビティは取り除かれ、静的センシティビティが戻されなければならない。スレッドの場合、関連した関数は再度呼び出されなければならない。その結果、その関数はwaitが呼ばれるまで実行されて中断するか、リターンする。
- sc_unwind_exceptionの挙動は上記強制終了プロセスの記述と同等。ただし、リセットされたプロセスは終了されてはならず、後に動作可能となるかもしれない。また、プロセス・ハンドルは有効なままでなければならない。特に、リセットは直ちに効果を発揮しなければならない。コール・スタックは巻き戻されなければならず、制御はresetを呼び出した関数に戻るか、カーネルに戻らなければならない。reset呼び出しは副作用を持つかもしれない。特に、互いにリセットするプロセス間の相互再帰を避けることに注意すべきである。sc_unwind_exceptionは直ちに再度投げられる仮定の下、補足されるかもしれない。
- リセットに対してsc_unwind_exceptionが投げられた場合、sc_unwind_exceptionクラスのメンバー関数is_resetは値tureを返さなければならない。
- メソッドはリセットされてもよい。関連した関数は、メソッドが自分自身をリセットする場合を除き、巻き戻すべきコール・スタックを持たないが。メソッドがリセットされたとき、そのプロセスの動的センシティビティは取り除かれ、静的センシティビティが戻されなければならない。
- スレッドもしくはメソッドは自分自身をリセットしてもよい（関連したプロセス・ハンドルのresetメソッドを呼び出すことにより）。その場合、コール・スタックは直ちに巻き戻されなければならない。
- 終了したプロセスに対してresetを呼び出しても、そのプロセスに影響があってはならない。（終了していない子プロセスがリセットされる場合あり。）
- エラボレーション中もしくはプロセスが初めて実行される前にresetを呼び出すことは許されている。しか

JEITA

killとresetの違い

Killing and Resetting a Process (Asynchronous)

- `proc_handle.kill()` terminates a process
  - immediate, asynchronous effect
  - an exception (`sc_kill_exception`) is thrown in the target process that unwinds its stack and exits its function
  - control returns to the initiator process
- `proc_handle.reset()` resets a process
  - same effect as kill
  - in addition resets sensitivity to time 0 state => static sensitivity
  - a thread process is re-run from the beginning until it waits or returns
  - control returns to the initiator process



7 November 17, 2010

cadence

NASCUG, Feb 21, 2007 Cadenceのプレゼン資料より

JEITA

ADDITION
6.6.6.2 Member functions to kill and reset processes

bool is_unwinding () const;

- is_unwindingは、カーネルがsc_unwind_exceptionを投げた時点からカーネルがその同じsc_unwind_exceptionを補足する時点までで、値trueを返さなければならない。そうでなければ、値falseを返さなければならない。
- throw_itの呼び出し後ではなく、killもしくはreset呼び出し後のコール・スタックの巻き戻し中はis_unwindingはtrueを返さなければならない。
- このメンバー関数は、プロセスが強制終了またはリセット、シミュレーションの終了の場合を区別するために使用される。
- 無効なプロセス・ハンドルに対して呼び出された場合、実装はwarningを発生させなければならず、is_unwindingは値falseを返さなければならない。

ADDITION
6.6.6.2 Member functions to kill and reset processes

void sync_reset_on (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);
void sync_reset_off (sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

- sync_reset_onはプロセスを同期リセット状態に入らせる。
- 同期リセット状態にある間、プロセスはイベント通知もしくはタイムアウトにかかわらず再開されるたびにリセットされなければならない。
- sync_reset_onの呼び出しはプロセスが即座にリセットされることを引き起こしてはならない。その後再開されるたびにのみリセットされなければならない。resetメソッドの呼び出しと同じ効果を持たなければならない。
- sync_reset_offはプロセスが同期リセット状態からの脱出を引き起こす(そのプロセス・インスタンスでreset_signal_isもしくはasync_reset_signal_isを使って指定されたどれかのリセット信号が有効な値を持たないかぎり)。sync_reset_offはいずれのリセット信号の効果を変更してはならないし、resetメソッドの明示的な呼び出しの効果を変更してはならない。
- プロセスが再開する前にsync_reset_onに引き続きsync_reset_offが呼ばれると、そのプロセスはsync_reset_on呼び出しによるリセットはされない。
- sync_reset_onとsync_resto_offはスレッドに対してのみ有効。メソッドに対するsync_reset_onとsync_reset_offの呼び出しは許されているが、実装がwainingを発生させるかもしれないが、何の効果があってはならない。特に、include_descendants引数が値SC_INCLUDE_DESCENDANTSで子プロセスはメソッドとプロセス混在の場合にもsync_reset_onとsync_reset_offの呼び出しは許されている。各子プロセスがメソッドかスレッドjかに応じて適切は動作がとられなければならない。
- 既に同期リセット状態にあるプロセスに対してsync_reset_onを呼んでも、そのプロセスに効果があってはならない。(子プロセスが同期リセット状態に入る場合あり。)
- プロセスが同期リセット状態から出るためには、ただ一回のsync_reset_offの呼び出しのみが要求される。
- 同期リセット状態にないプロセスに対してsync_reset_offを呼んでも、そのプロセスに効果があってはならない。(子プロセスが同期リセット状態を出る場合あり。)
- プロセスは自分自身をターゲットとしてsync_reset_onまたはsync_reset_offを呼んでもよい。
- 終了したプロセスに対してsync_reset_onもしくはsync_reset_offを呼んでも、そのプロセスに効果があってはならない(子プロセスが同期リセット状態に入る／出る場合あり。)
- エラボレーション中もしくはプロセスが初めて実行される前にsync_reset_on／sync_reset_offを呼び出すことは許されている。この場合、プロセスは同期リセット状態に入る／出る。同期リセット状態にあることはプロセスの初期化に影響があってはならない。すなわち、そのプロセスは最初の評価フェーズで再開されなければならず、それがリターンするもしくはwait呼び出しのポイントまで実行されなければならない。もしプロセスが初めて再開された時(wait呼び出しから)も依然として同期リセット状態にあるならば、そのプロセスは上記のようにリセットされなければならない。

JEITA

ADDITION
6.6.6.2 Member functions to kill and reset processes

template <typename T>
void throw_it (const T& user_defined_exception, sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS) ;

- throw_itは関係ある(？)プロセス内で例外を投げなければならない。
- 投げられる例外は関数throw_itの第1引数でなければならない。
- 例外はプロセス・ハンドルの下にあるプロセスのコンテクストで投げられねばならない(呼び出し側のコンテクストではない)。プロセスが自分自身へ例外を投げる場合を除いて、これは、2つのコンテクスト・スイッチを要求する。呼び出し側からプロセスへとプロセスから呼び出し側へである。
- 投げられる例外はstd::exceptionクラスを継承することが推薦される。
- throw_itは即座に効果を発揮しなければならない。すなわち、制御は呼び出し側からプロセスに渡り、例外が投げられ、throw_it関数から戻る前にその例外が補足されなければならない。
- プロセスはそれに関連した関数内で例外を補足しなければならない。プロセスが例外を補足しない場合はエラー。例外の補足後、その関連した関数は戻る(スレッドの場合はすなわち終了)かもしれないしwaitを呼び出すかもしれない。その関係する関数が初めの例外を処理する前に他の例外を投げることはエラー。
- プロセスが他のプロセスに例外を投げる時、制御はthrow_itを呼び出した関数へ戻らなければならない。プロセスが自分自身に例外を投げる場合、throw_itから戻った時、制御はその関数のキャッチブロックに渡されなければならない。
- 例外処理では、プロセスのコール・スタックは巻き戻され、どの局所オブジェクトもそれのデストラクタが呼ばれる。そのようなデストラクタは直接間接にかかわらず、プロセスに対するwaitもしくはnext_triggerを呼び出してはならない。しかし、最初の場所でthrow_itを呼び出したプロセスを含む他のプロセスに副作用を引き起こすかもしれない。
- throw_itはターゲットが未終了のスレッドのときにのみ効果がある。プロセスがメソッドもしくは終了したプロセスであってもthrow_itの呼び出しは許されており、何の効果もあってはならない。実装はwarningを出すかもしれない。これにはメソッドが自分自身に例外を投げるケースも含まれなければならない。特に、include_descendants引数が値SC_INCLUDE_DESCENDANTSで、子プロセスがメソッドとスレッドの混在している場合、いくつかは終了するかもしれない場合でも、throw_itを呼び出すことも許される。各子プロセスに応じて適切なアクションがとられなければならない。ターゲット・プロセスが未終了のメソッドの場合、そのプロセスはthrow_itの呼び出しによって終了されてはならない。
- throw_itはターゲット・プロセスが実行中に中断された場合、すなわち、waitが呼ばれたもしくはsuspendまたはdisableの呼び出しターゲットである場合、にのみ効果がある。他のコンテクストで呼ばれたthrow_itは実装がwarningを出すかもしれないが、何の効果もあってはならない。特に、エラボレーション中もしくはターゲット・プロセスが初めて実行される前にthrow_itを呼び出しても、warningが出る以外に効果があってはならない。
- 評価フェーズ中でプロセスと関連なる関数から以外のthrow_itの呼び出しはエラー。特に、エラボレーション中またはアッ

JEITA

kill、reset、sync_reset_on/offの違い

	動作	効果が表れるポイント（呼び出し時点を起点）	動作の違い
kill	プロセスを強制終了 直ちにrunnable setから削除される	即時	再びrunnableになることはない
reset	プロセスをリセット runnable setから削除される。 dynamic sensitivityが削除され、static sensitivityが戻される。methodに対しても同様。	即時	後でrunnableになるかもしれない
sync_reset_on	プロセスを同期リセット状態にする 同期リセット状態中のイベント通知、タイムアウトが起こっても同期リセット状態のまま 即座にリセットされるわけではない。 threadにのみ効果あり。methodには効果なし。	現在の評価フェーズ	
sync_reset_off	同期リセット状態から抜け出させる threadにのみ効果あり。methodには効果なし。	現在の評価フェーズ	

JEITA

ADDITION:
6.6.6.3 ADDITION: Class sc_unwind_exception

Class sc_unwind_exception is the type of the exception thrown by member functions kill and reset. A process instance is permitted to catch the sc_unwind_exception within its associated function body, in which case it shall re-throw the exception such that the implementation can properly execute the kill or reset. It shall be an error for an application to catch the sc_unwind_exception without re-throwing it.

sc_unwind_exception クラスは、killとresetメンバー関数よりスローされる例外のためのクラス定義である。
プロセスインスタンスは、それに関連づけられた関数のボディの中において、sc_unwind_exceptionをキャッチすることができる。
その例外をキャッチした場合には、実際のkillやresetが実行されるように、例外を再度スローする必要がある。
アプリケーションがsc_unwind_exceptionをキャッチしながら、再度スローしない場合にはエラーである。

virtual const char* what() const;
Member function what shall return an implementation-defined string describing the exception.

what()は、インプリメンテーションに依存した例外を定義する文字列型を返すメンバー関数である。

virtual bool is_reset() const;
Member function is_reset shall return the value true if the exception is thrown by reset and false if the exception is thrown by kill..

is_reset()は、resetによってスローされた例外をキャッチした場合にはtrueを返し、killによってスローされた例外をキャッチした場合には、falseを返す。

JEITA

ADDITION:
6.6.6.3 ADDITION: Class sc_unwind_exception

sc_unwind_exception();
sc_unwind_exception(const sc_unwind_exception&);
virtual ~sc_unwind_exception();
This exception shall only be thrown by the kernel. Hence the constructors and destructor shall be protected members.

この例外は、カーネルによってのみスローされる例外である。従って、コンストラクターとデストラクターはprotectedメンバーである。

```
Example:
SC_MODULE(m)
{
SC_CTOR(m)
{
SC_THREAD(run);
}
void run()
{
try {
...
}
catch (const sc_unwind_exception& ex) {
// Perform clean-up
if (ex.is_reset())
...
else
...
throw ex;
}
}
...
};
```

JEITA

ADDITION:
6.6.6.4 ADDITION: Interactions between member functions for process control

Member function When it takes effect on the target process instance
suspend The current evaluation phase
resume The current evaluation phase
disable The next evaluation phase
enable The current evaluation phase
kill Immediate
reset Immediate
throw_it Immediate
sync_reset_on The current evaluation phase
sync_reset_off The current evaluation phase

In Table 2, the phrase The current evaluation phase means that the effect on the target process becomes visible at the earliest when the process actually runs (for resume, enable, or sync_reset_on) or would next have run (for suspend or sync_reset_off), which could be in the current evaluation phase. The next evaluation phase implies that disable would not prevent a process from running in the current evaluation phase. Immediate means that an exception is thrown and any associated actions are completed in the target process before return to the caller.

テーブル2において、current evaluation phaseとは、ターゲットプロセスにおいて、効果がもっとも早くていつ実際に見えるようになるかを意味する物で、resume,enable,sync_reset_onの場合には、プロセスが実際に走る時、suspendとsync_reset_offの場合には、プロセスが次に走り始める時になる。いずれの場合にも、現在の評価フェーズの中において効果が現れることになる。disableの場合にのみ、next evaluation phaseと分類されているが、これは、disableが現在の評価フェーズで走っているプロセスを止める事はないことを意味している。
Immidieateとは、例外がスローされた時には、そのプロセスの中で関数が戻る前にそれぞれのアクションが完了することを意味する。

JEITA

ADDITION:
6.6.6.4 ADDITION: Interactions between member functions for process control

As described above, member functions kill and reset each achieve their effect by throwing an object of class sc_unwind_exception, while member function throw_it throws an exception of a user-defined class. Calls to kill, reset, and throw_it may be nested such that a process catching an exception may kill or reset another process instance or may throw an exception in another process instance before itself returning control to the kernel.

既に述べたように、killやresetのメンバー関数は、sc_unwind_exceptionクラスのオブジェクトをスローすることによって効果を発揮するが、throw_it()というメンバー関数はユーザー定義のクラスを例外としてスローする。
kill,reset,throw_it関数の呼び出しは、ネストしても構わない。すなわち、例外を受け取ったプロセスが制御をカーネルに戻す前に、他のプロセスインスタンスに対してkillやresetを呼び出したり例外をスローしてもいい。

Similarly, the destructor of an object that is going out-of-scope during stack unwinding (as a result of an sc_unwind_exception having been thrown) may itself kill or reset another process instance or may throw an exception in another process instance.

同様に、オブジェクトのデストラクターがsc_unwind_exceptionがスローされた結果として、スタックを消していく段階において、自分自身をkillしたり、別プロセスインスタンスをresetしたり、別プロセスインスタンスに例外をスローしてもよい。

Care should be taken to avoid mutual recursion between processes that kill or reset one another, as such mutual recursion can result in stack overflow.

ただし、プロセス間において互いにkillやresetを無限に呼び出して、スタックオーバーフローといった結果にならないように注意する必要がある。

JEITA

ADDITION:

6.6.6.4 ADDITION: Interactions between member functions for process control

The relative priorities of the process control member functions are given below, ordered from highest priority to lowest priority:
1) kill, reset, throw_it
2) disable, enable
3) suspend, resume
4) sync_reset_on, sync_reset_off

プロセス制御のメンバー関数間には、次のような優先順位がある。優先順位の高いものから並べてある。

Member functions kill, reset, and throw_it have immediate semantics that are executed immediately even if the target process instance is disabled, suspended, or in the synchronous reset state. In such a case the target process shall remain disabled, suspended, or in the synchronous reset state after the immediate action has been completed.

kill,reset,throw_itといったメンバー関数は即座実施されるという意味を持つ。従って、たとえそのプロセスインスタンスがdisabled,suspendedあるいは、同期リセットの状態に変更されようとも、必ず即座に実施される。その場合、即座に実施されたこれらのアクションが終了した後に、disabled,suspended,同期リセットの状態になる。

For example, a call to throw_it where the target process was disabled might cause the target process to wake up, catch the exception, call wait within its catch block, and finally yield control back to the kernel, all the while remaining in the disabled state.

例えば、既にdisableされているプロセスに対して、throw_itを呼んだ場合、そのプロセスを起こし、例外をキャッチし、そのキャッチブロックの中でwait()をコールし、最後にカーネルに戻ったとしても、最後にはdisabled 状態を保持する。

JEITA

ADDITION:

6.6.6.4 ADDITION: Interactions between member functions for process control

If reset is called while a target process instance is suspended, reset shall remove the dynamic sensitivity of the target process such that any event notification or time-out that occurred before the time of the reset will be ignored when determining the behavior of a subsequent call to resume for that same target process. In other words, a reset shall wipe the slate clean when determining the behavior of resume.

プロセスインスタンスがサスペンドしている時にresetが呼ばれた場合には、resetはターゲットプロセスに設定されている動的センシティビティリストを全て削除しなければならない。これにより、リセットが呼ばれる前に、そのプロセスをレジュームするために呼ばれていたイベント通知ややタイムアウト処理は全て無視される。つまり、resetを呼び出すと、resumeの動作のために設定されていた条件を全て削除することになる。

For each process instance, the implementation shall in effect maintain three independent flags to track the state of calls to the member functions disable/enable, suspend/resume, and sync_reset_on/sync_reset_off, and shall then act according to the relative priority of those calls as listed above.

それぞれのプロセスインスタンスに対して、disable/enable, suspend/resume,sync_reset_on/sync_reset_offのメンバー関数に対応した状態を保持するためのフラグを別々に持つようにカーネルが作られなければいけない。

It shall be an error to call the three member functions with immediate semantics during elaboration, before the initialization phase, or while the simulation is paused or stopped, but the remaining member functions may be called while in those states.

即時実施の意味を持つ三つのメンバー関数を、初期化フェーズの前のエラボレーション時に呼び出すとエラーである。シミュレーターがポーズしているときや止まっている時も同様である。しかし、他のメンバー関数に関しては、これらの状態の時に呼び出しても構わない。

JEITA

ADDITION:

6.6.6.4 ADDITION: Interactions between member functions for process control

Example:

```
sc_process_handle h;
...
h.sync_reset_on(); // Process instance put into the synchronous reset state
h.suspend(); // Process instance suspended
h.disable(); // Process instance disabled
wait(...);
h.reset(); // Process instance immediately reset but still disabled
...
h.enable(); // Process instance enabled but still suspended
...
h.resume(); // Process instance resumed but still in the synchronous reset state
```

JEITA

6.6.7 sc_get_current_process_handle

```
sc_process_handle sc_get_current_process_handle();
```

The value returned from function sc_get_current_process_handle shall depend on the context in which it is called. When called during elaboration from the body of a module constructor or from a function called from the body of a module constructor, sc_get_current_process_handle shall return a handle to the spawned or unspawned process instance most recently created within that module, if any.

sc_get_current_process_handle()関数の戻り値は、それがどこで呼ばれるかに依存する。
エラボレーション時に、moduleのコンストラクターまたは、moduleのコンストラクターから呼ばれた関数の中にてこの関数が呼ばれた場合には、そのモジュールの中にて最後に作成されたプロセスインスタンス(スタティックまたはダイナミック)のハンドルを返す。

CLARIFICATION: When called from one of the callbacks before_end_of_elaboration or end_of_elaboration, sc_get_current_process_handle shall return a handle to the spawned or unspawned process instance most recently created within that specific callback function, if any.

before_end_of_elaborationまたは、end_of_elaborationnoコールバックの中においてこの関数が呼ばれた場合にも、そのコールバック関数が定義されているモジュールの中で最後に作成されたプロセスインスタンス(スタティックまたはダイナミック)のハンドルを返す。

JEITA

6.6.7 sc_get_current_process_handle

If the most recently created process instance
was not within the current module, or in the case of before_end_of_elaboration or end_of_elaboration
was not created within that specific callback, an implementation may return either a handle to the most recently created process instance or an invalid handle.

もし、最後に作成されたプロセスインスタンスがそのモジュールの中に無かった場合、あるいは、before_end_of_elaboration または、end_of_elaboratioinの場合で、それらが定義されているモジュールの中に無かった場合、この関数の戻り値は、最後に作成されたプロセスインスタンスを返すか、invalidハンドルを返す。

When called during simulation,
sc_get_current_process_handle shall return a handle to the currently executing spawned or unspawned process instance, if any. If there is no such process instance, sc_get_current_process_handle shall return an invalid handle.

シミュレーション実行中にsc_get_current_process_handleが呼ばれた場合には、現在実行中のプロセスインスタンスのハンドルを返す。
もし、そういったプロセスインスタンスが無かった場合には、invalidハンドルを返す。

CLARIFICATION: When called from sc_main, sc_get_current_process_handle shall return an invalid handle.

sc_mainからsc_get_current_process_handleが呼ばれた場合には、invalid ハンドルを返す。

JEITA

ADDITION:
6.6.6.4 ADDITION: Interactions between member functions for process control

```
Example:
SC_MODULE(Mod)
{
...
SC_CTOR(Mod)
{

SC_METHOD(Run);
sensitive << in;
sc_process_handle h1 = sc_get_current_process_handle(); // Returns a handle to process Run
}
void Run()
{
sc_process_handle h2 = sc_get_current_process_handle(); // Returns a handle to process Run
if (h2.proc_kind() == SC_METHOD_PROC_)
... // Running a method process
sc_object* parent = h2.get_parent_object(); // Returns a pointer to the
// module instance
if (parent)
{
handle = sc_process_handle(parent); // Invalid handle – parent is not a process
if (handle.valid())
... // Executed if parent were a
// valid process
}
}
...
};
```

JEITA

ADDITION:
6.6.8 ADDITION: sc_is_unwinding

bool sc_is_unwinding();

Function sc_is_unwinding shall return the value returned from member function is_unwinding of the process handle returned from sc_get_current_process_handle. In other words, sc_is_unwinding shall return true if and only if the calling process instance is currently the target of a call to kill or reset.

sc_is_unwindoing は、sc_get_current_process_handle()にて取得されるプロセスハンドルに対してis_unwinding()メンバー関数を呼び出すのと同じである。すなわち、現在のプロセスインスタンスがkillやreestの対象であるときのみ、この関数はtrueを返す。

```
Example:
class wait_on_exit
{
~wait_on_exit() {
if( !sc_is_unwinding() ) // needed, because we might get killed
wait( 10, SC_NS ) ; // ... and this wait would be illegal
}
};
void some_module::some_process()
{
while( true )
{
try {
wait_on_exit w; // local object, destroyed before catch
// ...
} catch( const sc_unwind_exception& ) {
// some other cleanup
throw;
}
}
```

JEITA

ADDITION:
6.8 sc_event_and_list and sc_event_or_list

6.8.1 Description

The classes **sc_event_and_list** and **sc_event_or_list** are used to represent explicit event list objects which may be constructed and manipulated prior to being passed as arguments to the functions **next_trigger** (see 6.2.17) and **wait** (see 6.2.18). An event list object shall store pointers or references to zero or more event objects. An event list may contain multiple references to the same event object. The order of the events within the event list shall have no effect on the behavior of the event list when it is used to create dynamic sensitivity.

sc_event_and_listとsc_event_or_listクラスは、next_trigger（6.2.17参照）やwait（6.2.18参照）関数の引数に渡される前に生成されたり操作される明確なイベントリストを表します。イベントリストオブジェクトは、0かそれ以上のイベントオブジェクトへのポインターまたは参照を保存します。イベントリストは、同じイベントオブジェクトへの複数の参照を含むことができます。ダイナミックセンシティビティを生成するとき、イベントリストのイベントの順序はイベントリストの動作に影響しません。

JEITA

ADDITION:
6.8 sc_event_and_list and sc_event_or_list

6.8.3 Constraints and usage

The intended usage for objects of class **sc_event_and_list** and **sc_event_or_list** is that they are passed as arguments to the functions **wait** and **next_trigger**. Unlike the case of event expression objects, which are deleted automatically by the implementation (see 6.9.3), the application shall be responsible for deleting event list objects when they are no longer required. The application shall be responsible for ensuring that an event list object is still valid (that is, has not been destroyed) when a process instance resumes or is triggered as a direct result of the notification of one or more events in the event list. If the event list object has already been destroyed at this point, the behavior of the implementation shall be undefined.

sc_event_and_listとsc_event_or_listクラスのオブジェクトへの意図する使い方は，waitとnext_trigger関数の引数として渡されるものです。実装によって自動的に削除されるイベント式オブジェクトの場合と異なり，アプリケーションはイベントリストオブジェクトが不要になった時に削除する必要があります。アプリケーションは，プロセスインスタンスが再開する，もしくは1つあるいはそれ以上のイベントの通知の直接の結果でトリガされるとき，イベントリストオブジェクトがまだ有効である(まだ破壊されていない)ことを保証することに責任を持ちます。もし，この時点でイベントリストオブジェクトがすでに破壊されていたら，実装の動作は未定義です。

JEITA

ADDITION:
6.8 sc_event_and_list and sc_event_or_list

6.8.4 Constructors, destructor, assignment

sc_event_and_list();
sc_event_or_list();

Each default constructor shall construct an empty event list object. It shall be an error to pass an empty event list object as an argument to the functions **wait** or **next_trigger**.

それぞれのデフォルトコンストラクタは空のイベントリストオブジェクトを生成します。waitやnext_trigger関数に空のイベントリストオブジェクトを渡すとエラーになります。

sc_event_and_list(const sc_event_and_list&);
sc_event_or_list(const sc_event_or_list&);
sc_event_and_list& operator= (const sc_event_and_list&);
sc_event_or_list& operator= (const sc_event_or_list&);

The copy constructors and assignment operators shall permit event list objects to be initialized and assigned by the application.

コピーコンストラクタと代入演算子は、イベントリストの初期化やアプリケーションによる代入を可能にします。

sc_event_and_list(const sc_event&);
sc_event_or_list(const sc_event&);

These constructors shall construct event list objects containing the single event passed as an argument to the constructor.

これらのコンストラクタはコンストラクタへの引数に渡される単一のイベントを含むイベントリストを生成します。

~sc_event_and_list();
~sc_event_or_list();

An event list object may be constructed as a stack or as a heap variable. In the case of the heap, the application is responsible for deleting an event list object when it is no longer required.

イベントリストオブジェクトはスタックかヒープ変数として生成されます。ヒープの場合、アプリケーションはイベントリストが不要になった時に消去なければいけません。

ADDITION:
6.8 sc_event_and_list and sc_event_or_list

6.8.5 Member functions and operators

int size() const;

Member function size shall return the number of events in the event list.
メンバ関数sizeは、イベントリスト中のイベントの数を返します。

void swap(sc_event_and_list&);
void swap(sc_event_or_list&);

Member function swap shall exchange the current event list object *this with the event list passed as an argument.
メンバ関数swapは、現在のイベントリストオブジェクト*thisを引数に渡されたイベントリストと交換します。

sc_event_and_list& operator&= (const sc_event&);
sc_event_and_list& operator&= (const sc_event_and_list&);
sc_event_and_expr† operator& (const sc_event&) const;
sc_event_and_expr† operator& (const sc_event_and_list&) const;
sc_event_or_list& operator|= (const sc_event&);
sc_event_or_list& operator|= (const sc_event_or_list&);

sc_event_or_expr† operator| (const sc_event&) const;
sc_event_or_expr† operator| (const sc_event_or_list&) const;

Each of these operators shall append the event or event list passed as an argument to the current event list object *this, and shall return the resultant elongated event list as the value of the operator. In the case of the assignment operators &= and |=, the operator shall return a reference to the current object, which shall have been modified to contain the elongated event list. In the case of the remaining operators & and |, the current object *this shall not be modified.

これらの演算子は現在のイベントリストオブジェクト*thisの引数として渡されたイベントやイベントリストを追加し、そして演算子の値として結果のイベントリストを返します。
代入演算子&=と|=の場合、演算子はイベントリストを含むよう変更された現在のオブジェクトへの参照を返します。残りの&と|演算子の場合、カレントオブジェクト*thisは変更されません。

Example:

```
struct M: sc_module
{
    sc_port<sc_signal_in_if<int>, 0> p;      // Multiport

    M(sc_module_name _name)
    {
        SC_THREAD(T);
    }

    sc_event_or_list all_events() const
    {
        sc_event_or_list or_list;
        for (int i = 0; i < p.size(); i++)
            or_list |= p[i]->default_event();
        return or_list;
    }

    sc_event event1, event2;
    ...

    void T()
    {
        for (;;)
        {
            wait( all_events() );
            ...

            sc_event_and_list list;
            sc_assert( list.size() == 0 );

            list = list & event1;
            sc_assert( list.size() == 1 );

            list &= event2;
            sc_assert( list.size() == 2 );

            wait(list);
            sc_assert( list.size() == 2 );
            ...
        }
    }
    SC_HAS_PROCESS(M);
};
```

JEITA

CHANGE:
6.9 *sc_event_and_expr*† and *sc_event_or_expr*†

6.9.1 Description

The classes *sc_event_and_expr*† and *sc_event_or_expr*† provide the **&** and | operators used to construct the event expressions passed as arguments to the functions **wait** (see 6.2.17) and **next_trigger** (see 6.2.18). An event expression object shall store pointers or references to zero or more event objects. An event expression may contain multiple references to the same event object. The order of the events within the event expression shall have no effect on the behavior of the event expression when it is used to create dynamic sensitivity.

クラスsc_event_and_exprとsc_event_or_exprは、wait（6.2.17参照）とnext_trigger（6.2.18参照）関数に引数として渡されるイベント式で、&や | オペレーターを提供します 。イベント式オブジェクトは、0かそれ以上イベントオブジェクトのポインターまたは参照を保持します。イベント式は、同じイベントオブジェクトへの複数の参照を含むことができます。イベント式がダイナミックセンシティビティに用いられる場合、そのイベント式中のイベントの順序はイベント式の動作に影響しません。

JEITA

CHANGE:
6.9 *sc_event_and_expr†* and *sc_event_or_expr†*

6.9.3 Constraints on usage

An application shall not explicitly create an object of class *sc_event_and_expr†* or *sc_event_or_expr†*. An application wishing to create explicit event list objects should use the classes **sc_event_and_list** and **sc_event_or_list**.

Classes *sc_event_and_expr†* and *sc_event_or_expr†* are the return types of **operator&** and **operator|**, respectively, of classes **sc_event**, **sc_event_and_list**, and **sc_event_or_list**.

The existence of the type conversion operators from type *sc_event_and_expr†* to type **sc_event_and_list const &** and from type *sc_event_or_expr†* to type **sc_event_or_list const &** allow expressions of type *sc_event_and_expr†* and *sc_event_or_expr†* to be passed as arguments to the functions **wait** and **next_trigger**. The implementation shall delete the event expression object when the process instance resumes or is triggered as a direct result of the notification of one or more events in the event expression. In other words, the implementation shall be responsible for the creation, deletion, and memory management of event expression objects of type *sc_event_and_expr†* and *sc_event_or_expr†*.

アプリケーションは、sc_event_and_exprクラスまたはsc_event_or_exprのオブジェクトを明確には作りません。明確なイベントリストオブジェクトを作りたいアプリケーションは、sc_event_and_listクラスとsc_event_or_listを使わなければなりません。
sc_event_and_exprとsc_event_or_exprクラスは、それぞれsc_event、sc_event_and_list、sc_event_or_listの&演算子と|演算子の型を返します。
sc_event_and_expr型からsc_event_and_list const &型、そして、sc_event_or_expr型からsc_event_or_list const &型への型変換演算子により、sc_event_and_expr型とsc_event_or_expr型をwaitやnext_trigger関数の引数に渡すことを可能にします。実装は、プロセスインスタンスが再開されたときあるいはイベント式で1つかそれ以上のイベントの通知の結果としてトリガされたとき、イベント式オブジェクトを削除します。言い換えると、実装はsc_event_and_expr型とsc_event_or_expr型のイベント式オブジェクトの生成、削除とメモリ管理の責任を持つ必要があります。

CHANGE:
6.9 *sc_event_and_expr†* and *sc_event_or_expr†*

6.9.4 Operators

operator sc_event_and_list const &() const;
operator sc_event_or_list const &() const;

Each of these type conversion operators shall return a reference to an event list object equivalent to the event list expression represented by the current object ***this**, equivalent in the sense that the event list object shall contain references to the same set of events as the event list expression.

それぞれの型変換演算子は、イベントリスト式カレントオブジェクト*thisで表されるイベントリスト式と同じイベントリストオブジェクトへの参照を返します。

sc_event_and_expr† **operator&** (*sc_event_and_expr†* , sc_event const&);
sc_event_or_expr† **operator|** (*sc_event_or_expr†* , sc_event const&);

A call to either operator shall append the event passed as the second argument to the event expression passed as the first argument, and shall return the resultant elongated event expression as the value of the operator.

それぞれの演算子は第二引数で渡されたイベントを第一引数で渡されたイベント式に追加し、イベント式を演算子の値として返します。

CLARIFICATION:
Example:

```
sc_event event1, event2;

void thread_process()
{
    ...
    wait( event1 | event2 );
    // When the thread process resumes following the notification of event1 or event2,
    // the implementation shall delete the object of type sc_event_or_expr returned from operator|
    ...
}
```

6.10 sc_event

6.10.1 Description

An event is an object of class **sc_event**, used for process synchronization. A process instance may be triggered or resumed on the *occurrence* of an event, that is, when the event is notified. Any given event may be notified on many separate occasions.

ADDITION: Events can be *hierarchically named* or not. An event without a hierarchical name shall have an implementation-defined name. A hierarchically named event shall either have a parent in the object hierarchy (in which case the event would be returned from member function **get_child_events** of that parent) or be a top-level event (in which case the event would be returned by function **sc_get_top_level_events**). If an event has a parent, that parent shall be either a module instance or a process instance. An event without a hierarchical name shall not have a parent in the object hierarchy.

A *kernel event* is an event that is instantiated by the implementation, such as an event within a predefined primitive channel. A kernel event shall not be hierarchically named but shall have an implementation-defined name, regardless of whether it is instantiated during elaboration or during simulation.

eventはプロセスの同期に使用するsc_eventクラスのオブジェクトです。プロセスインスタンスはイベントが通知されるときの、イベントの発生によりトリガされるか再開されます。

イベントは、階層名でも，そうでないのも可能です。階層名でないイベントは実装定義の名前になります。階層名のイベントは、オブジェクト階層の親 (この場合、イベントは親のメンバ関数get_child_eventsで返されます)、あるいは最上位レベルのイベント(この場合は、イベントはsc_get_top_level_eventsで返されます)を持ちます。もしイベントに親がある場合、親はモジュールインスタンスあるいはプロセスインスタンスです。階層名のないイベントは、オブジェクト階層で親を持ちません。

カーネルイベントは、定義済みのプリミティブチャネルのイベントとして、実装によってインスタンスされるイベントです。カーネルイベントは階層名にならないですが、エラボレーション中かシミュレーション中のいつインスタンスされるかに関わらず実装定義の名前を持ちます。

6.10 sc_event

With the exception of kernel events, every event that is instantiated during elaboration or before the initialization phase shall be a hierarchically named event.

NOTE 1—Events without a hierarchical name are provided for backward compatibility with earlier versions of this standard and to avoid the potential performance impact of creating hierachical names during simulation.

NOTE 2—Although an event may have a parent of type **sc_object**, an event object is not itself an **sc_object**, and hence an event cannot be returned by member function **get_child_objects**. Member function **get_child_events** is provided for the purpose of retrieving the events within a given module or process instance.

NOTE 3—In the case of a hierarchically named event, the rules for hierachical naming imply that the name of the event is formed by concatenating the hierarchical name of the parent of the event with the basename of the event, with the two parts separated by a period character.

カーネルイベントは例外で、エラボレーション中もしくは初期化フェーズ前にインスタンスされた全てのイベントは、階層名イベントです。

注1−階層名を持たないイベントは，本規格より前のバージョンの下位互換性およびシミュレーション中に階層名を生成することによるパフォーマンスへの影響を避けるためのために提供されています。

注2−イベントはsc_object型の親を持つことができますが、イベントオブジェクト自体はsc_objectではないため、メンバ関数get_child_objectsによってイベントを返せません。メンバ関数get_child_eventsは指定されたモジュールやプロセスインスタンスでイベントを取得する目的で提供されています。

注3−階層名イベントの場合，階層名が意味するイベント名のルールは，ピリオドで区切られた2つの部分からなる、イベントの基底名とイベントの親の階層名の連結によって形成される。

JEITA

6.10 sc_event

6.10.3 Constraints on usage

Objects of class **sc_event** may be constructed during elaboration or simulation. CHANGE: Events may be notified during elaboration or simulation, except that it shall be an error to create an immediate notification during elaboration or from one of the callbacks **before_end_of_elaboration**, **end_of_elaboration**, or **start_of_simulation**.

sc_eventクラスのオブジェクトは、エラボレーションあるいはシミュレーション中に生成されます。イベントは、エラボレーション中あるいはbefore_end_of_elaboration、end_of_elaboration、もしくはstart_of_simulationのいずれか一つのコールバックからの通知の生成がエラーになる場合を除いて、エラボレーションあるいはシミュレーション中に通知されます。

JEITA

6.10 sc_event

6.10.4 ADDITION: Constructors, destructor, and event naming

sc_event();
sc_event(const char*);

Calling the constructor sc_event(const char*) with an empty string shall have the same effect as calling the default constructor, regardless of when it is called by the application.

When called by the application during elaboration or before the initialization phase, each of the above two constructors shall create a hierarchically named event.

When the default constructor is called by the application from the initialization phase onwards, whether or not a hierarchically named event is created shall be implementation-defined on a per-instance basis.

When the constructor sc_event(const char*) is called by the application from the initialization phase onward with a non-empty string argument, the implementation shall create a hierachically named event.

When a hierarchically named event is constructed, if a non-empty string is passed as a constructor argument, that string shall be used to set the string name of the event. Otherwise, the string name shall be set to "event". The string name shall be used to determine the hierarchical name as described in 6.17.

コンストラクタのsc_event（const char *）を空の文字列で呼び出すと、それがアプリケーションによって呼ばれたかに関わらず、デフォルトのコンストラクタを呼び出すのと同じことになります。
エラボレーション中、または初期化フェーズの前にアプリケーションから呼ばれた時、上記2つのコンストラクタは階層名イベントを生成します。
デフォルトのコンストラクタが、初期化フェーズ以降のアプリケーションから呼ばれた時、階層名もそうでないイベントも、インスタンスごとの実装定義になります。
コンストラクタsc_event（const char *）を空でない文字列の引数を指定して初期化フェーズ以降のアプリケーションによって呼び出されると、実装は階層名のイベントを生成しなければならない。
階層名のイベントが生成される時、空でない文字列が引数に渡された場合、文字列は、イベント名の文字列の設定に使われる。そうでない場合、文字列名は“event”に設定しなければならない。6.17で説明されるように文字列名が階層名を決定するために使用しなければならない。

6.10 sc_event

If a constructor needs to substitute a new string name in place of the original string name as the result of a name clash, the constructor shall generate a single warning.

An event that is not hierarchically named shall have a non-empty implementation-defined name. That name shall take the form of a hierachical name as described in 6.17 but may contain one or more characters that are not in the recommended character set for application-defined hierarchical names (such as punctuation characters and mathematical symbols). The intent is that an implementation may use special characters to distinguish implementation-defined names from application-defined names, although it is not obliged to do so.

Kernel events shall obey the same rules as application-defined events that are not hierarchically named.

NOTE—An implementation is not required to guarantee that each implementation-defined event name is unique. The intent is that the implementation will use characters not recommended in application-defined names to reduce the probability of a name clash between a hierarchical name and an implementation-defined name.

コンストラクタは、名前の衝突の結果として、元の文字列名の代わりに新しい文字列の名前を代入する必要がある場合は、コンストラクタは、単一の警告を生成しなければならない。
階層名でないイベントは、空でない実装定義の名前でなければならない。
その名前はとして6.17に記載される階層名の形式をとるが、アプリケーション定義で階層名に使うべきでない文字（句読点文字や数学記号など）を1つ以上含むことができる。その目的は、義務ではないが、アプリケーション定義の名前から実装定義の名前を区別するため、実装が特別な文字を使用できるためです。
カーネルイベントは、階層名でないアプリケーション定義のイベントと同じ規則に従わなければいけません。
注 – 実装はそれぞれの実装定義されたイベント名が同一であることを保証する必要はありません。目的は、実装では、階層名と実装定義の名前の衝突の確率を減らすため、アプリケーション定義の名前には推奨されない文字を使用するためです。

virtual ~sc_event();

The destructor shall delete the object and shall remove the event from the object hierarchy such that it is no longer a child of any module or process instance.

デストラクタは、オブジェクトを削除しなければならない、そしてもはやどのモジュールやプロセスインスタンスの子でないように、オブジェクト階層からイベントを除去しなければならない。

6.10 sc_event

6.10.5 ADDITION: Functions for naming and hierarchy traversal

const char* **name**() const;

Member function **name** shall return the hierarchical name of the event instance in the case of a hierarchically named event, or otherwise shall return the implementation-defined name of the event instance. The name shall always be non-empty.

メンバ関数nameは、または階層名イベントの場合には、イベントのインスタンスの階層名を返す、それ以外の場合は、イベントインスタンスの実装定義の名前を返す。名前は常に空でないものでなければならない。

const char* **basename**() const;

Member function **basename** shall return the string name of the event instance in the case of a hierarchically named event, or otherwise shall return an implementation-defined name. This is the string name created when the event instance was constructed. In the case of an implementation-defined name, the relationship between the strings returned from member functions **name** and **basename** is also implementation-defined

メンバ関数basenameは階層名イベントの場合は、イベントインスタンスの文字列名を返す、それ以外の場合は実装で定義された名前を返す。これは、イベントのインスタンスが生成されたときに作られた文字列の名前です。実装定義の名前の場合には、メンバ関数nameとbasenameから返される文字列の間の関係も実装定義です。

bool **in_hierarchy**() const;

Member function **in_hierarchy** shall return the value **true** if and only if the event is a hierarchically named event.

メンバ関数in_hierarchyはイベントが階層名のイベントの場合のみtrue値を返す。

JEITA

6.10 sc_event

The three functions below return information that supports the traversal of events that have a parent in the object hierarchy. An implementation shall allow each of these three functions to be called at any stage during elaboration or simulation. If called before elaboration is complete, they shall return information concerning the partially constructed object hierarchy as it exists at the time the functions are called. In other words, a function shall return pointers to any **sc_event** objects that have been constructed before the time the function is called but will exclude any objects constructed after the function is called.

以下の3つの関数は、オブジェクト階層内の親を持つイベントの動きをサポートする情報を返します。実装は、これらの3つの関数の各々が、エラボレーションまたはシミュレーション中の任意の段階で呼べるようにしなければいけません。エラボレーションが完了される前に呼ばれる場合は、関数が呼び出される時に存在する、部分的に構築されたオブジェクトの階層構造に関する情報を返します。言い換えれば、関数は関数が呼び出された後に構築されるオブジェクトを除外し、関数が呼び出された時点より前に構築されている任意のsc_eventオブジェクトへのポインタを返さなければいけません。

sc_object* **get_parent_object**() const;

Member function **get_parent_object** shall return a pointer to the **sc_object** that is the parent of the current **sc_event** object in the case of hierarchically named events, or otherwise shall return the null pointer. **get_parent_object** shall also return a null pointer for a top-level event. If the parent object is a process instance and that process has terminated, **get_parent_object** shall return either a pointer to the surviving process instance or a null pointer if it has been deleted.

メンバ関数get_parent_object、階層名のイベントの場合現在のsc_eventオブジェクトの親であるsc_objectへのポインタを、そうでない場合はnullポインタを返します。get_parent_objectはトップレベルのイベントに対してもnullポインタを返します。親オブジェクトが、プロセスのインスタンスでそのプロセスが終了しているが削除されている場合、get_parent_objectは存続するプロセスインスタンスまたはそれも削除されていたらnullポインタのいずれかを返します。

JEITA

6.10 sc_event

const std::vector<sc_event*>& **sc_get_top_level_events**();

Function **sc_get_top_level_events** shall return a **std::vector** containing pointers to all of the top-level **sc_event** objects, that is, events that are hierarchically named but have no parent.

sc_get_top_level_events関数は全てのトップレベルのsc_eventオブジェクト、つまり階層名で親を持たないイベントへのポインタを含むstd::vectorを返します。

sc_event* **sc_find_event**(const char*);

Function **sc_find_event** shall return a pointer to the hierarchically named **sc_event** object that has a name that exactly matches the value of the string argument or shall return the null pointer if there is no hierarchically named **sc_event** with that name. Function **sc_find_event** shall not return a pointer to an event that has an implementation-defined name.

sc_find_event関数は引数の文字列の値と正確に一致する名前を持つ階層名のsc_eventオブジェクトへのポインタ、もしくは階層名のsc_eventがない場合はnullポインタを返します。sc_find_event関数は実装定義の名前を持つイベントへのポインタを返してはいけません。

JEITA

6.10 sc_event

6.10.7 Event lists

CHANGE: *sc_event_and_expr*† **operator&** (const sc_event&) const;
CHANGE: *sc_event_and_expr*† **operator&** (const sc_event_and_list&) const;
CHANGE: *sc_event_or_expr*† **operator|** (const sc_event&) const;
CHANGE: *sc_event_or_expr*† **operator|** (const sc_event_or_list&) const;

A call to either operator shall return an event expression formed by appending the current event object to the event or event list passed as an argument.

NOTE—Event lists are used as arguments to functions **wait** (see 6.2.18) and **next_trigger** (see 6.2.17).

どちらの演算子の呼び出しでも現在のイベントオブジェクトに、引数で渡されたイベントあるいはイベントリストを追加して作られるイベント式を返します。

JEITA

6.11 sc_time

6.11.4 ADDITION: Function sc_max_time

The implementation shall provide a function **sc_max_time** with the following declaration:
実装は、次の定義を持つ関数sc_max_timeを提供します
const sc_time& **sc_max_time**();

The function **sc_max_time** shall return the maximum value of type **sc_time**, calculated after taking into account the time resolution. Since function **sc_max_time** necessarily returns an object of type **sc_time** that represents a non-zero time value, the time resolution cannot be modified after a call to **sc_max_time**. Every call to **sc_max_time** during a given simulation run shall return an object having the same value and representing the maximum simulation time. The actual value is implementation-defined. Whether each call to **sc_max_time** returns the same object or a different object is implementation-defined.

sc_max_time関数は、アカウントに時間分解能を考慮てから計算された、sc_time型の最大値を返します。
sc_max_time関数は、必ずゼロ以外の時間の値を表すsc_time型のオブジェクトを返すため、時間分解能はsc_max_timeの呼び出し後に変更できません。あるシミュレーションの実行中の全てのsc_max_timeの呼び出しは、同じ値の最大シミュレーション時間を表す持つオブジェクトを返します。実際の値は実装定義です。
sc_max_timeを呼び出すたびに同じオブジェクトか別のオブジェクトを返すかは実装定義です。

JEITA

6.12 sc_port

CORRECTION:
explicit **sc_port_b**(int , sc_port_policy);
sc_port_b(const char* , int , sc_port_policy);
virtual ~**sc_port_b**();

The constructor for class **sc_port_b** shall pass the values of its arguments though to the constructor for the base class sub-object. The character string argument shall be the string name of the instance in the module hierarchy, the **int** argument shall be the maximum number of channel instances, and the **sc_port_policy** argument shall be the port policy. The destructor shall do nothing.

sc_port_bクラスのコンストラクタはベース・クラス・サブオブジェクトのコンストラクタを通してその引数の値を渡します。文字列引数はモジュール階層のインスタンス名です、int引数はチャネルインスタンスの最大数です、そしてsc_port_policy引数はポートポリシーです。デストラクタは、何もしません。

JEITA

6.12 sc_port

6.12.7 Named port binding

Ports can be bound either using the functions listed in this subclause for named binding or using the **operator()** from class **sc_module** for positional binding. An implementation may defer the completion of port binding until a later time during elaboration because the port to which a port is bound may not yet itself have been bound. Such deferred port binding shall be completed by the implementation before the callbacks to function **end_of_elaboration**.

void **operator()** (IF&);
CORRECTION: virtual void **bind**(IF&);

Each of these two functions shall bind the port instance for which the function is called to the channel instance passed as an argument to the function. The actual argument can be an export, in which case the C++ compiler will call the implicit conversion **sc_export<IF>::operator IF&**. The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**.

