JEITA

EDAアニュアルレポート　2009

Annual Report on Electronic Design Automation

－　65nmから45nmテクノロジ世代のEDA技術の進展に向けて　－

2010年5月発行

作　　成

EDA技術専門委員会

EDA Technical Committee

発　　行

社団法人　電子情報技術産業協会

Japan Electronics and Information Technology Industries Association

【巻頭言】

「高付加価値なLSIの設計を支えるEDA技術の発展に向けて」

ＥＤＡ技術専門委員会　2009年度　委員長　　太田 光保

半導体を利用する産業の裾野は、情報機器・ネットワーク・車・医療等と広がり、その市場規模は非常に大なるものとなっている。半導体産業には、それらの産業の発展を支える基盤として、LSI の更なる高集積化・高性能化・低消費電力化等々が期待され、その果たすべき役割は益々高まっている。その中で、半導体の自動設計(EDA : Electronic Design Automation)技術分野には、それらを実現する鍵として、機能・論理の設計・検証を少ない期間・工数で高品質に実現する技術、微細化に伴う物理現象に対応した設計・検証を実現する技術、消費電力が少ない回路の設計・検証を支援・自動化する技術等々、多くの対応が求められている。また、これらの要求に対応して行くために、国内の半導体産業、大学、官庁間との密な連携はもとより、海外の関連業界・機関とも国際的な視野で協調、連携を図ることが重要となっている。

EDA 技術専門委員会は、電子情報技術産業協会（JEITA）における活動組織の一つとして、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、国内外の関係業界の発展に寄与することを目的とし、次の三つのテーマに取り組んでいる。

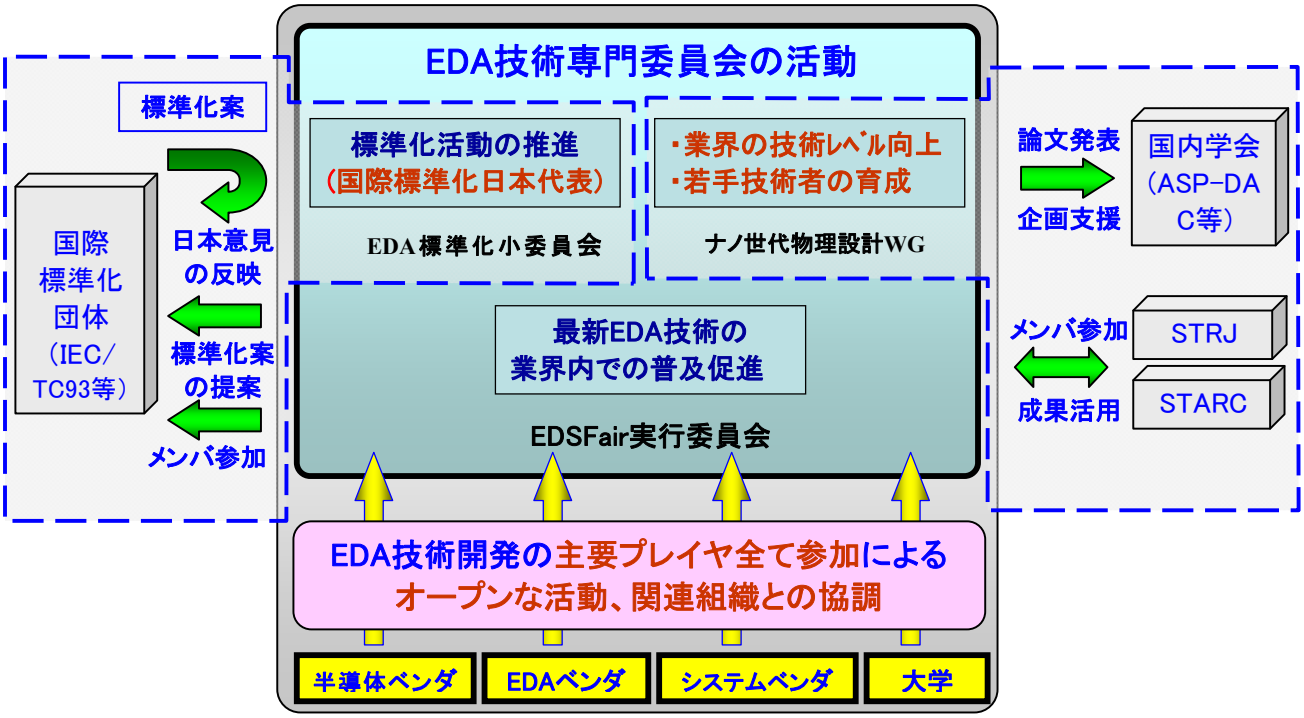
① システム LSI 設計技術に関する動向、関連情報についての調査・検討、課題解決への提案
② EDA 技術に関する標準化活動と、関連機関・団体への協力と貢献
③ EDA 技術および標準化の普及・推進のための、イベント開催・支援

①のテーマでは、ナノ世代物理設計ワーキンググループが、SSTA における配線ばらつきの取り扱い、配線セルフヒートのインパクト、統計的リーク電流解析手法などの、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。

②のテーマでは、EDA 標準化小委員会が、IEC、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 (International Electrotechnical Commission) TC93/WG2 の国内委員会（電子情報通信学会内に設置）としての役割も果たしている。また、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。

そして、③のテーマとしては、本年度も Electronic Design and Solution Fair2010(EDSFair2010)を、パシフィコ横浜にて開催した。この EDSFair は、電子機器の設計技術に関する展示会としては国内随一のもので、展示に加えて、会場内特設ステージでの当委員会企画のイベントや、同時開催される FPGA コンファレンス等々を通して、LSI 設計者に、最新の EDA 技術や関連情報との出会いの場の提供を行った。

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、高付加価値な LSI の設計を支える EDA 技術の発展に向け、ひいては日本の電子情報機器業界の発展に寄与すべく、本年度 16 社約 60 名の業界各社・有志メンバーの参画で運営してきた。

また、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2010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2009」は、EDA技術専門委員会の2009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)

2009 年度 JEITA/EDA 技術専門委員会 委員一覧

委員長	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　参事
副委員長	山本　一朗	ローム㈱	ＯＫＩセミコンダクタ㈱ 開発本部　開発システムユニット デザインクオリティ開発チーム チームリーダー
副委員長	青野　宏二	セイコーエプソン㈱	半導体事業部　ＩＣ事業推進部 部長
監事	山田　節	三洋電機㈱	研究開発本部 デジタル技術研究所 担当課長（ＬＳＩ設計技術担当）
幹事	吉田　正昭	ＮＥＣエレクトロニクス㈱	基盤技術開発本部 設計技術開発部 シニアプロフェッショナル
幹事	長尾　明	シャープ㈱	電子デバイス事業本部 ＮＢ事業化推進センター 新規モジュール開発部　副参事
幹事	齋藤　茂美	ソニー㈱	半導体事業本部　設計基盤技術部門 業務推進室 （2010 年 1 月退任、引き続き 特別委員就任）
幹事	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 （2010 年 3 月新任）

幹事	今井　浩史	㈱東芝	セミコンダクター社 システムＬＳＩ設計技術部 設計メソドロジ技術開発担当　参事
幹事	河村　薫	富士通マイクロエレクトロニクス㈱	共通テクノロジ開発統括部 主席部長
幹事	井下　順功	㈱ルネサステクノロジ	設計開発本部　設計技術統括部 ＤＦＭ／ディジタル ＥＤＡ技術開発部 グループマネージャ
幹事	江田　努	ローム㈱	ＹＴＣ開発システムユニット ＬＳＩ企画推進Ｇ　次席技術員
委員	野坂　啓介	図研エルミック㈱	営業本部　ＥＤＡ営業部 マネージャー
委員	下出　隆文	三洋半導体㈱	技術製造統括部 設計技術部 設計技術部　部長
委員	山城　治	㈱ジーダット	取締役
委員	山田　陽一	セイコーエプソン㈱	半導体事業部 ＩＣ設計技術センター　主事
委員	山﨑　安秀	凸版印刷㈱	トッパン・テクニカル・ デザインセンター 事業企画部　課長
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	三橋　明城男	メンター・グラフィックス・ジャパン㈱	フィールド・マーケティング・ マネージャー

委員	前野　酉治	㈱リコー	電子デバイスカンパニー 画像ＬＳＩ開発センター ＣＡＤ技術室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　ＣＷＢ事業推進室 テクニカルマーケティング ディレクター
特別委員	中森　勉	富士通マイクロエレクトロニクス㈱	基盤技術統括部 設計効率推進部　プロジェクト課長
特別委員	立岡　真人	富士通マイクロエレクトロニクス㈱	設計共通技術統括部 第一設計部
特別委員	金本　俊幾	㈱ルネサステクノロジ	設計開発本部　設計技術統括部 ＳＩＰ・アナログＥＤＡ技術開発部 主任技師
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

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（1985）
Commerce At Light Speed（1995）

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

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 事業計画

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	16 社/19 名	
特別委員	4 名	
客員	4 名	

委員会の役員

委員長:	パナソニック	太田　光保
副委員長(正):	ローム	山本　一郎
副委員長(代行):	セイコーエプロン	青野　宏二
監事:	三洋電機	山田　節

下部組織の役員

EDA 標準化小委員会	主査	ローム	山本　一郎
SystemC　WG	主査	東芝	今井　浩史
SystemVerilog　WG	主査	パナソニック	浜口　加寿美
Power Format WG	主査	富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	中森　勉
ナノ世代物理設計 WG	主査	ルネサステクノロジ	金本　俊幾
EDSFair 実行委員会	委員長	ソニー	齋藤　茂美
EDSFair 企画　WG	主査	三洋電機	山田　節
SDF　WG	主査	セイコーエプソン	山田　陽一

<方針>
1.情報収集力の強化(PowerFormat 取組みの反省)
　IEEE-SA の加入継続
　IEEE/DASC への加入検討
　国際会議(TC93, DASC)への積極的参加と国内開催 (WG2)
　標準化団体との定期的な会議開催
2. 委員会活動の活性化
　委員会を主体にした運営 EDSFair に係る JESA, ASP-DAC, FPGA/PLD Conf. などとの連携を再構築とイベント運営の見直し
　HP を活用したドキュメントの電子化 (情報共有強化)
　ペーパーレス化の推進
3. 委員会の継続的発展
　新公益法人制度への対応
　予算の最適運用

EDA 技術専門委員会メンバと担当　(敬称略)

委員長:	パナソニック	太田　光保	EDSFair/ASP-DAC 小委員会 主査、ASP-DAC OC EDSFair 検討会主査
副委員長:	ローム	山本　一郎	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:	セイコーエプソン	青野　宏二	EDA標準化小委員会 副主査、内規改訂、
監事:	三洋電機	山田　節	EDSFair 企画 WG 主査、EDSFair 実行委員
幹事:	ソニー	齋藤　茂美	EDSFair 実行委員長、SDF WG 委員
幹事:	富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	河村　薫	EDSFair 実行委員、広報パンフレット、ASP-DAC OC
幹事:	NECエレクトロニクス	吉田　正昭	EDSFair 企画 WG 委員、(ASP-DAC2010 Industry Liaison) システム･デザイン･フォーラム WG 委員
幹事:	ルネサステクノロジ	井下　順功	EDSFair 企画 WG 委員
幹事:	ローム	江田　努	
幹事:	シャープ	長尾　明	アニュアルレポート、EDSFair 企画 WG 委員 システム･デザイン･フォーラム WG 委員
幹事:	東芝	今井　浩史	ホームページ、メールシステム、予算実績管理
委員:	図研エルミック	野坂　啓介	
委員:	三洋半導体	下出　隆文	
委員:	ジーダット	山城　治	
委員:	セイコーエプソン	山田　陽一	SDF WG 主査、EDSFair 実行委員
委員:	凸版印刷	山﨑　安秀	
委員:	日本シノプシス	飯島　一彦	
委員:	ﾒﾝﾀｰｸﾞﾗﾌｨｯｸｽｼﾞｬﾊﾟﾝ	三橋　明城男	EDSFair 企画 WG 委員
委員:	リコー	前野　酉治	
特別委員:	NEC ｼｽﾃﾑﾃｸﾉﾛｼﾞｰ	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：	富士通ﾏｲｸﾛｴﾚｸﾄﾛﾆｸｽ	中森　勉	Power Format WG　主査
特別委員：	東芝	今井　浩史	SystemC WG　主査
特別委員：	ルネサステクノロジ	金本　俊幾	ナノ世代物理設計 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	古川　昇	
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　230,000 円 ＊16 社 =3,680,000 円

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
(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-DAC2010, DAC2009, IEICE, 軽井沢ワークショップ

2. 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, SystemVerilog, PowerFormat を継続
　・Verilog-HDL,VHDL,A-HDL 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
　・IEC/TC93 国際会議(9 月@京都)への参加
　・WG2 会議を開催(7 月@DAC(US), 1 月@EDSFair(Yokohama), 2 月@DVcon(US))
　・IEEE/DASC,IEEE-SA, OSCI, Accellera, Si2 等との連携を強化
　・IEEE/DASC への加入を検討

3. 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) 「EDSFair2010」(横浜)
　・日本エレクトロニクスショー協会へ運営委託
　・半導体部会/技術委員会を活用した各社への依頼とアナウンス
　・ASP-DAC, FPGA/PLD Conference との連携と新企画と提案
　・特設ステージの継続実施
(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 2010」を EDSFair2010 と同時開催
(3) 「ASP-DAC2010」(台湾)
　・Designer’s Forum との連携

4. 委員会活動の広報

(1) 広報パンフレットの配布@EDSFair2010
(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
　・学会での公演/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
2009/4	4/24(金)（東京地区） 議事録 ﾙﾈｻｽﾃｸﾉﾛｼﾞ ・09 年度事業計画説明、承認 ・09 年度小委員会/WG 計画説明、承認 ・08 年度会計収支と 09 年度会計予算説明、承認 ・08 年度版アニュアルレポート作成状況報告		・DATE2009（4/20-4/24） @ Nice ・軽井沢 WS (4/20-21)　@軽井沢
2009/5			
2009/6	6/19(金)（東京地区） 議事録 東芝 ・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 ・予算消費状況		・DAC2009 （7/26-7/31）@San Francisco, CA
2009/7			・STARC Forum&Symp. (8/25) @新横浜
2009/8			・DA シンポジウム 2009 (8/26-27) @加賀
2009/9	9/18(金) （関西地区） 議事録 NEC エレ ・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		・TC93 国際会議 (9/28-30) @京都
2009/10			
2009/11	11/20(金)（東京地区） 議事録 凸版印刷 ・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 ・EDSFair 用パンフレット作成手順説明		・ICCAD2009 (11/11-14) @Prague, Czech Republic
2009/12			・デザインガイア(12/2-4)@　高知
2010/1	1/15(金) （関西地区） 議事録 シャープ ・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 ・手順説明・予算消費状況		・ASP-DAC2010 (1/18-21) @台湾 ・EDSFair2010 (1/28-29) @横浜
2010/2			
2010/3	3/19(金) （東京地区） 議事録 日本シノプシス ・半導体部会/半導体技術委員会報告内容説明 ・09 年度小委員会/WG の年間活動報告 ・09 年度予算消費状況 ・10 年度事業計画説明 ・10 年度プロジェクト/研究会の年間活動計画説明	○	

1.2　2009 年度 JEITA/EDA 技術専門委員会ホームページ

1.2.1　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いただくことを目的とする。

1.2.2　ホームページの詳細

2006 年度よりホームページを一新し、よりわかりやすく、また、欲しい情報に簡易にアクセスできるような構成に変更を行った。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。ホームページより本委員会の成果であるドキュメントを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/