演算子()の実装は、仮想メンバ関数bindをコールすることで、その目的を達成しています。

CORRECTION: void **operator()** (sc_port_b<IF>&);
CHANGE: virtual void **bind**(sc_port_b<IF>&);

Each of these two functions shall bind the port instance for which the function is called to the port instance passed as an argument to the function. The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**.

演算子()の実装は、仮想メンバ関数bindをコールすることで、その目的を達成しています。

6.13 sc_export

6.13.7 Export binding

Exports can be bound using either of the two functions defined here. The notion of positional binding is not applicable to exports. Each of these functions shall bind the export immediately, in contrast to ports for which the implementation may need to defer the binding.

void **operator()** (IF&);
CHANGE: virtual void **bind**(IF&);

Each of these two functions shall bind the export instance for which the function is called to the channel instance passed as an argument to the function. The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**.

演算子()の実装は、仮想メンバ関数bindをコールすることで、その目的を達成しています。

NOTE—The actual argument could be an export, in which case **operator IF&** would be called as an implicit conversion.

JEITA

6.15 sc_prim_channel

6.15.3 Constraints on usage

Objects of class **sc_prim_channel** can only be constructed during elaboration. It shall be an error to instantiate a primitive channel during simulation.

A primitive channel should be *publicly* derived from class **sc_prim_channel**.

A primitive channel shall implement one or more interfaces.

CLARIFICATION: Member functions **request_update** and **async_request_update** may be called during elaboration or simulation. If called during elaboration, the update request shall be executed during the initialization phase.

メンバ関数request_updateとasync_request_updateは、エラボレーションまたはシミュレーション中に呼べます。エラボレーション中に呼ばれた場合、更新要求は、初期化フェーズ中に実行しなければいけません。

JEITA

6.15 sc_prim_channel

6.15.6 request_update and update

ADDITION: Member functions **request_update** and **async_request_update** each cause the scheduler to queue an update request. An application should not call both **request_update** and **async_request_update** for a given primitive channel instance at any time during elaboration or simulation, but may do so for different primitive channel instances. If both **request_update** and **async_request_update** are called for the same primitive channel instance, the behavior of the implementation is undefined.

メンバ関数request_updateおよびasync_request_updateそれぞれは、スケジューラのキューに更新要求を起こします。アプリケーションは、指定されたプリミティブのチャネルインスタンスに対して、エラボレーション中またはシミュレーション中の任意の時点でrequest_updateとasync_request_updateの両方を呼んではいけませんが、別のプリミティブチャネルインスタンスにはできます。request_updateとasync_request_updateの両方が同じプリミティブチャネルインスタンスで呼ばれた場合、実装の動作は未定義です。

void **request_update**();

CLARIFICATION: Member function **request_update** shall cause the scheduler to queue an update request for the current primitive channel (see 5.2.1.3). No more than one update request shall be queued for any given primitive channel instance in any given update phase, that is, multiple calls to **request_update** in the same evaluation phase shall result in a single update request.

メンバ関数request_updateは現在のプリミティブチャネル（5.2.1.3参照）のスケジューラに更新要求のキューを起こします。どんな特定の更新フェーズでも、特定のプリミティブチャネルインスタンスに1つ以上更新要求することはできません、それは同じ評価フェーズで複数のrequest_updateの呼び出しが、単一の更新要求ということです。

JEITA

6.15 sc_prim_channel

CLARIFICATION: void **async_request_update**();

Member function **async_request_update** shall cause the scheduler to queue an update request for the current primitive channel in a thread-safe manner with respect to the host operating system. The intent of **async_request_update** is that it may be called reliably from an operating system thread other than those in which the SystemC kernel and any SystemC thread processes are executing.

An application may call **async_request_update** at any time during elaboration or simulation, and these calls may be asynchronous with respect to the SystemC kernel. The implementation shall guarantee that each call to **async_request_update** behaves as if it were executed in an evaluation phase. No more than one update request shall be queued for any given primitive channel instance in any given update phase, that is, multiple calls to **async_request_update** received by the kernel between two consecutive update phases shall result in a single update request. The precise phase in which any given call to **async_request_update** is received and processed by the kernel is undefined. As a consequence, two calls to **async_request_update** for the same primitive channel and occurring within a narrow time window may result in a single update request, two update requests in consecutive delta cycles, or two update requests in non-consecutive delta cycles.

メンバ関数async_request_updateは、ホストオペレーティングシステムに対してスレッドセーフな方法で現在のプリミティブチャネルのスケジューラに更新要求のキューを起こします。async_request_updateの意図は、SystemCのカーネルと任意のSystemCのスレッドのプロセスが実行されているオペレーティングシステムのスレッドにおいて確実に呼べるためです。
アプリケーションは、エラボレーションまたはシミュレーション中の任意の時点でasync_request_updateを呼び出すことができます、これらの呼び出しは、SystemCカーネルに対して非同期かもしれません。実装は、各async_request_update呼び出しが、評価フェーズで実行されたかのように動作することを保証しなければならない。特定のプリミティブチャネルのインスタンス、特定の更新フェーズで、複数の更新要求を発行することはできない、それはカーネルによる2つの連続した更新フェーズでの複数のasync_request_updateの呼び出しが単一の更新要求となることを意味する。
async_request_updateの任意の呼び出しがカーネルによって正確にどのフェーズで受取られ処理されるかは定義されていません。その結果、狭い時間枠内、連続するデルタサイクルで2つの更新要求、または非連続のデルタサイクルで2つの更新要求内での、同じプリミティブチャネルに対するでの2つのasync_request_updateの呼び出しは、1個の更新要求になります。

6.15 sc_prim_channel

The application is responsible for synchronizing access to any shared memory across calls to **async_request_update** and **update**. The software thread that calls **async_request_update** would typically need to write a value to some variable that is read by function **update**. The implementation can give no guarantee with regard to the integrity of any such shared memory.

It is not recommended to call member function **async_request_update** from functions executed in the context of the SystemC kernel, such as SystemC thread or method processes, but only by operating system threads separate from the kernel. Calling **async_request_update** may cause a performance degradation compared to **request_update**.

アプリケーションはasync_request_updateとupdateの呼び出しをまたぐで任意の共有メモリへのアクセスを同期する責任があります。async_request_updateを呼び出すソフトウェアスレッドは通常、update関数によって読まれる変数に値を書き込む必要があります。実装は、このような共有メモリの整合性については一切の保証を与えません。SystemCスレッドやメソッドプロセスといった、SystemCカーネルのコンテキストで実行される関数からメンバ関数async_request_updateを呼び出すことは推奨されませんが、カーネルからわかれたオペレーティングシステムスレッドのみです。async_request_updateの呼び出しはrequest_updateに比べてパフォーマンスが低下する可能性があります。

JEITA

6.15 sc_prim_channel

virtual void **update**();

CLARIFICATION: Member function **update** shall be called back by the scheduler during the update phase in response to a call to **request_update** or **async_request_update**. An application may override this member function in a primitive channel. The definition of this function in class **sc_prim_channel** itself does nothing.

When overridden in a derived class, member function **update** shall not perform any of the following actions:

a) Call any member function of class **sc_prim_channel** with the exception of member function **update** itself if overridden within a base class of the current object

b) Call member function **notify()** of class **sc_event** with no arguments to create an immediate notification

c) ADDITION: Call any of the member functions of class **sc_process_handle** for process control (**suspend** or **kill**, for example)

プロセス制御(例えば，suspendやkill)のための、sc_process_handleクラスのいずれかのメンバ関数の呼出し

CLARIFICATION: Member function **update** should not call interface methods of other channel instances. In particular, member function **update** should not write to any signals.

メンバ関数のupdateでは、他のチャネルインスタンスのインターフェイスメソッドを呼び出してはいけません。特に、メンバ関数updateは、どんなsignalへも書き込まないでください。

JEITA

6.16 sc_object

6.16.1 Description

Class **sc_object** is the common base class for classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**, and for the implementation-defined classes associated with spawned and unspawned process instances. The set of **sc_object**s shall be organized into an *object hierarchy*, where each **sc_object** has no more than one parent but may have multiple siblings and multiple children. Only module objects and process objects can have children.

An **sc_object** is a *child* of a module instance if and only if that object lies *within* the module instance, as defined in 4.1.4. An **sc_object** is a *child* of a process instance if and only if that object was created during the execution of the function associated with that process instance. Object P is a *parent* of object C if and only if C is a child of P.

An **sc_object** that has no parent object is said to be a *top-level* object. Module instances, spawned process instances, and objects of an application-defined class derived from class **sc_object** may be top-level objects.

Each call to function **sc_spawn** shall create a spawned process instance that is either a child of the caller or a top-level object. The parent of the spawned process instance so created may be another spawned process instance, an unspawned process instance, or a module instance. Alternatively, the spawned process instance may be a top-level object.

Each **sc_object** shall have a unique hierarchical name reflecting its position in the object hierarchy.

CLARIFICATION: Except where explicitly forbidden (as is the case for the classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**), objects of a class derived from **sc_object** may be deleted at any time. When such an **sc_object** is deleted, it ceases to be a child.

明確に禁じられている（sc_module、sc_port、sc_exportクラスとsc_prim_channelの場合のように）場合を除き、sc_objectの派生クラスのオブジェクトはいつでも削除できる。そのようなsc_objectが削除されるとき、それは子でなくなります。

6.16 sc_object

6.16.4 Constructors and CORRECTION: destructor

sc_object();
sc_object(const char*);

Both constructors shall register the **sc_object** as part of the object hierarchy and shall construct a hierarchical name for the object using the string name passed as an argument. Calling the constructor **sc_object(const char*)** with an empty string shall have the same behavior as the default constructor, that is, the string name shall be set to **"object"**.

The rules for composing a hierarchical name are given in 6.17.
階層名を形成するルールは6.17に書かれます。
If the constructor needs to substitute a new string name in place of the original string name as the result of a name clash, the constructor shall generate a single warning.

CHANGE: **sc_object**(const sc_object& arg);

The copy constructor shall create a new **sc_object** as if it had been created by the call **sc_object(arg.basename())**. In other words, the new object shall be a child of a module instance if created from the constructor of that module, or a child of a process instance if created from the function associated with that process. The string name of the existing object shall be used as the seed for the string name of the new object. Attributes or children of the existing object shall not be copied to the new object.

コピーコンストラクタは、sc_object(arg.basename())の呼び出しで作成されるのと同様の新しいsc_objectを作成します。言い換えれば、新しいオブジェクトは、そのモジュールのコンストラクタ、またはそのプロセスに関連付けられている関数から生成した時のプロセスインスタンスの子から生成された場合、モジュールインスタンスの子になります。既存のオブジェクトの文字列名は、新しいオブジェクトの文字列名のシードとして使用します。属性または既存のオブジェクトの子は、新しいオブジェクトにコピーできません。

6.16 sc_object

CHANGE: sc_object& **operator**= (const sc_object&);

The assignment operator shall not modify the hierarchical name or the parent of the destination object in the object hierarchy. In other words, the destination object shall retain its current position in the object hierarchy. Attributes and children of the destination object shall not be modified. **operator**= shall return a reference to ***this**.

代入演算子は、オブジェクト階層での代入先オブジェクトの親を変更してはならない。言い換えれば、代入先オブジェクトは、オブジェクト階層内の現在の位置を保持しなければならない。代入先オブジェクトの属性と子は変更してはいけない。
=演算子は、*thisへの参照を返さなければならない。

CLARIFICATION: virtual ~**sc_object**();

The destructor shall delete the object, shall delete any attribute collection attached to the object, and shall remove the object from the object hierarchy such that it is no longer a child.

デストラクタはオブジェクトを削除し、オブジェクトにアタッチされている属性のコレクションを削除し、そしてもはや子でないように、オブジェクトの階層からオブジェクトを削除する。

JEITA

6.16 sc_object

6.16.7 Functions for object hierarchy traversal

sc_object* **get_parent_object()** const;

Member function **get_parent_object** shall return a pointer to the **sc_object** that is the parent of the current object in the object hierarchy. If the current object is a top-level object, member function **get_parent_object** shall return the null pointer. CLARIFICATION: If the parent object is a process instance and that process has terminated, **get_parent_object** shall return either a pointer to the surviving process instance or a null pointer if it has been deleted.

親オブジェクトがプロセスインスタンスでそして、そのプロセスが終わったならば、get_parent_objectは生き残っているプロセスインスタンスへのポインターか、そのプロセスが削除されていればnullポインターを返します。

JEITA

6.16 sc_object

6.17 CHANGE: Hierarachical naming of objects and events

This clause describes the rules that determine the hierarchical naming of objects of type **sc_object** and of hierarchically named events. The naming of events that are not hierarchically named is described in 6.10.4.

この章は、sc_object型の階層名イベントのオブジェクトの階層名を決定するルールを説明しています。階層名でないイベントの名前は、6.10.4に記載されている。

```
namespace sc_core {

    bool sc_hierarchical_name_exists( const char* );

} // namespace sc_core
```

Function **sc_hierarchical_name_exists** shall return the value **true** if and only if the value of the string argument exactly matches the hierarchical name of an **sc_object** or of a hierarchically named event. If the value of the string argument exactly matches an implementation-defined event name, function **sc_hierarchical_name_exists** shall return the value **false** unless the string argument also matches a hierarchical name.

sc_hierarchical_name_exists関数は、文字列引数の値が正確にsc_object、または階層名のイベントと一致する場合のみtrue値を返します。文字列引数の値が実装定義のイベント名と一致し文字列の引数が階層名と一致しない場合、sc_hierarchical_name_exists関数はfalse値を返します。

JEITA

6.16 sc_object

A hierarchical name shall be composed of a set of string names separated by the period character '.', starting with the string name of a top-level **sc_object** instance and including the string name of each module instance or process instance descending down through the object hierarchy until the current **sc_object** or **sc_event** is reached. The hierarchical name shall end with the string name of the **sc_object** or **sc_event** itself.

階層名は、トップレベルのsc_objectインスタンスの文字列名で始まり、現行のsc_objectもしくはsc_eventに達するまでオブジェクト階層に沿って下位の各モジュールのインスタンスまたはプロセスインスタンスの文字列名を含む、ピリオド'.'で区切られた文字列名のセットで構成されます。階層名はsc_objectもしくはsc_eventそれ自体の文字列で終わります。

Hierarchical names are case-sensitive.

階層名は、大文字と小文字が区別されます。

It shall be an error if a string name includes the period character (.) or any white-space characters. It is strongly recommended that an application limit the character set of a string name to the following:

a) The lower-case letters a-z
b) The upper-case letters A-Z
c) The decimal digits 0-9
d) The underscore character _

文字列名は、ピリオド(.)または任意の空白文字が含まれている場合はエラーです。アプリケーションが次のように文字列名の文字セットを制限することを強く勧めします:

JEITA

6.16 sc_object

An implementation may generate a warning if a string name contains characters outside this set but is not obliged to do so.

文字列名に、このセット以外の文字が含まれていたら実装はワーニングを生成できます、これは必須ではありません。

There shall be a single global namespace for hierarchical names. Each **sc_object** and each hierarchically named **sc_event** shall have a unique non-empty hierarchical name. An implementation shall not add any names to this namespace other than the hierarchical names of **sc_object**s explicitly constructed by an application and the hierarchical names of hierarchically named events.

階層名には単一のグローバルな名前空間があります。各sc_object、各階層名のsc_eventは、一意の空でない階層名を持つものとする。実装は、アプリケーションで明示的に生成されたsc_objectsの階層名や、階層名いべんとの階層名以外に、この名前空間に任意の名前を追加できません。

The constructor shall build a hierarchical name from the string name (either passed in as an argument or the default name **"object"** for an **sc_object** or **"event"** for an **sc_event**) and test whether that hierarchical name is unique. If it is unique, that hierarchical name shall become the hierarchical name of the object. If not, the constructor shall call function **sc_gen_unique_name**, passing the string name as a seed. It shall use the value returned as a replacement for the string name and shall repeat this process until a unique hierarchical name is generated.

コンストラクタは、(引数として渡される、またはsc_objectに対して“object”、sc_eventには“event”のデフォルト名の)文字列の名前から階層名を構築し、その階層名が一意であるかどうかをテストしなければならない。それが一意である場合、その階層名は、オブジェクトの階層名になります。そうでない場合は、コンストラクタは、シードとして文字列名を渡す、sc_gen_unique_name関数を呼び出します。これは、文字列名の代わりに返される値を使用し、一意の階層名が作られるまで、このプロセスを繰り返すものとします。

JEITA

6.16 sc_object

If function **sc_gen_unique_name** is called more than once in the course of constructing any given object, the choice of seed passed to **sc_gen_unique_name** on the second and subsequent calls shall be implementation-defined but shall in any case be either the string name passed as the seed on the first such call or shall be one of the string names returned from **sc_gen_unique_name** in the course of constructing the given object. In other words, the final string name shall have the original string name as a prefix.

sc_gen_unique_name関数が、特定のオブジェクトを生成する過程で複数回呼び出された場合、2回目以降の呼び出しでsc_gen_unique_nameに渡されるシードの選択は実装定義になるが、どのような場合でも最初の呼び出しでシードとして渡された文字列名か、指定されたオブジェクトを生成する過程でsc_gen_unique_nameから返される文字列名のいずれかでなければならない。言い換えれば、最後の文字列名は、接頭辞として元の文字列名を持つものとする。

JEITA

7.3 sc_signal_inout_if

7.3.3 Member functions

ADDITION: The following text describes the expected definitions of the member functions when overridden in a channel that implements this interface. The precise semantics will be channel-specific.

後のテキストは、このインターフェイスを実装するチャネルがオーバーライドされた時メンバ関数の期待する定義を説明しています。正確な意味は、チャネル特有のものが使われる。

virtual sc_writer_policy **get_writer_policy**() const { return SC_ONE_WRITER; }

Member function **get_writer_policy** shall return the value of the writer policy for the channel instance. Unless overridden in a channel, this function shall return the value SC_ONE_WRITER for backward compatibility with previous versions of this standard.

メンバ関数get_writer_policyは、チャネルインスタンスの書き込みポリシーの値を返さなければいけません。チャネルでオーバーライドされない限り、この関数は、この規格より以前のバージョンとの下位互換性のために値SC_ONE_WRITERを返さなければいけません。

virtual void **write**(const T&) = 0;

Member function **write** shall modify the value of the channel such that the channel appears to have the new value (as returned by member function **read**) in the next delta cycle but not before then. The new value is passed as an argument to the function.

ADDITION: Member function **write** shall honor the value of the writer policy set for the channel instance, that is, it shall permit either one writer or many writers.

メンバ関数writeは、チャネルのインスタンスに設定される、一つの書き込みか複数の書き込みのどちらを許すかの書き込みポリシーの値に従わなければいけない。

7.4 sc_signal

7.4.4 Reading and writing signals

A signal is *read* by calling member function **read** or **operator const T&** ().

ADDITION: A signal is *written* by calling member function **write** or **operator=** of the given signal object. If the template argument WRITER_POLICY has the value SC_ONE_WRITER, it shall be an error to write to a given signal instance from more than one process instance at any time during simulation. If the template argument WRITER_POLICY has the value SC_MANY_WRITERS, it shall be an error to write to a given signal instance from more than one process instance during any given evaluation phase, but different process instances may write to a given signal instance during different delta cycles. A signal may be written during elaboration (including the elaboration and simulation callbacks as described in 5.4) to initialize the value of the signal. A signal may be written from function **sc_main** during elaboration or while simulation is paused, that is, before or after the call to function **sc_start**.

信号は、メンバ関数writeの呼び出しまたは指定された信号オブジェクトの=演算子によって書かれます。テンプレート引数WRITER_POLICYの値がSC_ONE_WRITERの場合、シミュレーション中の任意の時点で複数のプロセスインスタンスから指定された信号のインスタンスへの書き込みはエラーになります。テンプレート引数WRITER_POLICYの値がSC_MANY_WRITERSの場合、任意の評価段階で複数のプロセスインスタンスから指定された信号のインスタンスへの書き込みはエラーになりますが、異なるプロセスのインスタンスは、異なるデルタサイクル中に特定の信号インスタンスに書き込むことができます。信号は、値を初期化するため(5.4で説明されるエラボレーションとシミュレーションのコールバックを含む)エラボレーション中に書けます。信号は、sc_start関数が呼び出される前または後の、エラボレーション中またはシミュレーションが一時停止している間にsc_main関数から書くこともできます。

JEITA

7.4 sc_signal

Signals are typically read and written during the evaluation phase but the value of the signal is only modified during the subsequent update phase. If and only if the value of the signal actually changes as a result of being written, an event (the *value-changed event*) shall be notified in the delta notification phase that immediately follows.

If a given signal is written on multiple occasions within a particular evaluation phase, CLARIFICATION: or during elaboration, or from function **sc_main**, the value to which the signal changes in the immediately following update phase shall be determined by the most recent write, that is, *the last write wins*.

もしある信号が、特定の評価フェーズ内に、あるいはエラボレーション中、あるいはsc_main関数から、複数回書き込まれた場合、信号の変化する値はすぐに次の更新の段階で、もっとも新しい書き込みによって決定されます、最後の書き込みが勝ります。

JEITA

7.4 sc_signal

7.4.6 register_port

virtual void **register_port**(sc_port_base&, const char*);

Member function **register_port** of class **sc_interface** shall be overridden in class **sc_signal**, and shall perform the following error check. ADDITION: If the template argument WRITER_POLICY has the value SC_ONE_WRITER, it shall be an error to bind more than one port of type **sc_signal_inout_if** to a given signal. If the WRITER_POLICY has the value SC_MANY_WRITERS, one or more ports of type **sc_signal_inout_if** may be bound to a given signal.

テンプレート引数WRITER_POLICYの値がSC_ONE_WRITERの場合、ある信号を複数のsc_signal_inout_if型のポートにバインドするとエラーになります。WRITER_POLICYの値が、SC_MANY_WRITERSの場合、sc_signal_inout_if型の1つまたは複数のポートが指定された信号にバインドすることができます。

JEITA

7.4 sc_signal

7.4.8 Member functions for writing

virtual void **write**(const T&);

Member function **write** shall modify the value of the signal such that the signal appears to have the new value (as returned by member function **read**) in the next delta cycle but not before then. This shall be accomplished using the update request mechanism of the primitive channel. The new value is passed as an argument to member function **write**. CLARIFICATION: Member function **write** may be called during elaboration, in which case the update request shall be executed during the initialization phase.

更新要求が初期化フェーズ中に実行される場合には、メンバ関数writeはエラボレーション中に呼び出されることができます。

operator=

The behavior of **operator=** shall be equivalent to the following definitions:

CLARIFICATION: sc_signal<T,WRITER_POLICY>& **operator=** (const T& arg) { write(arg); return *this; }

CLARIFICATION: sc_signal<T,WRITER_POLICY>& **operator=** (const sc_signal<T,WRITER_POLICY>& arg) { write(arg.read()); return *this; }

JEITA

7.8 sc_in

7.8.3 Member functions

The constructors shall pass their arguments to the corresponding constructor for the base class **sc_port**.

CHANGE: The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**. Member function **bind** shall call member function **bind** of the base class **sc_port**, passing through their parameters as arguments to function **bind**, in order to bind the object of class **sc_in** to the channel or port instance passed as an argument.

演算子()の実装は、仮想メンバ関数bindをコールすることで、その目的を達成しています。

JEITA

7.23 sc_fifo

7.23.1 Description

Class **sc_fifo** is a predefined primitive channel intended to model the behavior of a fifo, that is, a first-in-first-out buffer. CLARIFICATION: In this clause, *fifo* refers to an object of class **sc_fifo**. Each fifo has a number of *slots* for storing values. The number of slots is fixed when the object is constructed.

この条項で、fifoはsc_fifoクラスのオブジェクトを参照します。

JEITA

7.30 sc_event_queue

7.30.4 Constructors

CORRECTION:
sc_event_queue(sc_module_name name_= sc_module_name(sc_gen_unique_name("event_queue")));
This constructor shall pass the module name argument through to the constructor for the base class **sc_module**.

このコンストラクタは、ベースクラスsc_moduleのコンストラクタに対するモジュール名引数を渡します。

JEITA

8.2 Common characteristics

8.2.3 Base class default word length

The default word length of a data type base class shall be used where its default constructor is called (implicitly or explicitly). The default word length shall be set by the *length parameter* in context at the point of construction. A length parameter may be brought into context by creating a length context object. Length contexts shall have local scope and by default be activated immediately. Once activated, they shall remain in effect for as long as they are in scope, or until another length context is activated. Activation of a length context shall be deferred if its second constructor argument is SC_LATER (the default value is SC_NOW). A deferred length context can be activated by calling its member function **begin**.

Length contexts shall be managed by a global length context stack. When a length context is activated, it shall be placed at the top of the stack. A length context may be deactivated and removed from the top of the stack by calling its member function **end**. The **end** method shall only be called for the length context currently at the top of the context stack. CLARIFICATION: A length context is implicitly deactivated and removed from the stack when it goes out of scope. A deferred length context that has been activated by calling its member function **begin** should be explicitly deactivated and removed from the stack by calling its member function **end**. The current context shall always be the length context at the top of the stack.

長さコンテキストが暗黙的に無効にされ、それがスコープの外に出るときにスタックから削除されます。自身のメンバ関数beginを呼び出すことによりアクティブにされた延長コンテキストは、メンバ関数endを呼び出すことで明示的に無効にし、スタックから削除されなければいけません。現在のコンテキストは常にスタックの最上位にある長さコンテキストでなければいけません。

JEITA

8.3 String literals

A string literal representation may be used as the value of a SystemC numeric or vector type object. It shall consist of a standard prefix followed by a magnitude expressed as one or more digits.

The magnitude representation for SystemC integer types shall be based on that of C++ integer literals.

The magnitude representation for SystemC vector types shall be based on that of C++ unsigned integer literals.

The magnitude representation for SystemC fixed-point types shall be based on that of C++ floating literals but without the optional floating suffix.

CORRECTION: Where alphabetic characters appear in the prefix or magnitude representations, each individual character may be a lower case letter or an upper case letter. A string literal representation shall not be case sensitive.

アルファベット文字は、接頭辞または大きさの表現で表示される場合には、個々の文字は、小文字または大文字にできます。文字列リテラルの表現は、大文字と小文字を区別してはならない。

Table 5—String literal representation

sc_numrep	Prefix (not case sensitive)	Magnitude format
CLARIFICATION: SC_NOBASE	implementation-defined	implementation-defined
SC_DEC	0d	decimal

8.4 *sc_value_base †*

8.4.4 Member functions

virtual void **concat_clear_data**(bool to_ones=false);

Member function **concat_clear_data** shall set every bit in the *sc_value_base*† object to the state provided by the argument.

CHANGE: virtual bool **concat_get_ctrl**(*implementation-defined** dst_p , int low_i) const;

Member function **concat_get_ctrl** shall copy control data to the packed-array given as the first argument, starting at the bit position within the packed-array given by the second argument. The return value shall always be **false**. The type of the first argument shall be a pointer to an unsigned integral type.

最初の引数の型は符号なし整数型へのポインタです。

CHANGE: virtual bool **concat_get_data**(*implementation-defined** dst_p , int low_i) const;

Member function **concat_get_data** shall copy data to the packed-array given as the first argument, starting at the bit position within the packed-array given by the second argument. The return value shall be **true** if the data is non-zero; otherwise, it shall be **false**. The type of the first argument shall be a pointer to an unsigned integral type.

最初の引数の型が符号なし整数型へのポインタです。

JEITA

8.6 Finite-precision integer types

8.6.4.7 Arithmetic, bitwise, and comparison operators

U << i	sc_unsigned	sc_unsigned left-shift
S << U	sc_signed	sc_signed left-shift
U <<= i	sc_unsigned&	sc_unsigned assign left-shift
U >> i	sc_unsigned	sc_unsigned right-shift
S >> U	sc_signed	sc_signed right-shift
U >>= i	sc_unsigned&	sc_unsigned assign right-shift

CORRECTION: The left shift operator shall return a result with a word length that is equal to the word length of its **sc_unsigned** operand plus the right (integer) operand. Bits added on the right-hand side of the result shall be set to zero.

左シフト演算子は、sc_unsignedオペランドと右向き(整数)オペランドを足した結果の語長を返します。右側に追加されるビットはゼロに設定されます。

CORRECTION: The right shift operator shall return a result with a word length that is equal to the word length of its **sc_unsigned** operand. Bits added on the left-hand side of the result shall be set to zero.

右シフト演算子は、sc_unsignedオペランドの語長と等しい語長を返します。左側に追加されるビットはゼロに設定されます。

9~17章サマリ

- 9~17章の主な変更点（UtilityとTLM2.0に関する章）
  - sc_verbosityクラスの追加により、レポート生成時の出力レベルを設定することができるようになった。
  - sc_vectorクラスの追加により、システムの構築が容易になった。
  - バージョンとcopyright関連のdefine、定数の追加（TLMのdefine/定数とは独立して設定可能）
  - TLM関連の変更
    - 準拠レベルの用語を定義（TLM-2.0基本プロトコル準拠、TLM-2.0基本プロトコル準拠、TLM-2.0カスタム・プロトコル準拠）
    - 汎用ペイロードに、オプション・アトリビュートが追加され、DMIとデバッグ・トランスポート・インターフェースでターゲットがバイト・イネーブル等を考慮できるようになった（既存のモデルはそのまま使える）。

9. SystemC utilities

- 9.1.6 sc_trace
  - CORRECTION: sizeofの引数がlongになっていたものを、正しく修正
    - void sc_trace (sc_trace_file* , const sc_dt::int64& , const std::string& , int width = 8 * sizeof (sc_dt::int64));
    - void sc_trace (sc_trace_file* , const sc_dt::int64* , const std::string& , int width = 8 * sizeof (sc_dt::int64));
    - void sc_trace (sc_trace_file* , const sc_dt::uint64& , const std::string& , int width = 8 * sizeof (sc_dt::uint64));
    - void sc_trace (sc_trace_file* , const sc_dt::uint64* , const std::string& , int width = 8 * sizeof (sc_dt::uint64));
- 9.2.1 Class definition (sc_report)
  - ADDITION: verbosity の設定ができるようになった

```
enum sc_verbosity {
  SC_NONE = 0,
  SC_LOW = 100,
  SC_MEDIUM = 200,
  SC_HIGH = 300,
  SC_FULL = 400,
  SC_DEBUG = 500
};
class sc_report: public std::exception
{
public:
    :
   int get_verbosity () const;
    :
}
```

9. SystemC utilities

- 9.2.1 sc_verbosity
  - ADDITION:新規にクラス追加
  - sc_verbosityのenumは、verbosity (indicitive verbosity, indicativeのtypo？) のレベルを示すもので、sc_report_handlerクラスのメンバ関数set_verbosity_level 、reportへの引数として渡すことができる。

- 9.2.7 Member Functions
  - ADDITION: メンバ関数の一覧にget_verbosity () を追加
    - int get_verbosity () const;

- 9.3.2 Class Definition (sc_report_handler)
  - ADDITION: sc_verbosityに関する変更を反映

```
class sc_report_handler {
public:
    static void report ( sc_severity , const char* msg_type , const char* msg , const char* file , int line ) ;
    static void report ( sc_severity , const char* msg_type , const char* msg , int verbosity, const char* file ,
     int line ) ;
     :
    int set_verbosity_level ( int ) ;
    int get_verbosity_level () ;
…}
#define SC_REPORT_INFO_VERB ( msg_type , msg, verbosity ) ¥
sc_report_handler::report ( SC_INFO , msg_type , msg , verbosity, __FILE__ , __LINE__ )
```


9. SystemC utilities

- 9.3.5 report (ADDITION)
  - Verbosityを引数に持つreport関数の追加

```
static void report ( sc_severity , const char* msg_type , const char* msg , const char* file , int line ) ;
static void report ( sc_severity , const char* msg_type , const char* msg , int verbosity, const char* file, int
    line ) ;
```

  - メンバ関数reportは、レポートを生成し、以下に示す適切なアクションを起動する。
  - メンバ関数reportは、第一引数で渡されるseverityと、第二引数で渡されるメッセージタイプを使って、直前に呼ばれた関数呼び出し (set_actions, stop_after, suppress, force) の結果として実行されるべきアクションのセットを決定する。メンバ関数reportは、sc_reportクラスのオブジェクトを生成し、5つの引数全てを使って初期化を行う。また、このオブジェクトをメンバ関数set_handlerにより、セットされたハンドラに対して引き渡す。sc_reportクラスのオブジェクトは、メンバ関数reportの呼び出しが終われば消滅する。例外は、SC_CACHE_REPORTがセットされた場合で、この時、get_cached_reports関数の呼び出しにより、オブジェクトを取得することができる。この実装は、各インスタンス毎にsc_reportオブジェクトに対して独立したキャッシュを持たなければならず、また、プロセス外部からのreport関数を呼び出された場合は、単一のグローバルなレポートキャッシュを持たなければならない。そのようなキャッシュはそれぞれ、最新のレポートのみを格納する。
  - メンバ関数reportは、実行されるべきアクションのセットを決定する役割を持つ。set_handler関数によりセットされるhandler関数は、これらのアクションを実行する役割を持つ。メンバ関数reportは、レポート数の合計数を管理し、9.3.7に示すレポートを生成する。これらの合計数はアクションが実行されたかどうかに関わらず加算される。ただし、レポートがverbosityレベルにより無視された場合は除く。この場合の合計数は加算されない。
  - verbosity引数が指定され、そのseverity引数の値がSC_INFOの場合で、かつ、verbosity引数の値が最大verbosityレベルより大きい場合、メンバ関数reportはアクションを実行したり、合計数を加算したりしない。verbosity引数がなく、severityの値がSC_FIFOの場合、メンバ関数reportは、verbosity引数としてSC_MEDIUMを指定された場合と同様に動作する。severity引数がSC_INFO以外の値を採る場合、実装はVerbosity引数を無視する。
  - 以下略。

9. SystemC utilities

- 9.3.9 Verbosity level (ADDITION)
  - 最大verbosityレベルは、単一のグローバル数量(global quantity)で、severiyyレベルがSC_INFOであるレポートに対してのみ適用される。severityレベルがSC_INFOである個々のレポートで、verbosityレベルが最大verbosityレベルより大きなものは無視される。このため、実装では、そのレポートに関連するアクションは全て実行しない。
  - int set_verbosity_level (int) ;
    - メンバ関数set_verbosity_levelは、最大verbosityレベルを引数で渡された値に設定し、設定前の当たりを関数の戻り値として返す。
  - int get_verbosity_level () ;
    - メンバ関数get_verbosity_levelは、最大verbosityレベルの値を返す。
- 9.4 sc_exception
  - CLARIFICATION: sc_reportクラスがstd::exceptionクラスから派生していることを明記。

9. SystemC utilities

- 9.5 sc_vector (ADDITION)
  - sc_vectorクラスは、モジュール、チャネル、ポート、エキスポート、その他の任意の型のオブジェクトの配列を構築するために使用する。これは、反復処理や、パラメタライズされた構造の記述を容易にするための手段を提供する。sc_vectorクラスは、配列から要素を取り出したり、ポートで構成される配列をバインドしたりするためのメンバ関数を提供する。
  - sc_assemble_vector関数は(関連するプロキシ・クラスsc_vector_assemblyも含めて)、個々のモジュール内のオブジェクトのセットから、オブジェクトの配列を組み上げるための仕組みを提供する。これらは、多くの場面でvectorの代わりに使用できる(例えばポートのバインド)。
  - sc_vectorクラスは、モジュールで構成される配列を生成する際に、ユーザ定義のモジュール・コンストラクタ引数を引き渡すための仕組みを提供している。本節に限り、*vector*という用語は、適切なテンプレート引数Tを定義した、sc_vector<T>型を表すものとする。

9. SystemC utilities

- 9.5 sc_vector (ADDITION)
  - 9.5.3. 制限事項
    - テンプレートsc_vector<T>において、引数Tは、sc_objectから派生するものとする。カスタムの*creator*とともに使用する場合をのぞき、型Tはコンストラクタを提供するが、引数としてconst char*から変換可能な型を一つだけ渡す。アプリケーションが追加で要素のコンストラクタに対して引数を渡す必要がある場合は、*creator*関数か、または、関数オブジェクトを使用することができる。
    - sc_vector<T>型のオブジェクトを生成する場合、その制約はテンプレート引数Tに依存する。T型に対するあらゆる制約を満たした上で、vectorはシミュレーション時にダイナミックに生成できる。例えば、モジュールで構成されるvectorは、エラボレーション時にのみ生成できる。
    - vectorのサイズは、vectorが生成された時、または、メンバ関数initを呼び出した時に、一度だけセットすることができる。vectorは動的にサイズを変更することはできない。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - 9.5.4. コンストラクタとデコンストラクタ
    - sc_vector ();
    - explicit sc_vector (const char*);
    - sc_vector (const char* , size_type);
    - template< typename Creator >
      sc_vector (const char* , size_type , Creator);
    - virtual ~sc_vector ();
  - 説明
    - コンストラクタは、文字列引数をベースクラスsc_objectのコンストラクタに渡す。これは、モジュール階層における、vectorのインスタンス名をセットするためである。
    - デフォルト・コンストラクタは、固有の文字列を生成するために、sc_gen_unique_name ("vector") を呼び出す。その文字列は、ベースクラスsc_objectのコンストラクタに渡される。第二引数が与えられた場合、コンストラクタはメンバ関数initを呼び出し、第二引数の値をinitの第一引数として渡す。第二引数がない場合、コンストラクタは空のvectorを生成する。この場合、メンバ関数initを明示的に呼び出すことでイニシャライズができる。
    - 第二、第三引数が与えられた場合、コンストラクタはメンバ関数initを呼び出し、第二、第三引数の値を、initの第一、第二引数として、それぞれ引き渡す。
    - デコンストラクタは、vectorの全ての要素を削除する。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - 9.5.5. initとcreate_element
    - void **init** (size_type);
    - static T* **create_element** (const char* , size_type);
  - 説明
    - スタティックなメンバ関数create_elementは、第一引数で指定された名前でT型の新しいオブジェクトへのポインタを生成し、そのポインタを戻り値として返す。新たに生成されたオブジェクトは、vectorと同じ親を持つ。すなわちvectorの要素は、SystemCのオブジェクト階層上、vectorオブジェクトと兄弟である（同じ階層に位置する？）。この機能を提供する理由は、アプリケーションがユーザ定義のコンストラクタ引数を、vectorの各要素に引き渡す目的で、独自の関数を提供する場合を考慮したためである。メンバ関数initは、引数で与えられた数のT型のオブジェクトを生成し、vectorとこれらのオブジェクトを関連づける。各オブジェクトは、create_element関数を呼び出すことにより生成される。この時、第一引数は要素の名称としてセットされ、第二引数はvector中の要素の数としてセットされる。要素数は0からカウントされる。
    - 各要素の名称はsc_gen_unique_name (this->basename ()) 関数を呼び出すことで決まる。メンバ関数initの引数に0を指定して呼び出すことはできるが、効果はない。
    - 特定のvectorに対して、0より大きな値を引数として複数回呼び出すことはできない。第二引数を指定したコンストラクタはinitを呼び出すため、0より大きな値のサイズを指定してvectorを生成した上で、同様に0より大きな値のサイズを指定してinitを呼び出した場合はエラーになる。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - 9.5.5. initとcreate_element (続き)
    - template< typename Creator >
      void **init** (size_type , Creator c);
  - 説明
    - vectorの各要素を生成するため、メンバ関数テンプレートinitは、create_elementを呼び出す代わりに、第二引数を使用する。第二引数は、関数であったり、関数オブジェクトである場合がある。第二引数で渡されるcの値は、以下のようになっている必要がある。

      T* placeholder1 = c ((const char*) placeholder2, (size_type) placeholder3);
    - 言い換えると、実際の引数はcreate_elementの代わりに呼び出せなければならない。このことにより、文字列だけを渡すのではなく、追加のコンストラクタ引数をvectorの各要素に渡すために、creatorを使うことができるようになる。
    - V.init (N, sc_vector<T>::create_element) とV.init (N) の表記は全てのvector Vで同じである。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - 9.5.6. kind, size, get_elements
    - virtual const char* **kind** () const;
    - size_type **size** () const;
    - const std::**vector**<sc_object*>& **get_elements** () const;
  - 説明
    - メンバ関数kindはstring"sc_vector"を戻り値とする。
    - メンバ関数sizeは、vector中の要素数を戻り値とする。初期化されていないvectorの場合、サイズは0となる。一旦0以外の値がセットされた後は、サイズは修正できない。
    - sc_vectorクラスのメンバ関数get_elementsは、sc_vectorの要素へのポインタを含む、std::vectorへのconst型参照を戻り値とする。この時、ポインタは要素につき1つ存在し、順序は要素の順と同じである。std::vectorは、sc_vectorと同じサイズを持つ。この参照は、sc_vectorオブジェクトのライフタイム期間内で有効である。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - 9.5.7. operator [] と at
    - T& **operator []** (size_type);
    - const T& **operator []** (size_type) const;
    - T& **at** (size_type);
    - const T& **at** (size_type) const;
  - 説明
    - operator []とメンバ関数atは、vector内の各インデックスの位置に格納されたオブジェクトへの参照、またはconst-qualified参照を、戻り値とする。この参照は、vectorのライフタイム期間内で有効である。引数の値がvectorのサイズのより大きい場合、operator []の動作は保証されない。一方、メンバ関数atはこのエラーを検知し、エラーをレポートする。&V [i] + j == &V [i + j]の関係が成り立つかどうかは、全てのvector V、全てのインデックスiおよびjで未定義である。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - bind
    - template< typename ContainerType, typename ArgumentType >
      iterator **bind** (sc_vector_assembly<ContainerType,ArgumentType>);
    - template< typename BindableContainer >
      iterator **bind** (BindableContainer&);
    - template< typename BindableIterator >
      iterator **bind** (BindableIterator , BindableIterator);
    - template< typename BindableIterator >
      iterator **bind** (BindableIterator , BindableIterator , iterator);
    - template< typename ContainerType, typename ArgumentType >
      iterator **operator ()** (sc_vector_assembly<ContainerType,ArgumentType> c);
    - template< typename ArgumentContainer >
      iterator **operator ()** (ArgumentContainer&);
    - template< typename ArgumentIterator >
      iterator **operator ()** (ArgumentIterator , ArgumentIterator);
    - template< typename ArgumentIterator >
      terator **operator ()** (ArgumentIterator , ArgumentIterator , iterator);
  - 説明
    - メンバ関数bindとoperator () はそれぞれ、*thisで与えられるvectorの要素に対し、関数呼び出し時の引数で決まるvectorの要素を、要素単位でバインディングを行う。それぞれの場合において、実装では、現在のvectorの個々の要素が持つbindまたはoperator () を、それぞれ呼び出すことでバインドを行う。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - 説明 (続き)
    - 部分的なvectorのバインディングのため、以下のような形式がある。
    - 引数が一つまたは二つの場合、現在のオブジェクトの最初の要素からバインドを開始する。引数が三つの場合、第三引数で指定されたイテレータが参照する要素からバインドを開始する。第三引数が現在のオブジェクトの要素の範囲外の場合、動作は保証されない。
    - 引数が一つの場合、引数で渡されたコンテナの最初の要素に対して、バインドを開始する。この場合の第一引数は、vectorまたは、vector assemblyを指定することができ、そのコンテナの各要素をバインドする。第二引数および第三引数を持つ場合、第一引数で渡されたイテレータが参照する要素に対してバインドを行うが、第二引数で渡されたイテレータが参照する要素、あるいはその後続の要素は、バインドされない。
    - それぞれの場合において、メンバ関数bindとoperator () は、バインド終了後、現在のvectorである*this内で、バインドされていない最初の要素を返す。一つのvectorは、複数回バインドされてもよいが、これは個々の要素のバインドポリシーが許す場合にのみ可能である。言い換えると、bindを複数回呼び出すことは可能であるが、それぞれの呼び出しで、vectorの異なる要素をバインドする場合に限る。

```
iterator bind ( BindableContainer& );                          // 最初の要素から、最後までバインド
iterator bind ( BindableIterator , BindableIterator );         // 最初の要素から、第二引数までバインド
iterator bind ( BindableIterator , BindableIterator , iterator ); // 第三引数が参照する要素から、第二引数までバインド
```

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - sc_assemble_vector
    - template< typename T, typename MT >
      sc_vector_assembly<T,MT> sc_assemble_vector (sc_vector<T> & , MT (T::*member_ptr));
  - 説明
    - 関数sc_assemble_vectorはsc_vector_assemblyクラスのオブジェクトを戻り値とする。sc_vector_assemblyは、メンバ関数begin、end、cbegin、cend、size、get_elements、operator []、at、bind、operator () を提供することにより、sc_vectorに対するプロキシの役割を果たす。これらメンバ関数はそれぞれ、このプロキシで表現されるvectorに対して、sc_vectorクラスの対応するメンバ関数と同等の動作をする。
    - 関数sc_assemble_vectorの最初の引数は、sc_vector<T>型のオブジェクトであり、型Tは、sc_moduleクラスから派生した型である。言い換えると、最初の引数はモジュールで構成するvectorでなければならない。
    - アプリケーションが関数sc_assemble_vectorへ渡す第二引数は、ユーザ定義のモジュール・クラスのメンバである、サブ・オブジェクトのアドレスでなければならない。ユーザ定義のモジュールクラスは、最初の引数で渡されるvectorの要素の型を表す。言い換えると、第二引数は、最初の引数で指定されたモジュールのメンバでなければならない。

9. SystemC utilities

- 9.5 sc_vector (ADDITION、続き)
  - 説明 (続き)
    - sc_vector_assemblyクラスのオブジェクトが持つ、どのメンバ関数に対して呼び出しを行った場合でも、プロキシで表現されるvectorの要素に対して有効になる。すなわち、第二引数で指定されるサブ・オブジェクト・メンバは、実際には、第一引数で指定されるベクタの全てのメンバに対して展開される。これらのサブ・オブジェクトは、これらのメンバ関数の各動作という観点では、単一のvectorのメンバのように見える。
    - sc_vector_assemblyクラスのメンバ関数get_elementsは、型のオブジェクトstd::vector<sc_object*>を値で返す。このstd::vectorの各要素は、vector assembly中のメンバのポインタを、静的にsc_object*型にキャストすることでセットされる。
    - アプリケーションは、sc_assemble_vectorを呼び出した場合を除いては、sc_vector_assemblyクラスのオブジェクトを生成しない。
    - sc_vector_assemblyクラスはコピーできる。

9. SystemC utilities

- 9.6 Utility Function
  - ADDITION: バージョンとcopyright関連のdefine、定数の追加
    - #define SC_VERSION_MAJOR *implementation_defined_number // For example, 2*
    - #define SC_VERSION_MINOR *implementation_defined_number // 3*
    - #define SC_VERSION_PATCH *implementation_defined_number // 4*
    - #define SC_VERSION_ORIGINATOR *implementation_defined_string // "OSCI"*
    - #define SC_VERSION_RELEASE_DATE *implementation_defined_date // "20110411"*
    - #define SC_VERSION_PRERELEASE *implementation_defined_string // "beta"*
    - #define SC_IS_PRERELEASE *implementation_defined_bool // TRUE*
    - #define SC_VERSION *implementation_defined_string // "2.3.4_beta-OSCI"*
    - #define SC_COPYRIGHT *implementation_defined_string*
    - extern const unsigned int **sc_version_major;**
    - extern const unsigned int **sc_version_minor;**
    - extern const unsigned int **sc_version_patch;**
    - extern const std::string **sc_version_originator;**
    - extern const std::string **sc_version_release_date;**
    - extern const std::string **sc_version_prerelease;**
    - extern const bool **sc_is_prerelease;**
    - extern const std::string **sc_version_string;**
    - extern const std::string **sc_copyright_string;**

9. SystemC utilities

- 9.6 Utility Function (続き)
  - ADDITION: バージョンとcopyright関連のdefine、定数の追加
    - IEEE_1666_SYSTEMC マクロの値は、本バージョンでは201101L 固定とする。
    - 全ての実装が、ここで定義されているマクロを定義しなければならない。また、アプリケーションが、どの実装上で実行されているかがわかるように、実装依存のマクロを追加することを推奨する。
    - TLM-2.0でも、同様のマクロが定義されているが(11.8.2章参照、TLM_xxx)、その値と本章のマクロ(SC_xxx)の値は独立したものとして扱う。
    - *implementation_defined_number*の値は、 [0-9]で表される10進数字列で表し、クォーテーションマークで囲まない。
    - SC_VERSION_PRERELEASEは、 [A-Z] [a-z] [0-9]_ のセットで表され、クォーテーションマークで囲む。
    - SC_VERSION_RELEASE_DATEは、ISO 8601規格のフォーマットに従い、YYYYMMDD形式で表し、クォーテーションマークで囲む。
    - SC_IS_PRERELEASE フラグは、FALSEまたはTRUEの値を取り、クォーテーションマークでは囲まない。
    - バージョンは、“major.minor.patch_prerelease-originator”または、“major.minor.patch-originator”の形式でセットされる。prereleaseがあるかどうかは、SC_IS_PRERELEASEフラグの有無によって決まる。
    - SC_COPYRIGHTには適切な文字列をセットする。 使い方としては、アプリケーションがこのマクロを三書して、コンソールウインドウやlogファイルに出力することを想定している。
    - 各定数は、マクロで定義された値で初期化する。

9. SystemC utilities

- 9.6 Utility Function (続き)
  - ADDITION: バージョンとcopyright関連の関数の追加

const char* sc_release ()
  - sc_release関数は、Cストリングに変換したsc_version_stringの値を返す。

const char* sc_version ();
  - sc_version関数は、実装依存のストリングを返す。これは、このストリングがSystemCクラスライブラリ実装のバージョンに関する情報を示すことを意図している。アプリケーションは、コンソールウインドウやログファイルにこの情報を出力できる。

const char* sc_copyright ();
  - sc_copyright メソッドは、Cストリングに変換したコピーライト・ストリングの値を返す。

10. Overview of TLM-2.0

- 章のタイトルに"to TLM-2.0"を追加。
- 冒頭の説明を修正

  本ドキュメントはOSCIトランザクション・レベル・モデリング標準バージョン2.0のリファレンス・マニュアルである。

  → 以下の節は、OSCIトランザクション・レベル・モデリング標準バージョン2.0を定義する。この標準はTLM2.0としても知られている。
- 10.1 Compliance with the TLM-2.0 standard
  - CLARIFICATION:「TLM-2.0準拠」のレベルを定義

この標準は、TLM-2.0クラスに関連する、3つの準拠の概念を定義する。

a) TLM-2.0準拠実装は、本標準に含まれる全てのTLM-2.0のクラスとセマンティックスを提供する実装である。これにはTLM-2.0インタオペラビリティ・レイヤとTLM-2.0ユーティリティが含まれる。TLM-2.0準拠だけが、1666標準全体に対して完全準拠であるというわけではない。しかし、実際には、TLM-2.0クラスで使用するSystemCのサブセットに暗黙的に準拠している(1.3章を参照)。

b) TLM-2.0基本プロトコル準拠モデルは、本標準に記載されているTLM-2.0基本プロトコルの、全てのルールに従うアプリケーションの一部である。そのようなモデルは、一つ以上のSystemCモジュールから成り、これらのモジュールは、プロトコル・トレイツ・クラスtlm_base_protocolを使って特殊化された標準ソケットを備えており、16.2に定義されたあらゆるルールに正確に従う(以下に定義)。

10. Overview of TLM-2.0

- 10.1 Compliance with the TLM-2.0 standard (続き)

c) TLM-2.0カスタム・プロトコル準拠モデルは、ユーザ定義のプロトコル・トレイツ・クラス (tlm_base_protocol以外) で特殊化した標準ソケットを持つ、アプリケーションの一部である。このモデルは、汎用ペイロードを使用するが、必要であれば汎用ペイロードの拡張や、メモリ管理機構の追加を行う。カスタム・プロトコル準拠モデルは、基本プロトコルのどのルールにも従う必要はないが、このようなモデルは基本プロトコルのルールにできるだけ従うことを推奨する。これは、二つのこのようなモデルのインタフェースを行う際の開発工数を最小化するためである。このような推奨は非公式のものであるべきだが、TLM-2.0を使ってモデル化するプロトコルの種類については、事前に制限は設けておらず、これがTLM-2.0標準から利益を得るための秘訣でもある。

上記b) またはc) の場合、標準ソケットという用語は、tlm_initiator_socket、tlm_target_socket型、あるいはこれらの二つのクラスから派生したあらゆるクラスのオブジェクトを意味する。

TLM-2.0クラスライブラリの限られた機能だけを使うモデルは、本標準に準拠しているかもしれないが、TLM-2.0基本プロトコル準拠でも、TLM-2.0カスタム・プロトコル準拠でもない。

11. Introduction to TLM-2.0

- 章のタイトルに"to TLM-2.0"を追加。
- 11.3.6 Switching between loosely-timed and approximately-timed modeling
  - CLARIFICATION:typoを修正
    - approximately timed → approximately-timed
- 11.4 Initiators, targets, sockets, and transaction bridges
  - CLARIFICATION: Hopの説明を追加。

イニシエータとターゲット間のパスは幾つかのホップからなる。ホップとは隣り合った2つのコンポーネントの接続のことである。ホップは一つのターゲットソケットにバインドされる、一つのイニシエータソケットからなる。イニシエータとターゲット間のホップ数はそのパス上のインターコネクト・コンポーネント数より1だけ大きい。汎用ペイロードを使う場合、フォワード・パスとバックワード・パスはそれぞれ、常に反対方向の同じコンポーネントとソケットを通過する。

- 11.8 Header files and version numbers
  - CLARIFICATION:ヘッダの構成に関する説明を追加

アプリケーションは、ヘッダファイルtlm.hをインクルードする必要がある。これは、TLM-2.0インタオペラビリティ・レイヤのための全てのpublicな宣言を含む。TLM-2.0ユーティリティは、ヘッダファイルtlm.hには含まれない。このため、TLM-2.0ユーティリティを使う場合には、明示的に使用したいユーティリティのヘッダファイルをインクルードする必要がある。これらは、tlm_utilsという名称のディレクトリ以下にある。これらのヘッダファイルの正確なファイル名は、本標準の個別の節で定義する。

11. Introduction to TLM-2.0

- 11.8.1 Software version information
  - CLARIFICATION: ファイル名の修正
    The header file include/tlm/tlm_h/tlm_version.h shall contain a set of macros,
    → The header file tlm.h shall include a set of macros,
  - CLARIFICATION: 本節で定義するマクロと定数の値は、9.6.5章で定義しているSystemCの対応するマクロや定数の値とは独立なものでよい。

- 11.8.3 Rules
  d) e) CORRECTION: define名と本文の説明が合っていなかったものを修正
  IS_PRERELEASE → TLM_IS_PRERELEASE

12. TLM-2.0 Core Interfaces

- 12.1.2.7 The tlm_sync_enum return value
  b) CORRECTION: typoを修正
  tlm_quantum_keeper → tlm_quantumkeeper

- 12.2.4 template argument and tlm_generic_payload class
  k) ADDITION: 基本プロトコルの場合、汎用ペイロードのオプション・アトリビュートがTLM_MIN_PAYLOADの場合、イニシエータはコマンドやアドレスを除くと、汎用ペイロードに対して、他のいかなるアトリビュートも追加する必要はない。また、ターゲットやインターコネクト・コンポーネントは、これら全てのアトリビュートを無視できる。特に、レスポンス・ステータス・アトリビュートやDMI許可アトリビュートは無視できる。ターゲットが汎用ペイロードのオプション・アトリビュートをTLM_FULL_PAYLOAD_ACCEPTEDにセットした場合、ターゲットはレスポンス・ステータス・アトリビュートをセットしなければならず、イニシエータは15.8節に示すように、レスポンス・ステータスを確認しなければならない。DMI許可アトリビュートは、トランスポートやデバッグ・トランスポート・インタフェースと共に使われることしか想定していない。

- 12.3.4 Rules (Debug transport interface)
  o) ADDITION: 基本プロトコルの場合、 汎用ペイロード・オプション・アトリビュートが TLM_MIN_PAYLOADの場合、イニシエータはコマンド、アドレス、データ長、データポインタを除くと、汎用ペイロードに対して、他のいかなるアトリビュートも追加する必要はない。また、ターゲットやインターコネクト・コンポーネントは、これら全てのアトリビュートを無視できる。特に、レスポンス・ステータス・アトリビュートは無視できる。
  p) ADDITION: 基本プロトコルの場合イニシエータが汎用ペイロード・オプション・アトリビュートをTLM_FULL_PAYLOADにセットした場合、イニシエータはバイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅アトリビュートをセットし、さらに、15.8節に示す通り、DMI許可アトリビュートとレスポンス・ステータス・アトリビュートをデフォルト値にセットする。

12. TLM-2.0 Core Interfaces

- 12.3.4 Rules (Debug transport interface、続き)
  - q) ADDITION: 基本プロトコルの場合、ターゲットが汎用ペイロードのオプション・アトリビュートをTLM_FULL_PAYLOAD_ACCEPTEDにセットした場合、ターゲットはバイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅の各アトリビュートの値に従って動作しなければならない。また、DMI許可アトリビュート、レスポンス・ステータスアトリビュートは、15.8節に示す通りセットされなければならない。
  - r) イニシエータは、トランザクション・オブジェクトを継続する呼び出しでデバッグ・トランスポート・インタフェース、トランスポート・インターフェース、DMIで再利用することができる。
  - s) transport_dbgは、実際に読み書きされたバイト数を返す。この値はデータ長アトリビュートの値よりも小さい場合がある。ターゲットが実行できなかった場合は0を返す。CLARIFICATION: TLM_IGNORE_COMMANDの場合、ターゲットはtransport_dbgからの戻り値を自由に選択することができる。

13. TLM-2.0 global quantum

- 章のタイトルに"TLM-2.0"が追加されたのみ。

14. Combined TLM-2.0 interfaces and sockets

- 章のタイトルに"TLM-2.0"を追加。
- 14.2.2. class definition
  - ADDITION: get_base_portにconst付きのメソッドを追加
  - CHANGE: bindの型をvirtualを追加

```
class tlm_base_target_socket_b {…
    virtual       sc_core::sc_port_b<FW_IF> & get_base_port () = 0;
    virtual const sc_core::sc_port_b<FW_IF> & get_base_port () const = 0;
    virtual       BW_IF & get_base_interface () = 0;
    virtual const BW_IF & get_base_interface () const = 0;
 …}


class tlm_base_initiator_socket : … {…
    virtual void bind (base_target_socket_type& s);
    virtual void bind (base_type& s);
    virtual void bind (bw_interface_type& ifs);
    virtual       sc_core::sc_port_b<FW_IF> & get_base_port () { return *this; }
    virtual const sc_core::sc_port_b<FW_IF> &get_base_port () const { return *this; }
    virtual       BW_IF & get_base_interface () { return m_export; }
    virtual const BW_IF & get_base_interface () const { return m_export; }
    virtual       sc_core::sc_export<BW_IF> & get_base_export () { return m_export; }
    virtual const sc_core::sc_export<BW_IF> &get_base_export () const { return m_export; }
…}
```


14. Combined TLM-2.0 interfaces and sockets

- 章のタイトルに"TLM-2.0"を追加。
- 14.2.2. class definition (続き)

```
class tlm_base_target_socket: … { …
    virtual void bind (base_initiator_socket_type& s);
    virtual void bind (base_type& s);
    virtual void bind (fw_interface_type& ifs);
    virtual       sc_core::sc_port_b<BW_IF> & get_base_port () { return m_port; }
    virtual const sc_core::sc_port_b<BW_IF> &get_base_port () const { return *this; }
    virtual       FW_IF & get_base_interface () { return *this; }
    virtual const FW_IF & get_base_interface () const { return *this; }
    virtual       sc_core::sc_export<FW_IF> & get_base_export () { return *this; }
    virtual const sc_core::sc_export<FW_IF> &get_base_export () const { return *this; }
…}
```

- 14.2.4. Classes tlm_base_initiator_socket and tlm_base_target_socket
  - CHANGE (説明追加): それぞれの場合において、operator () の実装は、対応するvirtualメソッドbindを呼び出すことにより、有効になる。

15. TLM-2.0 generic payload

- 章のタイトルに"TLM-2.0"を追加。
- 15.2 Extensions and interoperability
  - CLARIFICATION: TLM-2.0準拠のレベルについての説明を追加

a) 汎用ペイロードをそのまま、無視可能な拡張と共に使用する。また、基本プロトコルのルールに従う。このようなモデルをTLM-2.0基本プロトコル準拠であると呼ぶ。

b) tlm_generic_payloadに対するtypedefを含む新しいプロトコル・トレイツ・クラスを定義する。このようなモデルは、TLM-2.0カスタム・プロトコル準拠であると呼ぶ。

c) 新しいプロトコル・トレイツ・クラスと新しいトランザクション・タイプを定義する。このようなモデルは、TLM-2.0クラス・ライブラリからは独立した機能を使う場合があり、TLM-2.0基本プロトコル準拠でもTLM-2.0カスタム・プロトコル準拠でもない。

15. TLM-2.0 generic payload

- 15.4 Class definition
  - ADDITION: DMI, debug用のenum、メソッドを追加

```
enum tlm_gp_option {
    TLM_MIN_PAYLOAD,
    TLM_FULL_PAYLOAD,
    TLM_FULL_PAYLOAD_ACCEPTED
  };
class tlm_generic_payload {…
// Access methods
tlm_gp_option get_gp_option() const;
void set_gp_option( const tlm_gp_option );
… }
```

- 15.5 Generic payload memory management
  - ADDITION: resetの動作として、オプション・アトリビュートも初期化することを追加

y) resetメソッドは自動削除マークが付いている拡張を全て削除し、対応する拡張ポインタにnullを設定する。各拡張は、extentionオブジェクトのfreeメソッドを呼び出すことにより削除される。拡張オブジェクトは、考えられる範囲では、ユーザが明示的に拡張オブジェクトに対するメモリ管理機能を提供したい場合にのみオーバロードされる。resetメソッドは、オプション・アトリビュートの値をTLM_MIN_PAYLOADにセットする。resetメソッドは通常の場合、トランザクションのライフタイムの終わりに拡張を削除するために、tlm_mm_interface クラスのfreeメソッドから呼び出される。

af) ADDITION: アトリビュートの種類の羅列で、optionを追加

15. TLM-2.0 generic payload

- 15.7 Default values and modifiability of attributes
  - CLARIFICATION: 基本プロトコルの汎用ペイロードに含まれるアトリビュートと配列の、デフォルト値と変更可否を表54、55にまとめる。

表54 Table 54— Default values and modifiability of attributes

Attribute	Default value	Modifiable by interconnect?	Modifiable by target?
ADDITION: Option	ADDITION: TLM_MIN_PAYLOAD	ADDITION: No	ADDITION: Yes
Command	TLM_IGNORE_COMMAND	No	No
Address	0	Yes	No
Data pointer	0	No	No
Data length	0	No	No
Byte enable pointer	0	No	No
Byte enable length	0	No	No
Streaming width	0	No	No
DMI allowed	false	Yes	Yes
Response status	TLM_INCOMPLETE_RESPONSE	No	Yes
Extension pointers	0	Yes	Yes

a) トランザクションオブジェクトをインタフェース・メソッド呼び出しを行う前に、CLARIFICATION:使用するコア・インタフェースのルールに従って、汎用ペイロードのアトリビュートに対して値をセットしておくことは、イニシエータの責任である。トランスポート・インタフェースの場合、汎用ペイロードの各アトリビュートは、拡張ポインタ以外を除いて全てセットされなければならない。DMIとデバッグ・トランスポート・インターフェースの場合は独自のルールがあり、それぞれ12.2.4、12.3.4小節に記載されている。オプション・アトリビュートはDMIとデバッグ・トランスポートが汎用ペイロードの最小セット、または、フルセットを利用するかどうかを決定するために使用する。トランザクション・オブジェクトがプールされ、再利用される場合においても、アトリビュートに正しい値が確実にセットされるよう、注意をはらう必要がある。

15. TLM-2.0 generic payload

- 15.8 Option attribute (ADDITION)
  - e) ～n) のサマリ (原文は次頁以降)

インターフェースの種類	オプション・アトリビュートの値	イニシエータのアクション	アトリビュートの書き換え	ターゲットのアクション
ダイレクト・メモリ・インターフェース	TLM_MIN_PAYLOAD		不可	コマンド・アトリビュートとアドレス・アトリビュート以外は全て値を無視
	TLM_FULL_PAYLOAD	バイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅、DMI許可、レスポンス・ステータス・アトリビュートをデフォルト値にセット	可	アトリビュートをTLM_FULL_PAYLOAD_ACCEPTEDにセット → 通常のルールに従って各アトリビュートを解釈 TLM_FULL_PAYLOAD_ACCEPTEDにセットしない → TLM_MIN_PAYLOADの場合と同じ動作
デバッグ・メモリ・インターフェース	TLM_MIN_PAYLOAD		不可	バイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅、DMI許可、レスポンス・ステータス・アトリビュートを無視
	TLM_FULL_PAYLOAD	DMI許可アトリビュートとレスポンス・ステータス・アトリビュートをデフォルト値にセット	可	アトリビュートをTLM_FULL_PAYLOAD_ACCEPTEDにセット → 通常のルールに従って各アトリビュートを解釈 TLM_FULL_PAYLOAD_ACCEPTEDにセットしない → TLM_MIN_PAYLOADの場合と同じ動作 (DMI許可、レスポンス・ステータス・アトリビュートはセット不可)
ブロッキング、または、ノンブロッキング・トランスポート・インターフェース	TLM_MIN_PAYLOAD		不可	通常のルールに従って各アトリビュートを解釈

15. TLM-2.0 generic payload

- 15.8 Option attribute（ADDITION）

a) オプション・アトリビュートは、DMIとデバッグ・トランスポート・インターフェースが汎用ペイロードのアトリビュートの最小セット、または、フルセットのどちらを使用するかを決めるために使われる。最小セットとは、TLM-2.0標準の過去のバージョンに対するバックワード・コンパチビリティをサポートするためにサポートされる。新規に開発するコンポーネントはアトリビュートの最小セット、または、フルセットのどちらを使うかを選択することができる。

b) set_gp_optionは引数で渡された値をオプション・アトリビュートにセットする。get_gp_optionメソッドは、オプション・アトリビュートの現在の値を返す。

c) オプション・アトリビュートのデフォルト値はTLM_MIN_PAYLOADである。

d) 一つの例外を除いて、イニシエータ、インターコネクト、ターゲット・コンポーネントはオプション・アトリビュートを無視する。これは、本標準の出版前に開発された既存のコンポーネント、本標準の出版後に開発される新しいコンポーネントの両方に適用される。唯一の例外は、イニシエータがTLM_FULL_PAYLOAD_ACCEPTED値をオプション・アトリビュートに持ったトランザクションを受け取った場合である。

e) 汎用ペイロード・トランザクションを、ダイレクト・メモリ・インターフェースを介して送る場合、ターゲットがレスポンス・ステータス・アトリビュートをセットすることを要求するイニシエータは、TLM_FULL_PAYLOADをオプション・アトリビュートにセットする。この場合、イニシエータは、バイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅、DMI許可、レスポンス・ステータスの各アトリビュートを、これらのデフォルト値にセットする（表54）。

15. TLM-2.0 generic payload

- 15.8 Option attribute（ADDITION、続き）

f) 汎用ペイロード・トランザクションを、デバッグ・トランスポート・インターフェースを介して送る場合、ターゲットがバイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅の各アトリビュートを使用することを要求するイニシエータ、または、ターゲットがDMI許可アトリビュートまたはレスポンス・ステータス・アトリビュートをセットすることを要求するイニシエータは、オプション・アトリビュートをTLM_FULL_PAYLOADにセットする。この場合、イニシエータはDMI許可アトリビュートとレスポンス・ステータス・アトリビュートをこれらのデフォルト値にセットする（表54）。

g) 汎用ペイロード・トランザクションを、ブロッキング、または、ノンブロッキング・トランスポート・インターフェースを介して送る場合、オプション・アトリビュートはTLM_MIN_PAYLOADにセットされていなければならず、いかなるコンポーネントも書き換えをしてはならない。

h) オプション・アトリビュートの値がTLM_MIN_PAYLOADの場合、オプション・アトリビュートの値はいかなるインターコネクト・コンポーネント、ターゲット・コンポーネントも修正してはならない。

i) ダイレクト・メモリ・インターフェースの場合で、オプション・アトリビュートがTLM_MIN_PAYLOAD であった場合、ターゲットはコマンド・アトリビュートとアドレス・アトリビュート以外は全て値を無視できる。

j) ダイレクト・メモリ・インターフェースの場合で、オプション・アトリビュートの値がTLM_FULL_PAYLOAD であった場合、ターゲットはその値をTLM_FULL_PAYLOAD_ACCEPTEDにセットできる。この場合、イニシエータとターゲットはレスポンス・ステータス・アトリビュートを15.17で示すルールに従って、セットしたり、動作したりしなければならない。

15. TLM-2.0 generic payload

- 15.8 Option attribute（ADDITION、続き）

k) デバッグ・トランスポート・インターフェースの場合で、オプション・アトリビュートの値がTLM_MIN_PAYLOAD であった場合、ターゲットはバイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅、DMI許可、レスポンス・ステータスの各アトリビュートを無視できる。

l) デバッグ・トランスポート・インターフェースの場合で、オプション・アトリビュートの値がTLM_FULL_PAYLOAD であった場合、ターゲットはその値をTLM_FULL_PAYLOAD_ACCEPTEDにセットできる。この場合、イニシエータとターゲットはバイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅、DMI許可、レスポンス・ステータスの各アトリビュートを、それぞれ15.13、15.14、15.15、15.16、15.17で示すルールに従って、セットしたり、動作したりしなければならない。

m) ターゲットがオプション・アトリビュートをTLM_FULL_PAYLOAD_ACCEPTEDにセットしない場合、ターゲットはバイト・イネーブル・ポインタ、バイト・イネーブル長、ストリーミング幅の各アトリビュートの値を無視しなければならず、また、DMI許可、レスポンス・ステータス・アトリビュートをセットしてはならない。

n) イニシエータがオプション・アトリビュートにTLM_FULL_PAYLOADをセットした場合で、ターゲットがその値をTLM_FULL_PAYLOAD_ACCEPTEDに設定しない場合、イニシエータはターゲットが、オプション・アトリビュートがTLM_MIN_PAYLOADの場合と同様の動作をするものと仮定する。この場合、ターゲットが汎用ペイロード・アトリビュートを誤って解釈するかもしれない。例えば、ターゲットはデバッグ・トランスポート・トランザクションに対して、バイト・イネーブル・アトリビュートを無視するかもしれない。

15. TLM-2.0 generic payload

- 15.8 Option attribute（ADDITION、続き）

o) インターコネクト・コンポーネントは、オプション・アトリビュートを書き換えてはならない（通常インターコネクト・コンポーネントとして動作するコンポーネントは、エラー・レスポンスを返すために、ターゲットとして動作する場合があるが、その場合は、オプション・アトリビュートの値をTLM_FULL_PAYLOAD_ACCEPTEDにセットしてよい）。

p) オプション・アトリビュートにTLM_FULL_PAYLOADをセットするイニシエータは、トランザクション・オブジェクトのライフタイムの終わりに、オプション・アトリビュートがTLM_MIN_PAYLOADに戻されていることを確認しなければならない。メモリ・マネージャが存在する場合、tlm_generic_payloadクラスのresetメソッドは、オプション・アトリビュートにTLM_MIN_PAYLOADをセットする。メモリ・マネージャが存在しない場合、イニシエータはオプション・アトリビュートを明示的にリセットする義務がある。

q) オプション・アトリビュートの値は、現在のトランザクション・インスタンスにのみ適用され、他のトランザクションまたは他のインターフェースに関しては、イニシエータ、インターコネクト、ターゲットの動作には一切影響を与えない。例えば、イニシエータが連続するデバッグ・トランスポート・トランザクションに対してTLM_MIN_PAYLOADや TLM_FULL_PAYLOADをセットすることができ、デバッグ・トランスポート・トランザクションでTLM_FULL_PAYLOAD、DMIトランザクションでTLM_FULL_PAYLOADをセットすることができる。同一のターゲットはTLM_FULL_PAYLOAD_ACCEPTEDをあるトランザクションに対してセットすることができるが、次のトランザクションでセットしないこともできる。各コンポーネントは、イニシエータやターゲットの機能を固定して考えるのではなく、個々のトランザクションをそれぞれ検査するべきである。

15. TLM-2.0 generic payload

- 15.11 Data pointer attribute
  - e) CHANGE: リードコマンドまたはライトコマンドにおいて、トランザクション・オブジェクトがnull(0)のデータ・ポインタ・アトリビュートでトランスポート・インタフェースを呼び出すとエラーとなる。
  - f) CHANGE: TLM_IGNORE_COMMANDの場合、データポインタはデータアレイを指すか、またはnullを指し示すことができる。
  - g) CHANGE: データポインタがnullでない場合、データ配列長はデータ長アトリビュート以下でなければならない。

- 15.12 Data length attribute
  - e) CHANGE: リードコマンドまたはライトコマンドにおいて、データ長アトリビュートは0にセットしてはならない。0バイトの転送を行う場合、コマンド・アトリビュートはTLM_IGNORE_COMMAND をセットしなければならない。
  - f) CHANGE: TLM_IGNORE_COMMANDの場合で、かつ、データ・ポインタがnullの場合、データ長アトリビュートの値は未定義である。
  - i) CHANGE: データ長アトリビュートのデフォルト値は0である。これはデータ・ポインタがnullでない限り、不正な値である。したがって、データ・ポインタがnullでない限り、データ長アトリビュートは、トランザクション・オブジェクトがインタフェース・メソッド呼び出しに渡される前に、明示的にセットされなければならない。

16. TLM-2.0 base protocol and phases

- 章のタイトルに"TLM-2.0"が追加されたのみ。

17. TLM-2.0 base protocol and phases

- 17章のタイトルに"TLM-2.0"を追加。
- 17.1.1.2のタイトルが、「ソケット・バインディング表」から、「許可されているソケット・バインディング」に変更 (CLARIFICATION)
- 17.1.2.1 Introduction
  - シンプル・ソケットを使用する場合、すべてのインタフェース・メソッドに対してコールバックを登録してもよいが、必須ではない。特に、シンプル・ターゲット・ソケットの場合、b_transportまたは、nb_transport_fwのどちらか片方のみを登録するだけでよい。この場合、登録されていない方のインタフェース・メソッドが呼び出された場合には、登録されている方のメソッドに自動変換される。この変換手法は単純なものではない。イニシエータとターゲットの双方の基本プロトコルの規則に準拠して変換が行われる。 CLARIFICATION:このため、クラステンプレートsimple_target_socketは、プロトコル・タイプをテンプレート・パラメタとして取得するが、このクラスとベースプロトコルの間には依存関係がある。これは、アプリケーションがブロッキング・トランスポート呼び出しとノンブロッキング・トランスポート呼び出しの変換を行う際にのみ問題になる。特に、クラスsimple_target_socketは、tlm_phase_enumタイプの値を使用する。ベース・プロトコル以外でそのような変換が必要なアプリケーションは、新しい便利ソケットを作成する必要がある。この場合のソケットは、おそらくはsimple_target_socketから派生するだろう。
- 17.1.2.5 Simple target socket b/nb conversion
  - b_transportかnb_transport_fwメソッドのどちらかがsimple_target_socketクラスのソケットを通して呼ばれたにも関わらず、対応するコールバック関数が登録されていない場合には、シンプル・ターゲット・ソケットは、2つのインタフェースを結ぶアダプタのように振舞う。
    CLARIFICATION:この場合にのみ、 simple_target_socketの実装はtlm_base_enumタイプの値と明示的な依存関係を持つ。

17. TLM-2.0 base protocol and phases

- 17.1.3.4 Rules
  d) タグ(int id)は、コールバック関数登録時のインタフェース・メソッドの最後の引数になる。しかし、ソケットのコールバック関数実装の本体には、最初の引数としてこのタグが付けられる。
  CLARIFICATION: idタグ引数の順番は、登録用メソッドとコールバック・メソッドの間で異なる。以下の例を参照。

```
CLARIFICATION: Example:
struct my_target: sc_module
{
tlm_utils::simple_target_socket_tagged<my_target> socket1;
tlm_utils::simple_target_socket_tagged<my_target> socket2;
SC_CTOR (my_target)
: socket1 ("socket1"), socket2 ("socket2")
{
socket1.register_b_transport (this, &my_target::b_transport, 1); // Registered with id = 1
socket2.register_b_transport (this, &my_target::b_transport, 2); // Registered with id = 2
}
void b_transport (int id, Transaction& trans, sc_time& delay); // May be called with id = 1 or id = 2
...
};
```

17. TLM-2.0 base protocol and phases

- 17.1.4.3. class definition
  - Initiator/target ソケットの、bindメソッドの型をvirtualに変更

```
class multi_passthrough_initiator_socket : … { …
    CHANGE: virtual void bind (base_target_socket_type& s) ; …
    CHANGE: virtual void bind (base_type& s) ;
…}
class multi_passthrough_target_socket: … { …
    CHANGE : virtual void bind (base_type& s) ;
…}
```

- 17.1.4.4. Rules
  - f) CHANGE (説明追加) : 各operator () の実装は、対応するvirtualメソッドbindを呼び出すことにより有効になる。
- 17.2.5. Class tlm_quantumkeeper
  - b) CORRECTION: tlm_quantum_keeper → tlm_quantumkeeper (スペルミス修正)

18.CLARIFICATION: TLM-1 Message passing interface and analysis ports

TLM1.0のmessage passing interfaceは、blockingとnon-blockingの
put,get,peek,transport,wirte,analysis interface,tlm_fifo chanel、analysis port , analysis fifoから成る。
TLM1.0 transport interfaceは、TLM2.0 transport interafceと異なる。

18.1 CLARIFICATION:
18.1 Put,get,peek,and transport interfaces

18.1.1 Description

TLM-1 message passing interfaceは、基本的に次の点でTLM-2 core interface と異なる。
TLM-2 core interfaceは、参照によってトランザクションオブジェクトを伝達し、トランザクションオブジェクトのライフタイムは、複数のインタフェイスメソッドのコール全体にわたる。
それに対して、TLM-1 interfaceは、message-passing(メッセージ伝達)の動作を実装する。
TLM-1のメッセージ伝達の動作とは、伝達する側と伝達される側の間でメモリをシェアをするべきでないという意味である、そして、TLM-1 interface methodコールにより実装されたメッセージ伝達の抽象度は、一つのSystemCモジュールの内部ステート変化を他のモジュールから隠すべきであるという意味である。
TLM-1 blocking interfaceは、同期メッセージ伝達を実現する。そして、TLM-1 non-blocking interfaceは、非同期メッセージ伝達を実現する。

TLM-1メッセージ伝達(トランザクション伝達と同じ)は、一方向である。
TLM-1の双方向transport interfaceは、逆方向に異なるメッセージを伝える2つの単方向message chanelの構成であるとみなすことが出来る。
ある抽象度において、getかputを使って伝達されたトランザクションであろうとなかろうと、トランザクションの受信者(recipient)は送信された値と正確に同じ値を受け取るべきであるということを意味する。
メッセージ伝達システムは、通常、厳格な値渡し動作を用いることで、この要求を実装するだろう。
しかしながら、TLM-1は、いわゆる効果的な値渡し動作を使う。トランザクションは、いくつかの場合において参照による伝達を行うが、送り手によって一回割り当てられたトランザクションオブジェクトは、伝達する側も伝達される側も変更を許されない。

JEITA

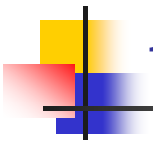
18.1.1 Description

TLM-1 は、メッセージ伝達 動作の整合性を促進するために、より一層の制約を与える。
トランザクションオブジェクトのデータ型は、ディープコピー動作をサポートするべきである。
もし、コピーがC++の初期化(コピーコンストラクタ)か値の代入により行われたのであれば、そのコピーへのその後の変更は、オリジナルのトランザクションオブジェクトを変更すべきでない。
(コピーへの変更がオリジナルに影響を与えるべきではない)
言い換えれば、トランザクションオブジェクトは、back-door通信の実装に使うことが出来るそれ自身の外部のメモリを共有するためのポインタ型や参照型を含むことが出来ない。
それどころか、もし、トランザクションオブジェクトがこのようなポインタや参照を含むなら、共有メモリを通した通信は、この標準のスコープの外側になる。(TLM-1標準からはずれる)

以下では、"sender"という単語は、putメソッドの場合の送信側(caller)、get、peekメソッドの場合の受信側(callee)をさす。
そして、"recipient"という単語は、putの場合の受信側(callee)、get,peekの場合の送信側(caller)を指す。
transportメソッドの場合、受信側(callee)は、recipientのリクエストとsenderのレスポンスであるのに対して、送信側(caller)は、senderのリクエストであり、recipientのレスポンスである。

トランザクションオブジェクトは、put,nb_put,nb_get,nb_peekやtransportメソッドの引数として、またはget,peekやtransportメソッドの戻り値として伝達される。

JEITA

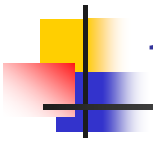


18.1.3 Blocking versus non-blocking interfaces

18.1.3 blocking versus non-locking interface

a) put,get,peek,transportメソッドは、blocking interfaceメソッドである。
b) nb_put,nb_can_put,ok_to_put,nb_get,nb_can_get,ok_to_get,nb_peek,nb_can_peek,ok_to_peekメソッドは、non_blocking interfacesメソッドである。
c) Blocking interfaceメソッドは、直接的、または間接的にwaitをコールする。
d) Blocking interfaceメソッドは、メソッドプロセスからコールされない。
e) Non-blocking interfaceメソッドは、直接的、または間接的にwaitをコールしない。
f) Blocking interfaceメソッドは、スレッドプロセスかメソッドプロセスからコールされる。

JEITA



18.1.4 Blocking put, get, peek, and transport

a) 連続した putメソッドのコールや連続した getメソッドのコール（同じport/exportを通した）は、いつも同じトランザクションオブジェクトが伝達されていようとも、異なるトランザクションのインスタンスをあらわす。言い換えれば、それぞれのputのコールでは、callerは、順に次のトランザクションを伝達する。そして、getからのそれぞれのreturnでは、calleeは、順に次のトランザクションを返す。
b) putメソッドは、recipientが、トランザクションオブジェクトを受け取るまで、retrunしない。言い換えれば、トランザクションオブジェクトは、recipientによって実行された、または、コピーされた、または、下流に伝達されたということまで(returnされない)。言い換えれば、putメソッドからのreturnによって、トランザクションオブジェクトの処理が、その段階で正しいかどうかは関係なく、callerは、calleeがcompleteしたとみなすことができる。トランザクションは、put メソッドのbodyの中で実行されるだろう、または、putメソッドは、その後の処理のためにトランザクションオブジェクトのコピーを行うであろう。いずれにしても、callerは、すぐに次のトランザクションを送ることが出来る。
c) getメソッドは、次のトランザクションオブジェクトが、retrunされる準備ができるまでretrunされない。言い換えれば、getメソッドからのreturnによって、処理する準備が出来た有効なトランザクションオブジェクトをreturnしているとcallerはみなすことが出来る。そして、callerは、すぐに次のトランザクションをgetすることができる。
d) peekメソッドは、次のトランザクションオブジェクトがreturnされる準備ができるまでretrunされない。しかしながら、getメソッドとは異なり、peekメソッドはcalleeからのトランザクションを削除できない。いいかえれば、連続したpeekのコール（同一のport/exportを通した、getのコールをはさまない）は、おなじトランザクションインスタンスを意味する同じトランザクションをretrunする。同様に1つのpeekのコールにつづいて1つのgetのコールが行われることは、同じトランザクションインスタンスを意味する同じトランザクションオブジェクトをそれぞれretrunすることである。
e) transportメソッドは、callerからcalleeへ送られたrequestオブジェクトはとcalleeからcallerへ送られたresponseオブジェクトのような、2つの単方向のトランザクションを結びつける。transportメソッドの実装は、request objectを伝達するputのcallと、それに続く response objectに対応したretrunであるgetのcallと等価である。calleeによりreturnされたresponse objectは、request objectを与えられたcalleeのresponseとして表現できる。言い換えれば、往復のトランザクションの両方は、transportメソッドの1回のコールによって実行される。

JEITA

18.1.5 Non-blocking interface method

a) 互いに依存しているnon-blocking interfaceは、calleeが、acceptの準備が出来た（nb_putの場合）かどうか、または、次のトランザクションをすぐにreturnする準備ができた（nb_get,nb_peek）かどうかを決めることが出来る。つまり、現在のnon-blockingメソッドコールの実行の一部分として決めることが出来る。
もしcalleeがすぐに返答できるなら、non-blockingメソッド（nb_put,nb_can_put,nb_get,nb_can_get,nb_peek,nb_can_peek）は、真を返す。そうでなければ、non-blockingメソッドは、偽を返す。そして、次のトランザクションのacceptやreturnをしない。

b) もし、nb_put,nb_get,nb_peek interfaceメソッドが真を返したなら、それらは、blockingメソッドput,get,peekと同じ動作をするだろう。waitの呼び出しなしですぐにretrunなければならない場合を除いて。

c) もしinterfaceメソッド（nb_put,nb_can_put,nb_get,nb_can_get,nb_peek,nb_can_peek）が偽を返したなら、トランザクションのreturnやacceptなしですぐに、（偽が）returnされるだろう。言い換えれば、特定のport/exportを通して行われるトランザクションの送信や受信に関連したcalleeのステートを変更しない。

d) もし、決まった時間に決まったport,export を通して互いにコールしたなら、nb_put,nb_can_putのinterfaceメソッドは、それぞれ同じ値（true or false）を返す。戻り値は、引数として伝達されたトランザクションオブジェクトの値に依存しない。

e) 同様に、もし、決まった時間に決まったport,export を通して互いにコールしたなら、nb_get,nb_can_get,nb_peek,nb_can_peek interfaceメソッドは、同じ値（true or false）を返す。言い換えれば、calleeは、nb_getのために真を返すけれども、nb_can_getのためにfalseを返すことは、許されていない。逆もまた同様である。そして、nb_can_getのためにtrueを返すけれども、nb_can_peekのためにfalseを返すことも同様である。

JEITA

18.1.5 Non-blocking interface method

f) ok_to_put, ok_to_get,ok_to_peekのinterfaceメソッドは、それぞれ、celleeが、次のtransactionのreturnやacceptの準備ができ次第、celleeによって通知されるsc_eventをreturnする。
この意図は、このイベントの通知が、同じport,exportを通した同じnon-blocking interfaceメソッドによって呼び出されることによって、次のtransactionのput,get,peekの準備するためのcallerのキューとして動作する。しかしながら、callerは、non-blocking interfaceメソッドが、これらのイベントの通知に従ってすぐにtrueを返す必要があるだろうとみなすことができない。

g) 同じblockingとnon-blocking interface メソッドは、互いの代わりに同一のport,exportを通してコールされたときに、一致した振る舞いをする義務は無い。同じ動作をすることが望ましいけれども。
たとえば、putがすぐにreturnする場合に、nb_put,nb_can_putのコールは、通常trueを返すだろう。しかし、そのように動作する義務は無い。同様に、putがwaitをコールする場合、nb_put,nb_can_putは通常falseを返す。しかし、そのように動作する義務は無い。

JEITA

18.1.6 Argumet passing and transaction lifetime

a) tlm_tag<t>*の引数型は、トランザクションタイプのみが異なるTLM-1 interfaceを持つ複数のインスタンスが存在する場合は、template instanceを区別するためにcallerによって使われるかもしれない。
callerは、メソッドのコールの曖昧さをなくすためのC++言語のルールによって、この引数が推定されるため、この引数の為に値を供給する義務はない。
インスタンスメソッドのbodyはこの引数を使うべきではない。

b) const参照型の引数として（const T& 型、Tは、クラステンプレートパラメータ）トランザクションの伝達を行う、put,nb_putやtransportなどのinterfaceメソッドの場合、callerは、実引数（put,nb_putの唯一の引数、transaportの第一引数）として伝達されたtransactionオブジェクトをメソッドコールの実行前に送信されるtransactionインスタンスであらわされる値で、初期化、または代入しなければならない。

c) get,peek,nb_get,nb_peek,transportのinterfaceメソッドの場合は、constではない参照型の引数（T&型、Tはクラステンプレートパラメータ）を通してトランザクションオブジェクトがreturnされる。calleeは、（get,peek,transportのためには必ず）または（nb_get,nb_peekのためにはたぶん）、transactionイスタンスをあらわしている仮引数に値を代入する。

d) それぞれのケースは、実引数または仮引数の初期化や代入の次におこることは、interfaceメソッドのコールからretrunまでに、callerやcalleeがそれぞれtransactionオブジェクトの値を（直接的、または間接的に）変更することだろう。
いくつかのケースでは、メソッドのコールがブロックされることがあることをふまえれば、interfaceメソッド自身がサスペンドしている間に実行されるかもしれない、実引数や仮引数へのアクセスを持つ並列なSystemCプロセス（callerとcalleeを含む）が存在するかもしれない。

e) voidでないreturnの型をもつ、get,peek,transportのinterfaceメソッドの場合、transactionオブジェクトは、戻り値である。そのため、トランザクションオブジェクトの変更の問題は、メソッドコール間では起こらない。

JEITA

18.1.6 Argumet passing and transaction lifetime

f) TLM-1 interfaceメソッドを伝達するtransactionオブジェクトのlifetimeは、C++言語のルールによって決定される。transactionオブジェクトは、interfaceメソッドコールの参照引数として伝達されるかinterfaceメソッドコールからの戻り値として伝達されるかのどちらかであることを思い出して欲しい。

g) TLM-1 interfaceメソッドに引数として伝達されたトランザクションオブジェクトは、実引数／仮引数が、ある値に初期化、代入されたときから、その後のinterfaceメソッドコールからのreturnまでの間、有効なtransactionインスタンスをあらわす。
たとえば、transportの場合、requestオブジェクトは、transportがコールされるまえの、初期化、または代入から、transportからのreturnされるまでが有効である。そして、responseオブジェクトは、transportの実装の中で、仮引数にある値が代入されたときからtransportのreturnまでの間が有効である。（標準では、この点を超えてトランザクションオブジェクトの有効性に関していかなる定義もしない、言い換えれば、トランザクションオブジェクトは、interfaceメソッドコールからretrunされるまでが有効である。しかし、この後のtransactionオブジェクトの運命を決めるのはアプリケーション次第である）

JEITA

18.1.7 Constrains on the transaction data type

a) この下記提案の意図は、senderとrecipientの間のメッセージ伝達動作の統合を保障するためである。そして、メモリのシェアーを通した通信の可能性も含む。

b) もし、transactionオブジェクトがポインタか参照を含むのであれば、トランザクションのrecipientは、これらのポインタや参照を通して、アクセス可能なストレージの内容を変更しない。

c) transactionオブジェクトのデータ型は、deep copy動作をしなければならない。もしrecipientがtransactionを意味する実引数や仮引数のコピー（C++初期化や代入を使って）を行ったとしたら、その後は変更はコピーに対して行い、オリジナルは変更すべきでない。

d) 上記、制約は、様々な方法で満たすことが出来る。例えば、transactionオブジェクトのデータ型は、copy-on-write動作によって実装できた。つまり、オリジナルのtransactionとそのコピーの両方は、1つの共有メモリのエリアへの内部ポインターを持つ。しかし、それぞれのオブジェクトへのいかなる代入も別々にコピーされて実行される。

e) もし、transactionオブジェクトのデータ型が、上記制約が支持されないとき、例えば、もし、ポインタのshadow copyを作るとき、通信プロトコルの保障や通信プロトコルに従ったメッセージ伝達動作の保障は、アプリケーションの責任である。

copy-on-write:
コピーを要求されてもとりあえず原本をそのまま参照させるが、原本またはコピーに本当に書き換えが発生した時点で、コピーを実行するというプログラミング最適化手法。

JEITA

18.2 TLM-1 fifo interface

18.2.1 Description

Class tlm_fifo_debug_if は、tlm_fifoへのdebug access に適したinterfaceである。
Class tlm_fifo_put_if と tlm_fifo_get_ifは、tlm_fifo_debug_ifとそれぞれtlm_put_ifとtlm_get_peek_ifの組み合わせたinterfaceである。
これら3つのinterfaceのそれぞれは、あらかじめ定義されたチャネル tlm_fifo によって実装されている。

JEITA

18.2 TLM-1 fifo interface

18.2.3 Member functions

次のメンバー関数は、全て純粋仮想関数である。
この記述は、このinterfaceの実装されるチャネルの中でオーバライドされることを期待した記述である。
この関数の正確な動作は、チャネル依存である。

メンバー関数 used は、get または nb_getを用いて、fifoから読み出し可能なitemの数をreturnする。

メンバー関数 size は、現在のfifoのサイズをreturnする。
すなわち、fifoがいつでも保持可能なitemの最大の数である。

メンバー関数 debug は、fifoの現在のステートに関連した特徴的な情報を標準出力に表示する。

メンバー関数 nb_peek は、fifoの中の与えられた位置にあるitemの参照をreturnする。
位置0は、getやnb_getで次に読み出されるitemである。そして、位置“ used()-1”は、putやnb_putによって挿入された最も新しいitemである。もし、与えられた位置にitemが無い場合や与えられた位置がマイナスであった場合は、nb_peekはfalseの値をreturnする。または、そうでなければ、真を返す。

メンバー関数 nb_poke は、引数として伝達された別のitemで、fifoの与えられた位置のitemを上書きする。
位置0は、getやnb_getで次に読み出されるitemである。そして、位置“ used()-1”は、putやnb_putによって挿入された最も新しいitemである。もし、与えられた位置にitemが無い場合や与えられた位置がマイナスであった場合は、nb_peekはfalseの値をreturnする。または、そうでなければ、真を返す。

JEITA

18.3 tlm_fifo

- 18.3.1 Description (ADDITION)
  - tlm_fifoクラスは，事前定義されたプリミティブチャネルであり，先入れ先出しのfifoの動作をモデル化する．TLM fifoのそれぞれアイテムを蓄積するための複数のスロットを備えている．スロットの数はオブジェクトがコンストラクトしたときに設定されるが，TLM fifoはコンストラクトされた後でもサイズ変更可能である．tlm_fifoとsc_fifoの主な違いは，第一にtlm_fifoはSystemC fifoのインターフェースとは違い，TLM-1のメッセージ・パッシング・インターフェースを利用している．第二にtlm_fifoは，固定サイズを持つよりも，サイズ変更またはサイズ固定しなくてもよい．以下の説明では，fifoという用語はtlm_fifoクラスのオブジェクトのこととする．
  - 各fifoは，tlm_put_if，tlm_get_if，tlm_peak_if，tlm_fifo_debug_ifのセマンティクスを実装し，上記インターフェースを通してトランザクション・オブジェクトを一つのfifoバッファに蓄える．putまたはnb_putの呼び出しで挿入したトランザクションをgetまたはnb_getを使って取り出す．
  - tlm_fifoはデルタサイクル・セマンティクスを実装する
- 18.3.3 Template parameter T (ADDITION)
  - tlm_fifoのテンプレートパラメータ Tの型は，C++の基本型かポインターであることと，下記ルールに従うこと
    - a) const T& operator= (const T&)
    - b) T型のコンストラクがあれば，デフォルトコンストラクタは定義しておく

© Copyright 2010 JEITA, All rights reserved
JEITA
206

18.3 tlm_fifo (cont.)