大阪大学のご協力を頂き、大阪大学のサーバーにホームページを設置させていただいてきた。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 年度にサーバーを外部業者のレンタルサーバーへ移行した。

（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

下記の研究会・小委員会等の活動状況を紹介している。

- EDA 標準化小委員会（EDA 標準化小委員会の他、SystemC WG、SystemVerilog WG、PowerFormat WG の活動が紹介されている）
- 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、英語版も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

関連の会議としては、次の関係の深い EDA 関連技術委員会の紹介が行われている。

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

また、関連機関として本委員会に関連のある 17 機関の紹介があり、さらにそれぞれのホームページへのリンクが行われている。

1.3　2009 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
2009 年 4 月	4/24（金）　11:00-13:30	4/24（金）　14:00-17:00　JEITA　504 会議室
5 月		
6 月	6/19（金）　11:00-13:00	6/19（金）　14:00-17:00　JEITA　506 会議室
7 月		
8 月		
9 月	9/18（金）　11:00-13:00	9/19（金）　14:00-17:00　電子会館　４階会議室
10 月		
11 月	11/20（金）　11:00-13:00	11/20（金）　14:00-17:00　JEITA　506 会議室
12 月		
2010 年 1 月	1/15（金）　11:00-13:00	1/15（金）　14:00-17:00　JEITA 関西支部　第２会議室
2 月		
3 月	3/19（金）　11:00-13:00	3/19（金）　14:00-17:00　日本教育会館　705 号室

月	EDA 標準化小委員会関連
2009 年 4 月	4/17(金) 13:00-17:00　第 1 回 SystemC-WG　日本教育会館　902 号室
5 月	5/29(金) 11:00-13:00 第 1 回 EDA 標準化小委員会　機会振興会館　64 号会議室
6 月	
7 月	7/24(金) 13:00-17:00　第 2 回 SystemC-WG　JEITA　502 会議室
8 月	8/7(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
9 月	9/4(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 9/11(金) 14:00-17:00　第 3 回 SystemC-WG　JEITA　503 会議室
10 月	10/16(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 月	11/13(金) 13:00-17:00、11/14(土) 9:00-12:00 SystemC-WG 集中審議 ヴィラ志摩
12 月	
2010 年 1 月	1/22(金) 14:00-17:00　第 4 回 SystemC-WG　JEITA　503 会議室
2 月	
3 月	3/12(金) 14:00-17:00　第 5 回 SystemC-WG　日本教育会館　808 号室 3/26(金) 11:00-13:00 第 5 回 EDA 標準化小委員会　機会振興会館　69 号会議室

月	ナノ世代物理設計 WG 関連
2009 年 4 月	
5 月	5/27(水) 10:00-17:00　第 1 回ナノ世代物理設計 WG　JEITA　503 会議室
6 月	6/26(金) 13:00-17:00　第 2 回ナノ世代物理設計 WG　日本教育会館　705 号室
7 月	
8 月	8/28(金) 10:00-17:00　第 3 回ナノ世代物理設計 WG　JEITA　関西支部　第一会議室
9 月	
10 月	10/30(金) 10:00-17:00　第 4 回ナノ世代物理設計 WG　JEITA　507 会議室
11 月	11/26(木) 10:00-17:00　第 5 回ナノ世代物理設計 WG　JEITA　関西支部　第一会議室
12 月	12/22(火) 10:00-17:00　第 6 回ナノ世代物理設計 WG　JEITA　507 会議室
2010 年 1 月	1/22(金) 10:00-17:00　第 7 回ナノ世代物理設計 WG　JEITA　507 会議室
2 月	2/26(金) 10:00-17:00　第 8 回ナノ世代物理設計 WG　JEITA　関西支部　第二会議室
3 月	3/26(金) 10:00-17:00　第 9 回ナノ世代物理設計 WG　日本教育会館　902 号室

月	EDS Fair 実行委員会関係
2009 年 4 月	

5 月	
6 月	
7 月	7/29(水) 15:00-20:00　第 1 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
8 月	8/28(金) 14:00-18:30　第 2 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
9 月	
10 月	10/15(木) 14:00-20:30　第 3 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	
12 月	12/16(水) 13:00-18:00　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
2010 年 1 月	
2 月	
3 月	3/5(金) 13:30-17:00　第 5 回 EDSFair2010 実行委員会　機械振興会館　62 号室

月	関連行事
2009 年 4 月	
5 月	
6 月	
7 月	7/23(木) 13:00-17:00　第 1 回 LPB 相互設計準備 WG　JEITA　409 会議室
8 月	8/27(木) 13:00-17:00　第 2 回 LPB 相互設計準備 WG　九段会館　あおいの間
9 月	9/17(木) 13:00-17:00　第 3 回 LPB 相互設計準備 WG　JEITA　409 会議室
10 月	10/ 8(木) 13:00-17:00　第 4 回 LPB 相互設計準備 WG　JEITA　406 会議室
11 月	11/ 5(木) 13:00-17:00　第 5 回 LPB 相互設計準備 WG　JEITA　409 会議室
12 月	
2010 年 1 月	1/20(水) 13:00-17:00　第 6 回 LPB 相互設計準備 WG　JEITA　409 会議室
2 月	
3 月	3/10(水) 13:00-17:00　第 7 回 LPB 相互設計準備 WG　日本教育会館　810 号室

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、これらの課題を解決するため各社が開発した手法やライブラリが、そのテクノロジが一般化した後も標準化されず、設計環境の開発・サポートコスト低減の障害となる事例や、半導体ベンダと顧客との情報授受がスムーズに行えない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(45 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
- 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。

- 製造ばらつきに起因するリーク電流変動の低減法
- 32nm プロセスにおける配線自己発熱の信号伝播遅延に対するインパクト調査
- 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)　RTN を考慮した回路特性ばらつき解析方法の検討

ランダムテレグラフノイズ（RTN）は素子の微細化に伴って、新たなデバイス・回路特性のばらつき要因として注目を集めている。本年度は、RTN を取り巻く確率・統計的側面を分析することにより、RTN が回路特性に与える影響を考察した。RTN のモンテカルロ解析アルゴリズムを新たに提案し、RTN はチップ内の欠陥歩留まりと同等の現象としてとらえるべきこと、すなわち RTN が検出の難しい内在欠陥の問題であり、RTN バーイン、欠陥素子のスクリーニングの課題に帰着することを指摘する。

(1)　RTN を簡易的にモンテカルロ解析するアルゴリズムを示した。
(2)　RTN はデジタル回路の遅延ばらつきを平均的に±1%程度大きくするだけである。
(3)　本実験の試行回数においては、10ppm の割合で、遅延時間が 2～4 倍となるサンプルが存

在した。

RTN は時間的に CMOS の特性を変化させる現象とみなされているが、LSI の欠陥と同様歩留まりに関わる扱いが必要になる。作りこまれた RTN 欠陥が検出されずに市場に出てから発見されることが最も懸念される状況である。その意味で、製品の最終テスター評価で、RTN ワースト状態が実現されることが対応できる最も重要なテーマになると思われる。

RTN の電子捕獲・放出時定数は共にミリ Sec のオーダーである。物理的に考えランダムな現象（指数分布に従うと言うのはその現われ）であり、高温においてこの電子捕獲・放出時定数は小さく出来ると考えられる。すなわち、RTN ワーストの状態の起こる確率が大きく出来る。

今後、統計的なデータ処理とあいまって、RTN ワーストの加速(エージング)方法やｐｐｍレベルの検出技術が大切になってくると思われる。

(2) 配線ばらつきを表現する Sensitivity SPEF フォーマットの調査

IEEE1481 の SSPEF 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 RLC の感度を線型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が実用になる寸法ばらつきの範囲を明確にし、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。

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論文誌にて発表した。

[1] “An Approach for Reducing Leakage Current Variation due to Manufacturing Variability,” IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, Vol.E92-A, No.12, Dec. 2009.

[2] “Impact of Self-heating in Wire Interconnection on Timing,” IEICE Trans. on Electronics, Vol.E93-C, No.3, Mar. 2010.

また、本年度は EDS フェア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、45nm 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	金 本 俊 幾	㈱ルネサステクノロジ
副 主 査	田 中　正 和	パナソニック㈱
委　　員	佐　方　　剛	富士通マイクロエレクトロニクス㈱
同	黒 川　　敦	三洋電機㈱
同	古 川　且 洋	㈱ジーダット
同	山 中　俊 輝	㈱リコー
同	増 田 弘 生	㈱ルネサステクノロジ
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

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。

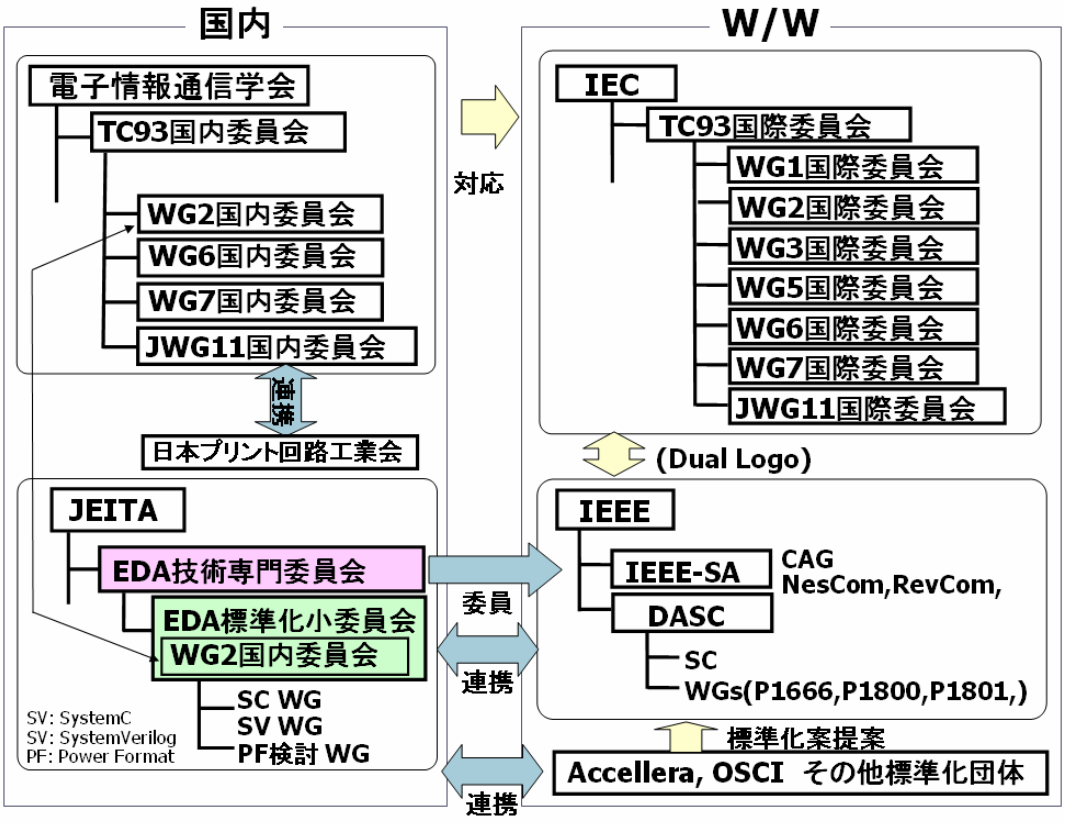
近年、設計記述言語は高度化し、普及が進んだ。しかし、設計生産性の更なる向上及び、それを支える EDA ツールの効率的な開発・利用を進めるためには、設計技術言語の国際標準化は依然として重要なテーマである。そこで、標準化関連の活動をより明確に位置づけるため、2000 年 11 月に本小委員会が設立された。

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）、及び 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 及び 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。（SystemVerilog ワーキンググループは、ミッションを完了したため 2008 年度末に解散した）



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 及び関連技術の標準化に関して、

・内外の動向を調査、検討し、
・技術及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

主査	山本　一郎	ロームグループ　OKI セミコンダクタ（株）
副主査	青野 宏二	セイコーエプソン（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）
委員	吉田　正昭	NEC エレクトロニクス（株）
委員	太田　光保	パナソニック（株）
委員	江田　努	ローム（株）
委員	山田　節	三洋電機（株）
委員	長尾 明	シャープ（株）
委員	齋藤 茂美	ソニー㈱（2010 年 1 月退任）
委員	大芝 克幸	ソニー㈱（2010 年 3 月新任）
委員	今井 浩史	（株）東芝
委員	河村　薫	富士通マイクロエレクトロニクス（株）
委員	井下 順功	（株）ルネサステクノロジ
特別委員	相京　隆	（株）半導体理工学研究センター （TC93/WG2　Member）
特別委員	長谷川　隆	富士通マイクロエレクトロニクス（株） （TC93/WG2　Member）
特別委員	浜口加寿美	パナソニック（株） （SystemVerilog WG 主査 TC93/WG2　Member）
特別委員	中森　勉	富士通マイクロエレクトロニクス（株） （Power Format WG 主査）
特別委員	金本　俊幾	（株）ルネサステクノロジ （ナノ世代物理設計 WG 主査）
特別委員	星野　民夫	（株）アプリスター （TC93/WG2　Member）
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） （TC93/WG2　Member）
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）２００９年度活動　　国内活動

EDA標準化小委員会としては、年６回の会合を行い、傘下のSystemCワーキンググループ、Power Format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、標準化活動に関する新たな試みとして、①「LSI、パッケージ、PCBを相互的に扱う設計環境」に関する検討及び、②設計言語俯瞰図の作成を行った。

また、「JIS C 0617電気用図記号の改正事業（電気用図記号標準化委員会）」に、山本主査をJEITA代表の委員として派遣し、電気用図記号標準（IEC60617）における、二値論理素子の日本語化（JIS規格化）を実現する作業を実施した。

①LSI、パッケージ、PCB を相互的に扱う設計環境：

LSI、パッケージ、PCBを相互的に扱う設計環境（以下､LPB統合設計環境）に関する検討の必要性を調査するため、LSIーパッケージーPCB統合設計環境（言語）標準化に関する準備WGを発足させた。この準備WGの結論として、「LPB統合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」と判明したため、LPB統合設計環境に対する本格検討に向け新たなワーキンググループを立ち上げると決定し、EDA技術専門委員会の了承を得た。

②設計言語俯瞰図：

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た。この俯瞰図の利用により、「ひとつの分野に、複数の言語が存在する」などの課題を把握することが容易となった。なお、設計言語俯瞰図をIEC/TC93国際会議で紹介し、この資料の活用を合意した。

（4）２００９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 京都国際会議への参加、JEITA-IEE/DASC 情報交換会の開催などを通じて、活発な交流を行った。

①IEC/TC93 京都国際会議：

2009 年 9 月の IEC/TC93 京都国際会議には、本委員会から TC93 国内委員会委員長である神戸客員、WG2 国際コ・コンベナの小島副主査、EDA 標準化小委員会主査の山本、他 2 名の５名が出席した。

②IEEE DASC との情報交換会

2009 年 6 月に Dennis Brophy 氏（Accellera 副議長）を招き、EDA 技術専門委員会及び、EDA 標準化小委員会の委員と情報交換会を開催した。会議では、Brophy 氏による講演（Accellera/IEEE 標準化活動の現況）の後、参加者を交えて情報の交換を図った。

③JEITA-IEEE/DASC 情報交換会

2010 年 1 月には、IEEE/DASC の Stan Krolikoski 委員長他を招き、DASC との情報交換会を日本で開催した。

④IEC/TC93/WG2 国際連携

小島副主査（ WG2 コ・コンベナ）を、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。

・DAC Meeting　　7 月 30 日　　サンフランシスコ

・DVCon Meeting　　2 月 23～25 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、EDSFair2009 合わせて 2010 年 1 月 29 日に IEEE/DASC との情報交換会を日本で開催した。会議には、米国からは 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、山本主査、小島副主査、神戸客員、今井委員（兼　SystemC WG 主査）、齋藤委員、河村委員、金本 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、SSPEF の標準化マイルストーン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 103 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 月シンガポールでの開催があり、本年度は、9 月に京都の京都リサーチパークで開催された。