- 18.3.4 Constructors and destructor (ADDITION)
  - コンストラクタとデストラクタの定義
    - explicit tlm_fifo (int size_ = 1);
    - explicit tlm_fifo (const char* name_, int size_ = 1);
    - virtual ~tlm_fifo ();
- 18.3.5 Member functions (ADDITION)
  - tlm_put_if, tlm_get_if,tlm_peek_ifのインターフェースのメンバー関数は，18.3.6で定義されたデルタサイクル・セマンティクスを加えて，18.1の定義に従って実装する．
  - 以下メンバー関数（各メンバー関数の説明は省略）
    - void nb_expand (unsigned int n = 1);
    - void nb_unbound (unsigned int n = 16);
    - bool nb_reduce (unsigned int n = 1);
    - bool nb_bound (unsigned int n);
    - bool nb_peek (T & , int n) const;
    - bool nb_poke (const T & , int n = 0);
    - int used () const;
    - int size () const;
    - void debug () const;
    - const char* kind () const;

© Copyright 2010 JEITA, All rights reserved

18.3 tlm_fifo (cont.)

- 18.3.6 Delta cycle semantics (ADDITION)
  - putまたはnb_putをコールすることでfifoに挿入したトランザクションは，次のデルタサイクル後に，get, nb_get, peek,またはnb_peekでのみ利用可能．
  - getやnb_getで空いたスロットは，次のデルタサイクルでputまたはnb_put利用できる．
  - 例として，評価フェーズに空のfifoに一連のputを呼び出すとfifoがフルになり，たとえnb_getが，falseを返したとしても，putはブロックされる．getも同様．
  - メンバー関数である，tlm_fifo_debug_if クラスのnb_peekとnb_pokeは，評価フェーズにputまたはnb_putによってfifoに挿入されたトランザクションにアクセスできない．同様に評価フェーズにgetまたはnb_getによってfifoから既に取り出されたトランザクションもアクセスできない．
  - example追加(コード例省略)
- 18.3.7 Member functions (ADDITION)
  - CLARIFICATION: sc_reportクラスがstd::exceptionクラスから派生していることを明記。

© Copyright 2010 JEITA, All rights reserved

18.3 tlm_fifo（cont.）

- 18.4 Analysis interface and analysis ports
  - CLARIFICATION: Class **tlm_analysis_port** is derived from class **sc_object**, not from class **sc_port**, so an analysis port is not technically a port. Analysis ports can be instantiated, deleted, bound, and unbound dynamically during simulation.
- 18.4.1 クラス定義
  - virtualを追加
    - CHANGE: virtual void **bind** (tlm_analysis_if<T> &) ;
    - CHANGE: virtual bool **unbind** (tlm_analysis_if<T> &) ;
- 18.4.2 ルール
  - d) CHANGE: operator()はbindと等価という表現から，operator()の実装は，仮想メソッドbindを呼び出すことで効果を発揮する とう表現に変更
  - g) CLARIFICATION: bindとunbindメソッドは，エラボレーション中またはシミュレーション中に呼び出すことができる．
  - *Example*
    - CORRECTION: subscriber2 = new Subscriber("subscriber2");
    - ※変更なしのように見えます．どこが修正されたのでしょうか．．．

JEITA

Annex B

- **B.3 application:** CLARIFICATION: A C++ program, written by an end user, that uses the SystemC or TLM-2.0 class libraries, that is, uses classes, calls functions, uses macros, and so forth. An application may use as few or as many features of C++ as is seen fit and as few or as many features of SystemC as is seen fit. (See 4.1.2.)
- **B.12** CLARIFICATION: **base-protocol-compliant:** Obeying all the rules of the TLM-2.0 base protocol. (See 10.1.)
- **B.13 bidirectional interface:** A TLM-1 transaction level interface in which a pair of transaction objects, the request and the response, are passed in opposite directions, each being passed according to the rules of the unidirectional interface. CLARIFICATION: For each transaction object, the transaction attributes are strictly read-only in the period between the first timing point and the end of the transaction lifetime. CLARIFICATION: (See 18.1.1.)
- **B.24 callee:** CLARIFICATION: In a function call, the function that is called by the caller, or the module in which that function is defined. The referent of the term may be a function or a module. This term is used in preference to target to refer to the function body as opposed to the target of a transaction.
- **B.40** CLARIFICATION: **custom-protocol-compliant:** Using the TLM-2.0 standard sockets (or classes derived from these) specialized with a traits class other than **tlm_base_protocol_types** and using the TLM-2.0 generic payload. (See 10.1.)
- **B.58** CLARIFICATION: **event expression:** A list of events, separated by either operator& or operator|, and passed as an argument to either the **wait** or the **next_trigger** method. (See 6.9.)
- **B.60** ADDITION: **event list:** An object of type **sc_event_and_list** or **sc_event_or_list** which may be passed as an argument to member function **wait** or **next_trigger**. (See 6.8.)

JEITA

Annex B (cont.)

- **B.68 generic payload:** CLARIFICATION: A specific set of transaction attributes and their semantics together defining a transaction payload which may be used to achieve a degree of interoperability between loosely-timed and approximately-timed models for components communicating over a memory-mapped bus. The same transaction class is used for all modeling styles. (See Clause 15.)
- **B.73** CLARIFICATION: **hop:** One initiator socket bound to one target socket. The path from an initiator to a target may consist of multiple hops, each hop connecting two adjacent components. The number of hops between an initiator and a target is always one greater than the number of interconnect components along that path. For example, if an initiator is connected directly to a target with no intervening interconnect components, the number of hops is one. (See 11.4.)
- **B.77 implementation:** CLARIFICATION: A specific concrete implementation of the full SystemC and TLM-2.0 class libraries, only the public shell of which need be exposed to the application (for example, parts may be precompiled and distributed as object code by a tool vendor). (See **kernel** and 4.1.2.)
- **B.101 loosely timed:** CLARIFICATION: A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely-timed model may include timer models and a notional arbitration interval or execution slot length. Some users adopt the practice of inserting random delays into loosely-timed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style. (See 11.3.2.)
- **B.103** CLARIFICATION: **mandatory extension:** A generic payload extension that is required to be present on every transaction that is sent through a socket of a given user-defined protocol type. (See 15.20.1.2.)

JEITA

Annex B (cont.)

- **B.113 mutex:** An instance of class **sc_mutex**, which is a CHANGE: predefined channel that models a mutual exclusion communication mechanism. (See 7.27.1.)
- **B.118** CLARIFICATION: **non-ignorable extension:** A generic payload extension that, if present, every component receiving the transaction is obliged to act upon. (See 15.20.1.2.)
- **B.134** CLARIFICATION: **phase transition:** A transition from one phase to another, where the phase is represented by the value of the phase argument to the non-blocking transport method. Each call to *nb_transport*, and each return from *nb_transport* with a return value of TLM_UPDATED, marks a phase transition. The base protocol does not permit calls to *nb_transport* where the value of the phase argument is unchanged from the previous state. (See 16.2.4.)
- **B.140** CLARIFICATION: **programmers view (PV) : The use case of the software programmer who requires a functionally accurate, loosely-timed model of the hardware platform for booting an operating system and running application software.**
- **B.148 resume:** To cause a thread or clocked thread process to continue execution starting with the executable statement immediately following the **wait** method at which it was suspended, dependent upon the sensitivity of the process. (See 6.2.11.) ADDITION: Also, a member function of class **sc_process_handle** that cancels the effect of a previous call to **suspend**. (See 6.6.6.1.)
- **B.169** ADDITION: **suspend**: To cause a process to cease execution by having the associated function call **wait**. Also, a member function of class **sc_process_handle** that causes a process to remain suspended and whose effect can be cancelled by calling **resume**. Also, to cause a process to cease execution by calling member function **suspend** of a process handle associated with the process itself. (See 6.6.6.1.)
- **B.172** ADDITION: **synchronous reset state**: The state entered by a process instance when its reset signal becomes active or when **sync_reset_on** is called. When in the synchronous reset state, a process instance is reset each time it is resumed. (See 6.6.6.2.)

JEITA

Annex B (cont.)

- **B.177 terminated:** The state of a thread or clocked thread process when the associated function executes to completion or executes a return statement and thus control returns to the kernel CHANGE:or after the process has been killed. Calling function **wait** does not terminate a thread process. (See **thread process, clocked thread process, method process**, 6.2.11, and 6.6.5.)
- **B.185 TLM-2.0:** The second major version of the OSCI Transaction Level Modeling standard. CHANGE: TLM-2.0 was first released in 2008 and the OSCI TLM-2.0 LRM was released in 2009. (See Clause 10.)
- **B.186** ADDITION: **TLM-2.0-compliant implementation**: An implementation that provides all of the TLM-2.0 classes described in this standard with the semantics described in this standard, including both the TLM-2.0 interoperability layer and the TLM-2.0 utilities. (See 10.1.)
- **B.199 unidirectional interface:** CLARIFICATION: A TLM-1 transaction level interface in which the attributes of the transaction object are strictly read-only in the period between the first timing point and the end of the transaction lifetime. Effectively, the information represented by the transaction object is strictly passed in one direction either from caller to callee or from callee to caller. In the case of **void put(const T&t)**, the first timing point is marked by the function call. In the case of **void get(T& t)**, the first timing point is marked by the return from the function. In the case of **T get()**, strictly speaking there are two separate transaction objects, and the return from the function marks the degenerate end-of-life of the first object and the first timing point of the second. (See 18.1.1.)
- **B.207 valid:** CLARIFICATION: The state of a process handle, or of an object passed to or returned from a function by pointer or by reference, during any period in which the handle or object is not deleted and its value or behavior remains accessible to the application. A process handle is *valid* when it is associated with a process instance. (See 4.3.3 and 6.6.1.)

JEITA

4.3.4 IEEE P1666 WG へのフィードバック

IEEE P1666 ワーキンググループへ提出したフィードバックと LRM への反映および WG からの回答を報告する。

セクション	フィードバック内容 LRM への反映/回答
5.5.3	The effect if “sc_set_stop_mode” is called more than one time from “before_end_of_elaboration”, “end_of_elaboration”, and “start_of_simulation” should be described. Also, we'd like to confirm the intent of this change from the current LRM."
	LRM 修正 If sc_set_stop_mode is called more than once, the most recent call shall take precedence.
6.2.13	It is difficult to understand the second sentence. Does it means that the order of reset is implementation defined?
	WG 回答 This sentence just spells out the consequences of the async reset, namely - async_reset_signal specifies (directly or indirectly) an object of type sc_signal - The reset occurs when that signal changes to its active value - Signals actually change value in the update phase - The reset is not executed in the update phase, but in the following evaluation phase - The actual call to reset() gets scheduled along with anything else due to run in that same evaluation phase, and thus is run in non-deterministic order. Asynchronous resets do not take priority over other processes due to run in the same delta.
6.5.2	The purpose of the namespace ""sc_unnamed"" should be described. We imagine that this class can be used to set vendor specific classes, and so on. If our understanding is correct, the intent of adding vendor specific classes, etc. in the LRM should be described.
	LRM 修正 6.1.2 と 6.5.6 の 10 パラグラフ目に説明追加。
6.5.6	We don't understand the content of this paragraph? What is “_1, _2, _3”?

	LRM 修正 The only purpose of namespace sc_unnamed is to allow the implementation to provide a set of argument placeholders for use with sc_bind. _1, _2, _3 ... shall be provided by the implementation to give access to the placeholders of the same names from the Boost libraries. These placeholders can be passed as arguments to sc_bind in order to defer the binding of function arguments until the call to the function object returned from sc_bind. Again, see the Boost documentation for details.
6.6.6.1	The following cases of calling “resume” do not seem to be descrived clear. (1) If “resume” are called multi processes at the same simulation time, what happens? (2) If “suspend” and “resume” are successively called to the process during wait, when does it become runnable? For example, wait(100) -> calls “suspend” after 10 ns -> calls “resume” after 50 ns, then does process become runnable after 40 ns?
	LRM 修正 プロセスコントロールの説明が大きく修正。例題も追加。
6.6.6.1	The explanation of timing of resume seems to be inconsistent with the explanation in “suspend”. In “resume”, “the process instance shall be made runnable in the current evaluation phase”. In “suspend”, “the process instance shall become runnable in the next evaluation phase in which resume is called”.
	LRM 修正 プロセスコントロールの説明が大きく修正。例題も追加。
6.6.6.1	The difference between suspend/resume and disable/enable is difficult to understand from the explanation in the LRM. Use cases can be helpful to understand these functions and purposes.
	LRM 修正 プロセスコントロールの説明が大きく修正。例題も追加。
6.6.6.2	Use cases of sync_reset_on/sync_reset_off are helpful to understand these functions and purposes.
	LRM 修正 プロセスコントロールの説明が大きく修正。例題も追加。
6.6.6.4	The “enable” row of Table 2 shows “The current evaluation phase”. Is this correct? Should it be “The next evaluation pahse”?

	WG 回答 It is correct, but it has to be read in the right context. The earliest time at which a call to enable() can affect the target process is indeed in the current evaluation phase. For example, a subsequent immediate notification within the current evaluation phase could make the target process runnable in the current evaluation phase. On the other hand, the earliest time at which a call to disable() can affect the target process is the next evaluation phase (the phase following the current evaluation phase)."
6.6.6.4	The reason of the priorities among kill, reset, and so on, should be described.
	WG 回答 I am not sure there is a compelling rationale. We just needed to make a choice."
6.6.7	In the CLARIFICATION sentences, LRM says, “if the most recently created process instance was not within the current module, ..., an implementation may return either a handle to the most recently created process instance or an invalid handle”. In this case, returning only “an invalid handle” seems to be sufficient. If not, the reason or explanation should be described.
	WG 回答 This is a corner case that we missed from IEEE 1666-2005. It is unimportant to the user (because we are talking about calling sc_get_current_process_handle() from a context where there is no process, i.e. a callback), but we thought it important not to overconstrain the implementation by being too specific. All this says is that the user can rely on sc_get_current_process_handle() not to crash.
6.10.4	LRM says, “when a hierarchically named event is constructed, if a empty string is passed as a constructor argument, the string name shall be set to “event””. It seems to be inconsistent with that event should have hierarchical name, or we may misread.
	WG 回答 See the 2nd para. When constructed during elaboration, every event shall be hierarchically named, even if constructed using sc_event() or sc_event(“”).

9.5	More examples of sc_vector can be helpful to understand its convenience instead of using vector of stl. Also, examples of sc_assemble_vector are helpful.
	例題の追加なし。
10.1	Since three compliances belong to different category; a) is about simulator implementation, b) and c) are about models, a) and b)/c) are explained in the different paragraph.
	LRM 修正 This standard defines three notions of compliance related to the TLM-2.0 classes, the first concerning compliance of the implementation and the latter two concerning compliance of the application.
全体	To enable dynamic process, we must define the macro “SC_INCLUDE_DYNAMIC_PROCESSES”. But the definition and the explanation are not given in the LRM. They should be in the LRM.
	実装依存のものなので、LRM から削除。

4.3.5 SystemC Japan 2010 アンケート報告

© Copyright 2011 JEITA, All rights reserved JEITA

SystemC Japan 2010の概要

- 主催：
  Open SystemC Initiative(OSCI)
- メインスポンサー
  日本ケイデンス・デザイン・システム社
  メンター・グラフィックス・ジャパン㈱
  フォルテ・デザイン・システムズ㈱
  日本シノプシス合同会社
- 協賛：
  アーム㈱
  カリプト・デザイン・システムズ㈱
  コ・フルエントデザイン社
  日本イヴ㈱
  ㈱エッチ・ディー・ラボ
  日本電気株式会社
  ㈱プライムゲート
- 日時：2010年7月2日(金)　12:30～19:20
- 会場：新横浜国際ホテル(定員300名)
- 参加者数：293名
- JEITAアンケート回答数：259名

© Copyright 2011 JEITA, All rights reserved JEITA

SystemC Japan 2010の概要

- 講演内容：
  - 「OSCI and IEEE P1666 Update」
    - OSCI Treasurer, P1666 Chair & Cadence Group Director Stan Krolikoski
  - 「日本におけるSystemCの標準化活動」
    - JEITA SystemC Working Group, 富士通セミコンダクター㈱ 立岡 真人
  - 「動作合成とバーチャル・プラットフォームの統合に向けて」
    - ㈱リコー 塚本 泰隆
  - 「シノプシスが提供する仮想化技術を用いたトータル・ソリューションの紹介」
    - 日本シノプシス合同会社 内田 憲法
  - 「ルネサスエレクトロニクスにおける高位設計適用事例」
    - ルネサスエレクトロニクス㈱ 野中 義弘
  - 「ケイデンスのTLM設計・検証ソリューション」
    - 日本ケイデンス・デザイン・システムズ社 後藤 謙治
  - 「ASIC適用に向けた高位設計技術への取り組み」
    - 富士通アドバンステクノロジ㈱ 中山 典保
  - 「CoFluent Studioによるハードウエア構成検討」
    - 日本システムウエア㈱ 斎藤 勇吉
  - 「ESL設計メソドロジによる高抽象度最適HW設計と検証手法」
    - メンター・グラフィックス・ジャパン㈱ 牧野 潔
  - 「経済省委託研究「画像・動画処理用C言語のLSI化の支援システム開発」事例」
    - ㈱プライムゲート 梅田 芳直
  - 「TLMレベルからの動作合成による実現化手法」
    - フォルテ・デザイン・システムズ㈱ 櫻井 至
  - 「ソニーで成功しているSystemC設計フロー」
    - ソニー㈱ 蛯原 均、丹木 秀和

© Copyright 2011 JEITA, All rights reserved JEITA

アンケート実施の目的

- SystemCの普及調査を目的として以下の方針のアンケートを実施した。
  - 国内SystemCユーザの動向の調査
  - 調査結果よりSystemCがどのくらい定着したか、また定着するために何が障害なのかを調査
  - SystemCの利用拡大を図る。

以上の調査結果と考察について報告する。

© Copyright 2011 JEITA, All rights reserved JEITA

SystemC Japan 2010　サマリー

- 講演内容について：
  - OSCIからはIEEE P1666の最新情報やその他の活動としてAMS 1.0リリース、CCI-WG (Configuration, Control & Inspection WG)の紹介
  - JEITA SystemC WGからは日本におけるSystemC標準化活動の報告
  - リコーより、Cynthesizerを用いてTLM2.0の高位合成とESLの統合について紹介
  - ルネサスエレクトロニクスからはSLECを用いた等価検証について報告
  - 富士通アドバンステクノロジからはCtoSを用いたECOの評価について報告
  - 日本システムウエアからはCoFluentを用いたアーキテクチャ探索の例について紹介
  - プライムゲートからはC言語LSI支援のシステム検討について紹介
  - ソニーからはSystemC設計について成功事例の紹介

© Copyright 2011 JEITA, All rights reserved

アンケート調査集計結果

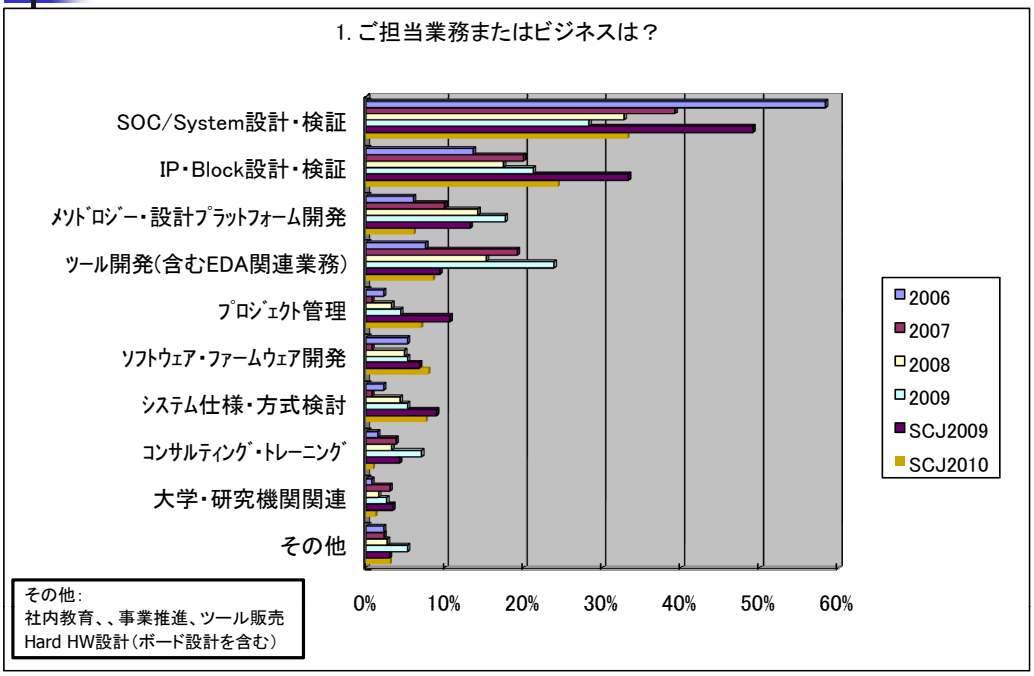
- 前回同様のアンケートと同様の内容でアンケートを実施し、今までの結果と合わせて聴講者の動向を分析しました。
  - 参加目的はユーザ事例聴講が増加した。
  - 利用されるトランザクションAPIはOSCI TLMが増加し、ほぼ半数となった。特にTLM利用者のうち2.0の利用者数が増加している。
  - 「導入の弊害となっている理由」は、言語の完成度の問題や対応ツールが増えたことにより激減。逆に、言語の習得が増加し、ツール費用が高いが増えた。
  - 「SystemCをより活用するために充実が必要なものは」については、デバッグ環境の不備が増加。

  上記以外の項目は、前年と同様の傾向がみられる。

今後もSystemCに関する調査は継続的に実施する必要がある。

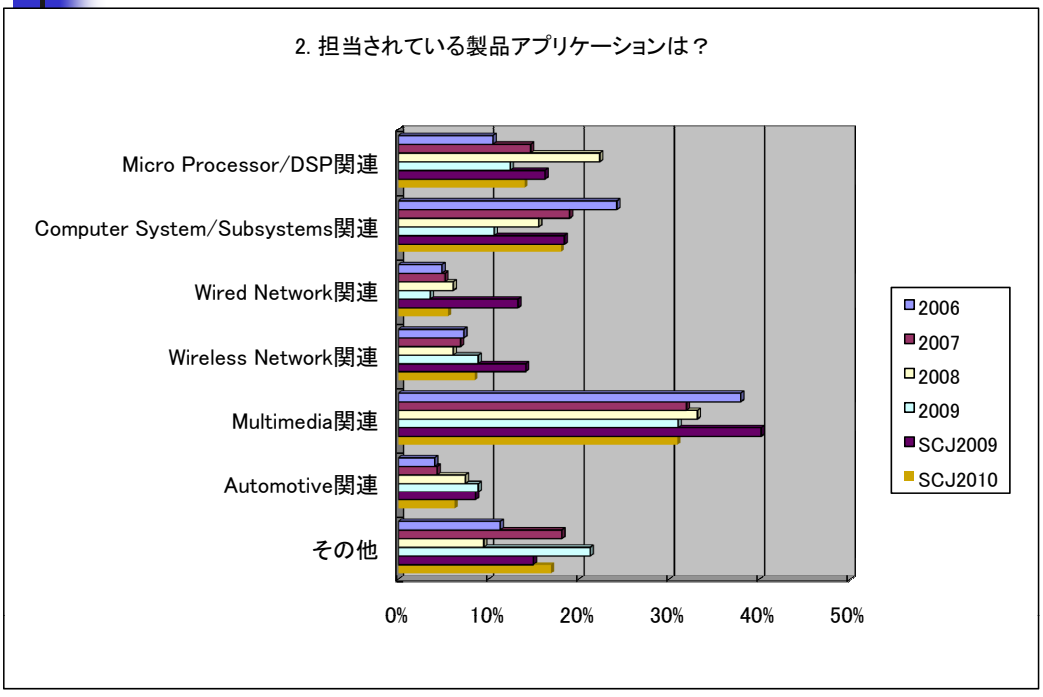
© Copyright 2011 JEITA, All rights reserved

① ご担当業務またはビジネスは?



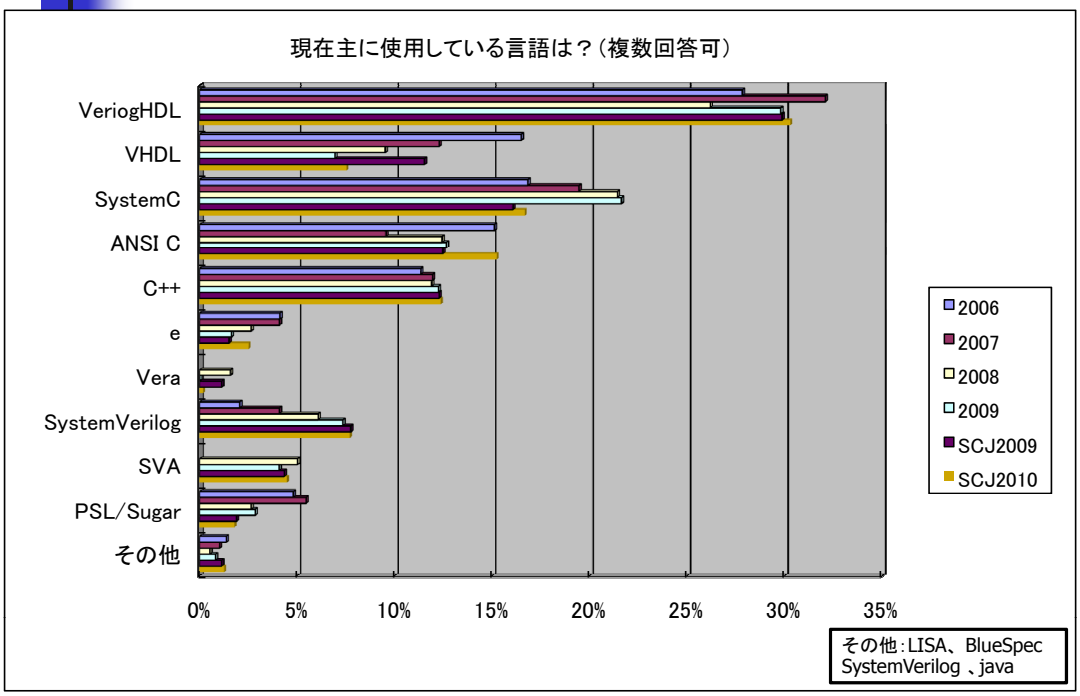
© Copyright 2011 JEITA, All rights reserved JEITA 7

② 担当されている製品アプリケーションは?



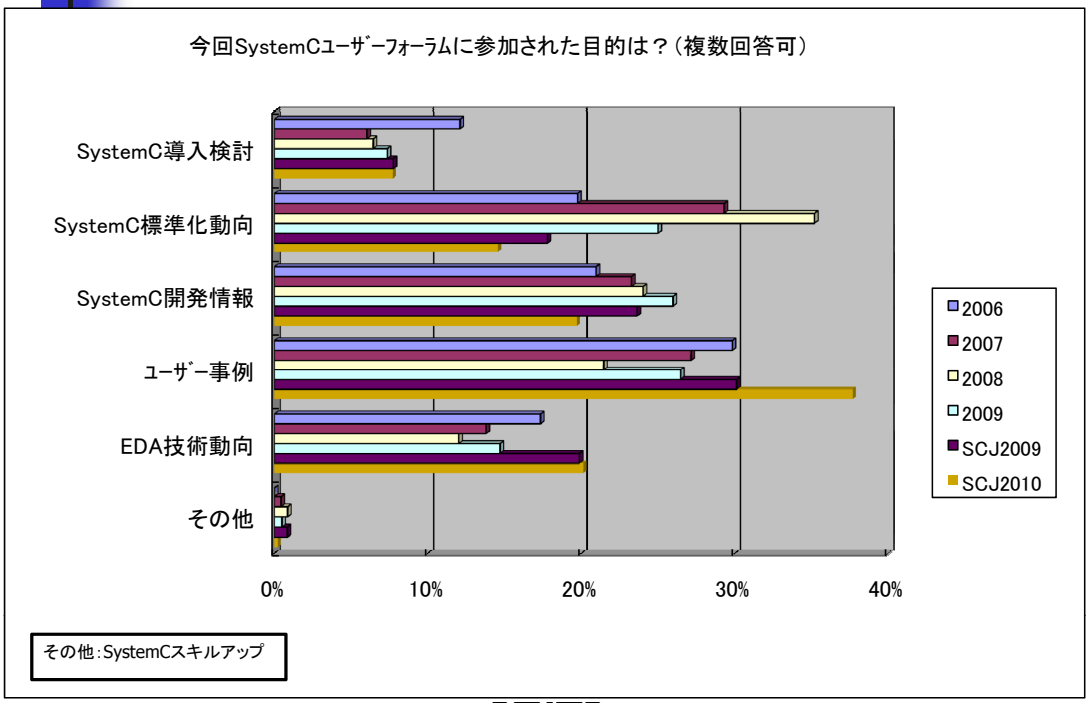
© Copyright 2011 JEITA, All rights reserved JEITA 8

③　現在主に使用している言語は?



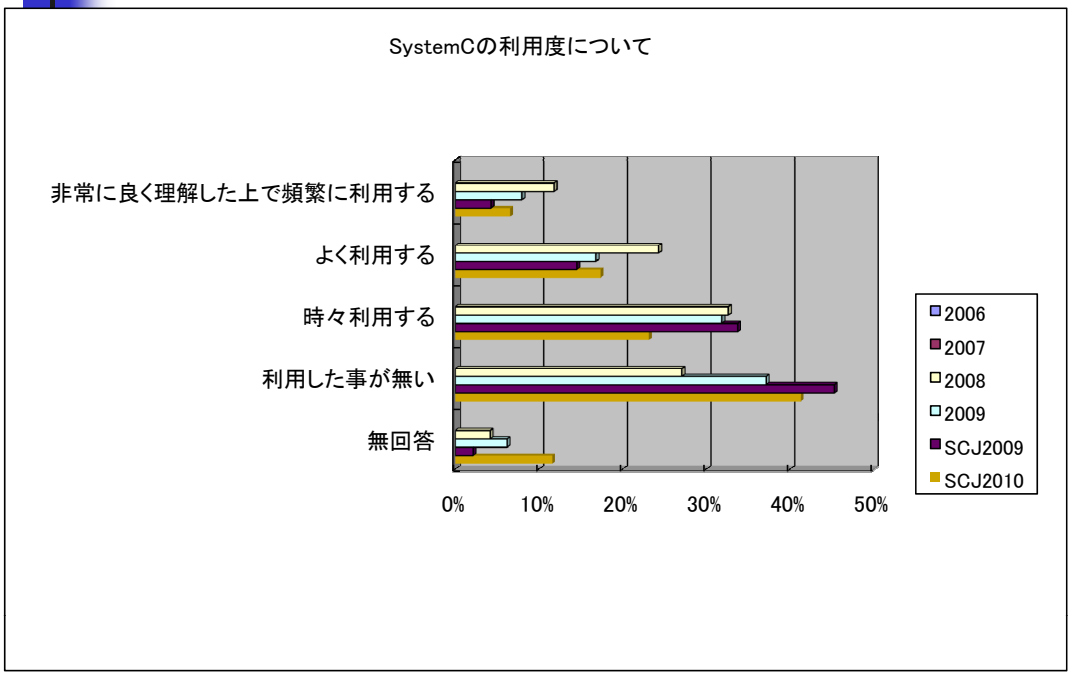
© Copyright 2011 JEITA, All rights reserved
JEITA

④SystemCユーザフォーラムに参加された目的は?



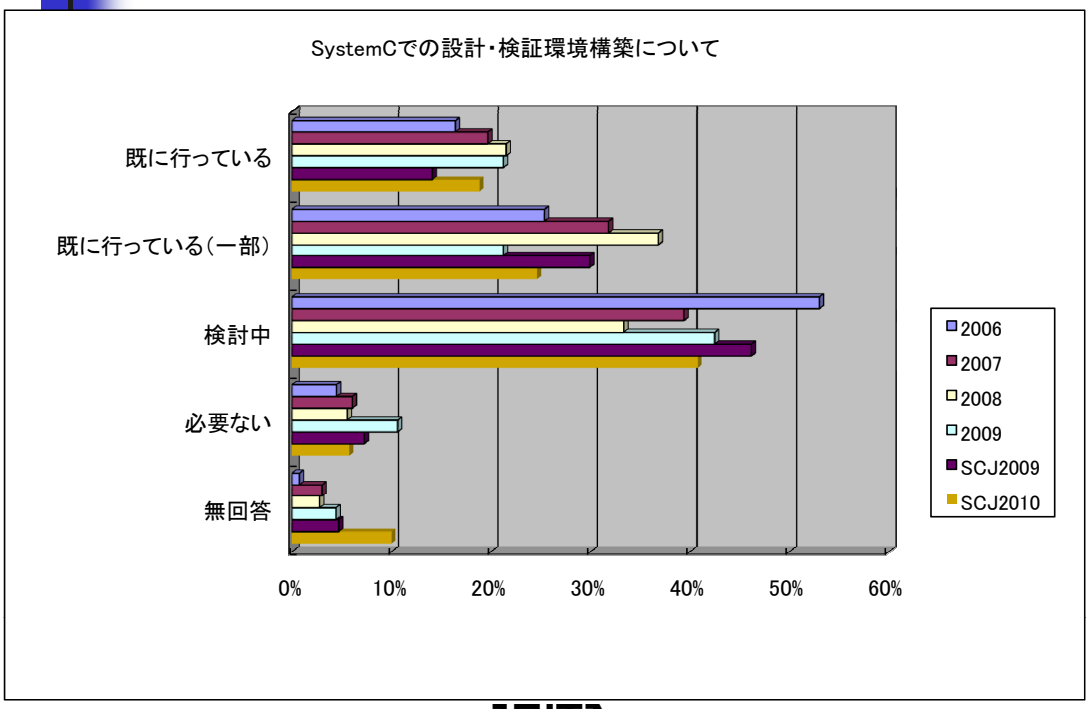
© Copyright 2011 JEITA, All rights reserved
JEITA

⑤　SystemCの利用度について



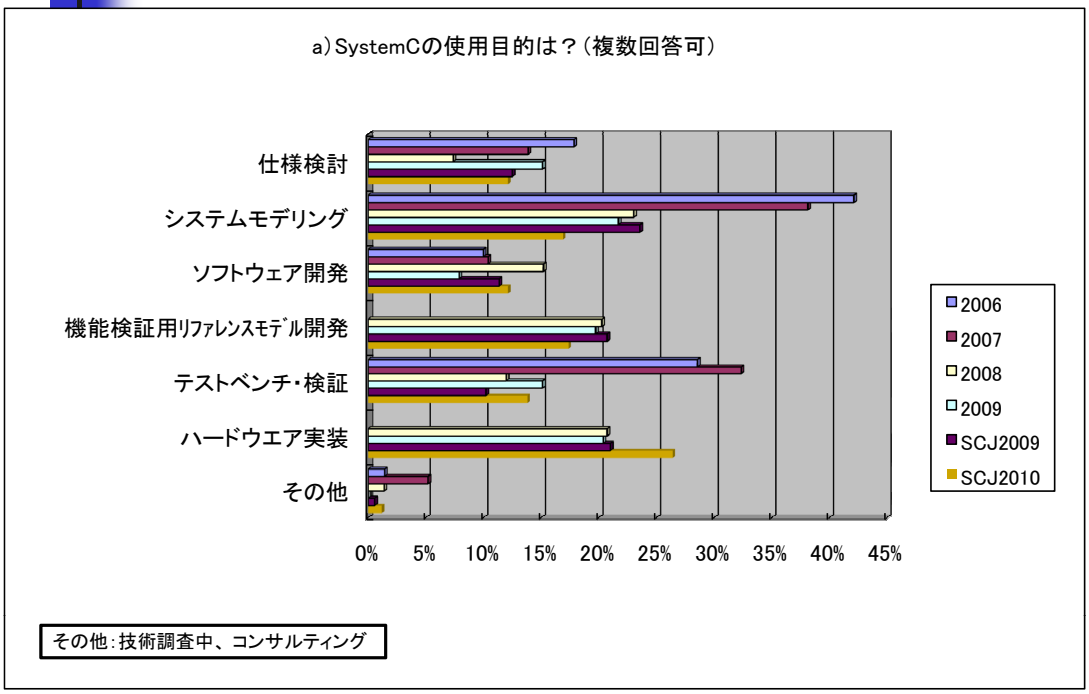
© Copyright 2011 JEITA, All rights reserved
JEITA

⑥　SystemCでの設計･検証環境構築について



© Copyright 2011 JEITA, All rights reserved
JEITA

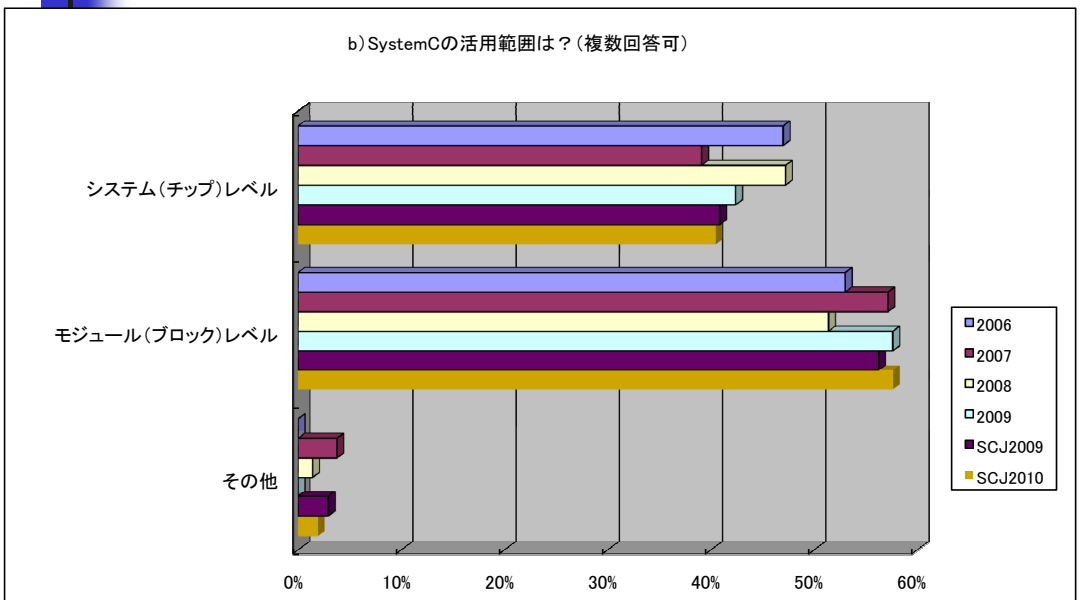
⑦　⑥で「既に行っている」または「検討中」と回答された方
a) SystemCの使用目的は?



その他：技術調査中、コンサルティング

© Copyright 2011 JEITA, All rights reserved

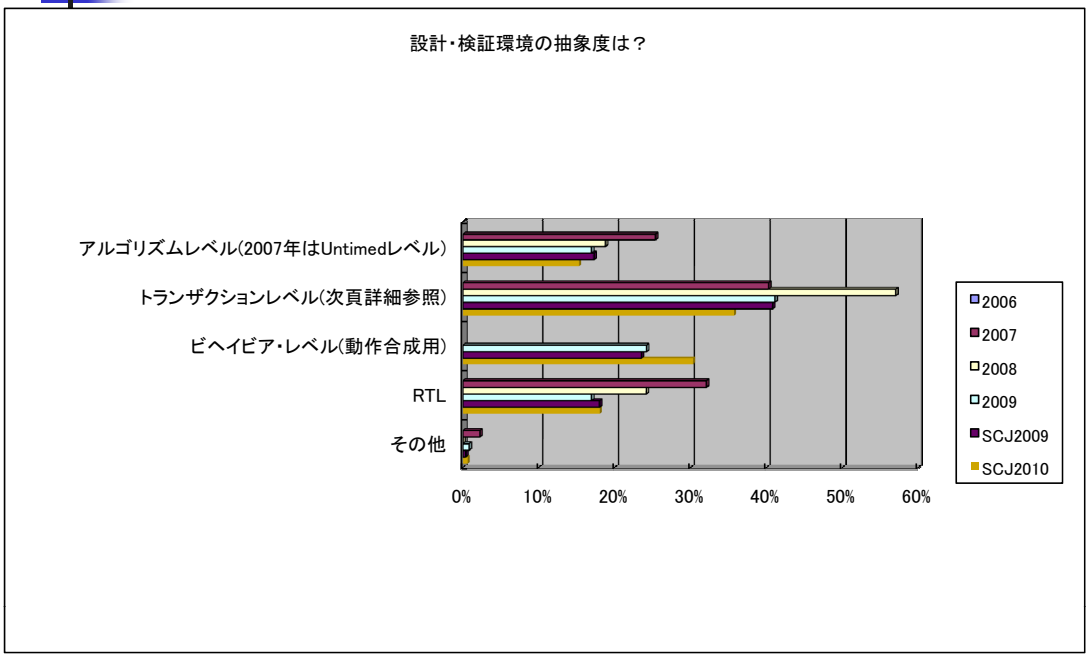
⑦　⑥で「既に行っている」または「検討中」と回答された方
b) SystemCの活用範囲は?