（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、昨年度、韓国が O メンバーから P メンバーに変わったことで、P メンバーが 4 カ国、O メンバーが 20 カ国となった。韓国以外の P メンバーとしては、日本、中国、米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。幹事国は米国（国際幹事：Victor Berman 氏）が、国際会議議長は日本（唐津氏）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：ハードウェア設計記述言語：

（a）VHDL 言語仕様、Verilog HDL の整合性等の検討、システム記述言語（SLDL）も議題に取り上げられてきた。（b）IC delay & power calculation system の検討。 日本からの提案 ALR 標準化;IS（国際規格）化完。現在は SystemC、SystemVerilog、Power Format が中心。

・WG3：設計データ交換表現

PDX（Product data eXchange）によるマテリアルデクラレーション関連への対応の議論。

・JWG11：記述の XML 化の流れへの取り組み方の議論。

・WG5：規格適合性（コンフォマンス）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。

（4）TC93 国内委員会と主要メンバ（2010 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）

WG2 コ・コンベナ：　小島 智（NEC システムテクノロジー（株））*

＜国内専門委員会＞

委員長：　神戸 尚志（近畿大学）　*

幹事：　古井 芳春（(株) シルバコ・ジャパン）

委員：　唐津　治夢（ＳＲＩインターナショナル）、高橋 満（(社) 電子情報通信学会）、
小島 智（NEC システムテクノロジー（株））*、
柴田　明一（(社) 日本電子回路工業会）、
小泉　徹（(社) 日本電子回路工業会）、山下　寛巳（Ｓ・Ｍ・Ｌ（株））、
星野　民夫（(株) アプリスター）*
山本　一郎（ロームグループ　OKI セミコンダクタ（株））*

・WG1：（モデルのハーモナイゼーション）

主査：　古井 芳春（(株) シルバコ・ジャパン）

委員：　小島 智（NEC システムテクノロジー（株））*

・WG2：（ハードウェア設計記述言語）

主査：　山本　一郎（ロームグループ　OKI セミコンダクタ（株））*

委員：　小島 智（国際コ・コンベナ、NEC システムテクノロジー（株））*
EDA 標準化小委員会メンバー（客員、特別委員を含む）

・WG3：（設計データ交換表現）

主査：　高橋 満（国際コ・コンベナ、(社) 電子情報通信学会）

・WG5：（規格適合性（コンフォマンス）テスト）

主査：　神戸 尚志（近畿大学）　*

・WG6：（再利用可能部品ライブラリ）

主査：　高橋 満（国際コ・コンベナ）

・WG７：（ システムテスト記述言語）

主査：　唐津 治夢（ＳＲＩインターナショナル）

・JWG11：記述の XML 化の流れへの取り組み方の議論。

主査：　柴田　明一（(社) 日本電子回路工業会）

※１：国内委員会の各 WG については、主査及び EDA 標準化小委員会からの参加者を示す。

※２：「*」印は EDA 技術専門委員会からの参加者を示す。

（５）TC93 京都会議の報告

2009 年の国際会議は、9 月に京都の京都リサーチパークで開催された。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、2010 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、山本主査から EDA 技術標準化小委員会で作成した設計言語俯瞰図を紹介し、設計言語標準化のロードマップとして活用すると合意した。

なお、設計言語俯瞰図は WG1 でも紹介され、言語ハーモナイゼーションに利用すると合意された。

（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）IEEE 1666-2005（SystemC Language Reference Manual） ⇒ IEC61691-7

２）IEEE 1076-1-2007（VHDL Analog and Mixed-Signal Extensions） ⇒ IEC61691-6

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(旧 EIAJ)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++言語を基本とする 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 のコア言語部分の標準化が完了した。

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、標準化の活動が開始され、2011 年に投票が予定されている。

(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 年度の活動報告書に一覧を記載)、IEEE 並びに OSCI に報告した。

- 2004 年度は 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 に伝えた。
- 2007 年度は 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 に伝えた。
- 2008 年度は 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 に伝えた。（詳細は「4.2.4 OSCI TLM2.0 へのフィードバック」を参照）また、ユーザーズマニュアルの抄訳を作成し、本アニュアルレポートにて公開した。
- 2009 年度は TLM 2.0 LRM のレビューを実施し、問題点や要望実行を IEEE P1666 SystemC WG へフィードバックした。本標準化は 2010 年度も継続して実施される。

② SystemC 技術調査

- 2003 年 11 月度に集中審議を行い、本ワーキンググループ参加各社の SystemC 利用状況について紹介しあい、業界内の現状ステータスについて理解を深めた。
- 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る。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る。
- 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」としてまとめ、アニュアルレポートにて公開した。
- 2007年度は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、アニュアルレポートにて公開した。
- 2008年度は引き続きSystemCを用いた推奨設計メソドロジについて検討し、昨年作成した合成編以外の部分について審議を完了し、本アニュアルレポートにて公開し

た。

- 2009年度は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。

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いる。
- 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。今回より、受講料を徴収することにした。(SystemC 単独の場合¥1,600、SystemVerilog と通しの場合¥2,000) 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。今回は 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。今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 年のアンケート調査結果からは、次のような事が読み取れた。(詳細については 4.2.6 を参照)

  - 今年度は SystemC の利用経験がない方の参加が多かった。
  - 過去のアンケートで、SystemC を利用しない理由としては、「HDL で十分」「言語の完成度が低い」等の回答があったが、今回のアンケートでは、これらが大幅に減少する一方で、「効果が不明」「検討する時間がない」といった理由が多く挙がった。これは、SystemC-WG で開発中の推奨設計メソドロジが貢献できるものと期待している。
  - TLM の標準化が一段落したこともあり、標準化に対する興味よりは、適用事例に期待して参加する方が多かった。

- SystemC の利用状況について引き続き調査を実施した。今年度は SystemC ユーザ・フォーラムは開催しなかったが、2009 年 7 月に開催された SystemC Japan の場をお借りして実施した。

(4) 参加メンバー

主査	今井 浩史	東芝
副主査	中西 早苗	NEC エレクトロニクス
委員	大島 良紀	ルネサステクノロジ
	西園寺 修	日本シノプシス
	清水 靖介	ロームグループ・OKI セミコンダクタ
	竹村 和祥	パナソニック
	立岡 真人	富士通マイクロエレクトロニクス
	旦木 秀和	ソニー
	長尾 文昭	三洋半導体
	牧野 潔	メンターグラフィクスジャパン
客員	今井 正治	大阪大学
		（計 11 名）

2.2.5 PowerFormat 検討グループ報告

(1) 背景

近年、System-LSI に対する低消費電力化の要求はますます強くなってきている。 バッテリー駆動の携帯機器用のみならず、環境保護の面から家電製品やサーバー用途に至るまで全ての領域で低消費電力化はLSI 設計における最大の課題のひとつとなってきた。 一方で、性能向上の要求とプロセスの微細化によるチップあたりの回路規模の増加、さらにはリーク電流の増加が低消費電力化を困難な問題にしている。

その結果、LSI インプリ時において従来から適用されてきたクロックゲーティングやマルチ Vth といった低消費電力技術に加え、マルチ VDD・パワーゲーティング・バックゲートバイアス等の高度な技術が一般に適用される割合が飛躍的に増加している。

市販のインプリツールでもこれらの技術のサポートが進んできたが、従来の RTL では表現することが出来ない電源接続（や分離）に関する情報を記述する共通のフォーマットが設計における重要なポイントのひとつになってきた。

2006 年 6 月、CADENCE 社は 十数社の半導体ベンダー、EDA ベンダーと共に PFI(Power Forward Initiative) を組織し、電源接続やマルチ VDD, パワーゲーティングを表現可能な標準形式 CPF(Common Power Format)を策定した。 一方、SYNOPSYS と MENTOR はこの動きに対抗し、Accellera で UPF(Unified Power Format) を作り、２つの「標準」フォーマットが出来てしまった。

現在、CPF は Si2 にて Rev 1.1 が公開され、CADENCE をはじめとした複数の EDA ベンダーでサポートされている。 また、UPF は Accellera にて Rev 1.0 が公開された後、UPF-2.0 が IEEE-1801 として 2009 年 3 月 20 日に標準化された。 MENTOR、SYNOPSYS をはじめとした複数の EDA ベンダーでのサポートが UPF-1.0 をサポートしており、2.0 も 2010 年中にサポートされる見込みである。

(2) 目的

日本を始め多くの半導体ベンダーでは、複数の EDA ベンダーのツールを使用して LSI 設計を行っているため、２つの「標準」PF の存在は今後障害となることが予想される。 フローのある部分のツールは CPF, 別の部分のツールは UPF が必要となれば、結局フォーマットのコンバージョンが必要となり、手間が減らず互換性も問題となる。

PF を統一し、全てのツールがそれをサポートするのが理想ではあるが、既に CPF/UPF 共に仕様が固まっており、Si2, Accellera 両陣営も統一化に対しては後ろ向きであるため、現時点では非常に難しい。 ツールへの CPU/UPF インプリメントも進んでしまっている状況である為、たとえ今すぐフォーマットが統一されたとしても全ツールがそのサポートを完了するには１～２年の時間が必要で、その間はやはり両フォーマット間のコンバージョンが必要となることが予想される。

以上の理由から、早急なフォーマットの統一は目指さず、両フォーマットのインターオペラビリティの確保を第一の目的とする。

(3) 活動内容

2009年度は IEC, IEEE, PFI 共に動きが無かったため、本WGも活動は無かった。

(4) 関連機関の動向

(5) 参加メンバ

主　査	中森　勉	富士通マイクロエレクトロニクス株式会社
委　員	北原　健	株式会社東芝
同	熊野義則	株式会社リコー
同	中村正昭	三洋半導体株式会社
同	古本光昭	メンター・グラフィックス・ジャパン株式会社
同	安井卓也	パナソニック株式会社
同	山縣暢英	ソニー・エルエスアイ・デザイン株式会社
同	山田陽一	セイコーエプソン株式会社
客　員	神戸尚志	IEC/TC93 国内委員長　(近畿大学)
特別委員	小島　智	IEC/TC93/WG2 ココンベナ　(NEC システムテクノロジー)

３．各種イベント(主催／協賛)報告

3.1 Electronic Design and Solution Fair 2010 (EDSFair2010)

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、2010 年 1 月 28 日（木）～29 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 2010 (略称:EDSFair2010)」 を開催した。

出展社数は、113 社／団体、来場者 9,300 人であった。

今回は、キャッチフレーズを 「Webにない“新しい”が、ここにある。」とし、新しいソリューションが求められる時代に対応した世界最先端技術・サービスの展示とともに、若手技術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の設置、産学官の技術交流を深める企画、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。

会場内の特設ステージでは、半導体ベンダの Green 戦略、全てが分かるローパワー設計技術、ソーラーや車載で注目されているパワー・ 高耐圧系アナログ回路設計と、地球環境に優しい設計技術をエグゼクティブの方々が登壇した。 例年好評の上流設計技術は、組込みシステムにおける ソフトウェア開発への ESL 活用について、日本を代表する技術者の方々が、初心者にもよく分かるように議論をした。 また、今回 10 回目の開催を記念して、10 周年特別企画「各社の No.1 設計者が語る"私の設計"」 を開催した。

第一線で活躍中のエンジニア、設計・開発を率いる管理職の方々、さらに若手エンジニアの方までを対象とした幅広い内容の企画により、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。

さらに、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、FPGA 関連の出展者を集めた、「FPGA ビレッジ」や、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。

また、海外出展企業を見学希望する来場者に、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 10～30 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:2010 年 1 月 28 日(木)～1 月 29 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:**9,300 名**(前年 9,117 名)

② 出展者数:**113 社/231 小間**(前年 143 社/317 小間)

③ 出展者セミナー:**84 セッション、延べ聴講者数 2,337 名**(前年 104 セッション、述べ聴講者数 2,070 名)

④ スイートルーム:**4 社**(前年 5 社)

⑤ ユニバーシティ・プラザ:**9 大学研究室**(前年 22 大学研究室)

⑥ (併催)第 17 回 FPGA/PLD Design Conference:**18 セッション、聴講者数 848 名**(前年 8 セッション、聴講者数 216 名)

⑦日別来場者数

1月 28 日(木) ※くもり／一時雨	1月 29 日(木) ※晴れ	合計
4,288 名 (前回 3,953 名)	5,012 名 (前回 5,164 名)	9,300 名(前回 9,117 名)

⑧来場者業種内訳

	2010 年	2009年	偏差
半導体・電子部品	**3,754**	3,747	0
機器メーカ	**2,627**	2,250	+17%
ツールベンダ	**569**	579	−2%
設計関連サービス	**1,024**	1,071	−4%
商社・営業	**539**	437	+23
出版・マスコミ	**95**	135	−30%
その他	**692**	898	−23%
計	**9,300**	9,117	+2%

⑨過去からの来場者数推移

開催年	初日	2 日目	合 計
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

世界経済が低迷する中で、初日の来場者数が前回比 8.5%増加。2 日目が前回比 2.9%減少。合計で総来場者数が前回比 2%増加した結果となった。

初日の増加の要因としては、前回までになかった施策として、10 周年特別企画「各社の No.1 設計者が語る“私の設計」の開催、「ワインの夕べ」の開催、日経 BP セミナーの併設、が考えられる。また、2 日目の減少は、同日午後に発生した東海道新幹線の架線切れ火災による東海道新幹線の運転見合わせ、およびダイヤの大幅な乱れが少なからず影響しているとみられる。会場内においても、午後から閉館に向けては通常来場者数が増えるものであるが、今回それが見られなかった。

全体として増加した要因には、電話・FAX での来場誘致、FPGA/PLD Design Conference の無料化、来場者プレゼント企画が貢献したものと考える。

3.1.2 出展カテゴリー

(1)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、他

(2)ハードウェア開発環境(EDA)

①EDA/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、他

②PCB/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、他

(3)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、他

(4)LSI テスト、計測器

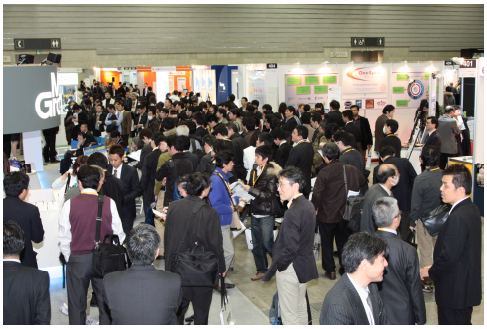
LSI テスタ、PCB テスタ、計測器、他

(5)IP コア、マクロ、セルライブラリ

(6)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、デバッガ、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、他

(7) 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、他
(8)設計インフラ(WS/PC、ネットワーク)
(9)設計データ管理ツール
　設計データ管理ツール、他
(10)マスクメーカ、ファウンダリメーカ
(11)大学(研究室)、コンソーシアム
(12)PR 関連
　出版物、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)
(株)アノーバ・ソリューションズ/ エートップテック(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
アンソフト・ジャパン(株)
アンシス・ジャパン(株)
イノテック(株)
Arteris Inc.
Duolog Technologies Ltd.
Jazz Semiconductor,Subsidiary of Tower
Rapid Bridge LLC
Target Compiler Technologies N.V.
Interoperable PDK Alliance
(株)エーイーティー
ATE サービス(株)
ENTASYS DESIGN INC.
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)
コフルエント デザイン
CyberTec(株)
Jasper Design Automation
サイバネットシステム(株)
サン・マイクロシステムズ(株)
(株)スピナカー・システムズ
CAST,Inc.
Verific Design Automation Inc.
スプリングソフト(株)
ダッソー・システムズ(株)
タナーリサーチジャパン(株)
DSM ソリューションズ(株)
DeFacTo Technologies S.A.
DOCEA POWER S.A.
Prolific, Inc.
デナリソフトウエア(株)
(株)電波新聞社
日本イヴ(株)
日本ケイデンス・デザイン・システムズ社
イノテック(株)
日本シノプシス合同会社
日本ワンスピン・ソリューションズ(株)
バークレー・デザイン・オートメーション(株)
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)
(財)福岡県産業・科学技術振興財団
富士通(株)
マグマ・デザイン・オートメーション(株)
MunEDA GmbH
メンター・グラフィックス・ジャパン(株)
リード・ビジネス・インフォメーション(株)

新興ベンダエリア

IC サービス(株)
CMSC,Inc.
IST,Inc.
IPextreme Inc. Constellations
Sidense Corp.
Tiempo
NXP Coolflux DSP
アラサン・チップ・システムズ
(株)エイアールテック
神戸大学工学部永田研究室

EDXACT SA
CLIOSOFT, INC.
（株）シンコム
CONCEPT ENGINEERING GMBH
シグナル・プロセス・ロジック（株）
（株）シグリード
シリコン・フロントライン・テクノロジー（株）
Synfora, Inc.
TEKLATECH A/S
Dorado Design Automation,Inc.
（株）トプスシステムズ
日本アールソフトデザイングループ（株）
日本リアルインテント（株）
POLYTEDA SOFTWARE CORPORATION
MICROLOGIC DESIGN AUTOMATION INC.

JEVeC ビレッジ

アートグラフィックス
（株）アストロン
（株）礎デザインオートメーション
エイシップ・ソリューションズ（株）
ギガヘルツテクノロジー（株）
ケイレックス・テクノロジー（株）/（株）システム・ジェイディー
（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）

FPGA ビレッジ／特定非営利活動法人 FPGA コンソーシアム

アルティウム リミテッド/（株）エー・ディ・ティ
アルデック・ジャパン（株）
特定非営利活動法人 FPGA コンソーシアム
（株）シャンテリー
立野電脳（株）
特殊電子回路（株）
（株）PALTEK
富士エレクトロニクス（株）
（株）プライムゲート
プロトタイピング・ジャパン（株）
三菱電機マイコン機器ソフトウエア（株）

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
九州工業大学　中村研究室
筑波大学　高度 IT 人材育成のための実践的ソフトウェア開発専修プログラム　東海大学　清水尚彦研究室
東京大学　VDEC
名古屋大学　高田・冨山研究室
広島大学　アルゴリズム論研究室
早稲田大学　後藤研究室/渡邊研究室/木村研究室/吉村研究室

50 音順 （※一字下げは共同出展）

3.1.4 出展傾向

出展者数は 113 社(新規出展企業 35 社)となった。前年比 21%減で 2004 年時の社数を下回った。小間数としては、27%減で EDSFair となってから過去一番少ない小間数であった。

減少の要因は、アメリカを発端とする世界同時不況の影響。各社の M&A 等が考えられる。

	出展者数	小間数
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

	2010 年		2009 年	
	社数	小間数	社数	小間数
通常出展	91 社	211 小間	107 社	278 小間
新興ベンダ(ブース)出展)	22 社	20 小間	36 社	39 小間

3.1.5 出展者セミナー

出展者セミナーは今回よりカテゴリー別にプログラムを組み、事前登録を可能とした。

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提供した。2010 年は 8 会場にて 84 セッションを開催した。

聴講者数:2,337 名

トラック別内訳	セッション数	28 日聴講者数計	29 日聴講者数計
システム設計・検証	23	183 名	321 名
ロジック設計&フィジカル設計 ／検証	11	141 名	215 名
AMS 設計・検証	7	35 名	146 名
PCB	4	41 名	77 名
テスト設計	2	36 名	55 名
機能検証	10	81 名	278 名
IP	6	41 名	34 名

LowPower	3	57 名	42 名
DFM	3	50 名	34 名
フリー	15	80 名	390 名

3.1.6 第 17 回 FPGA/PLD Design Conference

(1) 日　時:1 月 28 日(木)・29 日(金)

(2) 場　所:アネックスホール

(3) 聴講料:無料・事前申込不要

聴講者数:

1 月 28 日(木)	
基調講演	116 名
招待講演1	131 名
スペシャルセッション	175 名
招待講演 2/協賛講演	45 名

1 月 29 日(金)	
特別講演/協賛講演	42 名
招待講演 3/協賛講演	62 名
チュートリアル 1 /協賛講演	75 名
招待講演 4/協賛講演	119 名
チュートリアル 2	83 名

EDSFair と併設の第 17 回 FPGA/PLD Design Conference が 1 月 28 日(木)と 29 日(金)の両日、アネックスホールにて開催され、招待講演含む全 18 セッションで合計 848 名の聴講者を集めた。従来は聴講は有料であったが、今回は聴講無料で開催した。

1 月 28 日(木)

基調講演(10:30-11:30)
リコンフィギャラブルデバイスの現状と動向
天野 英晴 氏
[慶應義塾大学 教授]

招待講演 1 (13:00-13:45)
ASIC プロが語るザイリンクス FPGA の挑戦 ー日本産業の未来へ
菊池 秀夫 氏
[ザイリンクス(株) 副社長]

協賛講演(13:45-13:55)
進化した FPGA に ASIC 的検証手法を ～ModelSim Delux Edition～
菅沼 庸吉 氏
[(株)PALTEK エンジニアリング ディビジョン 課長]

スペシャルセッション (14:15-15:45)
FPGAはどう使うのが賢いか? ～FPGA 大喜利, 設計のコツ教えます～
モデレータ
佐藤 幸一 氏 [コニカミノルタテクノロジーセンター(株) システム技術研究所 アーキテクチャ開発室長]
パネラー
(ＦＰＧＡ) 大山 浩司 氏 [(株)アルティマ 技術統括部 開発企画課 課長]
(ツール) 藤野 博信 氏 [日本シノプシス合同会社 営業本部]
(基板設計) 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]
(ユーザー) 小熊 博 氏 [宮城県産業技術総合センター 機械電子情報技術部 情報技術開発班 研究員]

招待講演 2 (16:00-16:45)

Flash FPGAs: a different technological approach to PLDs’
片山 雅美 氏［アクテルジャパン(株) セールスディレクター］
協賛講演 (16:45-16:50)
Altium Designer における FPGA 設計
常木 竜太 氏［(株)エー・ディ・ティ ALTIUM 日本代理店 システム開発部 FAE 課］
FPGA 検証を効率的に行うアルデック社の取り組み
藤永 康博 氏［アルデック・ジャパン(株) 代表取締役］

1 月 29 日(金)

特別講演 (10:00～11:00)
回路の消費電力から暗号を解読するサイドチャネル攻撃の実際
佐藤 証 氏
[産業技術総合研究所 情報セキュリティ研究センター ハードウェアセキュリティ研究チーム長]
協賛講演 (11:00-11:05)
匠のエンジニア集団・プライムゲートの FPGA 開発実績紹介
鈴木 宏海 氏［(株)プライムゲート 営業部］
協賛講演 (11:05-11:10)
～ムダな苦労はもうしたくない！～ FPGA 開発を楽にする JTAG 活用ノウハウ
内藤 竜治 氏［特殊電子回路(株) 代表取締役］

招待講演 3 (11:30-12:15)
Lattice Semiconductor 社の戦略製品と戦略市場について
山本 好充 氏［ラティスセミコンダクター(株)代表取締役］
協賛講演 (12:15-12:25)
簡単評価！ Lattice リファレンスボードとデザイン・サービス
北村 塁 氏［富士エレクトロニクス(株) 技術開発部 lattice グループ マーケティング担当］

チュートリアル 1 (13:15-14:15)
FPGA 汎用プロトタイプボードを用いた SoC 開発事例
中島 孝二 氏［パイオニアマイクロテクノロジー(株) 第二技術部設計五課 副主事］
協賛講演 (14:15-14:25)
FAVS(FPGA Accelerated Verification Solution) ～StratixⅣと Virtex-6 で広がる ASIC プロトタイピング検証手法～
鳥本 元彦 氏［プロトタイピング・ジャパン(株) CEO］
協賛講演 (14:25-14:30)
FPGA を使った LSI 評価ボードのご紹介
澤田 朋子 氏［三菱電機マイコン機器ソフトウェア(株) 第 3 事業部開拓部営業グループ］

招待講演 4 (14:45-15:30)
市場要求と技術トレンドにみる、アルテラ FPGA＆ASIC の可能性
日隈 寛和 氏［日本アルテラ(株) 代表取締役社長］
協賛講演 (15:30-15:45)
FPGA を取り巻くトピックと新しい FPGA の挑戦
末吉 敏則 氏［(特非)FPGA コンソーシアム 理事長］

チュートリアル 2 (16:00-17:00)
FPGA ならではの部分再構成機能の使い方
堀 洋平 氏［中央大学 研究開発機構 専任研究員/機構助教］

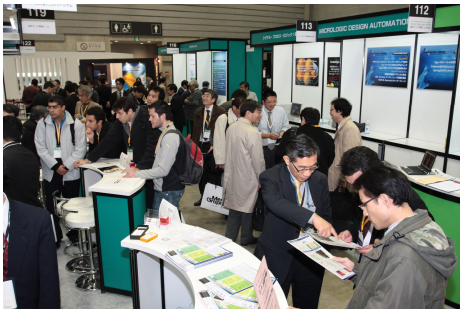
3.1.7 特別ゾーン

新興ベンダエリア

国内外新興企業 18 社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.
IPextreme Inc. Constellations
Sidense Corp.
Tiempo
NXP Coolfl ux DSP
アラサン・チップ・システムズ
(株)エイアールテック
神戸大学工学部永田研究室
EDXACT SA
CLIOSOFT, INC.
(株)シンコム
CONCEPT ENGINEERING GMBH
シグナル・プロセス・ロジック(株)
(株)シグリード
シリコン・フロントライン・テクノロジー(株)
Synfora, Inc.
TEKLATECH A/S
Dorado Design Automation,Inc.
(株)トプスシステムズ
日本アールソフトデザイングループ(株)
日本リアルインテント(株)
POLYTEDA SOFTWARE CORPORATION
MICROLOGIC DESIGN AUTOMATION 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
(株)礎デザインオートメーション
エイシップ・ソリューションズ(株)
ギガヘルツテクノロジー(株)
ケイレックス・テクノロジー(株)/(株)システム・ジェイディー
(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)

FPGA ビレッジ

特定非営利活動法人 FPGA コンソーシアムが出展者を集め、FPGA ゾーンを設置した。
今後もさらなる発展が見込まれる最新の FPGA 関連技術を紹介した。

出展者:

アルティウム リミテッド/(株)エー・ディ・ティ
アルデック・ジャパン(株)
特定非営利活動法人 FPGA コンソーシアム
(株)シャンテリー
立野電脳(株)
特殊電子回路(株)
(株)PALTEK
富士エレクトロニクス(株)
(株)プライムゲート
プロトタイピング・ジャパン(株)
三菱電機マイコン機器ソフトウエア(株)

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。

9 大学研究室

■ **非同期式回路の設計支援環境の構築**
会津大学 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 樋上・高橋研究室

■ **素子劣化を考慮したアナログ LSI 回路設計環境の構築**
九州工業大学 中村研究室

■ **動的再構成可能 LSI を活用したシステム構築および技術応用**
筑波大学 高度 IT 人材育成のための実践的ソフトウェア開発専修プログラム

■ **高位記述による LSI 設計手法**
東海大学 清水尚彦研究室

■ **VDEC の活動紹介**
東京大学 VDEC

■ **システムレベル設計環境 SystemBuilder**
名古屋大学 高田・冨山研究室

■ **多層プリント配線基板設計支援システム MULTI-PRIDE**
広島大学 アルゴリズム論研究室

■ **ICT アプリケーション LSI IP のための先端的設計技術**
早稲田大学 後藤研究室/渡邊研究室/木村研究室/吉村研究室

3.1.8 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。

新興ベンダ・ツアー参加企業:13社

内訳:米国企業:8 社、カナダ企業:1社、独企業:1 社、仏企業:1社、デンマーク:1 社、台湾企業:1社

実施回数:1月 28 日 1 回、1月 29 日 2回
ツアー参加人数:各ツアーとも 10－30 名

EDSFair2010 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)の執行役員開発第 1 部長、西口 信行氏、29 日は、エーエスケイアシスト(ASKAssist)の 代表責任者、の秋山 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。中には、日本語デモや、パンフレットなどの資料を準備していた企業もあり、出展者としてもこのツアーによるブース来場者を期待していた様子が見受けられた。 ツアー・ガイドは、必要に応じて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ーケ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 10－30 名の参加者があり、盛況であった。

・ツアーA: 23 名
・ツアーB:19 名
・ツアーC:33 名
※バーコード読み取り件数

3.1.9 特設ステージ

場　所:展示フロア内　参加無料

展示会場内では、特設ステージにて、第一線で活躍中のエンジニアの方はもちろん、設計・開発を率いる管理職の方々、さらに若手エンジニアの方までを対象とした幅広い内容の企画を開催しました。また EDSFair としては、今回 10 回目の開催となり、これを記念して、10 周年特別企画も開催しました。

1 月 28 日(木)

■10:50-12:50　10周年特別企画

「各社の No.1 設計者が語る"私の設計"」

道正　志郎　氏〔パナソニック(株) 本社 R&D 部門　戦略半導体開発センター　ハードウェア設計エキスパート〕

片倉　雅幸　氏〔ソニー(株) コンスーマープロダクツ&デバイスグループ　半導体事業本部　研究開発部門　主幹技師〕

大塚　竜志　氏〔富士通マイクロエレクトロニクス(株) ASSP 事業本部 Infotainment 事業部　第 1 設計部　プロフェッショナル〕

佐伯　貴範　氏〔NEC エレクトロニクス(株)　デジタルコンシューマ LSI 事業部　シニアプロフェッショナル〕

小林　昭男　氏〔三洋電機(株)　デジタルシステムカンパニー　DI事業部　部長〕

【司会】松澤　昭　教授〔国立大学法人東京工業大学　理工学研究科　電子物理工学専攻　工学博士〕

【10 周年記念特別企画 WG 主査】　河村　薫　氏〔富士通マイクロエレクトロニクス　(株)〕

聴講者数約 280 名

セッションの冒頭、EDSFair と同じく今年 10 周年を迎える Accellera の vice-chair である Mr. Dennis Brophy 氏と OSCI の treasurer である Dr. Stan Krolikoski 氏にお祝いのスピーチをお願いした。お二人は毎年 EDSFair の時期にあわせてパシフィコで開催している IEEE DASC(The Design Automation Standards Committee)ミーティング出席のため来日中であった。

5 名の発表者は、アナログ設計技術、ディジタル設計技術、商品開発の立場から、若手設計者に向けてメッセージを発信した。全発表終了後に、最優秀発表者としてパナソニックの道正　志郎氏を、優秀発表者として他の 4 名を齋藤実行委員長が表彰した。

来場者アンケートでは、16 名の人が 10 周年企画にコメントし、

- No.1 設計者のステージはとてもよかった。来年以降も続けてほしい。
- 特設企画の「各社の No.1 設計者」が良かったです。 設計者の生の声を聞ける機会は日本ではほとんどなかったので、このような企画は今後も継続して欲しいです。
- 特設ステージでの設計事例発表は興味深かった。来年以降も同様の他社設計事例が聞きたい。