その他：検討中、ボードレベル、検証環境

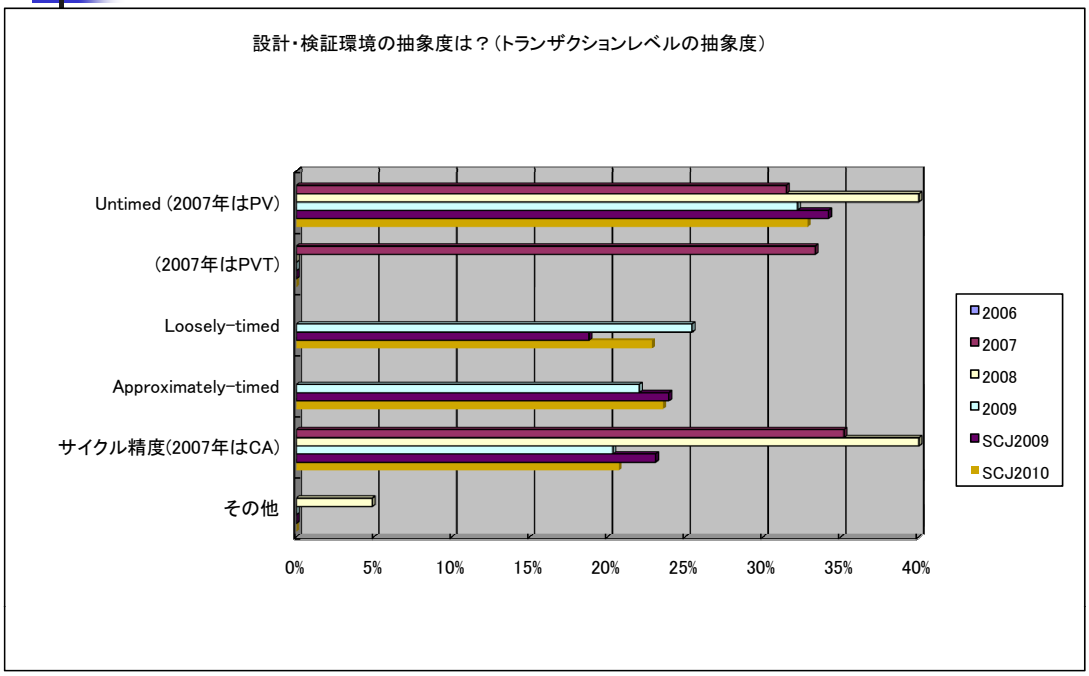
© Copyright 2011 JEITA, All rights reserved

⑦　⑥で「既に行っている」または「検討中」と回答された方
c）設計・検証の抽象度は？



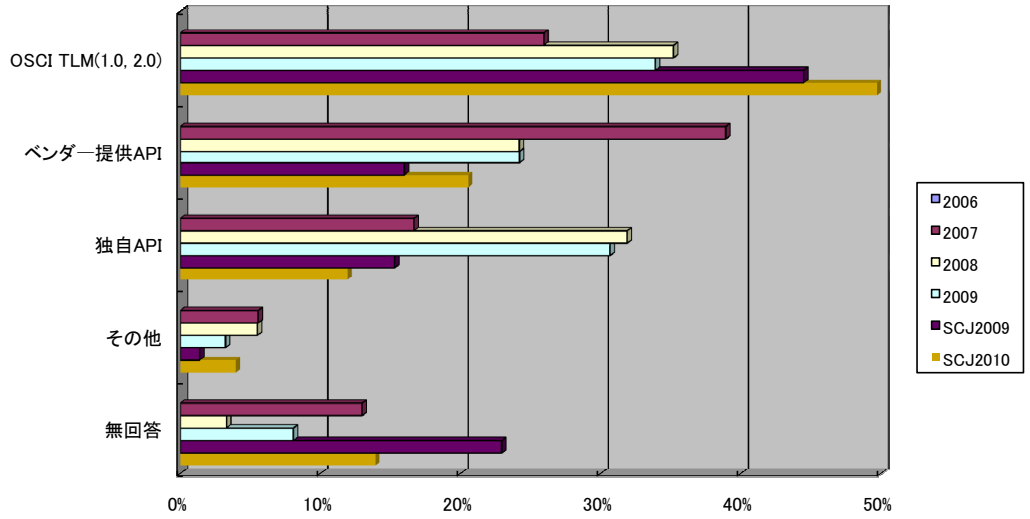
© Copyright 2011 JEITA, All rights reserved
JEITA

⑦　⑥で「既に行っている」または「検討中」と回答された方
c）設計・検証の抽象度は（トランザクションレベルの詳細）？



© Copyright 2011 JEITA, All rights reserved
JEITA

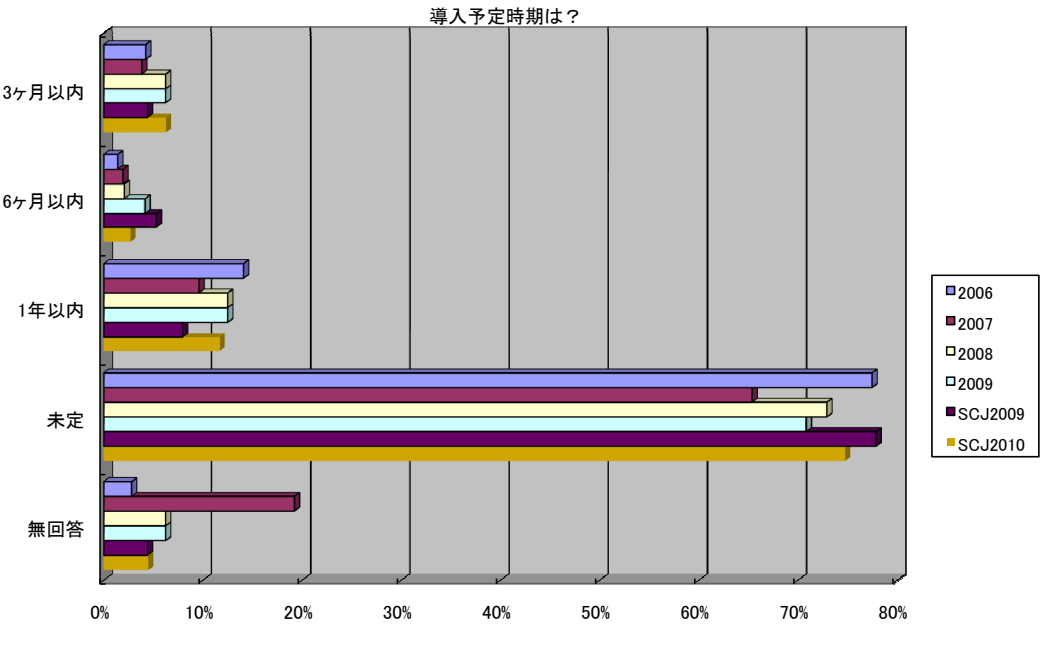
⑦ ⑥で「既に行っている」または「検討中」と回答された方
d) どのようなトランザクションAPIを使用していますか？



TLM利用者のうち2.0利用者は8割を超えた。

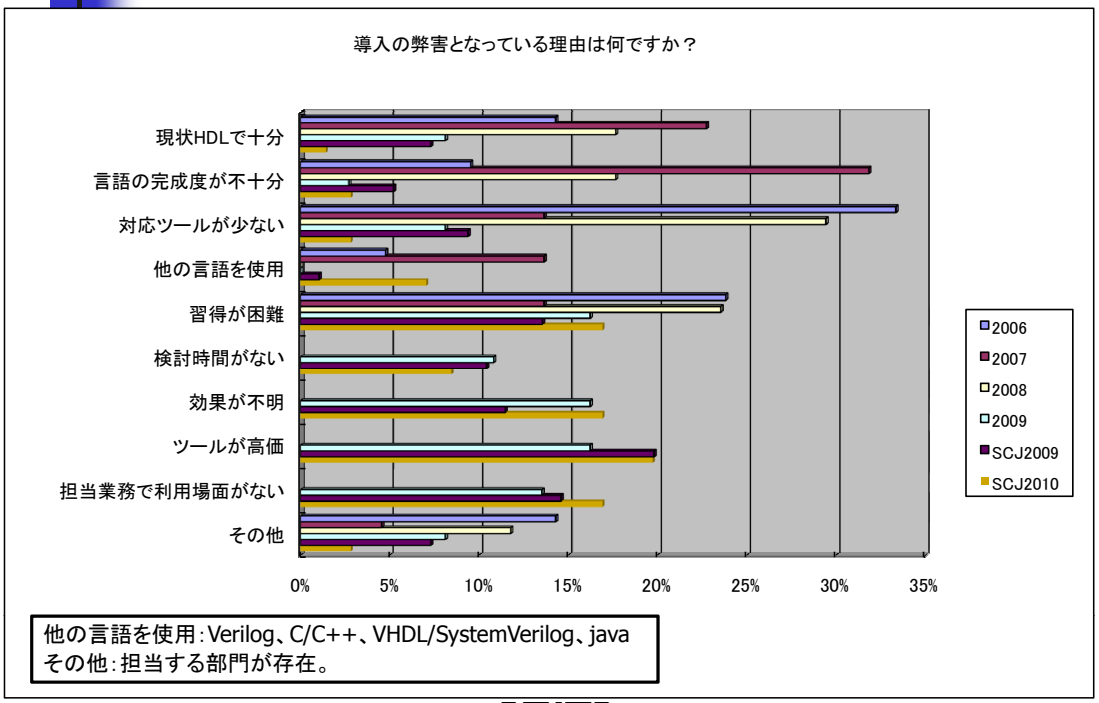
© Copyright 2011 JEITA, All rights reserved JEITA

⑧ ⑥で「検討中」と回答された方へ
導入予定時期は?



© Copyright 2011 JEITA, All rights reserved JEITA

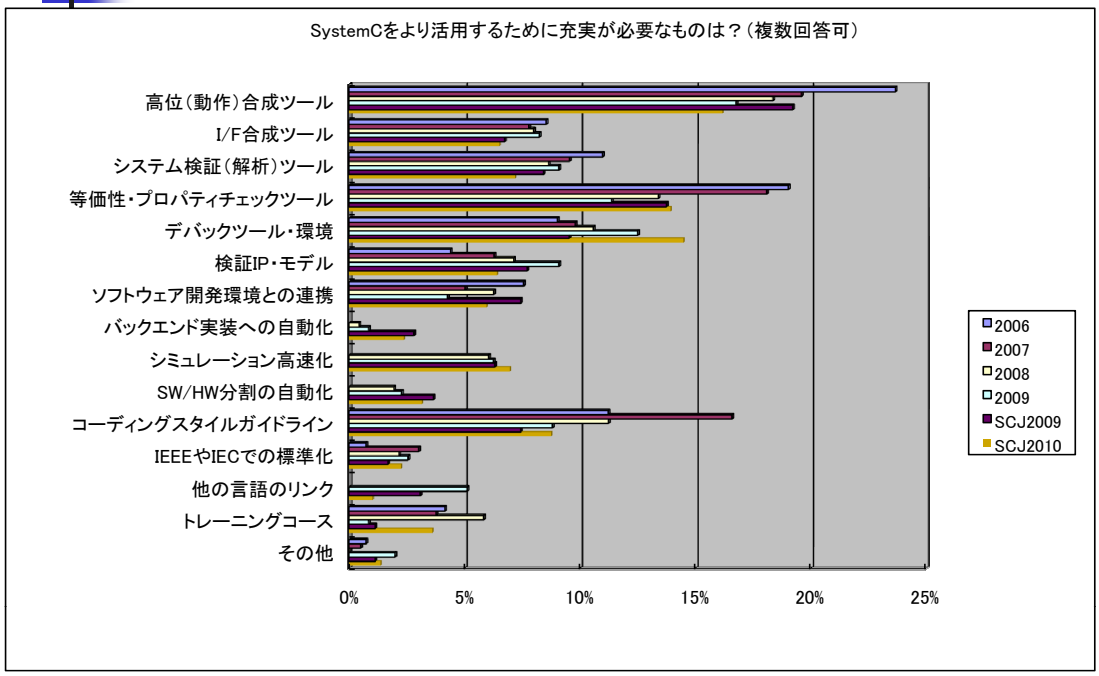
⑨　⑥で「必要ない」、「検討中」と回答された方へ導入の障害となっている理由は?



他の言語を使用：Verilog、C/C++、VHDL/SystemVerilog、java
その他：担当する部門が存在。

© Copyright 2011 JEITA, All rights reserved

⑩　SystemCをより活用する為に充実が必要なものは？



© Copyright 2011 JEITA, All rights reserved

「⑩SystemCをより活用する為に充実が必要なものは？」のその他

他の言語のリンク:

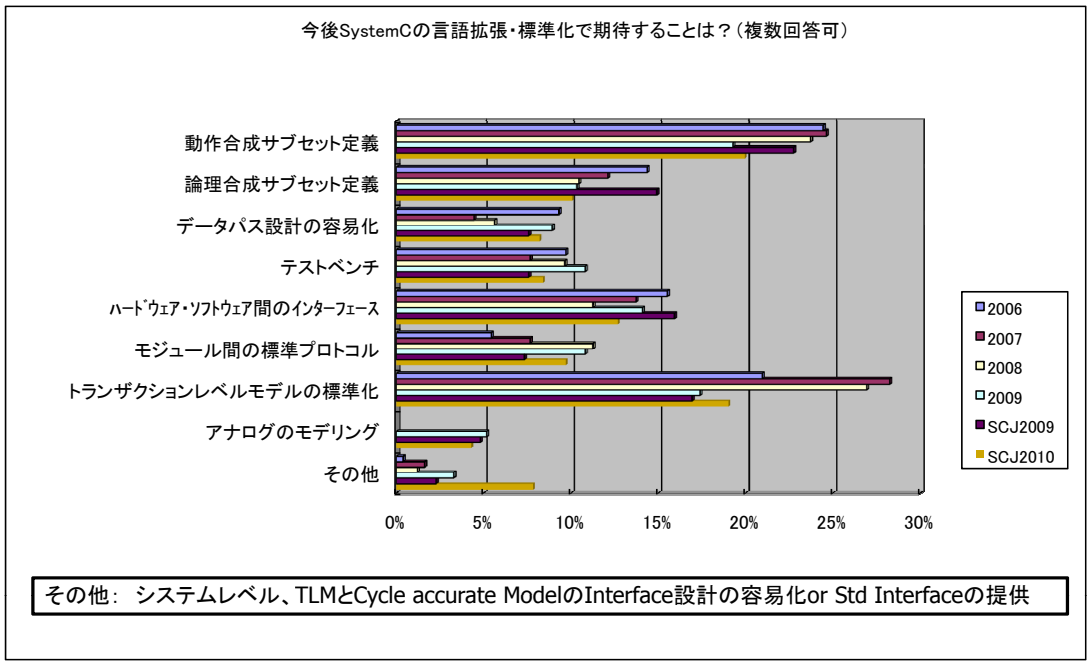
RTL、SystemVerilog 、Verilog、 XML/SPILIT、Matlab、VerilogHDL

その他:

- SCV
- FPGAツールチェーンへの安定した組み込み
- SystemCモデル自動生成（さらに上位より）
- ツールが十分に安価であること
- Modular I/F標準化
- 実装用ライブラリ ツールの低価格化・無償化
- 見積もり(Power,Spec)
- 非同期設計コーディングスタイル
- 全て。SystemCとしてのメリットを出した上で現在のRTL設計と同等の充実性が必要
- RTL混在を回避する為の標準策定

© Copyright 2011 JEITA, All rights reserved
JEITA

⑪　今後SystemCの言語拡張･標準化で期待することは?



その他：　システムレベル、TLMとCycle accurate ModelのInterface設計の容易化or Std Interfaceの提供

© Copyright 2011 JEITA, All rights reserved
JEITA

その他、ご意見、ご要望などございましたらご記入ください。

- SCVの開発に期待しています。SC_fixed, sc_ufixed対応がされていないので。
- SystemCを用いて設計した回路・プログラムの検証に関する手法・情報がもっと出れば良かった。
- 純粋なSW技術者が障壁なく利用できるように発展して欲しいです。
- 現在職務上は半導体開発より遠ざかっているが以前半導体技術を専攻にしたので，今後もこの分野に注意・注目して行きたい．今回はこのような有意なイベントに参加させていただき感謝している．EDAツールの購入予定はないが，試用版等になじんで，この分野での知見を拝借したい。
- ユーザ事例セッションとベンダーセッションを分離してほしい。
- FLD/FPGAの開発環境に含まれるツールとして、SystemCがサポートされるとありがたいです。
- 各種ツールの低価格化が必要。
- 未だにTLM中の各level (Untimed, Approximately, blah blah etc.)の定義、それらの違いについての理解が曖昧です。より解り易く、具体的なTLM定義書があればと思う次第です。更にTLM level/抽象度の違うModelを1つのSystem Simulation中に用いる場合、それらのInterfaceの設計に苦労しています。抽象度の違うModel間に適用可能な標準Interfaceがあればと思う今日このごろです。

© Copyright 2011 JEITA, All rights reserved

LPB相互設計ワークグループ

■背景と目的

システムの高速化と電源・インターフェースの低電圧化でタイミングやノイズに対するマージンが少なくなってきています。またコスト競争の激化によりコストと性能のバランス設計は益々重要となっています。これまで設計はLSI・パッケージ・ボード(以下LPB)それぞれ別々に設計ガイドに従って行われてきました。しかしながら設計マージンの減少に伴いあらかじめLPB各所個別の設計ガイドを定めることが難しくなり、LPB全体で協調して設計ターゲットを決める必要が出てきました。すなわちシステムの設計途上でシミュレーションによって設計指針を決める手法への変革が必要であり、その為には迅速でかつ正確なシミュレーションを行える環境が必要となります。

LPBが相互に協調して設計すべき部分の一例として電源網の設計があります。LPB上の電源網:Power Delivery Network(以下PDN)(図1)はノイズを小さくする為にインピーダンスを下げるように設計をします。LPB各部に存在するインダクタンス成分とキャパシタンス成分により共振が起こります。共振が発生する周波数ではインピーダンスが極端に大きくなります。すなわちLPB各部のPDNモデルを正確にシミュレーションに反映しないと全体のPDNの設計が正しく出来ません(図2)。このシミュレーションを実行するためにはLSIのPDN等価モデル、パッケージやボードの形状情報、構成材料の電気特性、構成部品の等価モデルなど様々な情報を揃えなければなりません。この際にLPBの各部分の設計情報の書式や仕様の違い、必要パラメータが共有されていない事などが効率的な設計が出来ない要因となっています。

このように、シミュレーションをとりまくモデルやインターフェースの方法・形式は定まったものが無く、セットアップにかかる時間とコストは膨大であり、LPBが協調する障壁となっています。

JEITAでは協調設計環境の効率的な構築を可能とするインターフェース方法を模索する為にLPB相互設計WGを設置し活動しています。

■発足・変遷

2009年7月準備WG発足、2010年4月より正式WG

(参加メンバ)16社／20名+特別委員3名　客員1大学／1名

主査:福場義憲／(株)東芝

委員構成は半導体・セット・EDA・大学と多様

■議論概要

設計のフローを模式化すると3つの階層にまとめられます。

1、製品企画・仕様からLPB各個の設計ターゲットを決める工程

2、LPB各個のCAD設計

3、LPB設計結果の解析(CAE)・評価・設計へのフィードバック

このうちEDA(CAD／CAE)の入力部分となるところは設計部門・ツール・ベンダーによって仕様が統一されておらず、この部分の共通仕様が模索されています。(図3)

流通するべき情報としては

・ネットリスト

・形状データ

・電気・物理・熱特性や制約などのパラメータ

これらがLPB環境で共通・統合化されて運用される必要があります。(図4)

■活動内容

WG内で3つのサブWGを構成して活動をしています。

◆インターフェースフォーマットSWG

各設計フェーズ、あるいは解析において必要となる項目と情報の洗い出しと整理、さらに、洗い出した項目や情報が既存フォーマットで表現可能かどうかを調査し、それを元にインターフェースフォーマットとしての共通仕様案を策定することを目的としています。

◆企画・広報SWG

本WGの活動内容を公開することを目的として活動しており、SDFやEDSフェア特設ステージの企画などを行っています。

◆リファレンスフローSWG

標準的なLPB相互設計フローを定義し、このフローにおける共通インターフェースフォーマットの活用方法を提案します。

以上、3つのSWGが協力しあって効率的なLPB相互設計環境の構築を可能とするインターフェースフォーマット案を検討しています。

図1. LPBの電源網(PDN)の等価回路

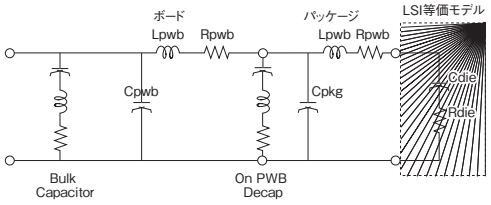


図2. ボード電源インピーダンスにおけるチップ電源モデル有無の影響

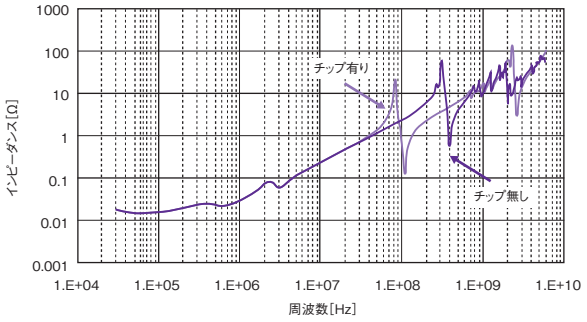


図3. 相互設計における共通化領域

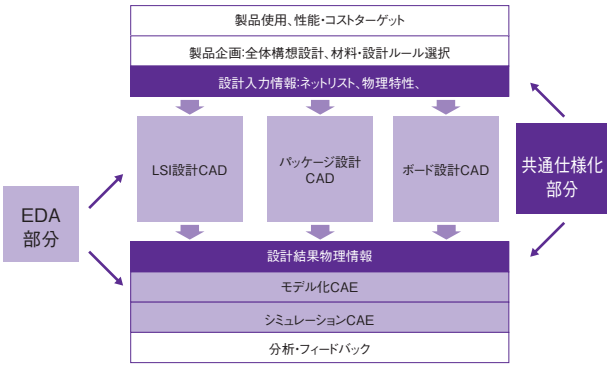
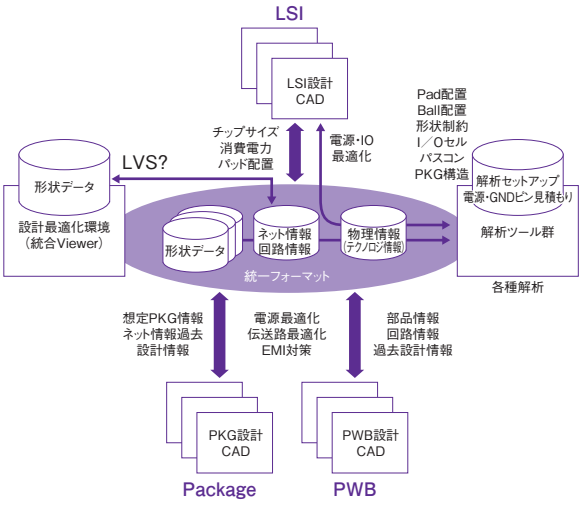


図4. 情報・データ共有イメージ



LSI-Package-Board相互設計WG紹介

2011年2月24日（木）

JEITA半導体技術委員会
EDA技術専門委員会
LPB相互設計　WG

主査：福場 義憲（東芝）

内容

1. LPB相互設計WG設立の背景
2. メンバー
3. 議論内容・ゴール
4. LPB相互設計WG活動内容とマイルストン
5. 標準化の内容と提案
6. 標準化の効果
7. まとめ

LPB相互設計WG設立の背景

■ 背景

- 国際市場における日本工業製品の競争力強化

1.電子機器(セット)の高性能化、低価格化、商品サイクルの短期化

2.日本製品の差別化部分： 機能・性能・品質

3. 1. 2. 両面をいかに実現するか？

- 電子機器産業界(特に日本)の現状と課題
  - LSI/パッケージ/ボード各分野での設計力はある。しかしながらそれらの連携がとれておらず全体としての最適化がすすまない。
  - 競争するべき分野と協調するべき分野が明確にされていない。
  - VOCの分散によりEDA技術活用の効率化が進まない

3

LPB相互設計WG設立の目的

●LPBが協力しあって設計を行う為の課題を抽出しその分析を行いソリューションの提案を行う。

●具体的にはLPB全体での接続記述、形状情報、解析設定条件に着目し、それらの定義や書式等の検討を行い提案を作成する。

●提案内容は公開し普及を図る。

●これにより日本のエレクトロニクス製品の国際的競争力を高めることを目標とする。

4

議論内容・ゴール

■ 初年度活動範囲

1. EDAツール間のインターフェースで問題となる部分の調査、共通仕様検討
2. LPBそれぞれの設計プロセスで必要となる情報の調査、共通仕様検討
3. ネット・物理形状・物性に関する情報の内容・仕様検討、ドキュメント化、公開

5

参加メンバー　17団体、24名

お名前	会社・学校	役職	正／副
福場義憲	東芝	主査	
田中修治	ソニー	副主査	
大坪祐治	図研	委員	正
古賀一成	図研	委員	副
市川浩司	デンソー	委員	
岡野資睦	東芝	委員	副
冨島敦史	東芝	委員	副
青木孝哲	東芝	委員	副
金子俊之	トッパンNECサーキットソリューションズ	委員	
齋藤義行	パナソニック	委員	
佐藤厚志	富士通セミコンダクター	委員	正
小澤要	富士通セミコンダクター	委員	副
中川裕之	富士通VLSI	委員	副
久島憲司	メンタージャパン	委員	
大槻 隆志	リコー	委員	
吉田　敦史	リコー	委員	
永野民雄	ルネサスエレクトロニクス	委員	正
木本務	ルネサスエレクトロニクス	委員	副
楠本学	NEC	委員	
林　靖二	キヤノン	委員	
須藤 俊夫	芝浦工業大学	客員	
板橋裕行	ATEサービス	特別委員	
人見忠明	アバッチデザインソリューションズ	特別委員	
渡辺了	アンソフトジャパン	特別委員	
小島智	NECシステムテクノロジ	オブザーバー	
吉田正昭	ルネサスエレクトロニクス	オブザーバー	

6

LPB相互設計WG活動内容とマイルストン

4月	5月	7月	9月	10月	11月	1月	3月

抽出フェーズ
- 設計工程における現状の問題点の調査・可視化・共有化

調査分析フェーズ
- 既存のツール間インターフェース方法・フォーマットの調査
- 各方法の内容比較/メリット・デメリットの検討

仕様作成フェーズ
- 共通仕様を実施する範囲の選定
- 共通仕様の方針、内容の決定

評価・公開フェーズ
- 参加メンバーによる評価
- 内容公開

1回目:4/21
問題定義
•問題点自由討議

2回目:5/26
課題抽出
•セット側要望
•半導体側要望

3回目:7/6
問題点分析
フォーマット基本案

4回目:9/3
フォーマット案レヴュー
SDFの計画

5回目:10/12

6回目:11/30

7回目:1/11

8回目:3/8

仮サブWG実施
•関西部会
•関東部会

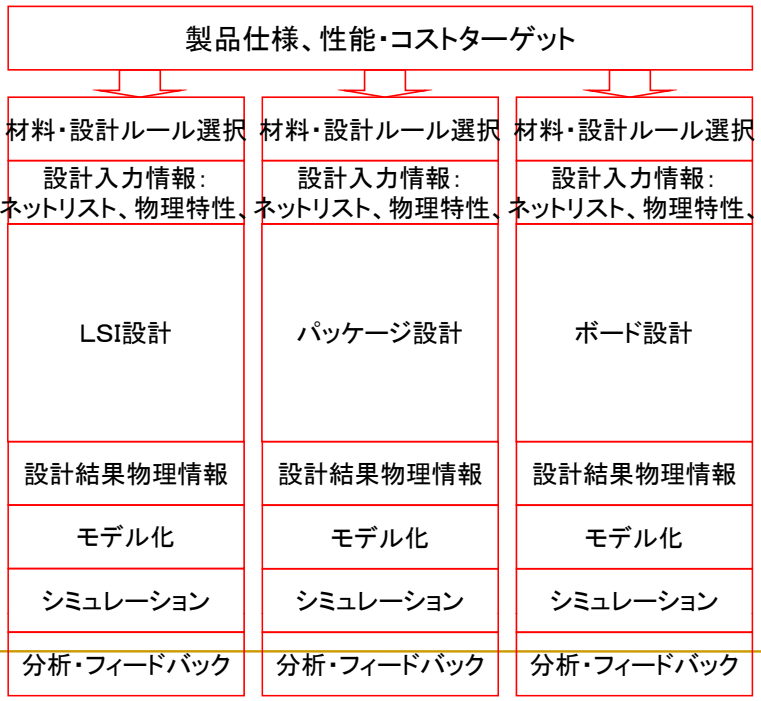
サブWG発足
•インターフェースフォーマット
•企画(SDF)
•リファレンスフロー

SFD2011(1/27)
1セッション担当
活動内容紹介

7

従来型個別設計

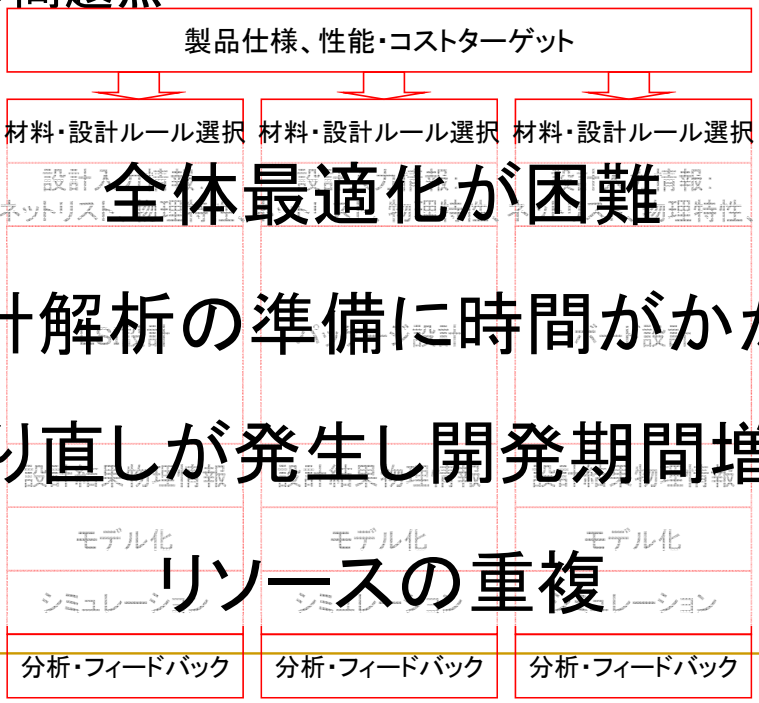
- 従来



8

従来型個別設計

- 従来の問題点



全体最適化が困難

設計解析の準備に時間がかかる

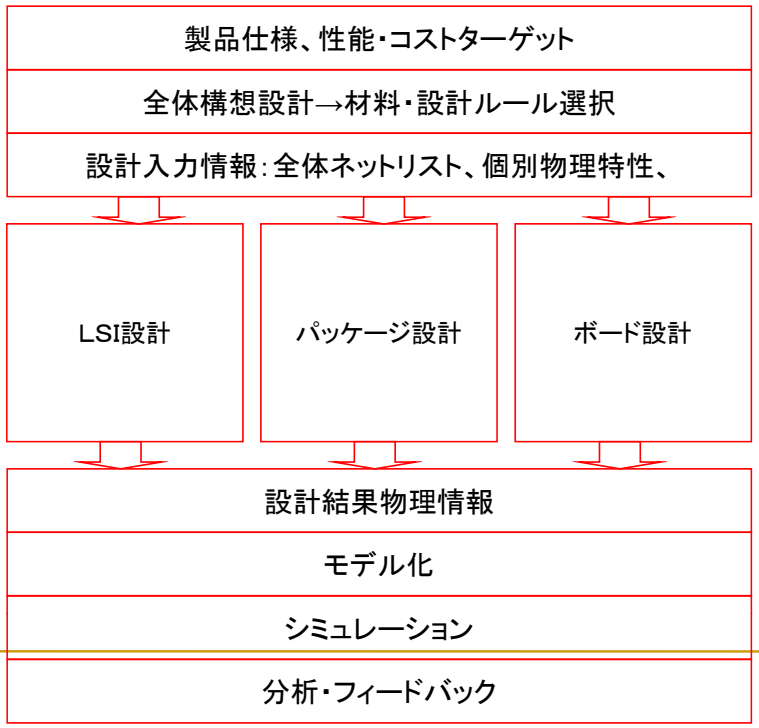
やり直しが発生し開発期間増大

リソースの重複

9

LPB相互設計

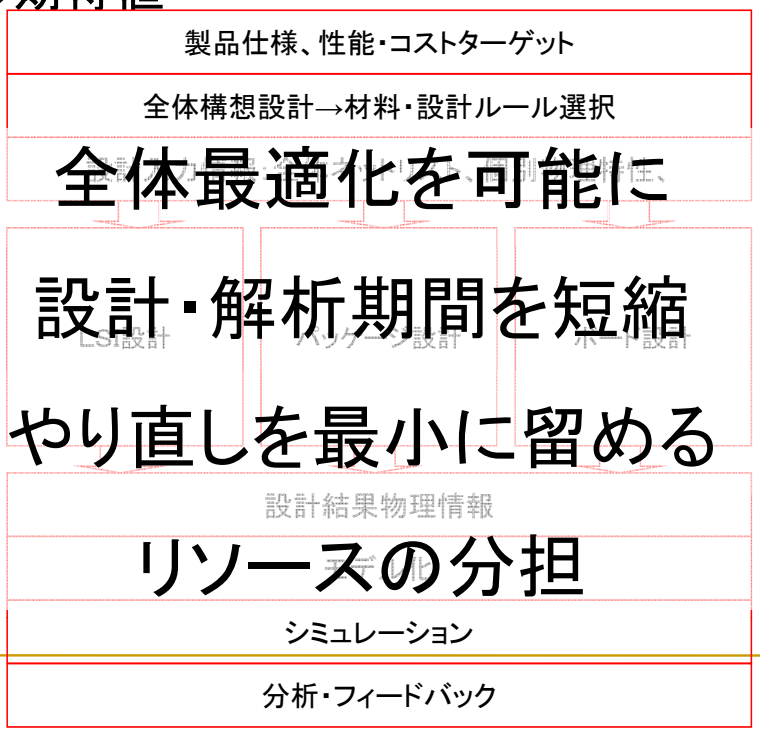
- 連携



10

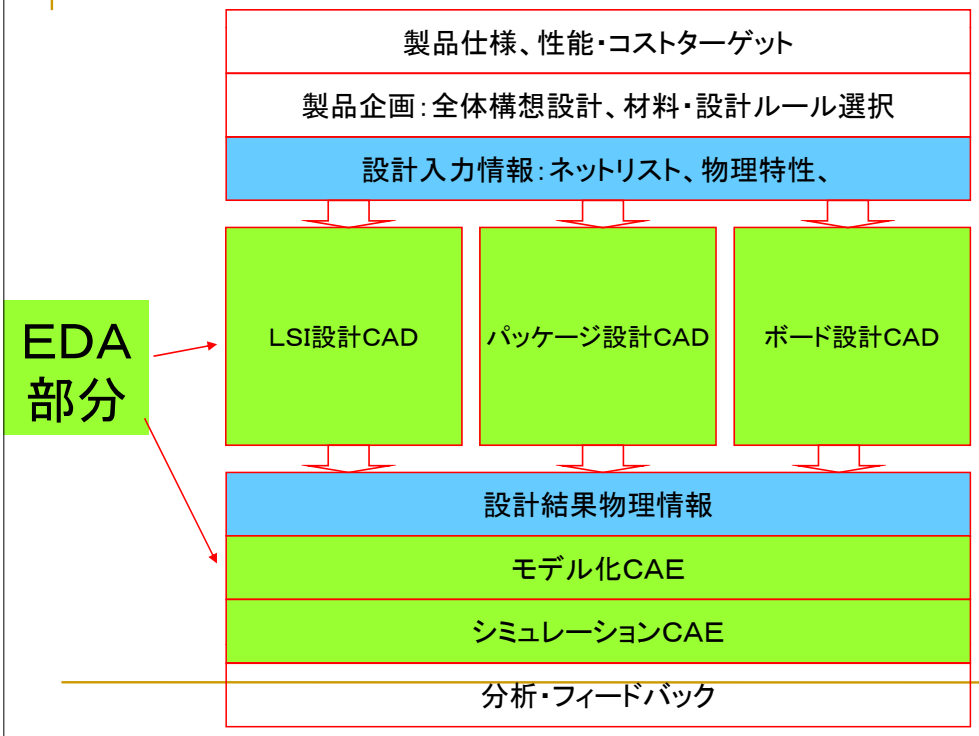
LPB相互設計

- 連携の期待値

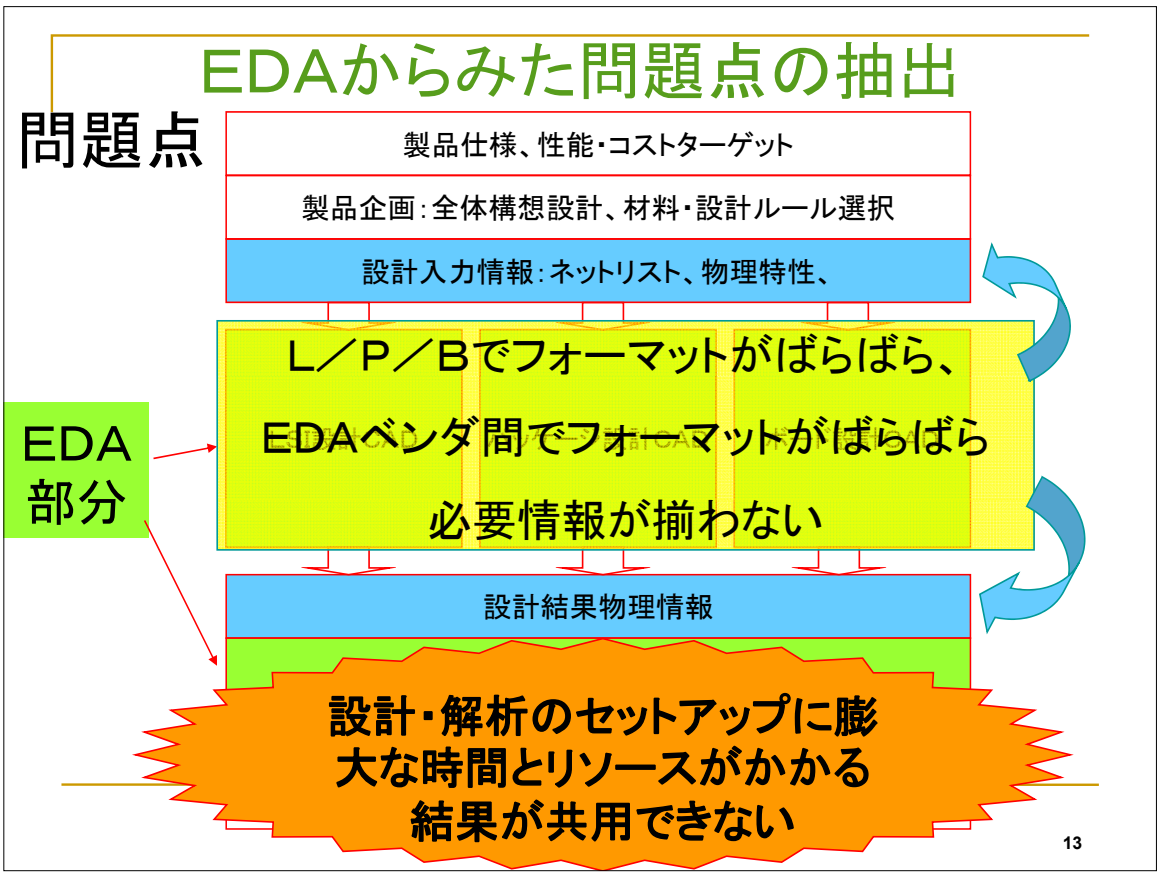


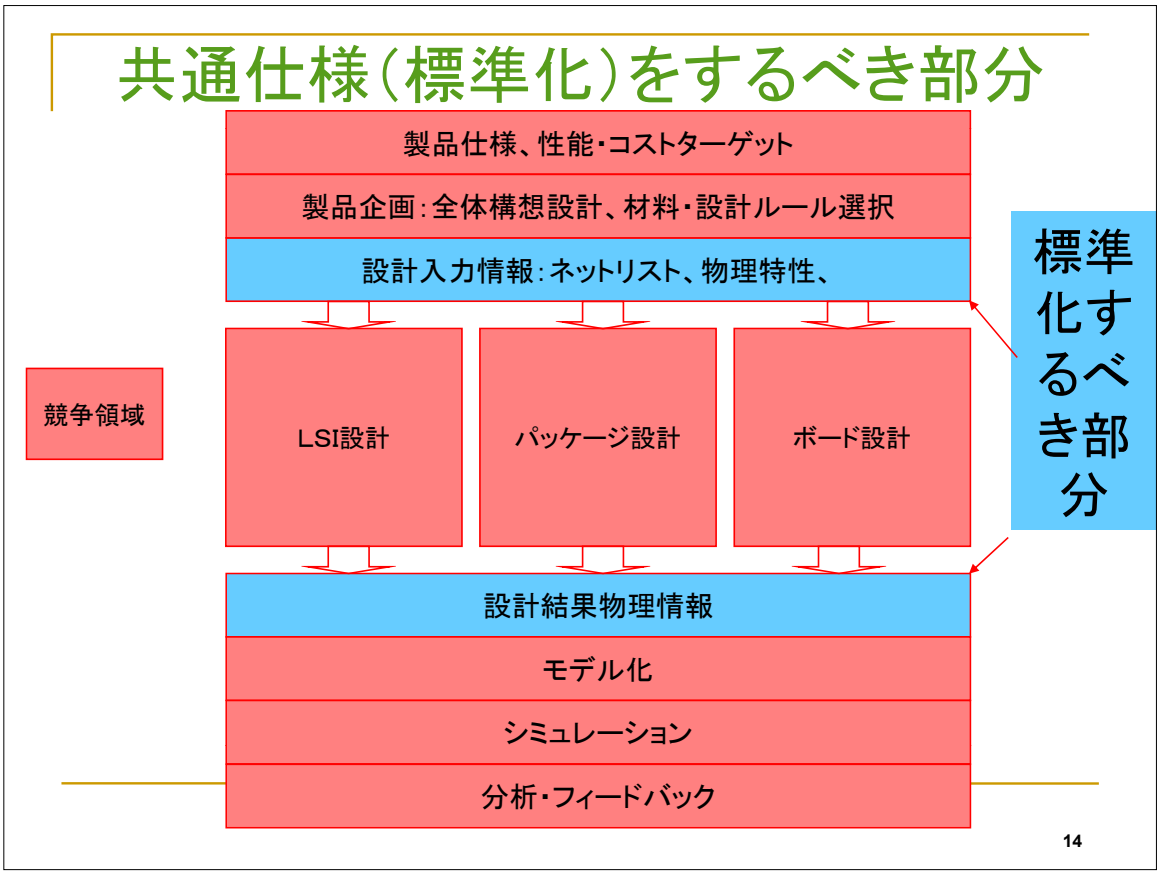
11

LPB相互設計におけるEDA部分

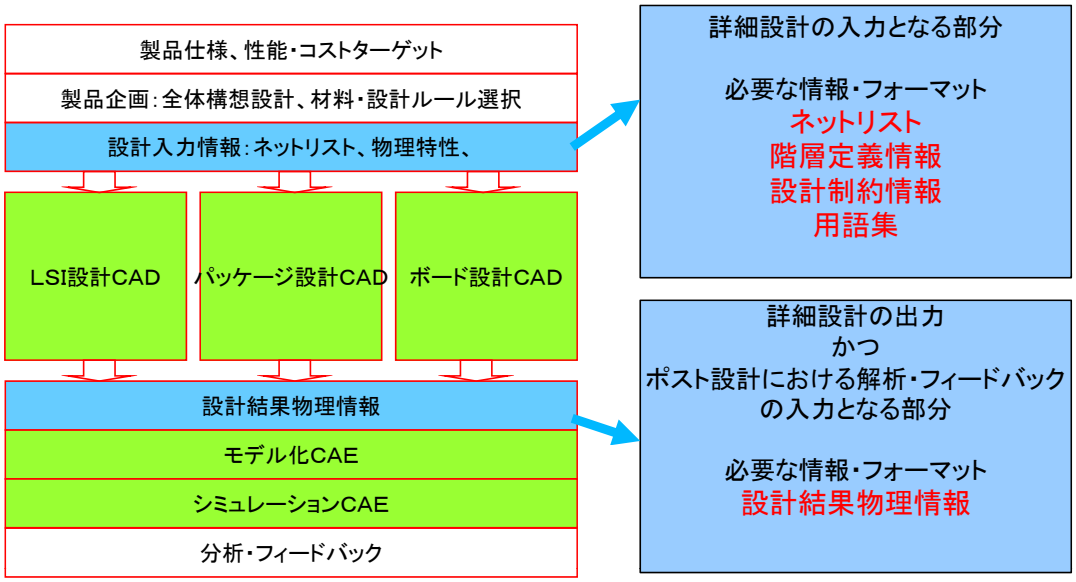


12


EDAからみた問題点の抽出
問題点
製品仕様、性能・コストターゲット
製品企画：全体構想設計、材料・設計ルール選択
設計入力情報：ネットリスト、物理特性、
L／P／Bでフォーマットがばらばら、
EDAベンダ間でフォーマットがばらばら
必要情報が揃わない
EDA
部分
設計結果物理情報
設計・解析のセットアップに膨
大な時間とリソースがかかる
結果が共用できない
13


共通仕様（標準化）をするべき部分
製品仕様、性能・コストターゲット
製品企画：全体構想設計、材料・設計ルール選択
設計入力情報：ネットリスト、物理特性、
競争領域
LSI設計
パッケージ設計
ボード設計
標準
化す
るべ
き部
分
設計結果物理情報
モデル化
シミュレーション
分析・フィードバック
14

共通仕様化（標準化）部分の提案



15

共通仕様（標準化）部分提案

- 設計入力　：　フォーマット3つ＋用語を定義

  1.ネットリスト：　Verilog(SPICE制約互換）

  2.階層定義情報：独自定義　XML出力

  3.設計制約情報：独自定義　（XML出力：協議中）

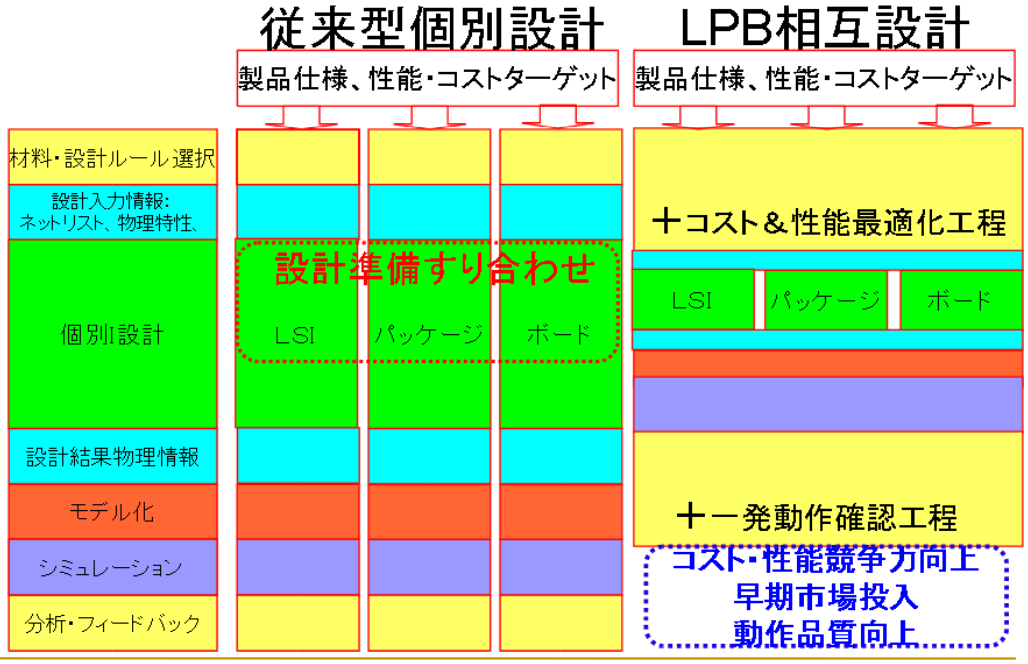
  4.用語集

- 設計出力＝解析入力

  1.設計結果物理情報　既存フォーマット XFL

16

標準化の効果の視覚化



17

1

標準化の効果のまとめ

- ✓ 入力情報が整備され詳細設計工程はすり合わせ準備がなくなるので短期化できる。
- ✓ 構想設計が促進されコスト・性能のバランス最適化が促進される。
- ✓ 形状情報が統一され、モデリングもスムースになる。後工程（シミュレーションフィードバック）がLPB全体で出来るようになって設計品質が向上する。

18

全体のまとめ

- LPB相互設計実現の為にJEITAで活発な議論と共通仕様作成の努力が成された。
- インターフェース部分について共通仕様の提案が成された。
- 今後はその実施と普及が課題である。LPB相互設計WGはこれに取り組むべく来期も継続したいという要望がでている。

19

LSI-Package-Board相互設計WG インターフェースフォーマットSWG インターフェースフォーマット一覧と機能評価結果 R1.0 用語集 R1.0

2011年3月11日

JEITA半導体技術委員会
EDA技術専門委員会
LPB相互設計 WG

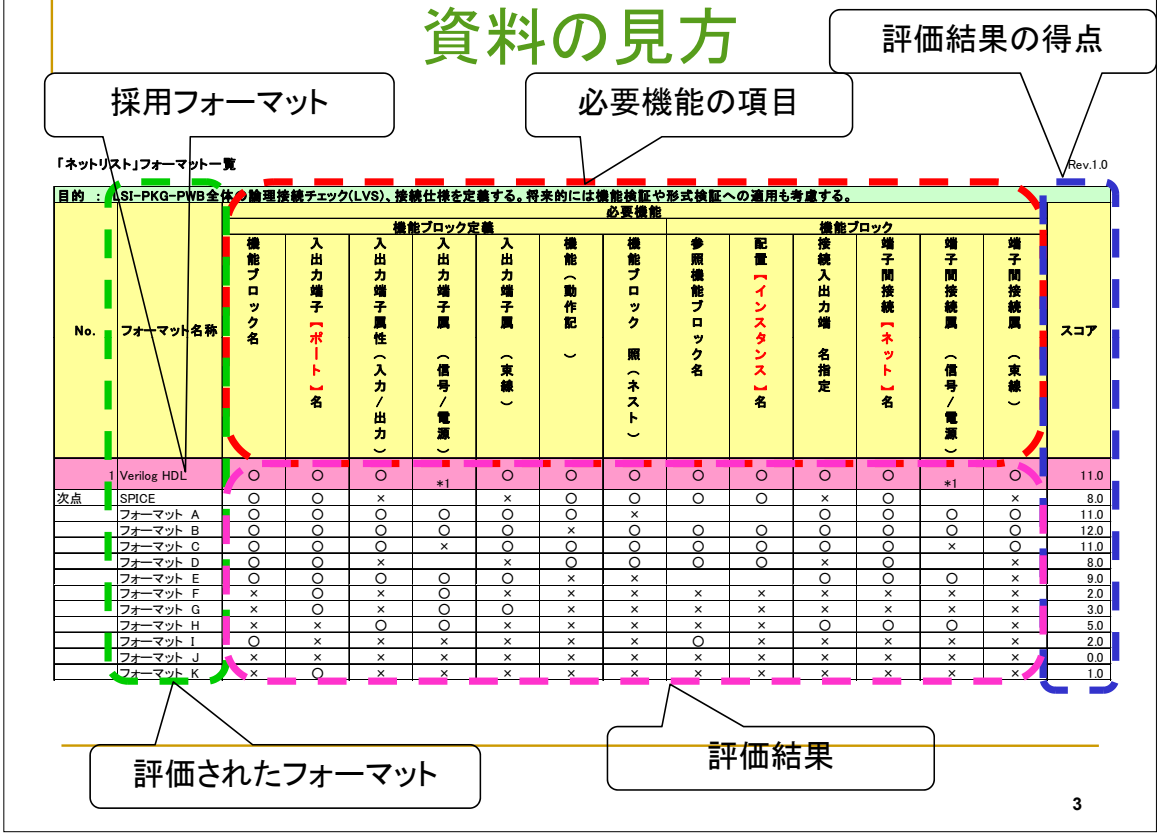
1

内容

1. 資料の見方
2. ネットリストフォーマット一覧と機能評価
3. 階層間定義フォーマット一覧と機能評価
4. 設計制約フォーマット一覧と機能評価
5. 形状フォーマット一覧と機能評価
6. 用語集

2

資料の見方



3

ネットリストフォーマット一覧と機能評価

「ネットリスト」フォーマット一覧 Rev.1.0

目的 ： LSI-PKG-PWB全体の論理接続チェック(LVS)、接続仕様を定義する。将来的には機能検証や形式検証への適用も考慮する。

No.	フォーマット名称	機能ブロック名	入出力端子【ポート】名	入出力端子属性（入力／出力）	入出力端子属（信号／電源）	入出力端子属（束線）	機能（動作記）	機能ブロック照（ネスト）	参照機能ブロック名	配置【インスタンス】名	接続入出力端名指定	端子間接続【ネット】名	端子間接続属（信号／電源）	端子間接続属（束線）	スコア
1	Verilog HDL	○	○	○	*1	○	○	○	○	○	○	○	*1	○	11.0
次点	SPICE	○	○	×		×	○	○	○	○	×	○		×	8.0
	フォーマット A	○	○	○	○	○	○	×			○	○	○	○	11.0
	フォーマット B	○	○	○	○	○	×	○	○	○	○	○	○	○	12.0
	フォーマット C	○	○	○	×	○	○	○	○	○	○	○	×	○	11.0
	フォーマット D	○	○	×		×	○	○	○	○	×	○		×	8.0
	フォーマット E	○	○	○	○	○	×	×			○	○	○	×	9.0
	フォーマット F	×	○	×	○	×	×	×	×	×	×	×	×	×	2.0
	フォーマット G	×	○	×	○	○	×	×	×	×	×	×	×	×	3.0
	フォーマット H	×	×	○	○	×	×	×	×	×	○	○	○	×	5.0
	フォーマット I	○	×	×	×	×	×	×	○	×	×	×	×	×	2.0
	フォーマット J	×	×	×	×	×	×	×	×	×	×	×	×	×	0.0
	フォーマット K	×	○	×	×	×	×	×	×	×	×	×	×	×	1.0

4

階層間定義フォーマット一覧と機能評価

「階層間定義」フォーマット一覧　　Rev.1.0

目的 ： PKG、PWBに搭載する部品の配置仕様（部品の端子の物理定義と上位階層での配置情報）を定義する。またネットリストでブラックボックスとして扱う部品のモデル（Sparaなど）の割付けも行う。

No.	フォーマット名称	必要機能																												スコア
		機能ブロック定義（外観部）																	機能ブロック参照・接続定義（接続部）												
		機能ブロック名	機能ブロック属性	配置領域（矩形・穴・凹凸）	機能概念	入出力端子（ボール・ピン）	入出力端子属性（入力/出力）	入出力端子属性（信号/電源）	入出力端子属性（差動）	入出力端子名	入出力端子配置	入出力端子（レイヤー）	機能の置き換え	接続不要端子名	機能概念端子名（CSPカテゴリ）	機能概念端子名（ボール・ピン・リード）	機能概念端子名（ボール・ピン・リード・配置）	機能概念端子名（ボール・ピン・リード・サイズ・属性）	機能概念端子名（ボール・ピン・リード・サイズ・属性）	機能概念端子名（配置領域）	機能概念端子名（配置領域）	機能概念端子名（配置領域）	スケール（縮尺率）	機能概念名	参照階層名（CSPカテゴリ）	配置マトリックス	参照インデックス	配置領域（矩形・サイズ）	配置領域（矩形＋ミラー）	配置領域（レイヤー）	配置領域（矩形・属性）	接続入出力端子名称変換	端子属性（ボール・ピン・リード・属性）	端子属性（ボール・ピン・リード・配置）	端子属性（配置領域）	端子属性（配置領域）	
1	独自フォーマット検討	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	34.0
	フォーマット L	○	×	○	○	○	○	○	×	×	○	○	×	-	-	-	-	-	-	○	○	×		-	○	○	×	○	×	○	×	×	16.0
	フォーマット M	○	×	○	○	○	○	○	×	×	○	○	×	-	-	-	-	-	-	○	○	×		-	○	○	○	○	×	○	○	×	19.0
	フォーマット N	○	-	○	○	○	○	○	○	×	○	○	×	-	-	-	-	-	-	○	○	×	○	○	-	○	○	×	○	×	○	○	×	19.0
	フォーマット O	○	-	×	×	○	○	×	○	×	×	×	×	-	-	-	-	-	-	×	×	×	○	○	-	×	×	×	○	×	○	×	○	9.0
	フォーマット P	○	-	×	×	○	○	×	○	×	×	×	×	-	-	-	-	-	-	×	×	×	○	○	-	×	×	×	○	×	○	○	×	9.0
	フォーマット Q	○	-	×	×	○	○		×	×	×	×	○	-	-	-	-	-	-	×	×	×	○	○	-	×	×	×	×	×	○	×	×	7.5
	フォーマット R	-	-	-	-	○	-	○	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2.0
	フォーマット S	-	-	-	-	○	-	○	○	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3.0
	フォーマット T	×	-	×	×	○	×	○	×	×	○	×	×	-	-	-	-	-	-	×	×	×	○	○	-	×	×	×	○	×	○	×	○	8.0
	フォーマット U	○	-	○	○	×	×	×	×	×	×	×	×	-	-	-	-	-	-	○	○	×	○	×	-	○	○	×	×	×	×	×	×	8.0
	フォーマット V	×	-	×	×	×	×	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	-	×	×	×	×	×	×	×	×	0.0
	フォーマット W	×	-	×	○	×	×	×	×	×	×	×	×	-	-	-	-	-	-	○	○	×			-	○	○	○	○	○	○	×	×	11.0
	フォーマット X	×	-	×	×	×	×	×	×	×	×	×	×	-	-	-	-	-	-	×	○	×	×	×	-	○	×		○	○	○	×	×	6.5

5

設計制約フォーマット一覧と機能評価

「設計制約」フォーマット一覧　　Rev.1.0

目的 ： LSI-PKG-PWB全体のテクノロジ（指定値、L/Sなどのデザインルール）、製造性のルールを定義する。電的特性などの制約は今後の課題とする。

No.	フォーマット名称	必要機能																														スコア
		基板デザインルール																			設計制約												
		層数	層厚さ	配線	パターン幅・間隔（最小値指定可）	接続形状	配線断面形状	ポリゴン領域（サイズ・ボリューム）	VIA配置（配置規則・ルール）	VIA配置（配置・コンセプト）	VIA形状	ボンディングワイヤ（配置）	ボンディングワイヤ（長さ・配置）	ボンディングワイヤ（長さ・配置）	端子属性（ボール・リード・サイズ）	モールド形状（配置・サイズ・サイズ）	実装属性（ボール・リード・ルール）	VIA属性（ドリル・ルール）	ボンディングワイヤ属性（長さ）	ボール属性（高さ・配置）	モールド属性	接続部の配置形状	接続部の配置形状（位置・高さ等）	接続部の配置形状（位置・高さ等）	コンスラッド属性（可否・ハーフ・穴可否）	インターフェース属性（配置数・寸法）	実装配置（ベア・チップ・VIA数）	信号伝送属性（シールド有無・特性インピーダンス）	配線禁止領域（シールド有無・配線禁止など）	端子（接続）する外部端子（LGA）情報	周囲温度	
1	独自フォーマット	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	9.0
	フォーマット Y	○	○	×	○	-	-	×	○	○	-	-	×	×	○	○	-	-	-	-	-	-	-	-	×	×	×	×	×	×	-	0
	フォーマット Z	○	○	○	○	-	-	○	○	○	-	-	○	○	○	○	-	-	-	-	-	-	-								-	15.0
	フォーマット AA	×	×	×	×	-	-	×	×	×	-	-	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	0.0
	フォーマット AB	×	×	×	×	-	-	×	×	×	-	-	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	0.0
	フォーマット AC	×	×	×	×	-	-	×	×	×	-	-	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	0.0
	フォーマット AD	○	×	○	○	-	-	×	○	○	-	-	×	×	×	×	×	×	-	-	-	-	-	-	○	×	×	×	○	-	-	-	0
	フォーマット AE	○	○	×	○	-	-	×	○	○	-	-	×	×	×	○	○	-	-	-	-	-	-	-	×	×	×	×	×	-	-	-	0
	フォーマット AF	×	×	×	×	-	-	×	×	×	-	-	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	0.0
	フォーマット AG	×	×	×	×	-	-	×	×	×	-	-	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	0.0
	フォーマット AH	○	×	○	○	-	-	×	○	○	-	-	×	×	×	×	×	×	-	-	-	-	-	-	-	×	×	×	×	-	-	-	0
	フォーマット AI	○	×	×	×	-	-	×	×	×	-	-	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	1.0
	フォーマット AJ	×	×	×	×	-	-	×	×	×	-	-	×	×	×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	0.0
	フォーマット AK	○	×	○	×	-	-	×	○	×	-	-	×		×	×	×	×	-	-	-	-	-	-	×	×	×	×	×	×	-	-	5

6

形状データフォーマット一覧と機能評価

「形状データ」フォーマット一覧　　Rev.1.0

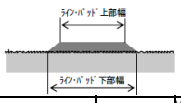
目的 ： PKG、PWBの電気的特性などの解析のための形状データを定義する。解析ツールへの共通・中間ファイルとして用いる。

No.	フォーマット名称	必要機能																										スコア
		機能ブロック名					層構成					層内図形									層外形状・搭載部品								
		機能ブロック名	機能ブロック属性	階層情報（ネット上位階層）	階層情報分岐	外形情報	層名	層厚さ	層順	層構成（導体/誘電体）	材質名	端子（形状・座標・コン等）	領域図形（ベタ形状・穴）	PG層図形（ベタ形状・穴・座標・線）	VIA形状（ランド・ドリルホール）	貫通ホール	PG層ボンゴン	直線・矢印図形（ライン）	PG層ネガ図形（ライン）	ボンディングワイヤ層情報	ボンディングワイヤ形状（高さ方向）	ワイヤボンド材質情報（材料名など）	端子・層情報	部品（DIE）層情報	部品図形名	部品搭載情報	部品図形情報（図形・ミラー・層）	部品端子（位置・形状・コン等・層）	部品搭載情報（サイズ・厚さ）	
1	XFL（アバッチ）	○	－	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	9.5
	フォーマット AL	×	×	×	×	×	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	5.0
	フォーマット AM	○	－	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	×	×	×	×	×	○	○	○	○	4.0
	フォーマット AN	○	－	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	1.0
	フォーマット AO	○	－	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	1.0
	フォーマット AP	○	－	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	1.0
	フォーマット AQ	○	－	○	○	×	○	×	○	○	×	○	○	×	○	○	○	×	×	×	×	×	×	×	×	×	×	×	11.0
	フォーマット AR	×	－	○	○	×	×	×	×	×	×	○	×	×	×	×	×	×	×	×	×	×	×	○	○	○	○	○	○	.0
	フォーマット AS	×	－	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	0.0
	フォーマット AT	×	－	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	0.0
	フォーマット AU	○	－	○	○	×	○	○	○	○		○	○	×	○	○	○	×	×	×	×	×	×	×	×	×	×	×	12.0
	フォーマット AV	○	－	○	○	○	○	×	×	×	×	×	○	×	×	○	×	×	×	×	×	×	×	×	×	×	×	×	.0
	フォーマット AW	×	－	×	×	○	○	×	○	×	×	×	○	○	×	×	○	○	×	×	×	×	×	×	×	×	×	×	×	.0
	フォーマット AX	×	－	×	○	×	×	×	×	×	×	○	×	×	×	×	○	×	×	×	×	×	○	×	○	○	×	○	×	.0

7

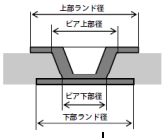
用語集

用語集　　Rev.1.0

No.	分類	用語・名称	定義・意味	キーワード	LSI例	PKG例	PWB例	関連語
1	ネットリスト	ポート（名）	外部とのインターフェースとなる入出力端子 同一階層では全てユニークな名前にする		ート	ピン名 ボール BGA リード	ピン名	
2	ネットリスト	リファレンス（名）	機能ブロックの定義名 階層にかかわらず全てユニークな名前にする		リファレンス セル名			
3	ネットリスト	インスタンス（名）	機能ブロックを配置する際に付ける名前 同一階層では全てユニークな　前にする		インスタンス	RefDes	リファレンス	
4	ネットリスト	ネット（名）	子間の接続につけるラベル 基本的には同一階層内での定義だが、階層に関係なく接続を表す場合もある（グローバルネット・定義）		ネット	ネット	ネット	
5	ネットリスト	バス	複数の信号を束ねた配線のこと 号名＋数字で表す（数字の省略で全信号を表す場合もある）		バス	バス	バス	
6	形状データ	DIE	SoCやメモリなどシリコンチップのこと		-	部品 コンポーネント LSI チップ	部品 コンポーネント	
7	形状データ	PKG	ッケージのこと					
8	形状データ	PWB	プリント基板のこと （PCBとも呼ばれるが有害物質「ポリ塩化ビフェニル」と区別するため避けられる傾向がある）					
9	形状データ	ボンドフィンガ	ボンディングワイヤの基板側の着地点の図形		-	リード 2ndパッド ワイヤボンドパッド ボンディングパッド	-	
10	解析条件	表面粗さ	導体と絶縁体の密着性を上げるために 導体表面を化学処理により粗した状態				アンカースパイク	
11	解析条件	配線断面構造	サブトラクティブ工法（銅箔の不要部を削除する工法）では、配線の上部と下部では導体の幅が異なる。 このような配線の構造を表す。					
12	形状データ	VIA	層間の導通接続を行う穴。PWBでは、部品搭載するための穴とは区別される。（部品搭載するための穴はスルーホール）		VIA ンタクト	VIA スルーホール レーザービア	（貫通）スルーホール VIA	
13	形状データ	ランド	部品の取り付けおよび接続に用いる導体パターン。 表面実装用のパッドおよび部品取り付け穴、ビアを囲む導体パターンなどがある。		-	ランド パッド	ランド	

8

用語集

No.	分類	用語・名称	定義・意味	キーワード	LSI例	PKG例	PWB例	関連語
14	形状データ	ドリルホール	VIAに関するドリルの径				ドリル径、キリ径	
15	解析条件	VIA3D構造	VIAに関する3次元的な各種形状の寸法値および、VIA内部への充填材料					
16	形状データ	フリップチップパッド	IEをフェイスダウン（回路面を基板側へ向ける）で搭載する際の基板側の接続点。 ※ワイヤタイプのボンドフィンガに相当。			FCパッド		
17	形状データ	認識マーク	IEをマウントするための手がかりとなるパターンマーク。			フィデューシャルマーク		
18	形状データ	ソルダーレジスト	プリント基板の半田付けが不要な部分に形成する保護膜のこと。			レジスト		
19	形状データ	スタブ	解メッキのために必要な、回路上不要な配線パターン。			メッキ引き出し線 プレーティングバー プレーティングスタブ		
20	形状データ	基板	担当分野によって意味合いが大きく異なる。 o.8に類似項目があるが、PWBというとプリント基板限定のイメージがある。		シリコン基板	インターポーザ パッケージ基板	マザーボード 実装ボード プリント基板	
21	ネットリスト	ラッツネスト	同一ネット間を結んだ直線。 子間を接続する配線設計を行う際の道標。		フライライン	ラッツ		
22	形状データ	リング	IEの周辺を囲んだ配線パターン。 源やグランドで使われることが多い。			グランドリング 電源リング		
23	形状データ	ボール	BGAタイプパッケージでの外部端子。			半田ボール ピン		
24	形状データ	サーマルボール	熱を逃がす為にDIE直下に配置されたボール					
25	形状データ	ゲートパターン	ッケージで封止する際に、樹脂を流す通り道			ゲート		
26	形状データ	サーマルVIA	熱を逃がす為にDIE直下に配置されたVIA		—	サーマルVIA	サーマルVIA	
27	形状データ	サーマルリリーフ	ハンダ付け用に熱を逃がさないための形状				サーマルVIA	

9

用語集

No.	分類	用語・名称	定義・意味	キーワード	LSI例	PKG例	PWB例	関連語
28	形状データ	プレーン	ッケージやボードで主に電源/グランドなどに用いる導体面のこと		—	プレーン ベタ（パタン）	プレーン 面、ベタ（パタン）、メタル	
29	形状データ	ボイド	プレーンに孔をあけるための図形形状のこと		—	抜きパタン 窓 ガス抜き穴	抜きパタン 窓 ガス抜き穴	
30	形状データ	残銅率	レイヤーごとの導体の占有面積比のこと		メタルカバレッジ メタルデンシティ	残銅率	残銅率	

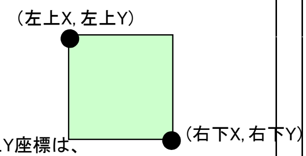
10

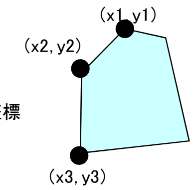
階層間定義フォーマット Ver.1.0

2011.01.24	リコー/吉田 初版
2011.02.04	東芝/青木 修正
2011.02.14	リコー/吉田 修正
2011.02.21	リコー/吉田 修正
2011.02.22	東芝/青木 修正

座標、表裏はすべてTOP Viewにて定義する

	コメント
# Data:2011.01.24	
<module name="機能ブロック名" type="機能ブロック属性">	階層間定義 開始 機能ブロック名 : 上位階層からの参照名。リファレンス名 機能ブロック属性 : LSI, PKG , PWB など
<unit> <distance unit="長さ単位" scale="縮尺値" /> <area unit="面積単位" scale="縮尺値" /> <time unit=時間単位" scale="縮尺値" /> ….	単位系定義 開始 : この定義は、module全体に有効
</unit>	単位系定義 終了
<shape>	物理形状定義 開始
<rectangle distunit="単位" scale="縮尺率" llx="左下X座標" lly="左下Y座標" urx="右上X座標" urx="右上Y座標"/>	外観 矩形定義(TOP View) 単位 : um , mm , mil など 縮尺率 : 0より大きい実数値 左下X座標、左下Y座標、右上X座標、右上Y座標は、 外観を表現するためのTop Viewからの相対座標
<polygon distunit="単位" scale="縮尺率" points="x座標,y座標 x座標,y座標…"/> ・・・・・	外観 ポリゴン定義 単位 : um , mm , mil など 縮尺率 : 0より大きい実数値 x座標,y座標 x座標,y座標…は、 外観を表すためのTOP Viewからの相対座標 座標記述の順に線で結んだ外観となる
	物理形状定義 終了
</shape>	
<socket inst="インスタンス名">	接続口定義 開始 インスタンス名 :文字列
<port name="端子名" id="端子番号" direction="端子方向" type="信号タイプ">	端子定義 開始 端子名 : 上位階層から入出力端子を参照する際の参照名 端子番号: 上位階層から参照する端子番号。1以上の整数値 端子方向: 上位階層から見た場合の信号の方向。In , out , inout 信号タイプ: power, ground , signalなど
<shape> <point distunit="単位" scale="縮尺率" x="x座標" y="y座標" layer="レイヤ" />	端子形状定義 開始(任意) 端子座標定義 単位 : um , mm , mil など 縮尺率 : 0より大きい実数値 x座標、y座標:参照先ファイルの端子のX座標、Y座標 レイヤ:参照先ファイルの端子の配置層
<rectangle distunit="単位" scale="縮尺率" llx="左下X座標" lly="左下Y座標" urx="右上X座標" ury="右上Y座標" layer="レイヤ" />	端子外観 矩形定義 左下X座標、左下Y座標、右上X座標、右上Y座標は、 外観を表現するための相対座標
<polygon distunit="単位" scale="縮尺率" points="x座標,y座標 x座標,y座標…" layer="レイヤ ・・・・・ ・・・・・ ・・・・・	端子外観 ポリゴン定義 単位 : um , mm , mil など 縮尺率 : 0より大きい実数値 x座標、y座標:外観を定義するための相対座標
</shape>	端子形状定義 終了
<electrical> <Zsingle impedance="特性インピーダンス" tpd="遅延"	端子の電気特性要求定義 開始 特性インピーダンス: 1端子への特性インピーダンス要求(差動の場合はpoertgroupへ) 遅延: 端子への信号遅延要求
</electrical>	端子の電気特性要求定義 終了
</port>	端子定義 終了
<portgroup name="グループ名" type="グループ属性" Zdiff="差動特性インピーダンス" skew="スキュー">	ポートグループ定義 開始 :バスグループ、差動信号、接続可能なポートグループを定義 端子グループ名 : 任意の英数字の文字列 グループ属性 : PGNET , BUS , DIFF, Zなど 差動特性インピーダンス:差動配線(DIFF)の要求インピーダンス定義 スキュー:差動配線(DIFF)、バス配線(BUS)の要求スキュー定義
<portref>端子名</portref> <portref>端子名</portref> <portref>端子名</portref> ・・・・	端子名、または端子番号にて定義
</portgroup>	ポートグループ定義 終了





```
   <specification>

      <power>消費電力</power>

   </specification>

<reference
         reffile="参照ファイル名"
         format="言語フォーマット">


  <unit>
    <distance="長さ単位系" scale="縮尺率" />
    <anglunit="角度単位系" scale="縮尺率" />

  </unit>

<placement module="機能ブロック名">

   <point x="x座標" y="y座標"
            flip="フリップタイプ"
          angle="回転角度"
           layer="層"
          mount="配置面"/>




   <rectangle llx="左下X座標" lly="左下Y座標"
            urx="右上X座標" ury="右上Y座標"
            layer="層"
          mount="配置面" />
...

</placement>

<connection inst="インスタンス名"
   portref="端子名" portid="端子番号">



<verilog:portref
   module="モジュール名" portname="ポート名"/>


<def:portref
  comp="コンポーネント名" pinname="ピン名"/>
<def:portref pin="ピン名"/>

<spice:portref
  subckt="サブサーキット名" portid="ポート番号"/>


<touchstone:portref portid="ポート番号"/>


<dxf:portref distunit="単位系" distscale="縮尺率"
   x="x座標" y="y座標" layer="レイヤ"  mount="配置面" />
<gds:portref distunit="単位系" distscale="縮尺率"
   x="x座標" y="y座標" layer="レイヤ"  mount="配置面" />



<xfl:portref module="機能ブロック名" name="端子名">



<ibs:portref   module="モデル名" portname="端子名"/>



<JLPB:portref
  module="機能ブロック名" portname="端子名"/>


<XXX:portref>
      (任意)
</XXX:portref>
</connection>

</reference>

</module>
```

動作仕様の開始

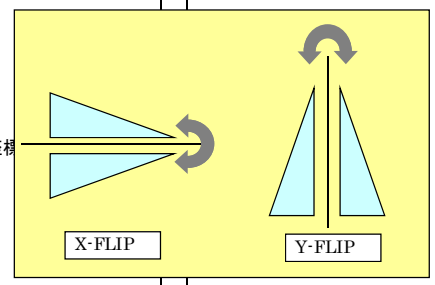
　消費電力:機能ブロックの消費電力

動作仕様の終了

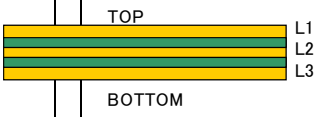
参照ファイル定義　開始
　参照ファイル名:機能ブロックが定義された参照先ファイル名
　言語フォーマット:機能ブロックの設計言語種類。
　　SPICE , XFL , VERILOGなど

単位系定義　開始
　長さ単位系　:　um mm cm mil 等
　角度単位系　:　degree radian 等
　縮尺率　:　0より大きい実数値
単位系定義　終了

配置制約定義　開始
機能ブロック名:参照先ファイルの機能ブロック名
　x座標、y座標:参照先ファイルの機能ブロック原点の配置座標
　フリップタイプ:X　…　X軸を中心ととしたフリップ(上下反転)
　　Y　…　Y軸を中心としたフリップ(左右反転)
　回転角度:参照先ファイルの機能ブロック原点の回転角度
　フリップ→回転の順番で実施される
　層　:　配置される層(L1, L2, L3 ...)
　配置面:TOP・・・対象レイヤのTOP面に機能ブロックを配置
　　BOTTOM・・・対象レイヤのBOTTOM面に機能ブロックを配置



　左下X座標、左下Y座標、右上X座標、右上Y座標は、
　物理形状を持たない機能ブロックの配置を表現するための絶対座標
　層　:　配置される層(L1, L2, L3 ...)
　配置面:TOP・・・対象レイヤのTOP面に機能ブロックを配置
　　BOTTOM・・・対象レイヤのBOTTOM面に機能ブロックを配置



配置制約定義　終了

端子接続開始
インスタンス名:　socket instにて定義する機能ブロック名
端子名:関連付けられる端子名
端子番号:関連付けられる端子番号

verilogの参照
　モジュール名:参照先ファイルのモジュール名
　ポート名:参照先ファイルのポート名

DEFの参照
コンポーネント名:参照先ファイルのコンポーネント
ピン名:参照先ファイルのピン名

SPICEの参照
サブサーキット名:参照先ファイルのサブサーキット名
ポート番号:参照先ファイルのポート番号

Touchestone形式S-parameterの参照
ポート番号;参照先ファイルのポート番号

DXF/gdsの参照
　単位　:　um , mm , mil など
　縮尺率　:　0より大きい実数値
　x座標、y座標:参照先ファイルの端子のX座標、Y座標
　レイヤ:参照先ファイルの端子の配置レイヤ
　配置面:TOP・・・対象レイヤのTOP面に機能ブロックを配置
　　BOTTOM・・・対象レイヤのBOTTOM面に機能ブロックを配置

XFLの参照
　機能ブロック名:参照先ファイルの機能ブロック名
　端子名:参照先ファイルの端子名

IBISの参照
　モデル名:参照先ファイルの機能ブロック名
　端子名:参照先ファイルの端子名

JEITA　LPB-WGの階層間フォーマットの参照
　機能ブロック名:参照先ファイルの機能ブロック名
　端子名:参照先ファイルの端子名

暗号化された接続方式、これをどう解釈するかはXXXを入力する側の
　ベンダと、XXXを提供する側のベンダとのライセンスで管理する

端子接続定義　終了

参照ファイル定義　開始

階層間定義　終了

「階層間定義」フォーマット一覧

Rev.1.0

		フォーマット名称	ラベル	有無
必要機能	機能ブロック定義	機能ブロック名	module/name	○
		機能ブロック属性	module/type	○
		座標原点（左下右上座標）	shape/rectangle/llx,lly	○
		座標単位系	disunit	○
		スケール（拡縮率）	scale	○
		入出力端子【ポート】名	port/name	○
		入出力端子属性（入力/出力）	port/direction	○
		入出力端子属性（信号/電源）	port/type	○
		入出力端子属性（束線）	portgroup/type	○
		入出力端子番号	port/id	○
		入出力端子座標	shape/point/x,y	○
		入出力端子層（レイヤー）	shape/point/layer	○
		端子間接続【ネット】名	ネットリストへ	
		電源マージ情報	portgroup/name, type	○
		機能参照（外部 Sパラ等）	reference/inst	→
		機能ブロック参照（ネスト）	必要なし	
	機能ブロック参照・接続定義	座標原点（左下右上座標）	reference/placement/point/x,y	○
		座標単位系	reference/disunit	○
		スケール（拡縮率）	reference/scale	○
		参照機能ブロック名	reference/placement/point/module	○
		配置【インスタンス】名	reference/inst	○
		言語フォーマット	reference/format	○
		配置座標（オフセット）	必要？	必要
		配置情報（回転・ミラー）	reference/placement/point/flip, angle	○
		配置層（レイヤー）	reference/placement/point/layer	○
		配置面	reference/placement/point/mount	○
		接続入出力端子名指定	connection/portref	○
		接続入出力端子番号指定	connection/portid	○
		端子間接続【ネット】名	ネットリストへ	
		端子間接続属性（信号/電源）	portgroup/type	←
		端子間接続属性（束線）	portgroup/type	←
		電源マージ情報	portgroup/type	←
		機能参照（外部 Sパラ等）	reference/inst	○
	電気特性（部品）	搭載部品種別名	module/name	ブロッ
		搭載部品端子情報（I/Oセルモデル）	portref/portname	○
		搭載部品モデル（電流源・インピーダンス）	portref/module	○
		搭載受動素子（Sパラメータ・ESR・ESL等）	portref/module	○
	熱特性（部品）	搭載部品種別名	module/name	ブロッ
		搭載部品情報（特性値）	portref/module	○
		搭載部品情報（形状・サイズ・厚さ）	reference/placement/point/x,y	○
		搭載部品情報（消費電力）	reference/placement/point/power	○
		周囲温度	thermal/ambient/temp	○
		メリット		
		デメリット		
		備考		

設計制約フォーマット Ver.1.0

2010.12.15 ルネサス/永野
2011.02.14 TNCSi/金子 追記
2011.02.21 ルネサス/永野 追記
2011.02.22 東芝/青木 XML化

```
# Data:2011.02.07                                                コメント
<Header                                                          ヘッダー定義 開始
   TechDefName="Tech_xyz"                                        制約定義の名称:文字列
   Version="1.1"                                                 仕様のバージョン:文字列
   Type="PKG";                                                   定義タイプ:LSI、PKG、PWB、など
  DesignUnits="mm"                                               長さ単位:mm、um、など
  AreaUnits="mm2"                                                面積単位:mm2、um2、など
  AngleUnits="degree"                                            角度単位:degree、radian、など
  TimeUnits="ns";                                                時間単位:us、ns、など
  ResistivityUnits="ohm*m"                                       体積抵抗率単位:Ω・m(ohm*m)、μΩ・cmなど
  TempUnits="degree Celsius"                                     温度単位:℃(degree Celsius)、ケルビン(kelvn)など
  …                                                               ・・・
/>                                                               ヘッダー定義 終了

<LayerDef>                                                       レイヤー定義 開始
 <Layer name="L1" order="1" thickness="0.015" material="Copper"/>  層名、層番、導体厚、材料名                    層厚定義から導体厚定義に変更、材料名を追加
 <Layer name="L2" order="2" thickness=" 0.03" material="Copper"/>  層名、層番、導体厚、材料名
 <Layer name="L3" order="3" thickness="0.03" material="Copper"/>   層名、層番、導体厚、材料名
 <Layer name="L4" order="4" thickness="0.015" material="Copper"/>  層名、層番、導体厚、材料名
 <Dielectric above="L1" below="L2" thickness="0.03" material="FR4"/>  層名1、層名2、層名1-2間の層間厚、材料名    層厚さの定義追記、材料名を追加
 <Dielectric above="L2" below="L3" thickness="0.3" material="FR4"/>   層名1、層名2、層名1-2間の層間厚、材料名
 <Dielectric above="L3" below="L4" thickness="0.03" material="FR4"/>  層名1、層名2、層名1-2間の層間厚、材料名
 <TopSurfaceDielectric thickness="0.03" material="ResistInk"/>    最上/下面(TOP/BOTTOM)、層間厚、材料名        表面の誘電体定義を追加
 <BottomSurfaceDielectric thickness="0.03" material="ResistInk" />  最上/下面(TOP/BOTTOM)、層間厚、材料名
  …                                                               ・・・
</LayerDef>                                                      レイヤー定義 終了

<LineDef>                                                        配線幅定義 開始
 <Width layer="L1" min="0.035" max="0.1" />                        層名、最小配線幅、最大配線幅
  …                                                               ・・・
</LineDef>                                                       配線幅定義 終了

<PolygonDef>                                                     ポリゴン最小定義 開始
 <Area layer="L1" min="0.0001" />                                  層名、最小寸法(面積)
  …                                                               ・・・
</PolygonDef>                                                    ポリゴン最小定義 終了

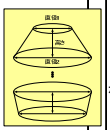
<ViaDef>                                                         VIA定義 開始
 <Via name="VIA12">                                                個別VIA定義 名称
  <Circle layer="L1" diameter="0.1" />                               形状、レイヤー名、外径、内径              サイズを外径と内径に修正
  <Circle layer="L2" diameter="0.15" />                              形状、レイヤー名、外径、内径
  <DrillHole material="Copper" />                                    VIAドリルホール物性値                     VIA物性値
 </Via>
 <Via name="VIA23">                                                個別VIA定義 名称
  <Rectangle layer="L2" width="0.15" height="0.15" />                形状、レイヤー名、外径、内径
  <Rectangle layer="L3" width="0.15" heignt="0.15" />                形状、レイヤー名、外径、内径
  <DrillHole material="Copper" />                                    VIAドリルホール物性値
 </Via>
 <Via name="BF1">                                                  個別VIA定義 名称
  <Rectangle layer="L1" width="0.15" height="0.1" />                 形状、レイヤー名、外径、内径
  <DrillHole material="Copper" />                                    VIAドリルホール物性値
 </Via>
  …                                                               ・・・
</ViaDef>                                                        VIA定義 終了

<SpacingDef>                                                     スペース定義 開始
 <LineToLine layer="L1" space="0.07"/>                             対象、Layer、スペース
 <LineToVia layer="L1" via="VIA12" space="0.07" />                 対象、Layer、VIA、スペース
 <LineToPolygon layer="L1" space="0.05"/>                          対象、Layer、スペース
  …                                                               ・・・
</SpacingDef>                                                    スペース定義 終了

<PitchDef>                                                       ピッチ定義 開始
  <ViaPitch via1="VIA12" via2="VIA12" pitch="0.1" />               VIAピッチ VIA1、VIA2、ピッチ
  …                                                               ・・・
</PitchDef>                                                      ピッチ定義 終了

<BondingWireDef>                                                 ボンディングワイヤ定義 開始
 <BondingWire name="WB1" Material="Au"                             個別ボンディングワイヤ定義 名称 材料名
   <Forward HoL="0.0" VeL="150.0" />                                 順方向、水平値、垂直値
   <Forward HoR="0.125" VeL="0.0" />                                 順方向、水平比、垂直値
   <Length min="3.0" max="5.0" />                                    最小長さ、最大長さ
 </BondingWire>
 <BondingWire "WB2" material="Au" >                                個別ボンディングワイヤ定義 名称 材料名
   <Forward VeL="150.0" Ang="85.0" />                                順方向、垂直値、角度
   <Forward HoR="0.125" VeL="0.0" />                                 順方向、水平比、垂直値
   <Backward HoR="0.5" Ang="20.0" />                                 逆方向、水平比、角度
   <Length min="3.0" max="5.0" />                                    最小長さ、最大長さ                        ボンディングワイヤ物性値
 </BongingWire>
  …                                                               ・・・
</BondingWireDef>                                                ボンディングワイヤ定義 終了

<BallDef>                                                        ボール定義 開始
  <Ball name="BGA" material="SnZu">                                個別ボール定義 名称 材料名
    <Frustum height="0.3" diam1="0.2" diam2="0.4" />                 高さ、直径1、直径2
  …                                                                ・・・
    <Frustum height="0.2" diam1="0.4" diam2="0.3" />                 高さ、直径1、直径2
  </Ball>                                                                                                    ボール物性値
  <Ball name="LGA" material="SnZu">                                個別ボール定義 名称 材料名
    <Frustum height="0.1" diam1="0.2= diam2="0.2" />                 高さ、直径1、直径2
  </Ball>                                                                                                    ボール物性値
  …                                                               ・・・
</BallDef>                                                       ボール定義 終了
```



```
<MoldDef>
  <Mold  width="40" depth="40" height="3" material="Resin"/>
</MoldDef>

<ConductorStruct>
  <TrapezoidalAngle layer="L1" angle="92" />
  …
  …
  <UpSurfaceRoughness layer="L1" RMS="0.000" />
  <DownSurfaceRoughness layer="L1" RMS="0.000" />
  …
  …
</ConductorStruct>

#-----------------------------------------

<ComponentDef>
  <Comp name="AAA" width="8.0" depth="4.0" height="4.0"
        power="1.5"
        property="ddr3-1066"
   />
  <Comp name="BBB" width="0.5" depth="0.5" height="0.5"/>
  …
</ComponentDef>

<HeightLimit>
  <Top  x1="-10.0" y1="-10.0" x2="10.0" y2="10.0" height="10.0"/>
  <Top  x1="5.0"   y1="-5.0" x2="10.0" y2="0.0" height="0.2" />
  <Bottom   />
  …
</HeightLimit>

<SwappablePin>
  <Group name="PSG01" part="AAA">
    <Pin name="A1" />
    <Pin name="A3" />
  </Group>
</SwappablePin>

<FixedPin>
  <Pin part="AAA" name="B1"/>
  <Pin part="BBB" name="1"/>
  …
</FixedPin>

<InterfaceStandardSetting>
  <Pin part="AAA" name="A1" standard="ddr3" />
  <Net name="DQ1" standard="xxx" voltage="1.5" waveform="666MHzRect" />
  …
</InterfaceStandardSetting>

<DifferentialPairConstraint>
    <DiffPair name="Diff01
          LengthTolerance="0.005
          MaxViaCount=2>
      <Pin part="AAA" name="A1" />
      <Pin part="AAA" name="A3" />
  </DiffPair>
  …
</DifferentialPairConstraint>

<ElectricalConstraint>
  <PinDelay part1="AAA" pin1="A1" part2="BBB" pin2="A1" delay="100" />
  <SingleImpedance  net="single_net" impedance="50" />
  <DiffImpedance net1="diff_net1_p" net2="diff_net1_n" impedance="100" />
  …
</ElectricalConstraint>

<WiringConstraint>
  <Net name="DQ1"
      MinWidth="0.15"
      MinGap="0.15"
      GuardShield="VSS"  />
  …
</WiringConstraint>

<SpecSetting>
    <Spec name="ddr3">
      <SignalLevel   Vihac="1.2"
                     Vilac="0.3"
                     Vihdc="1.1"
                     Vildc="0.35"
                     Vover="1.7"
                     Vunder="-0.3"   />
    <Skew clock="CLK" signal="DQS" delay="0.02" />
      <Jitter  up="0.2"  down="0.2" />
    </Spec>
  …
</SpecSetting>

<MaterialSetting>
  <Conductor  material="Copper" VolumeResistivity="1.68e-8" temperature="20" />
  <Conductor  material="Au" VolumeResistivity="2.21e-8" temperature="20" />
  <Insulator material="FR4" Permittivity="4.2" TanDelta="0.016" frequency="1G" />
  …
</MaterialSetting>

<TemperatureDef>
      <Ambient temperature="25.0" />
</TemperatureDef>
```

```
モールド定義  開始
    モールド形状W、D、H、材料名
モールド定義  終了

導体構造設定  開始
  導体厚断面設定：層名、側面角度α[degree]
    0>α>-180：下辺CAD寸法
    0<α<180：上辺CAD寸法
  表面粗さ設定：層名、上面(UP)下面(DOWN)、
                粗さ、2乗平均粗さ(RMS)
  …
  …
導体構造設定  終了

#-----------------------------------------

搭載部品定義  開始
  部品名称、W、D、H
    電源電圧  V
    属性  文字列

  部品名称、W、D、H
  …
搭載部品定義  終了

高さ制限定義  開始
  制限面、X1、Y1、X2、Y2、高さ
  制限面、X1、Y1、X2、Y2、高さ
  制限面
  …
搭載部品定義  終了

ピンスワップ定義  開始
  スワップ可能ピングループ定義、グループ名、部品
      ピン名
      ピン名
  スワップ可能ピングループ終了
ピンスワップ定義  終了

スワップ禁止ピン定義開始
  スワップ禁止、部品、ピン
  スワップ禁止、部品、ピン
  …
スワップ禁止ピン定義終了

インターフェース規格設定  開始
  ピン、部品、端子、規格
  ネット、ネット名、規格、動作電圧、波形名
  …
インターフェース規格設定  終了

差動配線定義  開始
  個別差動配線定義  名称
    配線差許容値、長さ
    最大VIA数、数
    部品、ピン
    部品、ピン

  …
差動配線定義  終了

信号遅延制約  開始
  ディレイ：部品1、端子1、部品2、端子2、ディレイ値
  特性インピーダンス(単線)：ネット、値
  特性インピーダンス(差動)：ネット1、ネット2、値
  …
信号遅延制約  終了

配線制約設定  開始
  ネット名
    最小配線幅
    最小ギャップ
    シールド配線ネット
  …
配線制約設定  終了

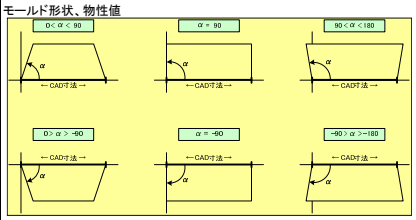
規定値、判定値設定  開始
  規格名
  Vihac規定電圧値
  Vilac規定電圧値
  Vihdc規定電圧値
  Vildc規定電圧値
  オーバーシュート規定電圧値
  アンダーシュート規定電圧値
  スキュー基準信号、対象信号、規定遅延時間
  ジッタ立上り規定時間、ジッタ立下り規定時間

  …
規定値、判定値設定  終了

材料設定  開始
  導体名、体積抵抗率、抵抗値想定温度
  導体名、体積抵抗率、抵抗値想定温度
  絶縁体名、誘電率、誘電正接、想定周波数
  …
材料設定  終了

温度定義  開始
    周囲温度
温度定義  終了
```

モールド形状、物性値



電源電圧を追加
属性項目を追加

Pin to Pinに変更

配線制約（＋シールド）追加

インターフェース規格

解析条件 物性値

周囲温度

解析条件 電気特性 搭載部品動作条件

Apache Design Solutions

PCB/Package Data Exchange File (XFL) Format

1. Reserved Characters

#	any data after # sign in each line is ignored so the user may use it for comments.
" "	any data between two double quotation marks are considered as a string.

2. Structure

The XFL consists of several sections and each section start with a dot (.) followed by a key word. **.version**, **.unit**, and **.scale** sections consist of one line each. All other sections end width **.end** followed by the same key word. The key word in the **.end** line may be omitted. Except the .version, any section may be omitted.

.version x y

.unit [**inch** | **mm**]

.scale value

.layer
definition
.end [**layer**]

.shape
definition
.end [**shape**]

.board_geom
definition
.end [**board_geom**]

.padstack
definition
.end [**padstack**]

.part
definition
.end [**part**]

.component
definition
.end [**component**]

.netattr
definition
.end [**netattr**]

.netlist
definition
.end [**netlist**]

.via
definition
.end [**via**]

.bondwire
definition
.end [**bondwire**]

.route
definition
.end [**route**]

3. Description

. version x y

A version of the XFL file. x is a major version number, y is a minor version number. **.version** must appear before any other section appears.

x integer
y integer

.unit value

A geometric unit used throughout the XFL file. The value can be either **inch** or **mm**. If this section or the value is omitted, **inch** is assumed as a default.

value string (**inch** or **mm**)

.scale value

A geometric scale used throughout the XFL file. Actual dimension of the data in the file is determined by dividing the number by the scale value. For example, if the unit is set to **inch** and the scale is 1000, the geometric data represents mil or 1/1000 of inch. If the unit is set to mm and the scale is 1000, it is micron or micrometer. Default is 1.

value double

.layer
definition
.end [layer]

Layer stack up. Describes a layer stack up from top to bottom (or from front to back). Each layer is defined as:

name thickness type conducting dielectric1 [dielectric2 dielectric3]

where

name	Name of the layer enclosed by double quotation marks. If unknown, write "".
thickness	Thickness of the layer. If unknown, write 0.
type	Type of the layer. A single character defines the layer type. **S** for signal layer **D** for dielectric layer **P** for power or ground layer if it can be differentiated from the signal layer
conducting	Conducting material name if entered as a string enclosed by double quotation marks. Conductivity if entered as a number. Unit is 1/ohm.mm
dielectric1	Dielectric material name if entered as a string enclosed by double quotation marks. If entered as a number, this field is permittivity and the next two fields dielectric2 and dielectric3 are permeability and loss tangent, respectively.

Note: *Throughout the XFL, signal layer numbers are the numbers that are numbered sequentially from the top by counting only the signal/power/ground layers.*

.shape
definition
.end [shape]

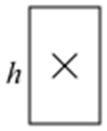
Shape definition. This section defines shapes that can be referenced from other sections of the file. Available shapes are polygon, rectangle, square, diamond, circle, annular, oblong, finger, bullet, and composite.
Each shape is defined as :

id keyword parameters

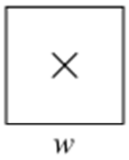
where the id is a number that will be referenced by others and it is sequentially numbered from 1. The keyword and parameters are described below.



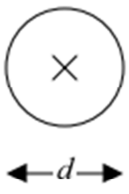
id **polygon** {*x1 y1 x2 y2 ...*}
Vertices are enclosed by { } and data can be shown in more than one lines. The last point does not need to be the same as the first point. Reference point is at (0,0).