など、全員が来年度の継続を希望した。

■エグゼクティブが語る！

□13:00-14:10　セッション 1

「エグゼクティブが語る我が社の Green 戦略」

地球温暖化防止に向けて、グリーン社会の実現が期待されています。本講演では、国内半導体各社がグリーン社会実現に向けどのように貢献しようとしているのかについて、各社のエグゼクティブの方々に戦略を語っ

ていただきます。

八木 春良 氏〔富士通マイクロエレクトロニクス(株) 取締役〕

山口 聖司 氏〔パナソニック(株) セミコンダクター社 システム LSI 事業本部 商品開発センター所長〕

中屋 雅夫 氏〔(株)ルネサステクノロジ 取締役〕

【司会】望月 洋介 氏〔日経 BP 社 電子・機械局長補佐〕

【オーガナイザ】吉田 正昭 氏〔NEC エレクトロニクス(株)〕／横山 昌生 氏〔シャープ(株)〕

聴講者数約 170 名

昨年のエグゼクティブパネル「電子産業の成長シナリオと EDA 業界の役割」に続き、半導体業界を代表する企業のエグゼクティブが登壇するセッションを企画した。半導体業界に限らず、昨今の社会活動において、「グリーン」は重要なキーワードであり、エグゼクティブ 3 氏が各社のグリーン戦略を語ったのは意義深い。

リーマンショック後の世界不況が続く中、半導体業界も非常に厳しい状況にあるが、このような場での業界および、関連業界関係者への発信は、今後も是非続けてほしいものである。

■ 今さら聞けないことがわかる！

□16:00-17:30 セッション 2

組込みシステムにおけるソフトウェア開発要件とESL技術の対応

～「ものをつくる」から、「ものにつくりあげる」へのシフト～

「ものづくり」という言葉が使われて久しくなります。4 ビットマイコン搭載機器において、「もの」の一部だったソフトウェアは、32 ビットのマルチコア時代を迎えた今、製品の魅力を大きく左右するキーとなっています。その開発基盤として注目を浴びているESL 技術、しかし果たしてソフトウェア開発要件を充分満たすことができるのでしょうか。このセッションでは、日ごろから HW / SW の統合やトレードオフに携わっている方、組込みソフト開発に携わっている方にお集まりいただき、組込みシステム開発における問題とその解決の方向性について議論いたします。

岩井 明史 氏〔(株)デンソー 電子プラットフォーム開発部 主幹〕

旦木 秀和 氏〔ソニー(株) 半導体事業本部 設計基盤技術部門 統括課長〕

松本 祐教 氏〔(株)トプスシステムズ 代表取締役社長〕

中村 和正 氏〔富士通マイクロエレクトロニクス(株) 共通技術本部 設計共通技術統括部 プロジェクト課長〕

牧野 潔 氏〔メンター・グラフィックス・ジャパン(株) ビジネス開発マネージャー〕

【司会】坂本 秀人 氏〔ESL(株) 代表取締役社長〕

【オーガナイザ】三橋 明城男 氏〔メンター・グラフィックス・ジャパン(株)〕

聴講者数約 260 名

大規模・複雑化する組み込みシステム開発に携わるハードウェア設計者とソフトウェア設計者、さらにその両者を統括するマネージャを対象とて二つのテーマについて議論を行った。

一つは、組み込システム機器開発においてキーとなる最適化技術の視点から、ESL を用いて組み込みシステム機器の最適設計がどこまでできるのか、ハードウェアとソフトウェアの統合、アーキテクチャ検討、性能評価などの側面から、ものに作り上げるための技術要件についてである。もう一つは、その最適設計を実現するために求められる技術以外の要件、すなわち最適化技術をうまく効率的に開発・適用・普及させるためのマネージメントの視点より、業界標準化やインフラの整備・組織・プロセス・人材・スキルなどについて議論を展開した。

上流設計のセッションは、毎年多くの聴衆が参加されており、ESL への期待が高まる中、導入をためらっていた会社(部署)への後押しになることを期待したい。

1 月 29 日(金)

■EDA ベンダエグゼクティブが語る！

□10:30～11:40 セッション 3

〈ローパワー設計の全てがわかる！！第一部〉
「ローパワー設計の現状、課題とその対策」

半導体ベンダーでのローパワー設計の実例 2 件の紹介、および STRJ ロードマップ委員会 WG1 が考える SOC の低消費電力設計技術の課題と解決策を紹介していただきます。これらの講演からローパワー設計の現状と課題、ロードマップの観点からの解決策を明らかにします。

吉田 裕 氏〔(株)ルネサステクノロジ 設計開発本部 主任技師〕
濱田 基嗣 氏〔(株)東芝 半導体研究開発センター 部長〕
隅谷 三喜夫 氏〔JEITA STRJ WG1:パナソニック(株) 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 グループマネージャー〕
【司会】小島 郁太郎 氏〔日経 BP 社 電子・機械局 編集委員〕
【オーガナイザ】吉田 正昭 氏〔NEC エレクトロニクス(株)〕／井下 順功 氏〔(株)ルネサステクノロジ〕
聴講者数約 290 名

□11:50-12:50 セッション 4 **日英同時通訳付き**

〈ローパワー設計の全てがわかる！！第二部〉
「EDA ベンダーが提供するローパワー設計技術」

セッション3に引き続き、ローパワー設計技術の課題に対し、EDA ベンダーの対応戦略を語っていただきます。EDA ベンダー各社に現在提供できている / 今後提供しようと考えているローパワー設計技術について明らかにしていただき、セッション 3 と合わせてローパワー設計技術の全体像を明らかにします。

Mr. Vic Kulkarni〔Apache Design Solutions Inc.,
GM and SVP of RTL Business unit〕
Mr. Bernard Murphy〔Atrenta Inc., CTO〕
Mr. Neil Hand〔Cadence Design Systems Inc.,
Director, Solutions Marketing, Low power〕
Dr. Anmol Mathur〔Calypto Design Systems, CTO and Founder〕
Mr. Robert Smith〔Magma Design Automation, Inc., VP, Product Marketing〕
Mr. Simon Bloch〔Mentor Graphics Corporation, VP and GM,
Design & Synthesis Division〕
Mr. George Zafiropoulos〔Synopsys, Inc., VP, Solution Marketing〕
隅谷 三喜夫 氏〔JEITA STRJ WG1:パナソニック(株) 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 グループマネージャー〕
【司会】小島 郁太郎 氏〔日経 BP 社 電子・機械局 編集委員〕
【オーガナイザ】吉田 正昭 氏〔NEC エレクトロニクス(株)〕／井下 順功 氏〔(株)ルネサステクノロジ〕
聴講者数約 250 名

「ローパワー設計のすべてがわかる！」というタイトル・セッションで、二部構成にて開催した。第一部では、半導体メーカーのローパワー設計事例や、JEITA STRJ によるローパワー設計のロードマップを紹介した。
第二部では、そのロードマップに対する EDA ベンダーの解決策が、7 社のエグゼクティブから説明された。ローパワー設計は、EDSFair2008 の「今さら消えないローパワー ～教えます。現場で使えるローパワー設計～」

に続いての企画であったが、今回もローパワー設計への関心は相変わらず高く、更にオーガナイザ、司会者の周到な準備により、両セッションともに立ち見が出るほどの盛況であった。

□15:30-17:00 セッション 5　**日英同時通訳付き**

「パワー ・ 高耐圧系アナログ回路の現状と課題」

近年、環境問題及び携帯機器への低消費電力化要求にて関心が高まっているパワー・ 高耐圧回路の現状及び技術課題を語り合っていただきます。携帯電話、ノート PC 及び携帯基地局に用いられる電源・パワーアンプ系回路にて高効率な回路を追求する立場から現状の課題や今後の技術革新を紹介していただき、回路・デバイス設計技術及び EDA 技術でどの様に課題を克服していくのかを明らかにします。

恩田 謙一 氏〔(株)日立製作所 日立研究所長付〕
松田 順一 氏〔旭化成東光パワーデバイス(株) 技術統括〕
中島 成 氏〔住友電気工業(株) 伝送デバイス研究所長〕
佐藤 伸久 氏〔日本ケイデンス・デザイン・システムズ社
テクニカルフィールドオペレーション本部 セールス AE ディレクター〕
Mr. Ernie Koeroghlian〔Mentor Graphics Corporation,
Product Architect, DSM CICD R&D〕
【司会】小林 春夫 教授〔群馬大学 大学院 工学研究科〕
【オーガナイザ】森井 一也 氏〔三洋半導体(株)〕

聴講者数約 230 名

セッションの前半では、パワー・高耐圧系アナログ・デバイス・メーカーの第一人者 3 名が「現在の開発状況や今後の技術革新」について語った。それを受けて、後半では、EDA ベンダーの技術エキスパート 2 名が、多くの課題提起に対しどの様なソリューションが提供できるのか、さらに今後どの様な技術革新を行っていくのかを明確に示した。

アナログのセッションは、EDSFair2007 の「アナログ難しそう。でも何が難しいの？」に続いての開催であったが、前回同様に多数の聴講者が集まり、今後も継続していく必要がある。

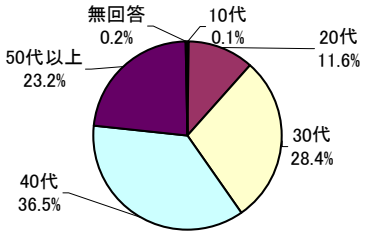
日時	領域	タイトル	モデレータ	講師・パネリスト（所属）	オーガナイザ	概要	聴講数（名）
28日（木） 午後 13:00– 14:10	Green戦略	エグゼクティブが語る我が社のGreen戦略	望月洋介 氏 （日経BP）	八木春良 氏（FLM） 山口聖司 氏（パナソニック） 中屋雅夫 氏（ルネサス）	吉田正昭 氏 （NEC-EL） 横山昌生 氏 （シャープ）	グリーン社会実現に向けた戦略を、国内半導体各社のエグゼクティブに語っていただきます。	169
28日（木） 午後 16:00– 17:30	ESL	組込みシステムにおけるソフトウェア開発要件とESL技術の対応 ～「ものをつくる」から、「ものにつくりあげる」へのシフト～	坂本秀人 氏 （ESL）	岩井明史 氏（デンソー） 旦木秀和 氏（ソニー） 松本祐数 氏（トプスシステム） 中村和正 氏（FML） 牧野潔 氏（メンター）	三橋明城男 氏 （メンター）	ソフトウェアが製品の魅力を左右する現在、 注目を浴びているESL技術は、果たしてソフトウェア開発要件を満たすことができるのか？	253
29日（金） 10:30– 11:40	Low Power （設計事例） （Road Map）	ローパワー設計の全てがわかる!! 第一部 「ローパワー設計の現状、課題とその対策」	小島郁太郎 氏 （日経BP）	吉田裕 氏（ルネサス） 濱田基嗣 氏（東芝） 隅谷 三喜夫氏（パナソニック）	吉田正昭 氏 （NEC-EL） 井下順功 氏 （ルネサス）	ローパワー設計の実例2件の紹介、STRJからの低消費電力設計の現状・課題と解決策の解説	287
29日（金） 11:50– 12:50	Low Power （EDA）	ローパワー設計の全てがわかる!! 第二部 「EDAベンダーが提供するローパワー設計技術」		Vic Kulkarni 氏(Apache) Neil Hand 氏(Cadence) Anmol Mathur 氏(Calypto) Robert Smith 氏 (Magma) Simon Bloch 氏 (Mentor) George Zafiropoulos 氏 (Synopsys) 隅谷 三喜夫氏（パナソニック）		EDAベンダー7社のエグゼクティブが、各社のローパワー設計の戦略を語ります。	249
29日（金） 15:30– 17:00	パワー素子 高耐圧素子 （EDA）	「検証メソドロジ入門から超並列計算機向けインターコネクトへの適用事例まで」	小林春夫 教授 （群馬大学）	恩田謙一 氏（日立） 松田順一 氏（旭化成東光） 中島成 氏（住友電工） 佐藤伸久 氏（日本ケイデンス） Ernie Koeroghlian 氏 (Mentor)	森井一也 氏 （三洋半導体）	電源・パワーアンプ回路の課題と今後の技術革新を紹介し、EDAによる克服方法を解説します。	231

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。

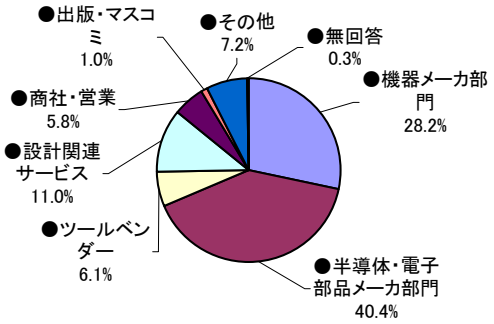
■年代

	2010(今回)	2009(参考)
10代	0.1%	0.1%
20代	11.6%	16.2%
30代	28.4%	29.2%
40代	36.5%	32.7%
50代以上	23.2%	18.3%
無回答	0.2%	3.5%
合計	100.0%	100.0%



■業種

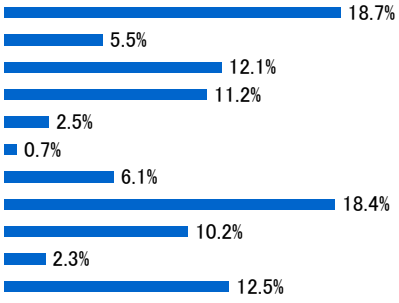
	2010(今回)	2009(参考)
●機器メーカ部門	28.2%	24.7%
●半導体・電子部品メーカ部門	40.4%	41.1%
●ツールベンダー	6.1%	6.3%
●設計関連サービス	11.0%	11.7%
●商社・営業	5.8%	4.8%
●出版・マスコミ	1.0%	1.5%
●その他	7.2%	6.2%
●無回答	0.3%	3.7%



※業種詳細

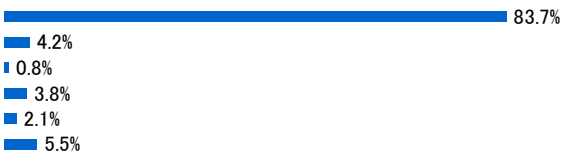
	2010(今回)	2009(参考)
●機器メーカ部門	28.2%	24.7%

コンピュータ関連機器	18.7%	21.9%
ネットワーク関連機器	5.5%	5.8%
一般民生機器	12.1%	14.7%
画像処理機器	11.2%	10.4%
医療機器	2.5%	2.2%
アミューズメント	0.7%	0.6%
自動車・輸送機器	6.1%	3.8%
産業機器(機械・精密機器等)	18.4%	17.1%
通信機器	10.2%	8.8%
放送機器	2.3%	3.2%
その他	12.5%	11.5%



	2010(今回)	2009(参考)
●半導体・電子部品メーカ部門	40.4%	41.1%

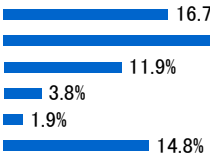
システムLSI、ASIC、マイコン、メモリ	83.7%	81.9%
FPGA/PLD	4.2%	5.2%
ディスプレイ	0.8%	1.7%
電子コンポーネント	3.8%	3.1%
プリント基板	2.1%	2.4%
その他	5.5%	5.8%



●ツールベンダー	2010(今回)	2009(参考)
	6.1%	6.3%

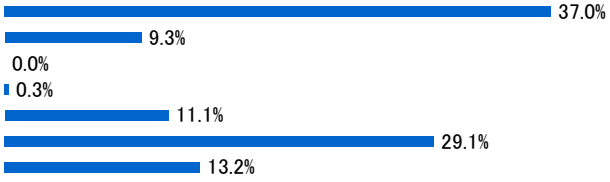
※ツール関連が主要営業品目である商社・代理店も含む

機能・論理設計ツール	16.7%	20.3%
LSI設計ツール	51.0%	49.1%
プリント基板設計ツール	11.9%	13.4%
マイコンツール	3.8%	2.2%
ハードウェア・ボード機器	1.9%	3.0%
その他	14.8%	12.1%