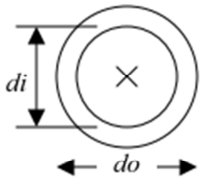
id **rectangle** *width height*
Each rectangle is defined by width and length and the definition must appear in one line. Reference point is at the center of the rectangle.



id **square** *width*
Each square is defined by width and the definition must appear in one line. Reference point is at the center of the square.

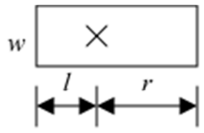


id **circle** *diameter*
Each circle is defined by radius and the definition must appear in one line. Reference point is at the center of the circle.



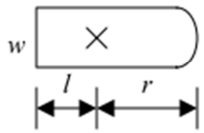
id **annular** *outer diameter inner diameter*

Each donut is defined by outer diameter and inner diameter and the definition must appear in one line. Reference point is at the center of the annular.



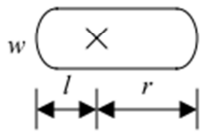
id **oblong** width left right

Each oblong is defined by width, left, and right. The definition must appear in one line. Reference point is shown at the left marked as X.



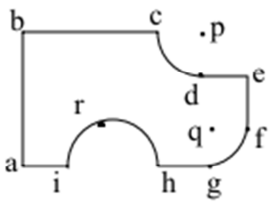
id **bullet** width left right

Each bullet is defined by width, left, and right. The definition must appear in one line. Reference point is shown at the left marked as X.



id **finger** width left right

Each finger is defined by width, left, and right. The definition must appear in one line. Reference point is shown at the left marked as X.



id **composite** {*1st point segment1 segment2 ...* }

Composite shape consists of lines and/or arcs. The definition is enclosed by { } and it can be shown in more than one lines. There are four types of segments – straight line, clockwise arc (**arc**), counter-clockwise arc (**rarc** or reverse arc) and arc defined by three points (**arc3**).

No keyword between two points indicates that they are connected by a straight line. A keyword **arc, rarc** or **arc3** between two points indicates that they are connected by a clockwise arc, a counter-clockwise arc or a three-point method arc, respectively. For **arc** and **rarc**, an arc origin appears after the arc ending point. For **arc3**, a middle point appears after the arc ending point.

The composite shape at the left can be described as:

2 **composite** {$x_a\ y_a\ x_b\ y_b\ x_c\ y_c$ **rarc** $x_d\ y_d\ x_p\ y_p\ x_e\ y_e\ x_f\ y_f$ **arc** $x_g\ y_g\ x_q\ y_q\ x_h\ y_h$ **arc3** $x_i\ y_i\ x_r\ y_r$}

where the number 2 at the beginning is a shape id. Straight lines a-b and b-c, then a counter-clockwise arc from c to d with an origin at p. Straight lines d-e and e-f, then a clockwise arc from f to g with an origin at q. A straight line g-h and a three-point method arc from h to i with a middle point at r. Finally straight line from i to a. i to a is not defined but a straight line will connect the last point to the first point by default.

The same shape can also be described as:

2 **composite** {$x_a\ y_a\ x_i\ y_i$ **arc3** $x_h\ y_h\ x_r\ y_r\ x_g\ y_g$ **rarc** $x_f\ y_f\ x_q\ y_q\ x_e\ y_e\ x_d\ y_d$ **arc** $x_c\ y_c\ x_p\ y_p\ x_b\ y_b$}

.board_geom
definition
.end [**board_geom**]

Exterior geometry of board or package is defined by one of the four methods shown below.

polygon {*x1 y1 x2 y2 ...*}
composite {*1st point segment1 segment2 ...* }
shape *shapeID x y rotation mirror*
shape *shapeID x y mirror rotation*

The method of defining **polygon** and **composite** are the same as described in the shape section.
The **shape** is placed by defining shape ID, x and y coordinates (global coordinates) of the shape origin (local origin), counter-clockwise rotation in degrees, and mirror indicator. Mirror indicators are:

X	mirror about X-axis
Y	mirror about Y-axis
N	no mirror

If the mirror indicator appears after the rotation, the mirror is performed after the rotation. If the mirror indicator appears before the rotation, the mirror is performed before the rotation.
Both mirror and rotation operations are performed with respect to the shape origin (local origin).

Holes (cutouts or voids) are defined by using one or more of the following methods.

void_polygon {*x1 y1 x2 y2 ...*}
void_rectangle *width length x y*
void_square *width x y*
void_diamond *width x y*
void_circle *diameter x y*
void_composite {*1st point segment1 segment2 ...* }
void_shape *shapeID x y rotation mirror*
void_shape *shapeID x y mirror rotation*

void_polygon, **void_composite**, and **void_shape** are defined the same way as **polygon**, **composite**, and **shape**, respectively.
void_rectangle, **void_square**, **void_diamond**, and **void_circle** are defined the same way as **rectangle**, **square**, **diamond**, and **circle** are defined in the shape section. They are placed at the coordinates x and y.

.padstack
definition
.end [**padstack**]

This section defines pad stacks and each pad stack is defined as:

pad stack id {pad1 pad2 ….}

where the pad stack id is a number that will be referenced by vias in the via section and it is sequentially numbered from 1. As described below, Pads are defined only by using the predefined shapes in the shape section.

signal layer # shape id shape rotation apshape id ap shape rot

where

signal layer # Signal layer number as appeared in the layer section.

shape id Shape ID.

shape rotation Counter-clockwise rotation angle of the shape in degrees.

ap shape id Anti-pad shape ID. Anti-pad definition is optional.

ap shape rot Counter-clockwise rotation angle of the anti-shape in degrees.

.part
definition
.end [**part**]

Describes a part. Each part is defined as:

part name [shape llx lly urx ury height [type [**noflip**] [material]]]
{pin1 pin2 ….}

where part name is a name of the part. shape, llx, lly, urx, ury, height and **noflip** are optional. If the keyword **noflip** appears, part will not be flipped when it is place at the back side.

shape Top view shape of the part.
R for rectangle
C for circle
D for round-corner diamond

llx, lly Lower-left coordinates of bounding box

urx, ury	Upper-right coordinates of bounding box
height	Height
type	Part type **R** for resistor **L** for inductor **C** for capacitor **S** for solder balls **D** for die **M** for molding compound **O** for other types
material	Optional. Material name enclosed by " ".

Pins are defined as:

pin name x y i/o type [pad_stack_id]

where

pin name	Name of the pin. If the name is not known, a sequential number will be used as a name.
x y	Location of the pin with respect to local origin
i/o type	Pin I/O type. **D** for driver pin **R** for receiver pin **B** for bi-directional pin **DT** for driver terminator **RT** for receiver terminator
pad_stack_id	Optional. Pad stack ID (0 if pad stack is unknown)

.component
definition
.end [**component**]

Describes a component placement. One component placement appears per line and each placed component is defined as:

name part name x y layer rotation [above/below]

where

U-name	U-name. Also known as location identifier or reference designator. No space is allowed in the name.

part name	Part name. The name is enclosed by double quotation marks (“ ”).
x y	Location of the component origin with respect to the board origin.
layer	Placement layer number. ***+n*** for above the layer ***-n*** for below the layer
rotation	Counter-clockwise rotation of the component in degrees with respect to the component origin (local origin).
above/below	Optional. This field is ignored if the layer is **F** or **B**. Placed above or below the layer. **above** for placed above the layer **below** for placed below the layer

.netattr
definition
.end [netattr]

Net attribute definition. This section defines net attributes such as net class, allowable delays, and other user properties.
Each attribute group is defined as:

id {attribute name1=value1 attribute name2=value2 ….}

where the id is an attribute group number that will be referenced by nets in the netlist section and it is sequentially numbered from 1. There may be more than one attribute defined for one group. Each attribute group definition may appear in more than one line. Attribute assignments appear in the braces { }. Each attribute assignment consists of attribute name, equal(=) sign, and the value.

.netlist
definition
.end [netlist]

This section defines net lists and each net is defined as:

net name net ype attribute id {node1 node2 ….}

where

net name — Net name.

net type — Net type.
S for signal net (**s** if broken)
P for power net (**p** if broken)
G for ground net (**g** if broken)

attribute id — Net attribute ID. Zero indicates that the attribute is unknown.

Each node is defined as:

U-name pin number i/o type [{x y layer}]

where

U-name — U-name. "-" (minus sign) indicates unknown U-name.

pin number — Pin number or pin name.

i/o type — Pin I/O type. Available types are **D**, **R**, **B**, **DT**, and **RT** as described in the part section. A place holder "**-**" (minus sign) must appears if the type defined for the pin in the part section is to be used.

Pin location may appear at the end of the node definition. It is optional and must be enclosed by { }. Set the layer number to negative for solder balls.

.via
definition
.end [via]

This section defines vias. One via definition appears per line and each via is defined as:

via name pad stack id pad stack rot shape id shape rot [thickness]

where

via name — Via name that will be referenced by routing in the route section.

pad stack id — Pad stack ID. Zero indicates that no pad stack exists for the via.

pad stack rot	Counter-clockwise rotation angle of the pad stack
shape id	Shape ID of the via barrel.
shape rot	Counter-clockwise rotation angle of the via barrel shape.
thickness	Optional. Via barrel wall thickness.





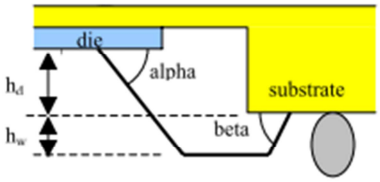


.bondwire
definition
.end [bondwire]

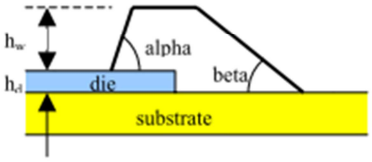
This section defines bond wire geometry. One bond wire definition appears per line and each bond wire is defined as:

id type material diameter h_w h_d alpha beta

where the id is a number that will be referenced by routing in the route section and it is sequentially numbered from 1. Others are explained below.



Die-down Configuration



Die-up Configuration

type	Bond wire type. **D** for die-down configuration **U** for die-up configuration
material	Material name or electric conductivity (1/ohm.mm). If it is a name, it must be enclosed by double quotation marks (“ ”).
diameter	Wire diameter.
h_w	Wire loop height.

h_d	Die height. $H_{die\_pad} - H_{top\_of\_top\_metal\_layer}$ for die-up $H_{die\_pad} - H_{bottom\_of\_bottom\_metal\_layer}$ for die-down
alpha	Die side angle in degrees.
beta	Substrate side angle in degrees.

.route
definition
.end [route]

This section defines routed nets and each routed net is defined as:

net name {segment1 segment2 ….}

where net name is one of the nets appeared in the netlist section. Each routed net consists of one or more of segments and each segment is defined by one of the following methods. All of them are in the form of

segment type signal layer number(s) segment definition

path *layer width* {*1st point segment1 segment2 ...* }
via *begin layer end layer via name x y rotation [mirror]*
bondwire *begin layer end layer bondwire id* x_d y_d x_s y_s

polygon *layer* {*x1 y1 x2 y2 ...*}
rectangle *layer width length x y*
square *layer width x y*
diamond *layer width x y*
circle *layer diameter x y*
annular *layer outer diameter inner diameter x y*
composite *layer* {*1st point segment1 segment2 ...* }
shape *layer shapeID x y rotation mirror*
shape *layer shapeID x y mirror rotation*

void_polygon *layer* {*x1 y1 x2 y2 ...*}
void_rectangle *layer width length x y*
void_square *layer width x y*
void_diamond *layer width x y*
void_circle *layer diameter x y*
void_composite *layer* {*1st point segment1 segment2 ...* }
void_shape *layer shapeID x y rotation mirror*
void_shape *layer shapeID x y mirror rotation*

The **path** segment is defined the same way as the composite shape is defined in the shape section except it has a *width* and the last point does not automatically connect to the first point.

The **via** segment requires *begin layer* and *end layer* numbers while other segments requires only one layer number. The via segment also requires

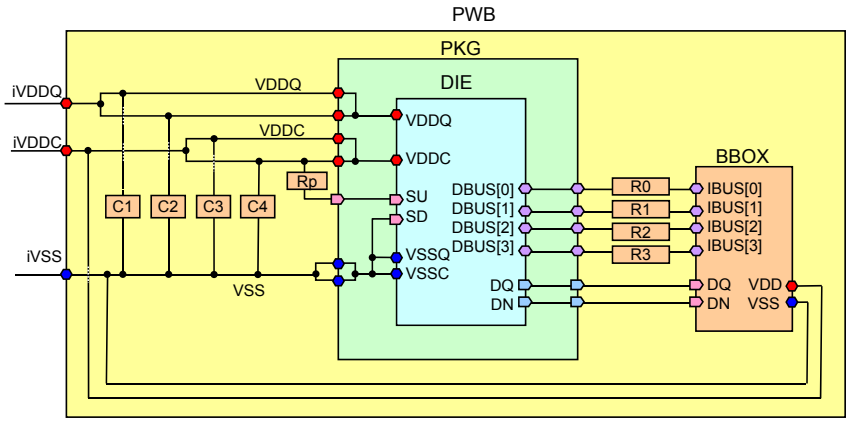
via name	Name of the via defined in the via section
x y	Location of the via
rotation	Counter-clockwise rotation angle of the pad stack in degrees
mirror	Optional. Padstack mirror flag. **Y** mirror padstack **N** do not mirror padstack

The **bondwire** segment requires

begin layer	Signal layer number of the one end of the wire. Use negative number to indicate the die.
end layer	Signal layer number of the substrate side of the wire.
bondwire id	Bond wire ID. (0 if unknown)
x_d y_d	Die side coordinate of the wire landing point
x_s y_s	Substrate side coordinate of the wire landing point

All other segment types are defined the same way as they are defined in the shape section or in the board_geom section.

LPB インターフェースフォーマット
サンプル回路のブロック図



DBUS	：	4bitsバス
DQ/DN	：	差動信号
SU	：	信号（PWBでPullUP）
SD	：	信号（PKG内でPullDown）
VDDC/VSSC	：	コア電源/グランド
VDDQ/VSSQ	：	IO電源/グランド

受動素子(コンデンサ)仕様

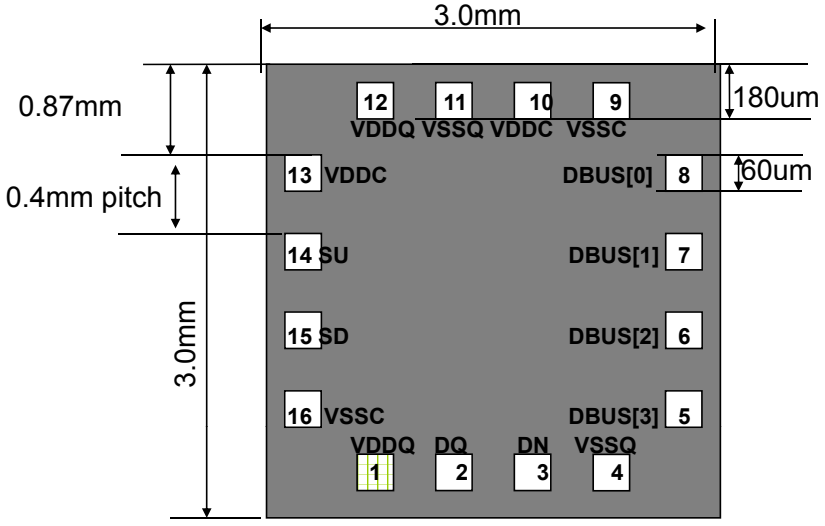
- module CF1D(A)
  - 外形サイズ 8.3x8.3mm
  - 容量　330uF
- module CF1D(B)
  - 外形サイズ 1.6x0.8mm
  - 容量　0.1uF

受動素子(抵抗器)仕様

- module RF1D(A)
- 外形サイズ 1608
- 抵抗値　10Ω

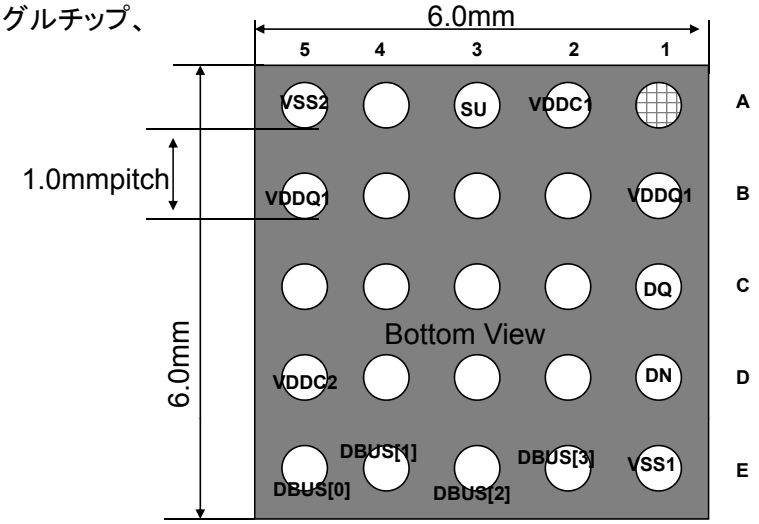
DIE仕様

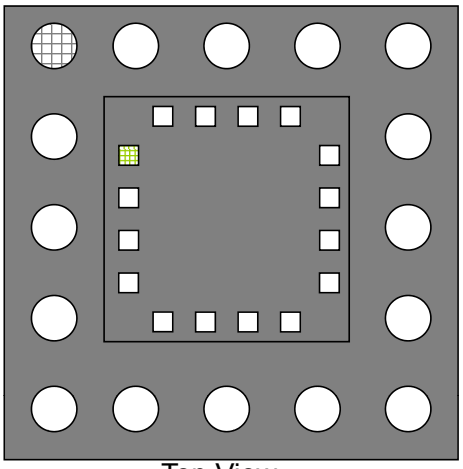
- modulde　DIE(SU,SD,DBUS,DQ,DN,VDDC,VSSC,VDDQ,VSSQ)；
- 信号(一般x2、差動×1、パス4bit×1)、電源(VSS、VCC、VSSQ、VCCQ)各2ピン
- 外部端子のみ、16ピン



PKG(4L)仕様

- modulde PKG(SU,DBUS,DQ,DN,VDDC1,VDDC2, VDDQ1,VDDQ2, VSS1, VSS2) ;
- 信号（一般x1、差動×1、パス4bit×1）、電源（VSS、VCC、VCCQ）
- モードピンをVSSにショート
- VSSとVSSQをマージ 5x5BGA（最外周以外サーマル）、A1をNC
- WB接続のシングルチップ、

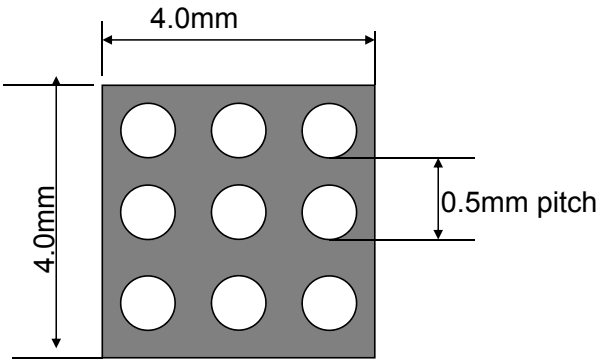




Top View

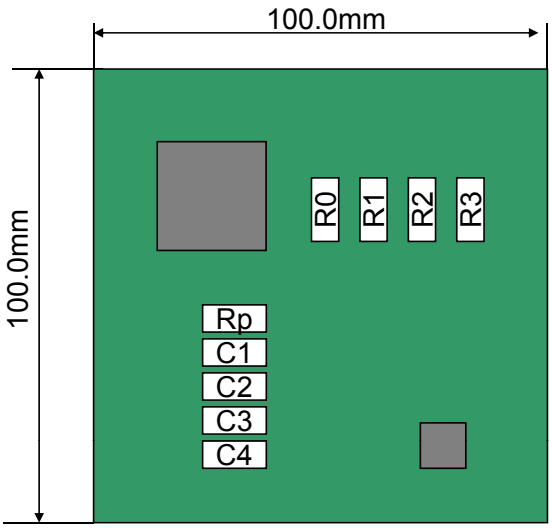
LSI(Blackbox)仕様

- modulde BBOX(IBUS,DQ,DN,VDD,VSS) ;
- SDRAM



基板(6L)仕様

- module PWB(iVDDC, iVDDQ, iVSS) ;
- 信号(一般x0、差動×1、パス4bit×1)、、電源(VSS、VCC、VCCQ)
- モードピンをVCCに接続
- その他PKGx1、抵抗、容量



ＬＰＢネットリストフォーマットのサンプル

```
/* 受動部品：コンデンサ */
module CF1D(A,B) ;
 inout A ;
 inout B ;
endmodule

/* 受動部品：抵抗器 */
module RF1D(A,B) ;
 inout A ;
 inout B ;
endmodule

/* DIE */
modulde DIE(SU,SD,DBUS,DQ,DN,VDDC,VSSC,VDDQ,VSSQ) ;
 input   SU ;
 input   SD ;
 inout [3:0] DBUS ;
 output DQ ;
 output DN ;
 inout   VDDC ; /* PG_NET VDDC */
 inout   VSSC ; /* PG_NET VSS */
 inout   VDDQ ; /* PG_NET VDDQ */
 inout   VSSQ ; /* PG_NET VSS */
endmodule

/* パッケージ */
modulde PKG(SU,DBUS,DQ,DN,VDDC1,VDDC2,VDDQ1,VDDQ2,VSS1,VSS2) ;
 input   SU ;
 input   SD ;
 inout [3:0] DBUS ;
 output DQ ;
 output DN ;
 inout   VDDC1 ; /* PG_NET VDDC */
 inout   VDDC2 ; /* PG_NET VDDC */
 inout   VDDQ1 ; /* PG_NET VDDQ */
 inout   VDDQ2 ; /* PG_NET VDDQ */
 inout   VSS1 ; /* PG_NET VSS */
 inout   VSS2 ; /* PG_NET VSS */

 wire VSS ;
 wire VDDC ;
 wire VDDQ ;
```

```
  assign VDDC1 = VDDC ;
  assign VDDC2 = VDDC ;
  assign VDDQ1 = VDDQ ;
  assign VDDQ2 = VDDQ ;
  assign VSS1   = VSS   ;
  assign VSS2   = VSS   ;

  DIE X1 ( .SU(SU), .SD(VSS), .DBUS(DBUS), .DQ(DQ), .DN(DN),
           .VDDC(VDDC), .VSSC(VSS), .VDDQ(VDDQ), .VSSQ(VSS) );
endmodule

/* LSI (Black box) */
modulde BBOX(IBUS,DQ,DN,VDD,VSS) ;
  inout   [3:0] IBUS ;
  input   DQ ;
  input   DN ;
  inout   VDD ; /* PG_NET VDDC */
  inout   VSS ; /* PG_NET VSS */
endmodule

/* 基板（最上位階層）*/
module PWB(iVDDC,iVDDQ,iVSS) ;
  inout iVDDC ; /* PG_NET VDDC */
  inout iVDDQ ; /* PG_NET VDDQ */
  inout iVSS ; /* PG_NET VSS */

  wire [3:0] DBUS ;
  wire [3:0] IBUS ;
  wire SU ;
  wire DQ ;
  wire DN ;

  CF1DM C1 ( .A(iVDDQ), .B(iVSS) ) ;
  CF1DM C2 ( .A(iVDDQ), .B(iVSS) ) ;
  CF1DM C3 ( .A(iVDDC), .B(iVSS) ) ;
  CF1DM C4 ( .A(iVDDC), .B(iVSS) ) ;

  RF1DM Rp ( .A(iVDDC), .B(SU) ) ;
  RF1DM R0 ( .A(DBUS[0]), B(IBUS[0]) ) ;
  RF1DM R1 ( .A(DBUS[1]), B(IBUS[1]) ) ;
  RF1DM R2 ( .A(DBUS[2]), B(IBUS[2]) ) ;
```

```
  RF1DM R3 ( .A(DBUS[3]), B(IBUS[3]) ) ;

  PKG   X1 ( .SU(SU), .DBUS(DBUS), .DQ(DQ), .DN(DN),
             .VDDC1(iVDDC), .VDDC2(iVDDC),
             .VDDQ1(iVDDQ), .VDDQ2(iVDDQ),
             .VSS1(iVSS) , .VSS2(iVSS) ) ;
  BBOX X2 ( .IBUS(IBUS), .DQ(DQ), .DN(DN), .VDD(iVDDC), .VSS(iVSS) ) ;
endmodule
```

階層間定義フォーマット案　(Ver 1.0)

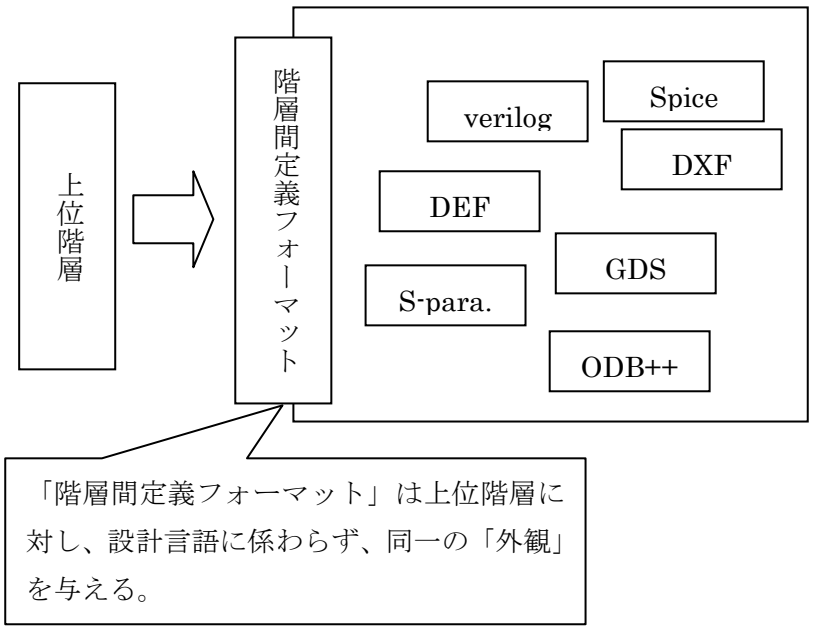
東芝セミコンダクター社　　青木孝哲
富士通セミコンダクター株式会社　　佐藤厚志
株式会社リコー　　吉田敦史

2011/02/22

die やパッケージなどの設計単位（以後、機能ブロックと呼ぶ）は、設計段階（設計フェーズ）や目的によって様々な言語によって表現される。例えば、論理回路は上位設計段階では Verilog や SystemC と言った論理的な言語で表現されるが、下位設計段階では DEF や GDS のような図形データとして表現される。またモデリングされた状態では、Spice ネットや Touch-stone 形式の S パラメータで表現される。これらの設計言語は、それぞれ異なった「外観」を持つ。

	外観（上位階層からどのように見えるか）		
	入出力ポート	物理形状	. . .
Verilog	有	無	
GDS	無	有	
S-Para	無	無	

ここで定義する「階層間定義フォーマット」は、このような様々な設計言語で表現された下位階層の機能ブロックを wrap し、同一の「外観」と上位階層と接続するための手続き（プロトコル）を与える。



プロトコル設定言語

本提案書では、次の言語を提案する。

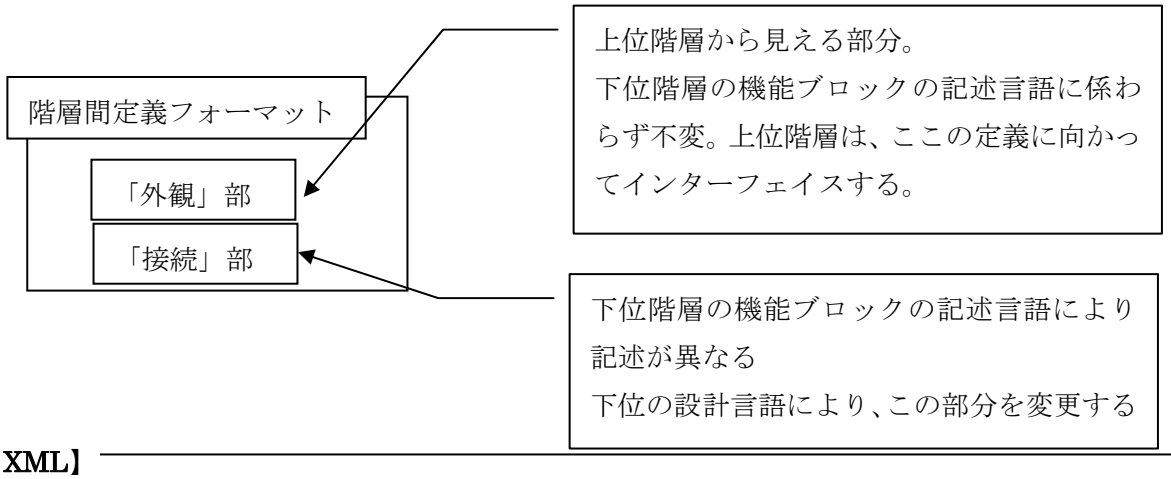
１．XML(Extensible Markup Language)を使用しスキーマ定義を行う。提案書内では、【XML】マークで識別する。XML の詳細は言語仕様書を参照のこと。

「#」がコメント行
セクションは<キーワード>から<End キーワード>で定義
セクション内のステートメントは、<キーワード>から「;」まで（改行は文末にあらず）
「スペース」「タブ」がデリミタ
「″」･･･「″」が文字列定義

本提案書で表記されているフォーマット案は、フォーマット策定のための素案でり、厳密なスキーマ定義により設計されたものではない。実運用時には厳密な設計が必要である。

「階層間定義フォーマット」の構成

「階層間論理フォーマット」は、物理形状や信号の入出力端子など、上位階層に対する機能ブロックの見せ方（見え方）を定義する「外観」部と、上位階層と下位階層の機能ブロックの接続手続きを定義する「接続」部で構成される。



【XML】

```
<module　name="機能ブロック名" type="機能ブロック属性">
    <単位系>
     <物理形状>                          ┐
     <接続口>                            ├「外観」部
     <ポートグループ>                    ┘
     <解析条件>                          ┐「仕様」部
     <下位機能ブロック参照>              ┐
     <端子接続>                          ┘「接続」部
</module>

機能ブロック名　　;　　上位階層からの参照名。リファレンス名
機能ブロック属性　　:　　DIE , PWB , PKG　など
```

単位系

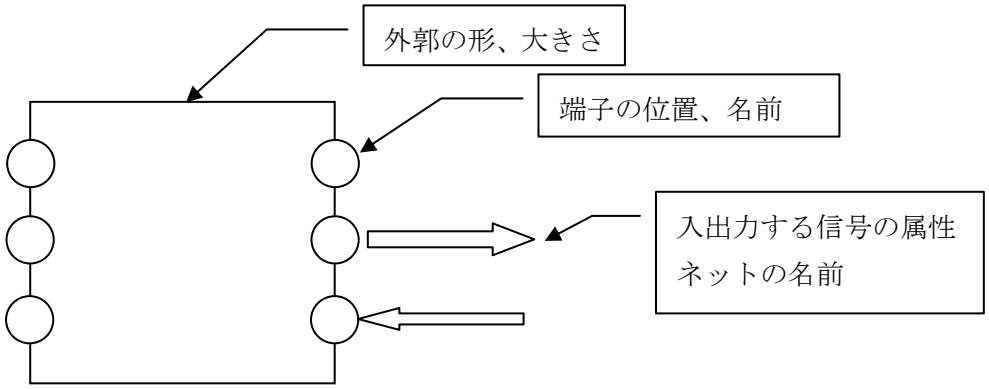
インターフェイスファイル内で使用する単位系を定義する。

【XML】

```
<unit>
    <distance unit="長さ単位" scale="縮尺値" />
    <area　　unit="面積単位" scale="縮尺値" />
    <time　　unit=時間単位"　scale="縮尺値" />
        ....
</unit>
```

「外観」部

信号が入出力する端子の数や位置、信号の属性（Power/Ground/clock/analog etc)、作動信号やアドレス線の束などの機能ブロックの「外観」を定義する。上位階層は、ここの定義に向かってインターフェイスする。
外観を構成する要素は、表現形態（言語）に係わらず不変。



物理形状

機能ブロックの外郭形状。上位階層に見せる（見せたい）物理形状。
下位階層の機能ブロックが DEF や XFL 等の図形データで定義されている場合は、ここで定義した物理形状と競合する場合がある。競合した場合の処理（考え方）は、明確化が必要。

【XML】

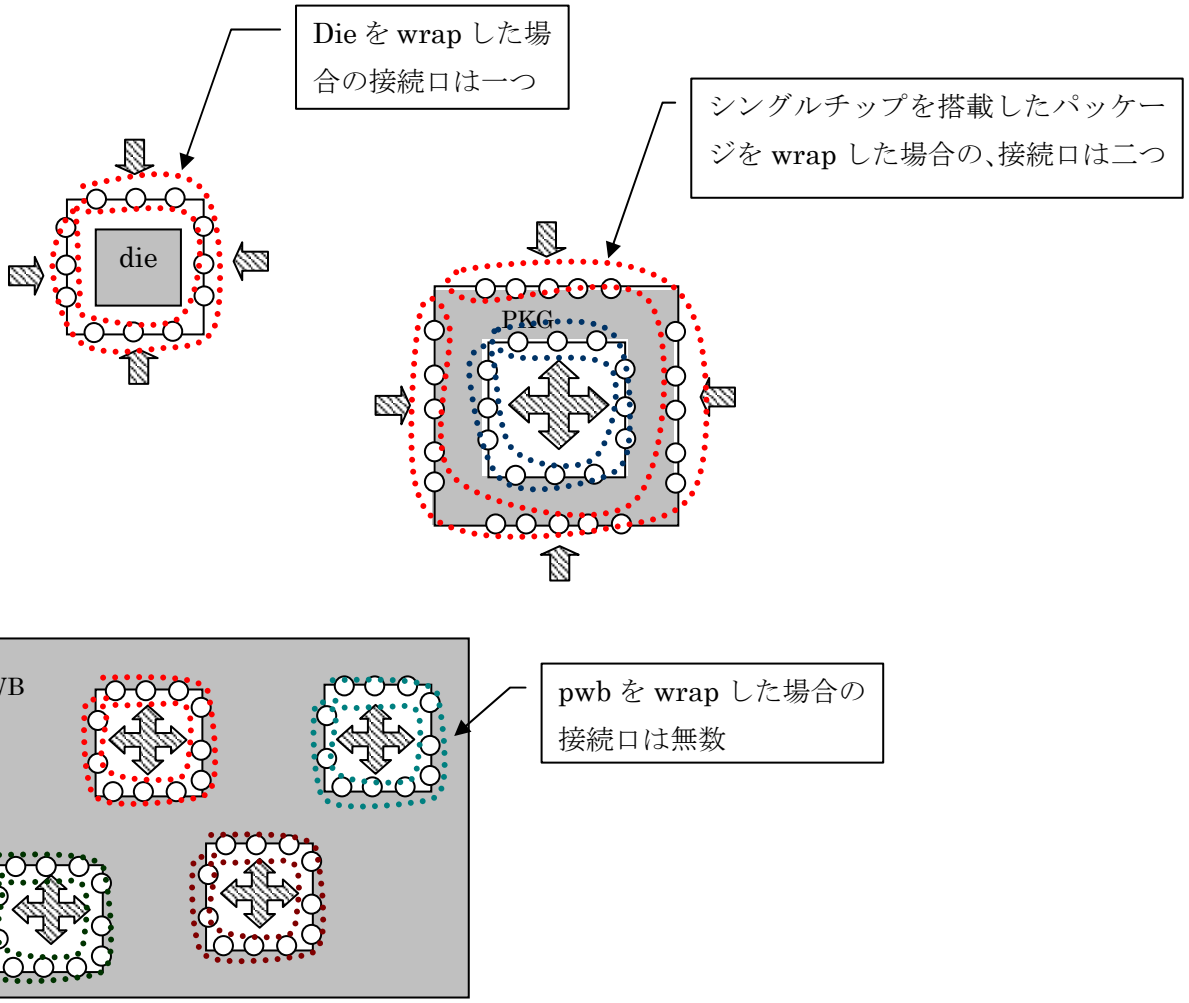
```
<shape>
    <rectangle distunit="単位" scale="縮尺率”
                llx="左下 X 座標" lly="左下 Y 座標"
                urx="右上 X 座標" urx="右上 Y 座標"/>
    <polygon distunit="単位" scale="縮尺率"
                points="x 座標,y 座標  x 座標,y 座標..."/>
    など
<shape/>
単位　：　um , mm , mil など
縮尺率　：　0 より大きい実数値
```

接続口（ソケット）

上位階層に見せる端子の集合を定義する。モデルによっては複数の「接続口」（ソケット）を持つケースがため、ここの接続口に対し、インスタンス名と参照する機能ブロック名を与える。

【XML】

```
<socket inst=”インスタンス名”>
        <入出力端子>
</socket>
```



入出力端子

端子の実態を定義する。設計言語により異なる端子定義の差異を、ここで吸収する。また、Verilog や Spice ネットのように物理的な座標を持たない言語の端子に座標を与える。

【XML】

```
<port name="端子名" id="端子番号" direction="端子方向"   type=”信号タイプ”>
        <端子形状・座標>
        ＜電気特性要求＞
</port>
```

端子名　：　上位階層から入出力端子を参照する際の参照名（リファレンス名)。

端子番号：　上位階層から参照する端子番号。１以上の整数値。

端子方向：　上位階層から見た場合の信号の方向。In , out , inout

信号タイプ：　power, ground , signal など

端子形状・座標

端子の座標もしくは形状。下位階層の機能ブロックが図形データで定義されている場合は、ここで定義した物理形状と競合する場合がある。競合した場合の処理（考え方）は、明確化が必要。

【XML】

```
<shape>
  <point distunit="単位" scale="縮尺率" x="x 座標" y="y 座標" layer="層" />
  <rectangle distunit="単位" scale="縮尺率”
            llx="左下 X 座標" lly="左下 Y 座標"
            urx="右上 X 座標" ury="右上 Y 座標"
            Layer="層" />
   など
<shape/>
```

電気特性要求

端子への電気特性要求を各ピンごとに定義する。定義する内容は今後、議論が必要。

【XML】

```
<electrical>
<Zsingle impedance＝”特性インピーダンス” tpd=”遅延”>
</electrical>
```

特性インピーダンス：　１端子にて特性インピーダンス要求する場合、ここにその値を記述（差動インピーダンス要求の場合は、portgroup にて定義する）

遅延：端子の信号遅延を定義する

ポートグループ

等電位電源やアドレスバス、差動信号等のネットの集合。グループに与えた属性により、そのグループに属する信号を出力するポートの（上位階層での）扱いが異なる。例えば「等電位」グループに属する信号同士は、上位階層でショートさせても（纏めても）エラーとしない。

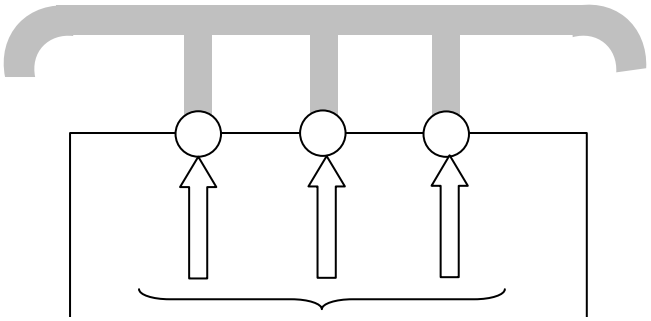
【XML】

```
<portgroup name="グループ名" type="グループ属性"
  Zdiff=”差動特性インピーダンス” skew=”スキュー”>
        <portref>ポート名</netref>
        <portref>ポート名</netref>
        ......
</portgroup>
```

グループ属性　：　PGNET , BUS , DIFF など

差動特性インピーダンス：type=”DIFF”の場合のみ有効

スキュー：　type=”DIFF”,”BUS”の場合のみ有効



電源（PGNET)属性のグループに属するポートに接続する信号は、下位階層では、別々でも、上位階層では、マージされる（場合もある)。

個々のグループ属性に対し、どのような対応を行うかは、このフォーマットを扱う処理系に依存するが、期待する処理は予め定めておいても良いだろう。

「仕様」部

解析温度条件、消費電力値などの解析条件や動作仕様を定義する。この部分で、この論理ブロックを動作仕様を与える。どの様な仕様定義を行うかは、今後の議論が必要。ここでは、温度条件と消費電力値の例のみを示す。

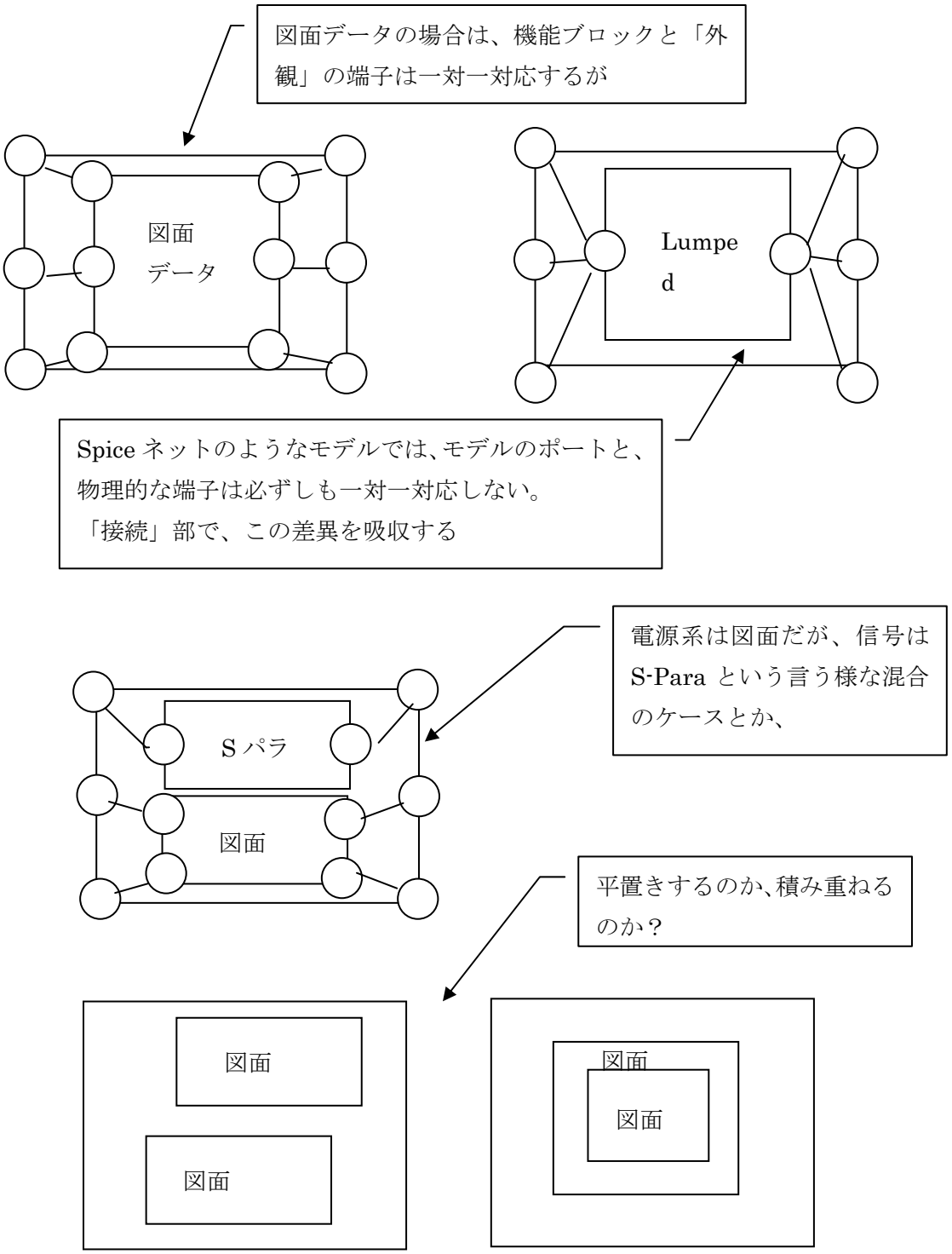
<specification>
　<消費電力＞
　.....
</specification>

消費電力

<power>"消費電力"</power>

「接続」部

上位階層と下位階層の機能ブロックの接続手続きを定義する。信号接続以外に、物理形状の座標変換やスケール変換も行う。



下位機能ブロック参照

下位階層の機能ブロックの呼び出しを行う。

【XML】

```
<reference reffile="参照ファイル名"
          format="言語フォーマット">
    <単位>
    <配置制約>
    <端子接続>
</reference>
```

参照ファイル名：参照する機能ブロックが定義されたファイル名

言語フォーマット：機能ブロックの設計言語種類。SPICE , XFL , VERILOG など

単位

DXF のような明確な単位系を持たない図形データに対して座標単位を定義する。

【XML】

```
<unit>
    <distance="長さ単位系" scale="縮尺率" />
    <anglunit="角度単位系" scale="縮尺率" />
</unit>
```

長さ単位系　：　um mm cm mil 等

角度単位系　：　degree radian mil 等

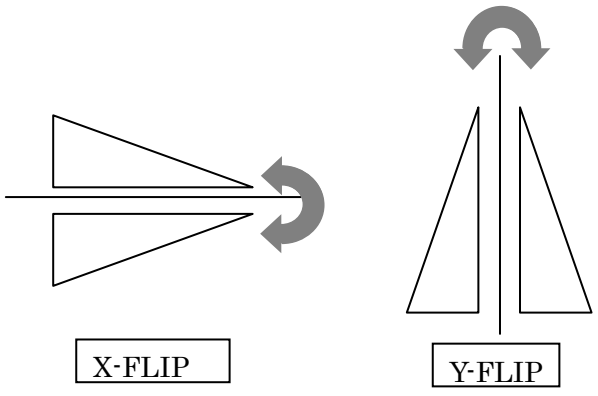
配置制約

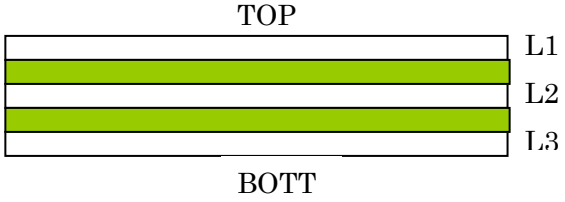
下位機能ブロックの配置制約を与える。DXF や DEF の様な図形データに対しては、<point>を使用し原点の座標を与える。Spice ネットのように物理形状を持たない機能ブロックに対しては、<rectangle>を使用し配置された物理形状を与える。

【XML】

```
<placement module=”機能ブロック名”>
    <point x="x 座標" y="y 座標" flip="フリップタイプ”
                                angle="回転角度"  layer="レイヤ" mount="配置面"/>
    <rectangle llx="左下 X 座標" lly="左下 Y 座標"
                urx="右上 X 座標" ury="右上 Y 座標"
                layer="層"  mount="配置面"/>
    など
</placement>
```