●設計関連サービス	2010(今回)	2009(参考)
	11.0%	11.7%

デザインハウス	37.0%	45.6%
IPプロバイダ	9.3%	5.3%
IP流通サービス	0.0%	0.0%
ネット環境	0.3%	0.2%
教育・コンサルタント	11.1%	13.0%
ソフト開発	29.1%	24.2%
その他	13.2%	11.6%



●商社・営業	2010(今回)	2009(参考)
	5.8%	4.8%

※ツール関連が主要営業品目である商社・代理店は除く

電子機器	10.1%	9.1%
半導体	52.3%	52.3%
電子部品	9.5%	10.2%
ツール	12.6%	9.7%
その他	15.6%	18.8%

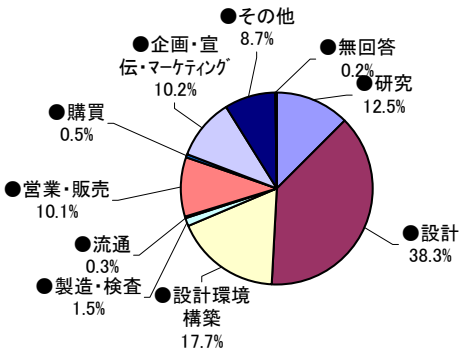
●出版・マスコミ	2010(今回)	2009(参考)
	1.0%	1.5%

●その他	2010(今回)	2009(参考)
	7.2%	6.2%

●無回答	2010(今回)	2009(参考)
	0.3%	3.7%

■職務

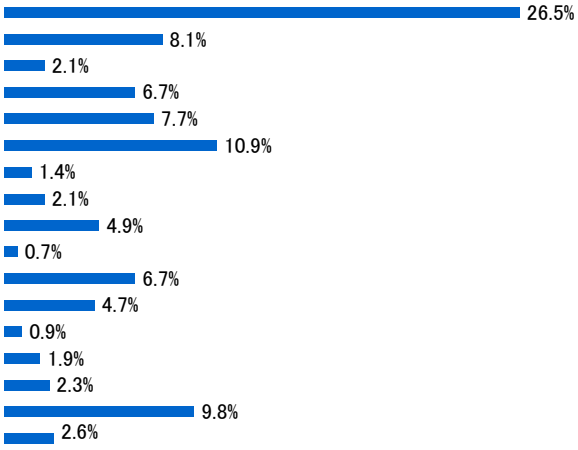
	2010(今回)	2009(参考)
●研究	12.5%	12.1%
●設計	38.3%	37.6%
●設計環境構築	17.7%	19.6%
●製造・検査	1.5%	1.4%
●流通	0.3%	0.1%
●営業・販売	10.1%	9.5%
●購買	0.5%	0.3%
●企画・宣伝・ﾏｰｹﾃｨﾝｸﾞ	10.2%	8.3%
●その他	8.7%	7.4%
●無回答	0.2%	3.7%



※職務詳細

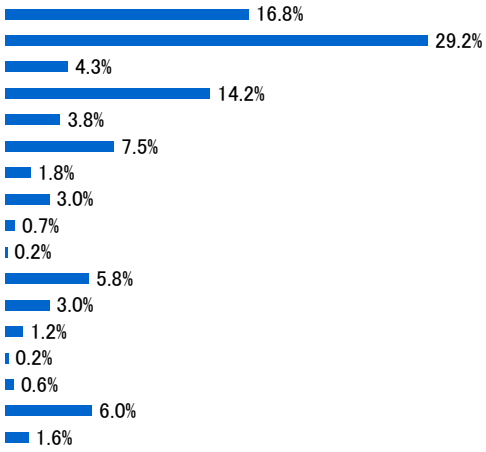
●研究

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

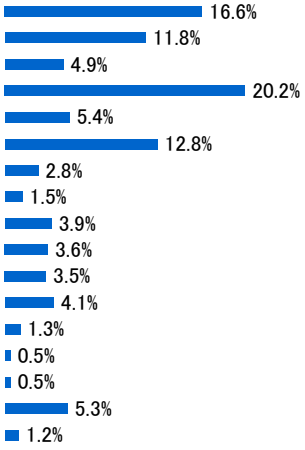


●設計

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

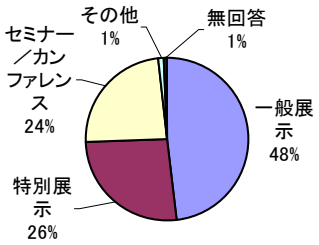


●設計環境構築	2010(今回) 17.7%	2009(参考) 19.6%
システムレベル	16.6%	14.6%
機能(RTL)	11.8%	12.1%
論理(ゲートレベル)	4.9%	6.4%
レイアウト	20.2%	21.4%
テスト	5.4%	5.4%
アナログ	12.8%	10.6%
カスタム	2.8%	4.3%
IPマクロ	1.5%	1.5%
リソ/マスク/プロセス/製造	3.9%	4.3%
TCAD	3.6%	1.7%
FPGA/PLD	3.5%	2.8%
PCB	4.1%	4.9%
IC Package	1.3%	1.5%
SiP	0.5%	0.6%
装置実装	0.5%	0.7%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



■ご来場の目的

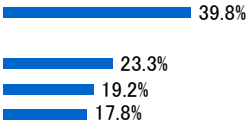
	2010(今回)	2009(参考)
●一般展示	48.2%	40.0%
●特別展示	26.2%	32.2%
●セミナー／カンファレンス	24.1%	24.2%
●その他	1.0%	0.5%
●無回答	0.6%	3.1%



※ご来場の目的詳細

●展示関連	2010(今回) 48.2%	2009(参考) 40.0%
展示ブース	88.2%	87.1%
スイートデモ	11.8%	12.9%

●特別展示	2010(今回) 26.2%	2009(参考) 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	23.3%	15.1%
JEVeCビレッジ	19.2%	16.8%
ユニバーシティ・プラザ	17.8%	13.3%

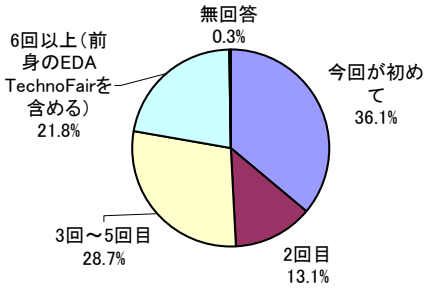


●セミナー／カンファレンス	2010(今回) 24.1%	2009(参考) 24.2%
出展者セミナー	75.8%	52.0%
キーノートスピーチ		24.9%
FPGA/PLD Design Conference	24.2%	11.3%
システム・デザイン・フォーラム		11.8%

●無回答	2010(今回) 0.6%	2009(参考) 3.1%

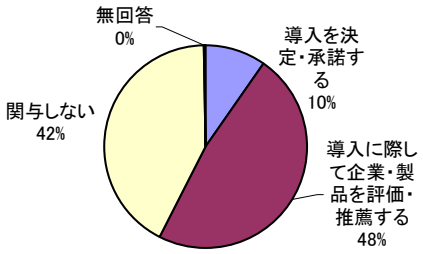
■来場頻度を教えてください

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



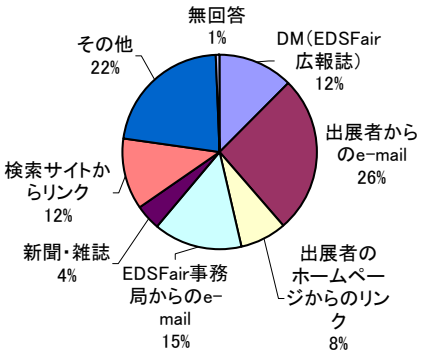
■あなたの製品導入権限について教えてください

	2010(今回)	2009(参考)
導入を決定・承諾する	9.8%	9.4%
導入に際して企業・製品を評価・推薦する	47.8%	45.2%
関与しない	42.0%	41.1%
無回答	0.4%	4.4%
合計	100.0%	100.0%



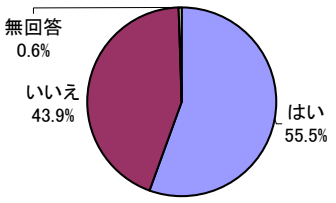
■Electronic Design and Solution Fairをどちらでお知りになりましたか？

	2010(今回)	2009(参考)
DM(EDSFair広報誌)	12.4%	11.1%
出展者からのe-mail	26.3%	24.4%
出展者のホームページからのリンク	7.7%	8.3%
EDSFair事務局からのe-mail	14.8%	14.4%
新聞・雑誌	4.1%	4.5%
検索サイトからリンク	11.9%	11.3%
その他	22.3%	21.6%
無回答	0.5%	4.3%
合計	100.0%	100.0%



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

	2010(今回)	2009(参考)
はい	55.5%	55.6%
いいえ	43.9%	39.9%
無回答	0.6%	4.6%
合計	100.0%	100.0%



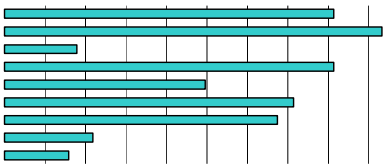
3.1.10 出展者アンケート回答 集計結果（実行委員会後 FIX データにて）

出展社アンケートの回答結果を示す。

出展会社 113 社（窓口数 91）の内、63 件の回答。

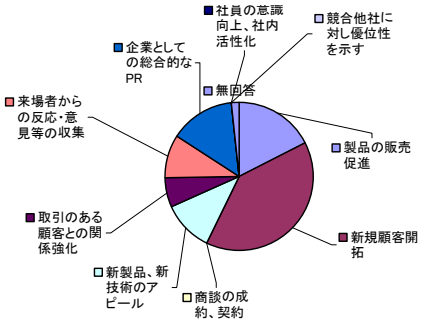
Q1-1. 今回の出展にあたっての目的について該当するものをお選びください。（いくつでも）

	（今回回答数）	2010年度	2009年度（参考）
製品の販売促進	41名	16.3%	14.6%
新規顧客開拓	47名	18.7%	18.2%
商談の成約、契約	9名	3.6%	6.1%
新製品、新技術のアピール	41名	16.3%	13.4%
取引のある顧客との関係強化	25名	9.9%	12.5%
来場者からの反応・意見等の収集	36名	14.3%	11.9%
企業としての総合的なPR	34名	13.5%	13.4%
競合他社に対し優位性を示す	11名	4.4%	5.8%
社員の意識向上、社内活性化	8名	3.2%	4.3%
合計	**252名**	**100.0%**	**100.0%**



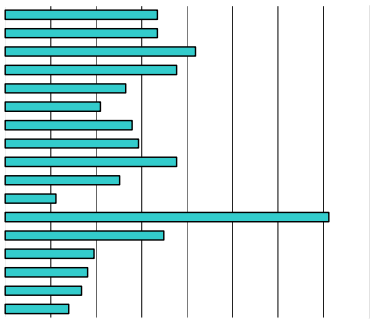
Q1-2. なかでも今回の出展にあたってもっとも重視した目的一箇所をお選びください。

	（今回回答数）	2010年度	2009年度（参考）
製品の販売促進	11名	17.5%	18.8%
新規顧客開拓	25名	39.7%	37.5%
商談の成約、契約	0名	0.0%	1.6%
新製品、新技術のアピール	7名	11.1%	10.9%
取引のある顧客との関係強化	4名	6.3%	9.4%
来場者からの反応・意見等の収集	6名	9.5%	3.1%
企業としての総合的なPR	9名	14.3%	18.8%
競合他社に対し優位性を示す	0名	0.0%	0.0%
社員の意識向上、社内活性化	0名	0.0%	0.0%
無回答	1名	1.6%	0.0%
合計	**63名**	**100.0%**	**100.0%**



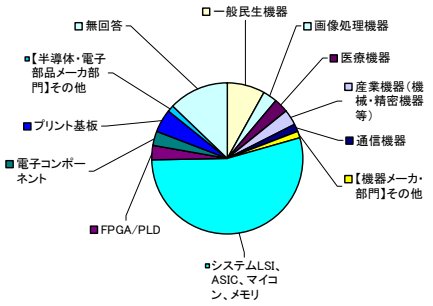
Q2. 出展にあたってターゲットとした来場者の業種について該当するものをお選びください。（いくつでも）

	（今回回答数）	2010年度	2009年度（参考）
【機器メーカ・部門】コンピュータ関連機器	24名	6.7%	6.2%
【機器メーカ・部門】ネットワーク関連機器	24名	6.7%	5.9%
【機器メーカ・部門】一般民生機器	30名	8.4%	8.5%
【機器メーカ・部門】画像処理機器	27名	7.5%	6.7%
【機器メーカ・部門】医療機器	19名	5.3%	4.9%
【機器メーカ・部門】アミューズメント	15名	4.2%	4.9%
【機器メーカ・部門】自動車・輸送機器	20名	5.6%	6.4%
【機器メーカ・部門】産業機器（機械・精密機器等）	21名	5.9%	6.2%
【機器メーカ・部門】通信機器	27名	7.5%	7.7%
【機器メーカ・部門】放送機器	18名	5.0%	4.9%
【機器メーカ・部門】その他	8名	2.2%	1.5%
【半導体・電子部品メーカ部門】システムLSI、ASIC、マイコン、メ	51名	14.2%	14.4%
【半導体・電子部品メーカ部門】FPGA/PLD	25名	7.0%	6.7%
【半導体・電子部品メーカ部門】ディスプレイ	14名	3.9%	4.9%
【半導体・電子部品メーカ部門】電子コンポーネント	13名	3.6%	3.8%
【半導体・電子部品メーカ部門】プリント基板	12名	3.4%	4.4%
【半導体・電子部品メーカ部門】その他	10名	2.8%	2.3%
合計	**358名**	**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。

	（今回回答数）	2010年度	2009年度（参考）
【機器メーカ・部門】コンピュータ関連機器	0名	0.0%	1.6%
【機器メーカ・部門】ネットワーク関連機器	0名	0.0%	0.0%
【機器メーカ・部門】一般民生機器	5名	7.9%	9.4%
【機器メーカ・部門】画像処理機器	2名	3.2%	4.7%
【機器メーカ・部門】医療機器	2名	3.2%	0.0%
【機器メーカ・部門】アミューズメント	0名	0.0%	0.0%
【機器メーカ・部門】自動車・輸送機器	0名	0.0%	1.6%
【機器メーカ・部門】産業機器（機械・精密機器等）	2名	3.2%	3.1%
【機器メーカ・部門】通信機器	1名	1.6%	1.6%
【機器メーカ・部門】放送機器	0名	0.0%	0.0%
【機器メーカ・部門】その他	1名	1.6%	0.0%
【半導体・電子部品メーカ部門】システムLSI、ASIC、マイコン、メ	34名	54.0%	64.1%
【半導体・電子部品メーカ部門】FPGA/PLD	2名	3.2%	7.8%
【半導体・電子部品メーカ部門】ディスプレイ	0名	0.0%	1.6%
【半導体・電子部品メーカ部門】電子コンポーネント	2名	3.2%	1.6%
【半導体・電子部品メーカ部門】プリント基板	3名	4.8%	3.1%
【半導体・電子部品メーカ部門】その他	1名	1.6%	0.0%
無回答	8名	12.7%	0.0%
合計	**63名**	**100.0%**	**100.0%**