機能ブロック	参照先ファイルの機能ブロック名
x 座標/y 座標	機能ブロックの原点の配置座標
フリップタイプ	X … X 軸を中心ととしたフリップ（上下反転）
	Y … Y 軸を中心としたフリップ（左右反転）
回転角度	原点を中心とした回転角度
	「FLIP->回転」の順で配置を行う
レイヤ	配置されるレイヤ
配置面	TOP ... 対象レイヤの TOP 面に機能ブロックを配置
	BOTTOM … 対象レイヤの BOTTOM 面に機能ブロックを配置



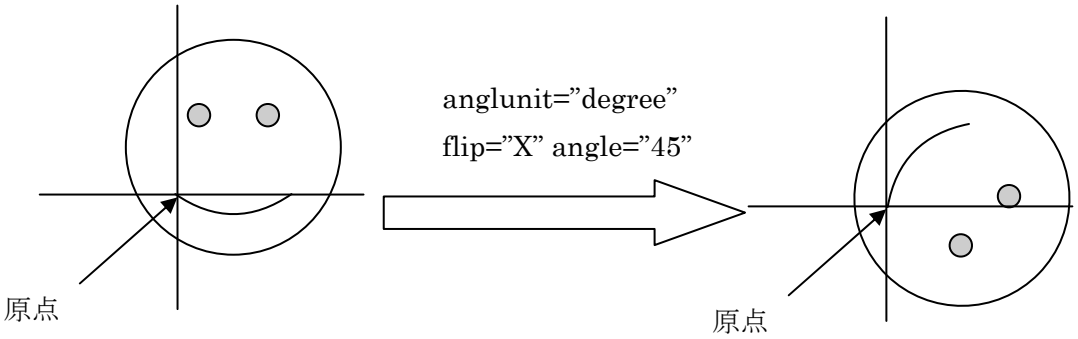


NOTE1

このフォーマット内では、層構造は定義していない。

NOTE2

ここでは、「FLIP->回転」の順で配置を行うこととする。



端子接続

「外観」で定義した端子(<port>)と、下位の機能ブロックを接続する。機能ブロックを定義する設計言語により、接続記述方式は異なる。

【XML】

```
<connection inst=”インスタント名” portname="入出力端子名">
    <下位機能ブロック端子参照>
</connection>
```

下位機能ブロック端子参照

「下位機能ブロック端子参照」は、設計言語により異なる。以下に、代表的な例を挙げる。

例えば、verilog や VHDL 等のポートが明確に定義されている言語は、「モジュール名」と「ポート名」の組み合わせで参照される。

【XML】

```
<verilog:portref module="モジュール名" portname="ポート名"/>
```

~~DEF では、コンポーネント名とピン名の組み合わせで参照するケースと、ピンを直接参照する~~ケースが考えられる。

【XML】

```
<def:portref comp=”コンポーネント名” pin=”ピン名”/>
<def:portref pin=”ピン名”/>
```

Spice ネットでは、サブサーキット名とポート番号（端子の定義順）での参照となる。

【XML】

```
<spice:portref subckt="サブサーキット名" portid="ポート番号"/>
```

Touch stone 形式の S パラメータでは、端子番号だけの参照となる。

【XML】
```
<touchstone:portref portid="ポート番号"/>
```

DXF や GDS 等の明確なポート定義を持たない図形表現言語では、座標による参照となる。

【XML】
```
<dxf:portref distunit=”単位系” distscale=”縮尺率”
                                    x="x 座標" y="y 座標" layer=”レイヤ”/>
<gds:portref distunit=”単位系” distscale=”縮尺率”
                                    x="x 座標" y="y 座標" layer=”レイヤ”/>
```

XFL 形式では、下記のようになる。

【XML】
```
<xfl:portref module=”機能ブロック名”  name=”端子名”>
```

IBIS 形式では、下記のようになる。

【XML】
```
<ibs:portref     model="モデル名" portname="端子名"/>
```

この文書にて定めている階層間定義を呼び出す LPB 形式では、下記のようになる。

【XML】
```
<JLPB:portref module="機能ブロック名" portname="端子名"/>
```

以上のように、設計言語ごとにポートへの参照方法はことなる。設計言語ごとに個別に端子参照の記述方式は定める必要が有るだろう。代表的なオープンフォーマットの言語の場合は、予め LPB SWG で定めれば良いだろう。

ベンダ固有の非公開フォーマットの場合は、ベンダに記述形式を定めてもらうことで、対応可能と思われる。

例えば、あるベンダが提供する XXX 形式のバイナリデータファイルの場合

【XML】
```
<XXX:portref>
  暗号化された接続方式、これをどう解釈するかは XXX を入力する側の
      ベンダと、XXX を提供する側のベンダとのライセンスで管理する
</XXX:portref>
```

階層間接続フォーマット

タグセット・スキーマ定義案

2011/02/23

```
<?xml version="1.0" encoding="Shift_JIS"?>
<!-- Yet another Chip-Package-Bard Connection Protocol(YACPBCP) -->
<!--      2011/02/22                                                  -->
<!--      DRAFT 1.5                                                    -->
<!--      takahiro.aoki@toshiba.co.jp                                 -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.jeita.or.jp/LPB/YACPBCP"
           xmlns="http://www.jeita.or.jp/LPB/YACPBCP"
           xmlns:Verilog="http://www.jeita.or.jp/LPB/Verilog/"
           xmlns:VHDL="http://www.jeita.or.jp/LPB/VHDL/"
           xmlns:Spice="http://www.jeita.or.jp/LPB/Spice/"
           xmlns:TouchStone="http://www.jeita.or.jp/LPB/TouchStone/"
           xmlns:EDIF="http://www.jeita.or.jp/LPB/EDIF/"
           xmlns:DEF="http://www.jeita.or.jp/LPB/DEF/"
           xmlns:LEF="http://www.jeita.or.jp/LPB/LEF/"
           xmlns:GDS="http://www.jeita.or.jp/LPB/GDS/"
           xmlns:DXF="http://www.jeita.or.jp/LPB/DXF/"
           xmlns:AIF="http://www.jeita.or.jp/LPB/AIF/"
           xmlns:XFL="http://www.jeita.or.jp/LPB/XFL/"
           xmlns:ODBpp="http://www.jeita.or.jp/LPB/ODBpp/"
           xmlns:MCM="http://www.jeita.or.jp/LPB/MCM/"
           xmlns:bga="http://www.jeita.or.jp/LPB/bga/"
           xmlns:ndf="http://www.jeita.or.jp/LPB/ndf/"
           xmlns:cdm="http://www.jeita.or.jp/LPB/cdm/"
           xmlns:spd="http://www.jeita.or.jp/LPB/spd/"
           xmlns:spd2="http://www.jeita.or.jp/LPB/spd2/"
           xmlns:spd2="http://www.jeita.or.jp/LPB/ibs/"
           elementFormDefault="qualified">

<xs:import namespace="http://www.jeita.or.jp/LPB/Verilog/"       schemaLocation="Verilog.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/VHDL/"          schemaLocation="VHDL.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/Spice/"         schemaLocation="Spice.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/TouchStone/"    schemaLocation="TouchStone.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/EDIF/"          schemaLocation="EDIF.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/DEF/"           schemaLocation="DEF.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/LEF/"           schemaLocation="LEF.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/GDS/"           schemaLocation="GDS.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/DXF/"           schemaLocation="DXF.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/AIF/"           schemaLocation="AIF.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/XFL/"           schemaLocation="XFL.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/ODBpp/"         schemaLocation="ODBpp.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/MCM/"           schemaLocation="MCM.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/bga/"           schemaLocation="bga.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/ndf/"           schemaLocation="ndf.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/cdm/"           schemaLocation="cdm.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/spd/"           schemaLocation="spd.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/spd2/"          schemaLocation="spd2.xsd"/>
<xs:import namespace="http://www.jeita.or.jp/LPB/ibs/"           schemaLocation="ibs.xsd"/>


<!-- 階層間接続フォーマット構造 -->
<xs:element name="module">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="unit"              minOccurs="0"/>
      <xs:element ref="shape"             minOccurs="0"/>
      <xs:element ref="socket"            minOccurs="0" maxOccurs="unbounded"/>
```

```
        <xs:element ref="electrical"       minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="portgroup"        minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="specification" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="reference"        minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="connection"       minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="type">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="DIE"/>
            <xs:enumeration value="PKG"/>
            <xs:enumeration value="PWB"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>



  <!-- 物理形状 -->
  <xs:element name="shape">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="point"/>
        <xs:element ref="rectangle"/>
        <xs:element ref="polygon"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="rectangle">
    <xs:complexType>
      <xs:attributeGroup ref="DistanceUnit"/>
      <xs:attribute name="llx"     type="xs:double" use="required"/>
      <xs:attribute name="lly"     type="xs:double" use="required"/>
      <xs:attribute name="urx"     type="xs:double" use="required"/>
      <xs:attribute name="ury"     type="xs:double" use="required"/>
      <xs:attribute name="layer" type="xs:string"/>
      <xs:attributeGroup ref="MountType"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="polygon">
    <xs:complexType>
      <xs:attributeGroup ref="DistanceUnit"/>
      <xs:attribute name="points" type="xs:string" use="required"/>
      <xs:attribute name="layer"   type="xs:string"/>
      <xs:attributeGroup ref="MountType"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="point">
    <xs:complexType>
      <xs:attributeGroup ref="DistanceUnit"/>
      <xs:attributeGroup ref="AngleUnit"/>
      <xs:attribute name="x"       type="xs:double" use="required"/>
      <xs:attribute name="y"       type="xs:double" use="required"/>
      <xs:attribute name="angle" type="xs:double" default="0.0"/>
      <xs:attribute name="layer" type="xs:string"/>
      <xs:attributeGroup ref="MountType"/>
      <xs:attributeGroup ref="FlipType"/>
    </xs:complexType>
  </xs:element>

  <xs:attributeGroup name="FlipType">
      <xs:attribute name="flip" default="none">
```

```
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="none"/>
            <xs:enumeration value="X"/>
            <xs:enumeration value="Y"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
</xs:attributeGroup>

<xs:attributeGroup name="MountType">
      <xs:attribute name="mount" default="TOP">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="TOP"/>
            <xs:enumeration value="BOTTOM"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
</xs:attributeGroup>

<xs:attributeGroup name="PowerUnit">
      <xs:attribute name="power" default="w">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="w"/>
            <xs:enumeration value="mw"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
</xs:attributeGroup>

<xs:attributeGroup name="DistanceUnit">
      <xs:attribute name="distunit" default="um">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="um"/>
            <xs:enumeration value="mm"/>
            <xs:enumeration value="cm"/>
            <xs:enumeration value="mil"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="distscale" type="xs:double" default="1.0"/>
</xs:attributeGroup>

<xs:attributeGroup name="AngleUnit">
      <xs:attribute name="anglunit" default="degree">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="degree"/>
            <xs:enumeration value="radian"/>
            <xs:enumeration value="mil"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="anglscale" type="xs:double" default="1.0"/>
</xs:attributeGroup>



<!-- 入出力端子 -->
<xs:element name="socket">
   <xs:complexType>
      <xs:sequence>
          <xs:element ref="port" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="inst"       type="xs:string" use="required"/>
```

```
  </xs:complexType>
</xs:element>

<xs:element name="port">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="shape"          minOccurs="0"/>
      <xs:element ref="electrical"   minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:attribute name="name"         type="xs:string"              use="required"/>
  <xs:attribute name="id"           type="xs:positiveInteger" use="required"/>
  <xs:attribute name="direction" default="inout">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="in"/>
          <xs:enumeration value="out"/>
          <xs:enumeration value="inout"/>
        </xs:restriction>
      </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="type" default="signal">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="power"/>
        <xs:enumeration value="ground"/>
        <xs:enumeration value="signal"/>
        <xs:enumeration value="NC"/>
        <xs:enumeration value="open"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="x" type="xs:double" />
  <xs:attribute name="y" type="xs:double" />
</xs:element>

<!-- 電気特性 -->
<xs:element name="electrical">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Zsingle" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Zsingle">
  <xs:complexType>
      <xs:attribute name="impedance" type="xs:double" use="required" />
      <xs:attribute name="tpd"          type="xs:double" use="required" />
  </xs:complexType>
</xs:element>

<!-- ポートグループ -->
<xs:element name="portgroup">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="portref" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="PGNET"/>
          <xs:enumeration value="BUS"/>
          <xs:enumeration value="DIFF"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
```

```
      <xs:attribute name="Zdiff" type="xs:double" />
      <xs:attribute name="skew"    type="xs:double" />
    </xs:complexType>
</xs:element>

<xs:element name="portref" type="xs:string"/>

<!-- 動作仕様-->
<xs:element name="specification">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="power" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>


<!-- 下位機能ブロック参照     -->

<xs:element name="reference">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="unit"              minOccurs="0"/>
        <xs:element ref="placement"     minOccurs="0"/>
        <xs:element ref="connection"    minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="inst"       type="xs:string" use="required"/>
      <xs:attribute name="reffile" type="xs:string" use="required"/>
      <xs:attribute name="format">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Verilog"/>
            <xs:enumeration value="VHDL"/>
            <xs:enumeration value="Spice"/>
            <xs:enumeration value="TouchStone"/>
            <xs:enumeration value="EDIF"/>
            <xs:enumeration value="DEF"/>
            <xs:enumeration value="LEF"/>
            <xs:enumeration value="GDS"/>
            <xs:enumeration value="DXF"/>
            <xs:enumeration value="AIF"/>
            <xs:enumeration value="XFL"/>
            <xs:enumeration value="ODBpp"/>
            <xs:enumeration value="MCM"/>
            <xs:enumeration value="bga"/>
            <xs:enumeration value="ndf"/>
            <xs:enumeration value="cdm"/>
            <xs:enumeration value="spd"/>
            <xs:enumeration value="spd2"/>
            <xs:enumeration value="ibs"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
</xs:element>


<!-- 単位 -->
<xs:element name="unit">
    <xs:complexType>
      <xs:sequence>

        <xs:element name="distance"    minOccurs="0">
          <xs:complexType>
              <xs:attribute name="unit" default="um">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:enumeration value="um"/>
```

```
                    <xs:enumeration value="mm"/>
                    <xs:enumeration value="cm"/>
                    <xs:enumeration value="mil"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:attribute>
              <xs:attribute name="scale" type="xs:double" default="1.0"/>
            </xs:complexType>
          </xs:element>

          <xs:element name="angle"    minOccurs="0">
            <xs:complexType>
              <xs:attribute name="unit" default="degree">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:enumeration value="degree"/>
                    <xs:enumeration value="radian"/>
                    <xs:enumeration value="mil"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:attribute>
              <xs:attribute name="scale" type="xs:double" default="1.0"/>
            </xs:complexType>
          </xs:element>

          <xs:element name="time"    minOccurs="0">
            <xs:complexType>
              <xs:attribute name="unit" default="ps">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:enumeration value="ps"/>
                    <xs:enumeration value="ns"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:attribute>
              <xs:attribute name="scale" type="xs:double" default="1.0"/>
            </xs:complexType>
          </xs:element>

        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="distunit">
      <xs:complexType>
        <xs:attributeGroup name="DistanceUnit"/>
      </xs:complexType>
    </xs:element>

    <xs:element name="anglunit">
      <xs:complexType>
        <xs:attributeGroup name="AngleUnit"/>
      </xs:complexType>
    </xs:element>


    <!-- 配置制約 -->
    <xs:element name="placement">
      <xs:complexType>
        <xs:choice>
          <xs:element ref="point"/>
          <xs:element ref="rectangle"/>
          <xs:element ref="polygon"/>
        </xs:choice>
        <xs:attribute name="module" type="xs:string" />
      </xs:complexType>
    </xs:element>
```

```
<!-- 端子接続 -->
<xs:element name="connection">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="portref"                    minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="Verilog:portref"           minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="VHDL:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="Spice:portref"              minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="TouchStone:portref"       minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="EDIF:portref"               minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="DEF:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="LEF:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="GDS:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="DXF:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="AIF:portref"               minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="XFL:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="ODBpp:portref"              minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="MCM:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="bga:portref"               minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="ndf:portref"               minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="cdm:portref"                minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="spd:portref"               minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="spd2:portref"               minOccurs="1" maxOccurs="unbounded"/>
      <xs:element ref="ibs:portref"              minOccurs="1" maxOccurs="unbounded"/>
    </xs:choice>
  </xs:complexType>
  <xs:attribute name="inst"            type="xs:string" use="required"/>
  <xs:attribute name="portname"      type="xs:string" use="required"/>
  <xs:attribute name="portref"       type="xs:string"/>
</xs:element>

<xs:element name="portref">
  <xs:complexType>
    <xs:attribute name="module"     type="xs:string"/>
    <xs:attribute name="subckt"     type="xs:string"/>
    <xs:attribute name="portname" type="xs:string"/>
    <xs:attribute name="portid"     type="xs:positiveInteger"/>
    <xs:attribute name="distunit"   type="xs:string"/>
    <xs:attribute name="distscale" type="xs:double" default="1.0"/>
    <xs:attribute name="x"            type="xs:double"/>
    <xs:attribute name="y"            type="xs:double"/>
    <xs:attribute name="layer"      type="xs:string"/>
  </xs:complexType>
</xs:element>


<!-- end of schema -->
</xs:schema>
```

階層接続フォーマットサンプル

```
<?xml version="1.0" encoding="Shift_JIS"?>

<JEITA>

<!-- ############################################################################## -->
<!-- #                                                                            # -->
<!-- ############################################################################## -->
<module name="LSI" type="DIE">
  <unit>
    <distance unit="um"        scale="1.0"/>
    <angle      unit="degree" scale="1.0"/>
  </unit>
  <shape>
    <rectangle llx="-1500" lly="-1500" urx="1500" ury="1500" />
  </shape>
  <socket inst="DIEPORTS">
<!--net name="VDDQ" type="power">
      <port id="1"   name="P1"   type="power"   direction="inout"/>
<!--net name="DQ" type="signal">
      <port id="2"   name="P2"   type="signal" direction="out"/>
<!--net name="DN" type="signal">
      <port id="3"   name="P3"   type="signal" direction="out"/>
  <!--net name="VSSQ" type="ground">
      <port id="4"   name="P4"   type="ground" direction="inout"/>
<!--net name="DBUS[3]" type="signal">
      <port id="5"   name="P5"   type="signal" direction="inout"/>
<!--net name="DBUS[2]" type="signal">
      <port id="6"   name="P6"   type="signal" direction="inout"/>
<!-- name="DBUS[1]" type="signal"-->
      <port id="7"   name="P7"   type="signal" direction="inout"/>
<!--"DBUS[0]" type="signal"-->
      <port id="8"   name="P8"   type="signal" direction="inout"/>
  <!--net name="VSSC" type="ground">
      <port id="9"   name="P9"   type="ground" direction="inout"/>
  <!--net name="VDDC" type="power">
      <port id="10" name="P10" type="power"   direction="inout"/>
  <!--net name="VSSQ" type="ground">
      <port id="11" name="P11" type="ground" direction="inout"/>
  <!--net name="VDDQ" type="power">
      <port id="12" name="P12" type="power"   direction="inout"/>
  <!--net name="VDDC" type="power">
      <port id="13" name="P13" type="power"   direction="inout"/>
```

```
<!--net name="SU" type="signal">
        <port id="14" name="P14" type="signal" direction="in"/>
<!--net name="SD" type="signal">
        <port id="15" name="P15" type="signal" direction="in"/>
    <!--net name="VSSC" type="ground">
        <port id="16" name="P16" type="ground" direction="inout"/>
    </socket>

    <portgroup    name="XXBUS" type="BUS">
      <portref>P8</portref>
      <portref>P7</portref>
      <portref>P6</portref>
      <portref>P5</portref>
    </portgroup>
    <portgroup    name="XXDIFF" type="DIFF">
      <portref>P2</portref>
      <portref>P3</portref>
    </portgroup>

    <portgroup    name="VDDC" type="PGNET">
      <portref>P9</portref>
      <portref>P14</portref>
    </portgroup>

    <portgroup    name="VSS" type="PGNET">
      <portref>P9</portref>
      <portref>P11</portref>
      <portref>P15</portref>
    </portgroup>

<specification>

<power>1.0</power>

</specification>

<reference reffile="sample_verilog.txt" format="verilog">
        <connection inst="DIEPORT" portref="P8">
                <verilog:portref module="DIE" portname="DBUS[0]"/>
        </connection>
        <connection inst="DIEPORT" portref="P7">
                <verilog:portref module="DIE" portname="DBUS[1]"/>
```

```
    </connection>
    <connection inst="DIEPORT" portref="P6">
        <verilog:portref module="DIE" portname="DBUS[2]"/>
    </connection>
    <connection inst="DIEPORT"   portref="P5">
        <verilog:portref module="DIE" portname="DBUS[3]"/>
    </connection>
    <connection inst="DIEPORT"   portref="P2">
        <verilog:portref module="DIE" portname="DQ"/>
    </connection>
    <connection inst="DIEPORT"   portref="P3">
        <verilog:portref module="DIE" portname="DN"/>
    </connection>
    <connection inst="DIEPORT"   portref="P14">
      <verilog:portref module="DIE" portname="SU"/>
    </connection>
    <connection inst="DIEPORT"   portref="P15">
        <verilog:portref module="DIE" portname="SD"/>
    </connection>
    <connection inst="DIEPORT"   portref="P10">
        <verilog:portref module="DIE" portname="VDDC"/>
    </connection>
    <connection inst="DIEPORT"   portref="P9">
      <verilog:portref module="DIE" portname="VSSC"/>
    </connection>
    <connection inst="DIEPORT"   portref="P12">
        <verilog:portref module="DIE" portname="VDDQ"/>
    </connection>
    <connection inst="DIEPORT"   portref="P11">
        <verilog:portref module="DIE" portname="VSSQ"/>
    </connection>
  </reference>
</module>

<!-- ############################################################################## -->
<!-- #                                                                            # -->
<!-- ############################################################################## -->
<module name="Package" type="PKG">
  <unit>
    <distance unit="um"        scale="1.0"/>
    <angle      unit="degree" scale="1.0"/>
  </unit>
```

```
<shape>
  <rectangle llx="-3000" lly="-3000" urx="3000" ury="3000" />
</shape>
<socket inst="BGA">
    <port id="1"    name="A1" type="NC"                                x="-2000"    y="2000"   />
    <port id="2"    name="A2" type="power"    direction="inout"   x="-1000"    y="2000"   />
    <port id="3"    name="A3" type="signal"   direction="in"   x="0"          y="2000"   />
    <port id="4"    name="A4" type="open"                              x="1000"      y="2000"   />
    <port id="5"    name="A5" type="ground"   direction="inout"   x="2000"      y="2000"   />
    <port id="6"    name="B1" type="power"    direction="inout"   x="-2000"    y="1000"   />
    <port id="7"    name="B2" type="thermal" direction="in"   x="-1000"    y="1000"   />
    <port id="8"    name="B3" type="thermal" direction="in"   x="0"          y="1000"   />
    <port id="9"    name="B4" type="thermal" direction="in"   x="1000"      y="1000"   />
    <port id="10"   name="B5" type="power"    direction="inout"   x="2000"      y="1000"   />
    <port id="11"   name="C1" type="signal"   direction="out" x="-2000"    y="0"   />
    <port id="12"   name="C2" type="thermal" direction="in"   x="-1000"    y="0"   />
    <port id="13"   name="C3" type="thermal" direction="in"   x="0"          y="0"   />
    <port id="14"   name="C4" type="thermal" direction="in"   x="1000"      y="0"   />
    <port id="15"   name="C5" type="open"                              x="2000"      y="0"   />
    <port id="16"   name="D1" type="signal"   direction="out" x="-2000"    y="-1000"   />
    <port id="17"   name="D2" type="thermal" direction="in"   x="-1000"    y="-1000"   />
    <port id="18"   name="D3" type="thermal" direction="in"   x="0"          y="-1000"   />
    <port id="19"   name="D4" type="thermal" direction="in"   x="1000"      y="-1000"   />
    <port id="20"   name="D5" type="power"    direction="inout"   x="2000"      y="-1000"   />
    <port id="21"   name="E1" type="ground"   direction="inout"   x="-2000"    y="-2000"   />
    <port id="22"   name="E2" type="signal"   direction="inout" x="-1000"    y="-2000"   />
    <port id="23"   name="E3" type="signal"   direction="inout" x="0"          y="-2000"   />
    <port id="24"   name="E4" type="signal"   direction="inout" x="1000"      y="-2000"   />
    <port id="25"   name="E5" type="signal"   direction="inout" x="2000"      y="-2000"   />
</socket>

<socket inst="DIEPORT">
    <port id="26"   name="P1"   itype="power"   direction="inout" />
    <port id="27"   name="P2"   type="signal"   direction="out" />
    <port id="28"   name="P3"   type="signal"   direction="out"   />
    <port id="29"   name="P4"   tyep="grond"    direction="inout" />
    <port id="30"   name="P5"   type="signal"   direction="inout" />
    <port id="31"   name="P6"   type="signal"   direction="inout" />
    <port id="32"   name="P7"   type="signal"   direction="inout" />
    <port id="33"   name="P8"   type="signal"   direction="inout" />
    <port id="34"   name="P9"   type="ground"   direction="inout"   />
    <port id="35"   name="P10" type="power"    direction="inout" />
```

```
    <port id="36"   name="P11" type="ground"   direction="inout" />
    <port id="37"   name="P12" type="power"    direction="inout"   />
    <port id="38"   name="P13" type="power"    direction="inout"   />
    <port id="39"   name="P14" type="signal"   direction="in" />
    <port id="40"   name="P15" type="signal"   direction="in"/>
    <port id="41"   name="P16" type="ground"   direction="inout" />
</socket>


<portgroup   name="XXBUS" type="BUS" skew="50p">
  <portref>E5</portref>
  <portref>E4</portref>
  <portref>E3</portref>
  <portref>E2</portref>
</portgroup>
<portgroup   name="XXDIFF" type="DIFF" Zdiff="100" skew="50p">
  <portref>C1</portref>
  <portref>D1</portref>
</portgroup>

<portgroup   name="VDDC" type="PGNET">
  <portref>A2</portref>
  <portref>D5</portref>
  <portref>A3</portref>
</portgroup>

<reference reffile="sample_verilog.txt" format="verilog">
  <connection inst="BGA"   portref="E5">
    <verilog:portref module="PKG" portname="DBUS[0]"/>
  </connection>
  <connection inst="BGA"   portref="E4">
    <verilog:portref module="PKG" portname="DBUS[1]"/>
  </connection>
  <connection inst="BGA"   portref="E3">
    <verilog:portref module="PKG" portname="DBUS[2]"/>
  </connection>
  <connection inst="BGA"   portref="E2">
    <verilog:portref module="PKG" portname="DBUS[3]"/>
  </connection>
  <connection inst="BGA"   portref="C1">
    <verilog:portref module="PKG" portname="DQ"/>
  </connection>
```

```
    <connection inst="BGA"    portref="D1">
      <verilog:portref module="PKG" portname="DN"/>
    </connection>
    <connection inst="BGA"    portref="A3">
      <verilog:portref module="PKG" portname="SU"/>
    </connection>
    <connection inst="BGA"    portref="A2">
      <verilog:portref module="PKG" portname="VDDC1"/>
    </connection>
    <connection inst="BGA"    portref="D5">
      <verilog:portref module="PKG" portname="VDDC2"/>
    </connection>
    <connection inst="BGA"    portref="B1">
      <verilog:portref module="PKG" portname="VDDQ1"/>
    </connection>
    <connection inst="BGA"    portref="B5">
      <verilog:portref module="PKG" portname="VDDQ2"/>
    </connection>
    <connection inst="BGA"    portref="A5">
      <verilog:portref module="PKG" portname="VSS1"/>
    </connection>
    <connection inst="BGA"    portref="E1">
      <verilog:portref module="PKG" portname="VSS2"/>
    </connection>

  </reference>

 <reference reffile="XML での階層間定義例_20110223_verilog.xml" format="JLPB">
<placement module="DIE">
    <rectangle llx="-1500" lly="-1500" urx="1500" ury="1500" layer="L1", mount="TOP" />
</placement>

    <connection inst="DIEPORT" portref="P8">
          <JLPB:portref module="DIE" portname="P8"/>
    </connection>
    <connection inst="DIEPORT" portref="P7">
          <verilog:portref module="DIE" portname="P7"/>
    </connection>
    <connection inst="DIEPORT" portref="P6">
          <verilog:portref module="DIE" portname="P6"/>
    </connection>
    <connection inst="DIEPORT" portref="P5">
```

```
            <verilog:portref module="DIE" portname="P5"/>
        </connection>
        <connection inst="DIEPORT" portref="P2">
            <verilog:portref module="DIE" portname="P2"/>
        </connection>
        <connection inst="DIEPORT" portref="P3">
            <verilog:portref module="DIE" portname="P3"/>
        </connection>
        <connection inst="DIEPORT" portref="P14">
          <verilog:portref module="DIE" portname="P14"/>
        </connection>
        <connection inst="DIEPORT" portref="P15">
            <verilog:portref module="DIE" portname="P15"/>
        </connection>
        <connection inst="DIEPORT" portref="P10">
            <verilog:portref module="DIE" portname="P10"/>
        </connection>
        <connection inst="DIEPORT" portref="P9">
          <verilog:portref module="DIE" portname="P9"/>
        </connection>
        <connection inst="DIEPORT" portref="P12">
            <verilog:portref module="DIE" portname="P12"/>
        </connection>
        <connection inst="DIEPORT" portref="P11">
            <verilog:portref module="DIE" portname="P11"/>
        </connection>

</reference>

</module>


<!-- ############################################################################## -->
<!-- #                                                                            # -->
<!-- ############################################################################## -->
<module name="Board" type="PWB">
    <unit>
        <distance unit="mm"         scale="1.0"/>
        <angle       unit="degree" scale="1.0"/>
    </unit>
    <shape>
        <rectangle llx="-50" lly="-50" urx="50" ury="50" />
```

```
</shape>

<socket inst="SUPPLY">
    <port id="1"   name="iVDDQ" direction="inout" type="power"   x="" y=""   />
    <port id="2"   name="iVDDC" direction="inout" type="power"   x="" y=""   />
    <port id="3"   name="iVSS"   direction="inout" type="ground" x="" y=""   />
</socket>
<socket inst="BGA">
     <port   id="4"   name="A1"                        type="NC"          x="-32"   y="28" />
  <!--net name="VDDC"      type="power"   >
     <port   id="5"   name="A2" direction="inout"   type="power"       x="-32"   y="29" />
  <!--net name="SU"         type="signal" >
     <port   id="6"   name="A3" direction="in"   type="signal"    x="-32"   y="30" />
     <port   id="7"   name="A4"                        type="open"       x="-32"   y="31" />
  <!--net name="VSS"      type="ground"   >
     <port   id="8"   name="A5" direction="inout"   type="ground"    x="-32"   y="32" />
     <port   id="9"   name="B1" direction="inout"   type="power"      x="-31"   y="28" />
  <!--net name="VDDQ"      type="power"   >
     <port id="10"   name="B2" direction="in"   type="thermal"   x="-31"   y="29" />
     <port id="11"   name="B3" direction="in"   type="thermal"   x="-31"   y="30" />
     <port id="12"   name="B4" direction="in"   type="thermal"   x="-31"   y="31" />
  <!--net name="VDDQ"      type="power"   >
     <port id="13"   name="B5" direction="inout"   type="power"      x="-31"   y="32" />
  <!--net name="DQ"         type="signal" >
     <port id="14"   name="C1" direction="out" type="signal"    x="-30"   y="28" />
     <port id="15"   name="C2" direction="in"   type="thermal"   x="-30"   y="29" />
     <port id="16"   name="C3" direction="in"   type="thermal"   x="-30"   y="30" />
     <port id="17"   name="C4" direction="in"   type="thermal"   x="-30"   y="31" />
     <port id="18"   name="C5"                        type="open"       x="-30"   y="32" />
  <!--net name="DN"         type="signal" >
     <port id="19"   name="D1" direction="out" type="signal"    x="-29"   y="28" />
     <port id="20"   name="D2" direction="in"   type="thermal"   x="-29"   y="29" />
     <port id="21"   name="D3" direction="in"   type="thermal"   x="-29"   y="30" />
     <port id="22"   name="D4" direction="in"   type="thermal"   x="-29"   y="31" />
  <!--net name="VDDC"      type="power"   >
     <port id="23"   name="D5" direction="inout"   type="power"      x="-29"   y="32" />
  <!--net name="VSS"      type="ground"   >
     <port id="24"   name="E1" direction="inout"   type="ground"    x="-28"   y="28" />
  <!--net name="DBUS[3]" type="signal" >
     <port id="25"   name="E2" direction="inout" type="signal"    x="-28"   y="29" />
  <!--net name="DBUS[2]" type="signal" >
     <port id="26"   name="E3" direction="inout" type="signal"    x="-28"   y="30" />
```

```
    <!--net name="DBUS[1]" type="signal" >
      <port id="27"   name="E4" direction="inout" type="signal"     x="-28"   y="31" />
    <!--net name="DBUS[0]" type="signal">
      <port id="28"   name="E5" direction="inout" type="signal"     x="-28"   y="32" />
  </socket>
  <socket inst="BBOX">
    <!--net name="DBUS[0]" type="signal">
      <port id="29"   name="A1" direction="inout"   type="signal"   x="37.5"   y="-37.5" />
    <!--net name="DBUS[1]" type="signal" >
      <port id="30"   name="A2" direction="inout"   type="signal"   x="38.0"   y="-37.5" />
    <!--net name="DQ"          type="signal" >
      <port id="31"   name="A3" direction="in"      type="signal"   x="38.5"   y="-37.5" />
    <!--net name="DBUS[2]" type="signal" >
      <port id="32"   name="B1" direction="inout"   type="signal"   x="37.5"   y="-38.0" />
    <!--net name="DBUS[3]" type="signal" >
      <port id="33"   name="B2" direction="inout"   type="signal"   x="38.0"   y="-38.0" />
    <!--net name="DN"          type="signal" >
      <port id="34"   name="B3" direction="in"      type="signal"   x="38.5"   y="-38.0" />
    <!--net name="VDDC"      type="power"   >
      <port id="35"   name="C1" direction="inout"   type="power"   x="37.5"   y="-39.5" />
    <!--net name="VSS"      type="ground"   >
      <port id="36"   name="C2" direction="inout"   type="ground"   x="38.0"   y="-39.5" />
      <port id="37"   name="C3"                       type="NC"   x="38.5"   y="-39.5" />
  </socket>

  <netgroup   name="XXBUS" type="BUS" skew="100p">
    <portref>P8</portref>
    <portref>P7</portref>
    <portref>P6</portref>
    <portref>P5</portref>
  </netgroup>
  <netgroup   name="XXDIFF" type="DIFF"   Zdiff="100" skew="100p">
    <portref>DQ</portref>
    <portref>DN</portref>
  </netgroup>
  <netgroup   name="VDDC" type="PGNET">
    <portref>VDDC</portref>
    <portref>SU</portref>
  </netgroup>

  <reference reffile="XML での階層間定義例_20110223_verilog.xml" format="JLPB">
<placement module="PKG">
```

```
    <rectangle llx="-3000" lly="-3000" urx="3000" ury="3000"    layer="L1" mount="TOP"/>
</placement>

    <connection inst="BGA" portref="E5">
      <JLPB:portref module="Package" portname="E5"/>
    </connection>
    <connection inst="BGA" portref="E4">
      <JLPB:portref module="Package" portname="E4"/>
    </connection>
    <connection inst="BGA" portref="E3">
      <JLPB:portref module="Package" portname="E3"/>
    </connection>
    <connection inst="BGA" portref="E2">
      <JLPB:portref module="Package" portname="E2"/>
    </connection>
    <connection inst="BGA" portref="C1">
      <JLPB:portref module="Package" portname="C1"/>
    </connection>
    <connection inst="BGA" portref="D1">
      <JLPB:portref module="Package" portname="D1"/>
    </connection>
    <connection inst="BGA" portref="A3">
      <JLPB:portref module="Package" portname="A3"/>
    </connection>
    <connection inst="BGA" portref="A2">
      <JLPB:portref module="Package" portname="A2"/>
    </connection>
    <connection inst="BGA" portref="D5">
      <JLPB:portref module="Package" portname="D5"/>
    </connection>
    <connection inst="BGA" portref="B1">
      <JLPB:portref module="Package" portname="B1"/>
    </connection>
    <connection inst="BGA" portref="B5">
      <JLPB:portref module="Package" portname="B5"/>
    </connection>
    <connection inst="BGA" portref="A5">
      <JLPB:portref module="Package" portname="A5"/>
    </connection>
    <connection inst="BGA" portref="E1">
      <JLPB:portref module="Package" portname="E1"/>
    </connection>
```

```
</reference>

  <reference reffile="sample_verilog.txt" format="verilog">
  <placement module="BBOX">
    <rectangle llx="-2000" lly="-2000" urx="2000" ury="2000" layer="L1" mount="TOP" />
  </placement>

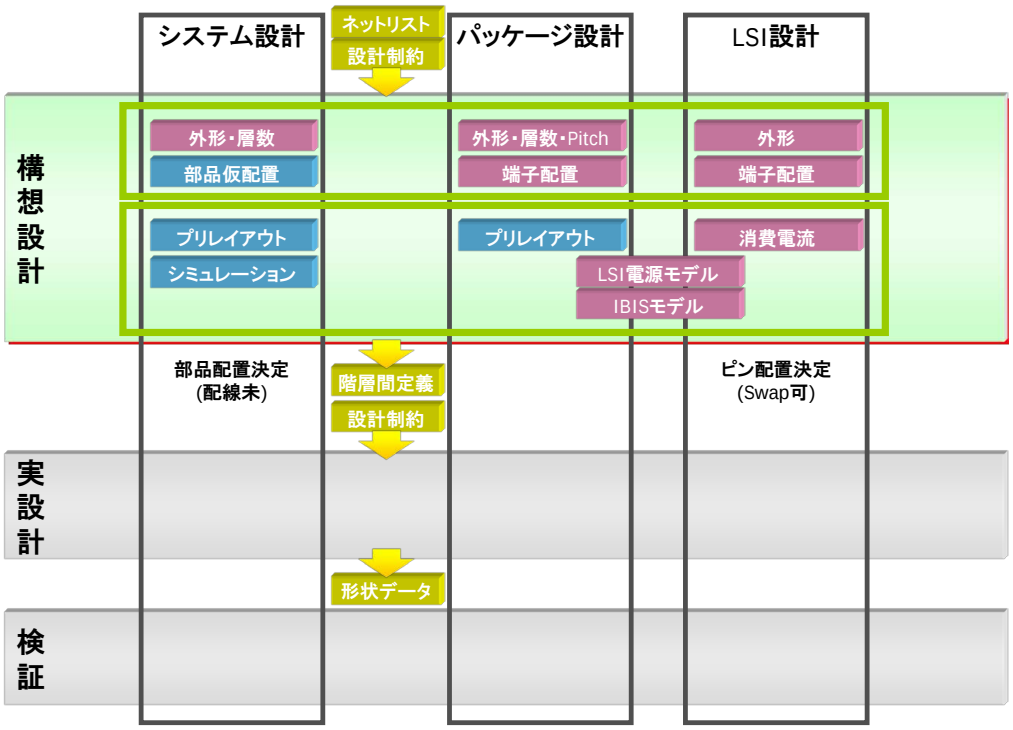
      <connection inst="BBOX" portref="A1">
      <verilog:portref module="BBOX" portname="DBUS[0]"/>
    </connection>
      <connection inst="BBOX" portref="A2">
      <verilog:portref module="BBOX" portname="DBUS[1]"/>
    </connection>
      <connection inst="BBOX" portref="B1">
      <verilog:portref module="BBOX" portname="DBUS[2]"/>
    </connection>
      <connection inst="BBOX" portref="B2">
      <verilog:portref module="BBOX" portname="DBUS[3]"/>
    </connection>
      <connection inst="BBOX" portref="A3">
      <verilog:portref module="BBOX" portname="A3"/>
      <verilog:portref module="BBOX" portname="DQ"/>
    </connection>
      <connection inst="BBOX" portref="B3">
      <verilog:portref module="BBOX" portname="DN"/>
    </connection>
      <connection inst="BBOX" portref="C1">
      <verilog:portref module="BBOX" portname="VDD"/>
    </connection>
      <connection inst="BBOX" portref="C2">
      <verilog:portref module="BBOX" portname="VSS"/>
    </connection>
</reference>

<reference reffile="sample_verilog.txt" format="verilog">
      <connection inst="SUPPLY" portref="iVDDC">
      <verilog:portref module="PWB" portname="iVDDC"/>
    </connection>
      <connection inst="SUPPLY" portref="iVDDQ">
      <verilog:portref module="PWB" portname="iVDDQ"/>
    </connection>
```

```
        <connection inst="SUPPLY" portref="iVSS">
        <verilog:portref module="PWB" portname="iVSS"/>
      </connection>
</reference>

</module>
</JEITA>
```

LPB リファレンスフロー 想定フロー



想定フローのInput/Output：ネットリスト関連

- 構想設計のInputとして、ネットリストがあること、を推奨する

		機能ブロック定義							機能ブロック参照・接続定義					
		機能ブロック名	入出力端子【ポート】名	入出力端子属性（入力／出力）	入出力端子属性（信号／電源）	入出力端子属性（束線）	機能（動作記述）	機能ブロック参照（ネスト）	参照機能ブロック名	配置【インスタンス】名	接続入出力端子名指定	端子間接続【ネット】名	端子間接続属性（信号／電源）	端子間接続属性（束線）
構想設計	Input													
	Output													
実設計	Input													
	Output													

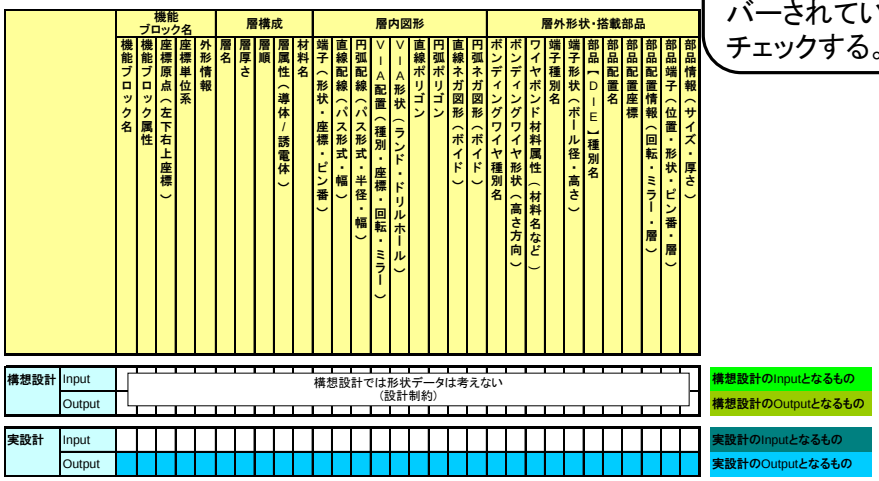
構想設計のInputとなるもの
構想設計のOutputとなるもの
実設計のInputとなるもの
実設計のOutputとなるもの

•制定されたフォーマットがカバーする範囲を色分けする。
•すべての項目がカバーされているかをチェックする。

想定フローのInput/Output：形状データ関連

- 形状データは、構想設計では考えない（構想設計は設計制約ベース）
  形状データは、全て実設計のOutputとなる

•制定されたフォーマットがカバーする範囲を色分けする。
•すべての項目がカバーされているかをチェックする。



想定フローのInput/Output：階層間定義関連

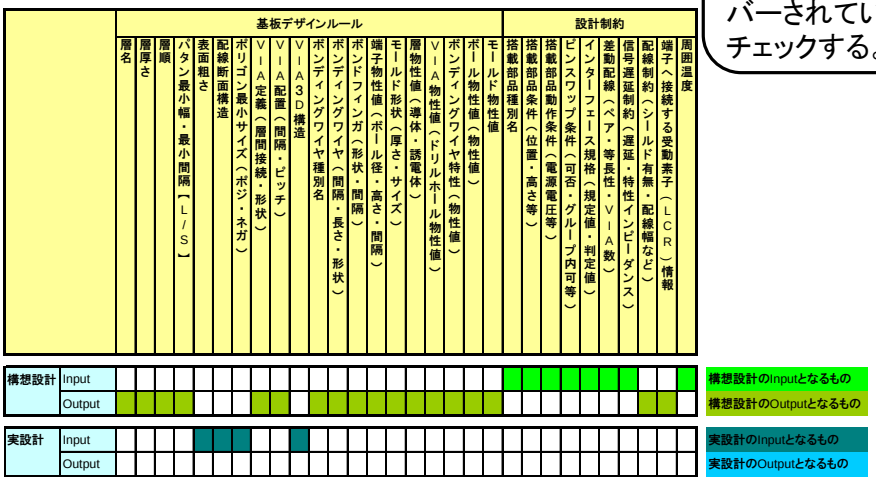
- 既存搭載部品の情報は、構想設計のInputとなる
  それ以外は、構想設計のOutput

•制定されたフォーマットがカバーする範囲を色分けする。
•すべての項目がカバーされているかをチェックする。

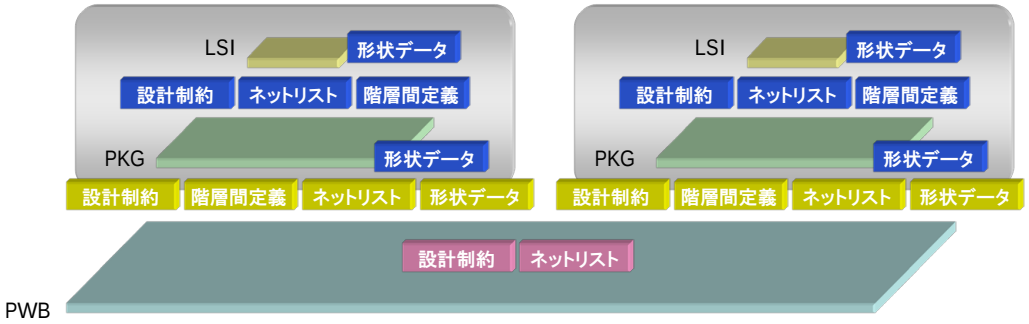
想定フローのInput/Output：設計制約関連

・ **設計制約は、基本的には構想設計のOutput**

•制定されたフォーマットがカバーする範囲を色分けする。
•すべての項目がカバーされているかをチェックする。



LSI-PKG-PWBとインターフェースファイル



•制定されたフォーマットがどの接続部分で機能するかを図式化する。

EDAアニュアルレポート　2010

2011年6月発行

禁無断転載

発　　行　　一般社団法人 電子情報技術産業協会　電子デバイス部
〒100-0004
東京都千代田区大手町一丁目1番3号
大手センタービル
電話　03-5218-1061　　FAX　03-5218-1080

作　　成　　三協印刷株式会社
〒152-0002
東京都目黒区目黒本町5-20-7
電話　03-3793-5971　　FAX　03-3793-6242

Copyright 2010 by Japan Electronics and Information Technology Industries Association

本書中に記載の会社名および商標名は、各社の登録商標、商標です。