3.1.11

EDSFair2010 実行委員会

委 員 長	齋藤茂美	ソニー（株）
副委員長	末吉敏則	熊本大学
副委員長	藤井浩充	日本シノプシス合同会社
委　　員	坂井　仁	イノテック（株）
委　　員	太田裕彦	（株）ジーダット
委　　員	山田陽一	セイコーエプソン（株）
委　　員	中根麻子	TOOL（株）
委　　員	平沢　寿美子	日本ケイデンス・デザイン・システムズ社
委　　員	遠藤裕之	丸紅情報システムズ（株）
委　　員	柴田　多英子	メンター・グラフィックス・ジャパン（株）
委　　員	太田光保	パナソニック（株）
委　　員	山田　節	三洋電機（株）
委　　員	河村　薫	富士通マイクロエレクトロニクス（株）
委　　員	古川　昇	（社）電子情報技術産業協会

JEITA EDA 技術専門委員会 企画 WG

主　　査	山田　節	三洋電機（株）
委　　員	吉田正昭	NEC エレクトロニクス（株）
委　　員	森井一也	三洋半導体（株）
委　　員	長尾　明	シャープ（株）
委　　員	横山昌生	シャープ（株）
委　　員	三橋　明城男	メンター・グラフィックス・ジャパン（株）
委　　員	井下順功	（株）ルネサステクノロジ
委　　員	齋藤茂美	ソニー（株）
委　　員	太田光保	パナソニック（株）
委　　員	河村　薫	富士通マイクロエレクトロニクス（株）

10 周年特別企画 WG

主　　査	河村　薫	富士通マイクロエレクトロニクス（株）
委　　員	吉田正昭	NEC エレクトロニクス（株）
委　　員	山田　節	三洋電機（株）
委　　員	齋藤茂美	ソニー（株）
委　　員	今井浩史	（株）東芝 セミコンダクター社
委　　員	石崎芳典	（社）日本エレクトロニクスショー協会
委　　員	藤井浩充	日本シノプシス合同会社
委　　員	太田光保	パナソニック（株）
委　　員	井下順功	（株）ルネサステクノロジ

第 17 回 FPGA/PLD Design Conference 実行委員会

委 員 長	末吉敏則	熊本大学
副委員長	浅井　剛	（株）ネクスト・ディメンション
副委員長	水尾　学	（株）セプト
委　　員	松本　仁	三菱電機（株）
委　　員	佐藤幸一	コニカミノルタテクノロジーセンター（株）
委　　員	小熊　博	宮城県産業技術総合センター
委　　員	水上明彦	（特非）FPGA コンソーシアム
委　　員	大山浩司	（株）アルティマ
委　　員	大桃　穣	（株）トッパン・テクニカル・デザインセンター
委　　員	阿部浩明	富士通（株）
委　　員	実吉智裕	（株）アットマークテクノ
委　　員	神田　隆	東京エレクトロンデバイス（株）
委　　員	宮澤武廣	三菱電機マイコン機器ソフトウェア（株）

3.1.12 まとめ

EDSFair2010 については、2008 年度の EDA 技術専門委員会での議論や EDSFair2009 実行委員会からの提案もあり、期初、開催法の検討も行ったが、結果的には、従来と同時期、同形態で開催した。

2008 年後半からの厳しい経済状況を反映して、出展者数は前回比 21%減であったが、実行委員会、事務局による取り組みが功を奏し、前回をやや上回る（＋2%）来場者があった。日経 Tech-On! 記事でも「最近は来場者数が減少するイベントが多い中で、大健闘と言える結果である」と評価された。

特に EDSFair10 周年を記念しての特別企画「各社の No.1 設計者が語る“私の設計”」は、多くの来場者を集め、メディアからも注目された。毎回、好評の EDA 技術専門委員会、企画 WG による特設ステージの 5 セッションも前回比 10%超の聴講があり、EDSFair の活性化、来場者増に貢献した。

来場者アンケートでみる、来場者の来場目的への期待達成度も一項目以外は前回以上の結果であった。出展者アンケートによる、出展目的に対する期待達成度についても、特に主な出展目的である、製品の販売促進、新規顧客獲得、新製品・技術のアピールなどの項目で、ここ数年来でもっとも高い数字となったことは強調しておきたい。この結果については、出展者数減（ESFair の規模縮小）に伴い、各出展者が期待レベルを下げたとの面が全くないとは言えないが、上記、特設ステージの企画に加え、事務局提案による新規取り組みによる来場者数の前回比微増との結果が、出展目的期待度アップの最大要因と考えられる。

EDSFair2010 実行委員会では、出展誘致 WG、来場誘致 WG を設けず、事務局中心の運営をお願いした。今後もぜひその方向とすべきである。次年度以降の JEITA　EDA 技術専門委員会体制下では、現在、EDA 技術専門委員会メンバーが担当している特設ステージ企画についても、担当範囲を見直すなどの検討が必要である。

3.2 システム・デザイン・フォーラム 2010

3.2.1 はじめに

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、例年、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。

まず、1990 年から 1994 年にかけて 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とした“EDA 標準化フォーラム”を 4 回開催し、つづいて、EDA 技術専門委員会の活動に係る内容の発表、討論の場を目的として“EDA フォーラム”を 1999 年から 2002 年にかけて 2 回開催してきた。また、2004 年には、最新の設計技術、課題を設計事例とともに紹介する“システム・デザイン・セミナー”を、2005 年には “システム・デザイン・フォーラム 2005”を、2 日間の日程で開催した。この“システム・デザイン・フォーラム 2005”では、1 日目に SystemVerilog ユーザ・フォーラムと SystemC ユーザ・フォーラムを、2 日目に 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行っている。2006 年には、“システム・デザイン・フォーラム 2006”として、“SystemVerilog ユーザ・フォーラム”と“SystemC ユーザ・フォーラム”の 2 セッション構成で、両設計言語の標準化動向の紹介、チュートリアル、設計適用事例の紹介を行った。2007 年は“システム・デザイン・フォーラム 2007”として、“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 セッションを開催した。2008 年は“システム・デザイン・フォーラム 2008”として、“SystemC ユーザ・フォーラム”と“Power Format フォーラム”の 2 セッションを開催した。“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行った。“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。昨年は“システム・デザイン・フォーラム 2009” として、「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」を開催した。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した。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。

今年は、従来通りの開催として、システム・デザイン・フォーラム 2010 を検討してきたが、以下の理由にて、開催を見送ることとした。

- 今年度のEDA-TCの各活動のフェーズから見て、有償で開催するシステム・デザイン・フォーラムに合致するテーマが無かった。
- EDS Fair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。

3.2.2 システム・デザイン・フォーラム 2010 活動報告

今年はシステム・デザイン・フォーラムを非開催であったため、システム・デザイン・フォーラム WG 活動について、ここに報告する。

- **開催可否検討**

2010 年は、テーマ設定、予算逼迫の課題があり、まず開催可否検討から取り組む方針とした。開催可否は、５月、６月の 2 回の WG にて、以下の基本目標に従い、十分な検討を行なった。

＜SDF2010 基本目標＞

EDA 技術専門員会の活動をベースとし、委員会の標準化活動の成果を国内外に広くアピールする。

更に、来場のシステム技術者、設計技術者にメリットがあるテーマを設定し、EDSFair の更なる発展にも貢献する。

- **SDF2010 テーマ候補の選定、検討**

SystemC　WG、ナノ世代物理設計 WG、PowerFormat　WG にて各テーマ候補の検討を実施した。SystemC WG は、例年通りの、有償で提供するに適切なテーマがない、という結論に至った。ナノ世代物理設計 WG は、次の活動を検討する年に当たり、新規事項で半日のフォーラムを開催する題材が準備困難な状況にあった。また、PowerFormat WG は、EDA ベンダーが無償の技術セミナーを開催する段階となったことから、一般的な技術情報としての提供を、有償のセッションにて行なう必要がなくなっていた。各 WG からの検討状況を踏まえて、無償開催（特設ステージ）ができないか、継続検討していくこととした。

- **特設ステージ前提のテーマ再検討**

EDSFair 特設ステージでの検討会を検討するため、企画 WG にて事前確認を実施した。その結果、システム・デザイン・フォーラムとしては、標準化の成果を発表したいものの、特設ステージの考え方を配慮した中で、システム・デザイン・フォーラムの基本目標を達

成できるテーマ設定が必要となることを、確認した。

これを踏まえた再検討の結果、「基本目標に添った形で開催したいが、今回のテーマ選定結果を鑑みると、特設ステージの期待に応えうるテーマ提案は困難」という認識に至った。

これらの結果より、システム・デザイン・フォーラム 2010WG として、2010 年は非開催の方針で進めるという結論に達することとなった。

3.2.4 まとめ

今年は、残念ながら上記検討結果より、非開催となったシステム・デザイン・フォーラムであるが、EDA 技術標準化推進、業界内での普及･促進活動の１つとして、引き続き位置づけられるものと思われる。

昨年のフォーラム終了後のアンケートを振り返ってみると、SystemC ユーザ・フォーラム 2009 の満足度は、「満足」だけを取り上げると、17%から 26%へ増加がみられている。また、ナノ世代物理設計フォーラムの満足度（満足＋まあ満足）が 76%となっており、今後の参加希望についての設問でも、「希望する」（参加する＋内容次第で参加する）が 88%となり、フォーラム内容の充実と開催への期待が確認されている。その他、意見、要望の設問でも、「有料でも 2,000～3,000 円であれば、ありがたい」、「DFM、DFY Technology フォーラムを開いて欲しい」等、今後に対する期待をいただいている。

来年は、引き続き、SDF 基本目標に沿ったテーマの選定と、無償開催の可否検討を検討事項と捉えて、活動を進めていきたい。

3.2.5 システム・デザイン・フォーラム 2010WG 委員（敬称略、順不同）

主　査	山 田 陽 一	セイコーエプソン㈱
委　員	斎 藤 茂 美	ソニー（株）
同	長 尾　明	シャープ㈱
同	中 西 早 苗	NEC エレクトロニクス㈱
同	金 本 俊 幾	㈱ルネサステクノロジ
同	中 森　勉	富士通マイクロエレクトロニクス㈱
同	今 井 正 治	大阪大学
同	若 林 一 敏	日本電気㈱
アドバイザ	江 田　努	ローム㈱
オブザーバ	太 田 光 保	パナソニック㈱
事 務 局	小田 佳代子	日本エレクトロニクスショー協会

４．添付資料

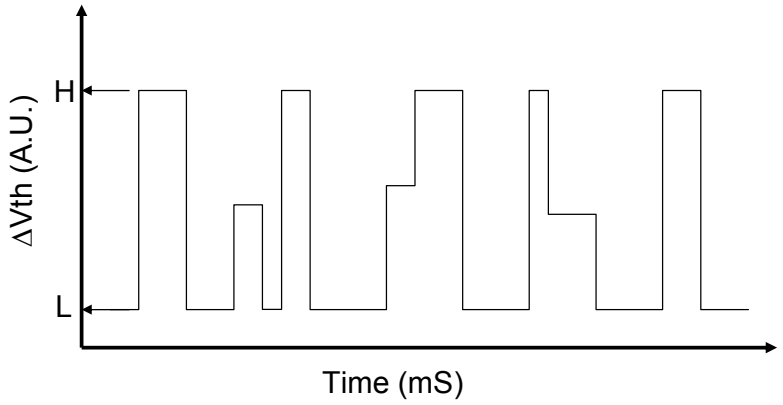
1. RTN (Random Telegraph Noise) のインバータ遅延ばらつきに与える影響の解析

動機と目的

- 動機
  - ランダム・テレグラフ・ノイズ（RTN）が微細CMOSの遅延ばらつきに与える影響がクローズアップされてきた。
  - この影響をどう取り扱うかの方法論、設計手法に関する研究と議論が必要。
- 目的
  - 22nmプロセス技術レベルを想定し、RTNによるインバータ回路遅延のばらつきを定量化すること。
  - 上記結果に基き、RTNに起因した設計上の課題を明確にすること。

RTNの概要

- MOSFETのゲート酸化膜中あるいは界面に、トラップ準位を発生させる欠陥が確率的に存在。
- トラップ準位への電子・正孔の捕獲・放出が確率過程的に発生。
- 電子・正孔のトラップによるMOSFETのしきい値電圧シフトと、それによる回路遅延ばらつきが発生。



RTNに関わる確率統計データ

- トラップ準位を持つTr.の確率分布
  - Poisson分布…Takeuchi [1]
- トラップ準位への電子・正孔捕獲によるしきい値電圧シフト量の分布
  - 対数正規分布…Tega [2]
  - 指数分布と重ね合わせ…Takeuchi [1]
- トラップ準位への捕獲・放出時定数の確率分布
  - 指数分布…Takeuchi [3], Campbell [4]
- 加工技術レベル依存性（トレンド）
  - Linearトレンド…Ghetti [5]
- 形状依存性 (LW依存性）
  - ∝1／SQRT(LW)… Tega [2]

MOSFETゲート領域に存在するトラップ準位数

- 55nmプロセスの例。ドレイン電流の変化レベルの水準数により測定。[1]
- トラップ準位数の分布はポアソン分布で良く近似できる。

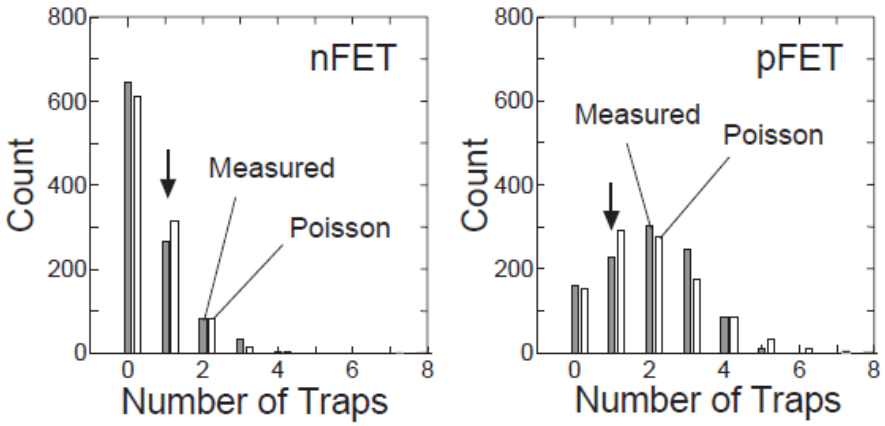
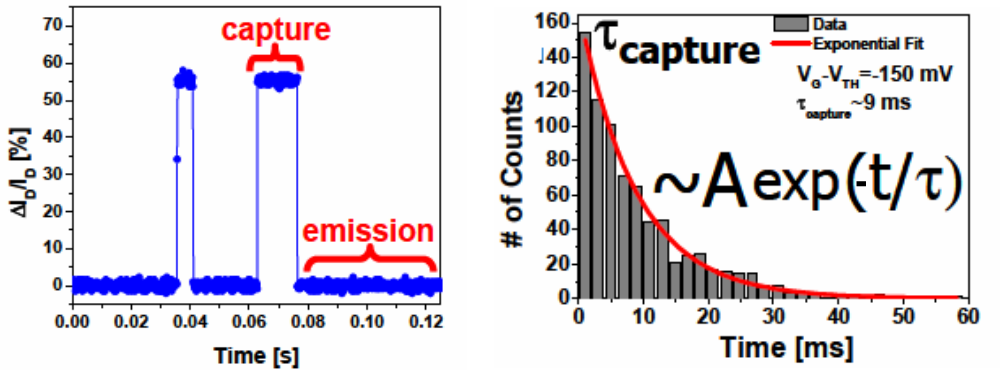


Fig.4 Measured number distributions of traps per single transistor.

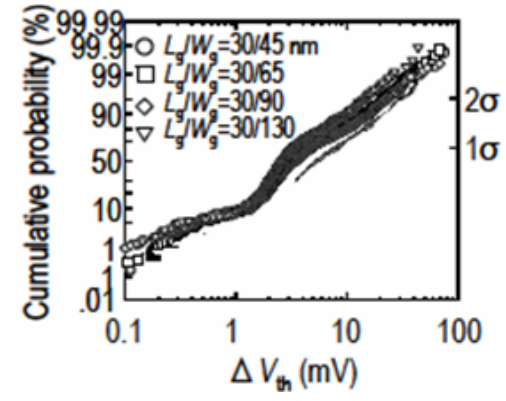
キャリア捕獲・放出時間（時定数）分布

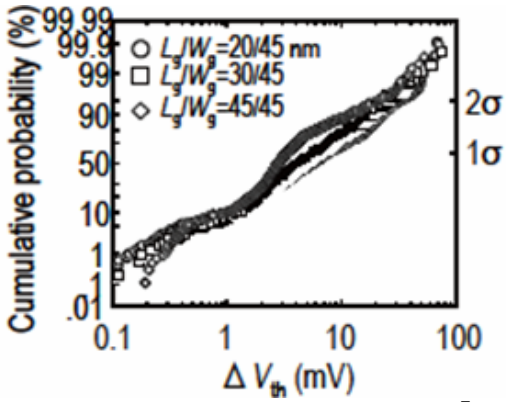
- トラップ準位へのキャリア捕獲・放出時定数は指数統計分布に従う確率過程現象。[2]
- 温度・バイアス電圧依存性を持つ。



トラップ準位特性の分布

- 1キャリア捕獲によるしきい値電圧シフト量（ΔV_{th}）は、欠陥位置に応じて変化し、分布を示す 。[3]
- 上記しきい値電圧シフト量分布はLognormal（対数正規）分布にほぼ従う。



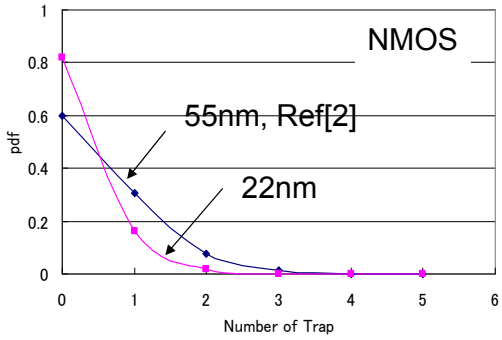


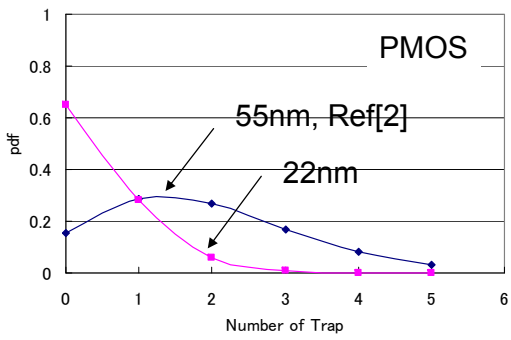
MOSFET当りの欠陥密度予測

- ゲート領域の欠陥密度はプロセスが微細化されても変わらないと仮定。
- 平均トラップ個数はゲート面積に比例。

$$Poisson\ Distribution;\ f(n)=\frac{e^{-\lambda}\lambda^{n}}{n!}$$

$$\lambda_{22nm}:\lambda_{55nm}=(LW)_{22nm}:(LW)_{55nm}$$





MOSFET当りのトラップ密度予測;数値表

- 22nmプロセスにおける単位ゲート面積当りのトラップ密度を予測。
- 22nm CMOS トラップ密度:

  NMOS: 平均トラップ個数 λ＝0.1984 個
  PMOS: 平均トラップ個数 λ＝0.4321 個

		Tech.(nm)	Area(nm^2)	λ
NMOS	Ref.[2]	55	6804	1.5
	This Work	22	900	0.1984
PMOS	Ref.[2]	55	6804	2.1
	This Work	22	1400	0.4321

インバータ:N & PMOSトラップ数のケース分けとその発生確率

- ポアソン分布から予測されるNMOS、PMOS Tr.のトラップ数は、97%の確率で合計2個以下と少ない。
- 従って、本解析では下記P1-P6の6ケースについて解析を行った。

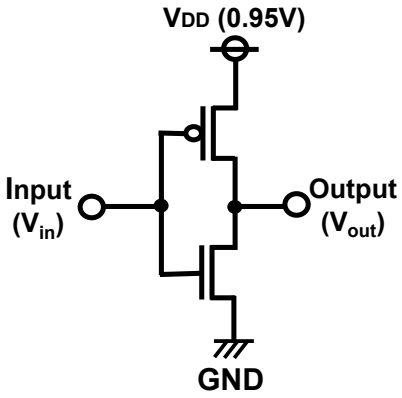
NMOSトラップ準位発生確率式 Poisson分布 λ＝ 0.1984
PMOSトラップ準位発生確率式 Poisson分布 λ＝ 0.4321

	欠陥個数				
	NMOS	PMOS	NMOS確率	PMOS確率	総合確率
P1	0	0	0.8200	0.6491	0.5323
P2	0	1	0.8200	0.2805	0.2300
P3	0	2	0.8200	0.0606	0.0497
P4	1	0	0.1627	0.6491	0.1056
P5	1	1	0.1627	0.2805	0.0456
P6	2	0	0.0161	0.6491	0.0105
Total					0.9738

インバータ回路遅延ばらつき解析

- 目的
  - 22nmCMOSインバータ回路遅延特性のRTN起因ばらつきを解析する。
- 方法
  - SPICEモデルパラメータ:PTM 22nm LP(Bulk)
  - 負荷容量:10fF
  - P1～P6各欠陥でのΔV_{th}量をLognormal分布(Page7 右図L/W=20/45より取得: μ=ln(1.57e-3), σ=1.456)に基きランダムに発生し、各々の遅延分布を得る
  - P1～P6の遅延分布を発生確率(総合確率)で重み付けの上重ね合わせ、最終的な遅延分布を得る
- 評価パラメータ
  - 立ち上がり遅延、立下り遅延の分布

SPICEモデルパラメータ等

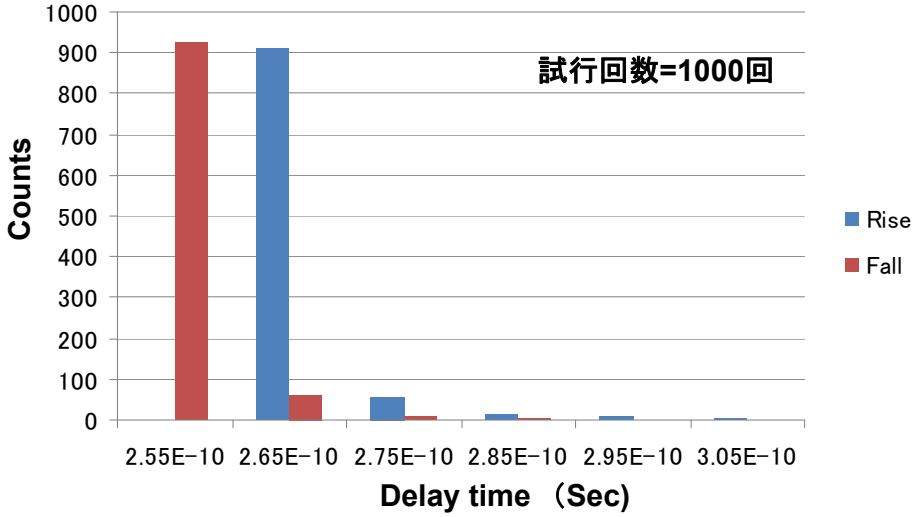


項目	値
SPICEモデルパラメータ	PTM 22nm LP (Bulk)
SPICE電流モデル	BSIM4
電源電圧V_{dd}	0.95V
基本(×1インバータ)のトランジスタサイズ	P W/L=70nm/20nm
	N W/L=45nm/20nm

ヒストグラム解析結果

RTNによるインバータ遅延時間の平均値シフト量は1%以下（無視できる量）

	w_RTN	w/o_RTN	Diff.	Ratio
Rise Delay	2.67139E-10	2.64790E-10	2.34900E-12	8.87118E-03
Fall Delay	2.56218E-10	2.58250E-10	-2.03200E-12	-7.86834E-03
Mean Delay	2.61678E-10	2.61520E-10	1.58264E-13	6.05168E-04

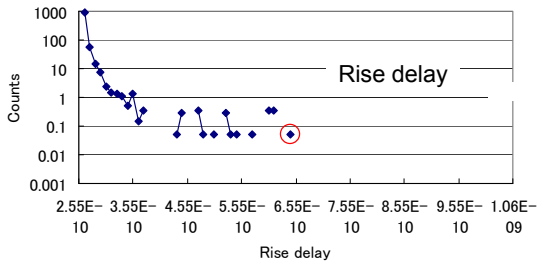


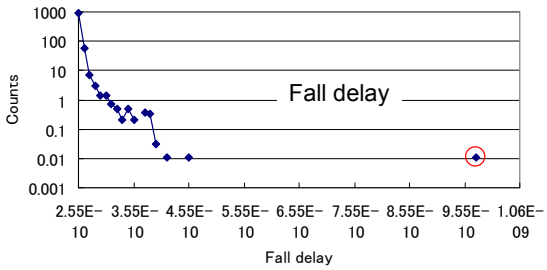
ヒストグラム解析結果：対数表示

・ 大多数のインバータは遅延特性がRTNにより劣化しない。
・ 本実験の試行回数においては、10ppmの割合で、遅延時間が2 ~ 4倍となるサンプルが存在した。

	Worst Delar	Typical delay	Ratio	Probability
Rise delay	6.45E-10	2.64790E-10	2.44E+00	70ppm
Fall delay	9.75E-10	2.58250E-10	3.78E+00	10ppm

○:Worst delay





残された検討項目

- 解析感度の向上：
  - モデル・分布の妥当性の評価
  - 試行回数、精度の向上
  - トラップケース数の拡大
- 回路遅延劣化への対策：
  - バーイン不良回路の加速検出とスクリーニング方法
  - 設計への考察
    - SRAMマクロ
    - 冗長やその他手法
- 実験検証：
  - 検証方法の確立（加速、統計解析、等）

まとめ

- RTNを取り巻く現象を分類し、確率・統計的に定量化を行い、解析した。
- 上記解析より、RTNがインバータ回路遅延特性に与える影響（ばらつき）を実験した。
- RTNは、その影響度が大きい場合、チップ内の欠陥と同様の現象として特性歩留まりに影響を持つ可能性を述べた。
- ただし、RTNは検出の難しい内在欠陥の現象であり、スクリーニング等対策が課題となると考えられる。

参考文献

[1] K. Takeuchi, T. Nagumo, S. Yokogawa, K. Imai and Y. Hayashi, “Single-Charge-Based Modeling of Transistor Characteristics Fluctuations Based on Statistical Measurement of RTN Amplitude,” 2009 Symposium on VLSI Technology, pp. 54 - 55, June 2009

[2] N. Tega, H. Miki, F. Pagette, D.J. Frank, A. Ray, M.J. Rooks, W. Haensch, K. Torii, “Increasing threshold voltage variation due to random telegraph noise in FETs as gate lengths scale to 20nm,” 2009 Symposium on VLSI Technology, pp. 50-51, June 2009

[3] T. Nagumo, K. Takeuchi, S. Yokogawa, K. Imai and Y. Hayashi, “New Analysis Methods for Comprehensive Understanding of Random Telegraph Noise,” IEDM 2009, pp. 759-762 , Dec. 2009.

[4] J.P. Campbell, J. Qin, K.P. Cheung, L.C. Yu, J.S. Suehle, A. Oates, K. Sheng, “Random Telegraph Noise in Highly Scaled nMOSFETs,” 47th Annual International Reliability Physics Symposium, pp.382-388, 2009.

[5] A. Ghetti, C. Monzio Compagnoni, F. Biancardi, A. L. Lacaita, S. Beltrami, L. Chiavarone, A.S. Spinelli, A. Visconti, “Scaling trends for random telegraph noise in deca-nanometer Flash memories,” IEDM 2008, pp.1-4, Dec. 2008.

2. 配線ばらつき感度/SSPEF フォーマットの適用範囲調査

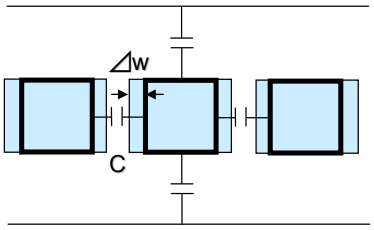
動機と目的

- 動機
  - 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つきSPEF：線型モデルが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の誤差を見積り、信号伝播遅延への影響を評価。
  - 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プロセスばらつき量の上限を明らかにする。

動機と目的

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

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

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

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

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

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

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

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

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

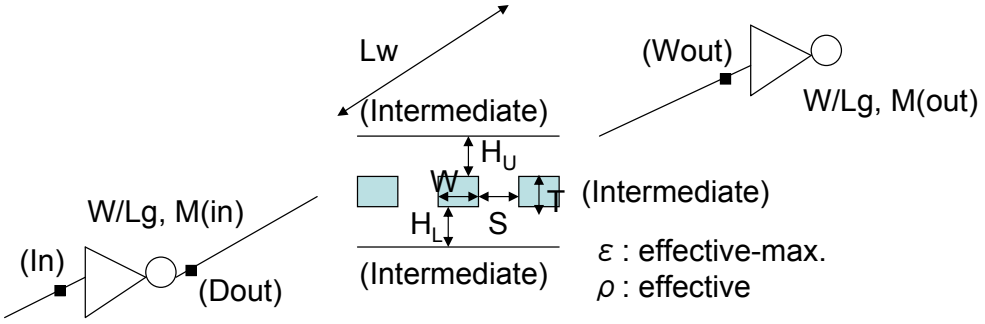
$$a=(\partial R/\partial T)/R_0$$

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

$$\Delta T=T-T_0$$

※本評価における感度モデルは、各ばらつきパラメータTypical近傍のRC感度で直線近似する

評価フロー



1. Typical のH, W(S), T に対してRC (/um)を求める。
2. 22nm相当のW/Lgに換算したNangate45nm(Low Power) ライブラリ、およびPTM22nmパラメータを使用。
3. 下記設計ポリシーにより、Lw(配線長), M(Multiplier)(in, out) を求める。

Lw, M(in, out) を求める設計ポリシー:

(1) Min = Mout

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

評価条件

算出条件は以下の通り。

・対象配線層：Intermediate層。
・上下配線層は平行平板。
・フリンジ容量,配線側壁の抵抗への影響は今回省略。
・同層配線は最小配線ピッチ(X1)と2倍ピッチ(X2)の2条件。

RC抽出: ITRS2007より22nmプロセスでのRCを算出。

		2007	2010	2013	2016
Intermediate wiring width (nm)	w	68	45	32	22
Intermediate wiring pitch (nm)	p	136	90	64	44
Intermediate wiring thickness (nm)	t	122.4	81	60.8	44
Height (dielectric thickness) between Intermediate wiring levels (nm)	h	108.8	72	54.4	39.6
Barrier/cladding thickness (for Cu intermediate wiring) (nm)		5.2	3.3	2.4	1.7
Cu thinning at minimum intermediate pitch due to erosion (nm), 10% × height, 50% area density, 500 μm square array		12	8	6	4
Conductor effective resistivity (μΩ-cm) Cu intermediate wiring including effect of width-dependent scattering and a conformal barrier of thickness specified below		3.43	4.08	4.83	6.01

評価条件

感度係数モデルは以下の関係を前提とした。

・容量変動量　：ばらつき量に比例。

$$C = C_{typ}\left(1 + \frac{\delta C}{\delta p}\right)$$

・抵抗変動量　：ばらつき量に反比例。

$$R = R_{typ} \Big/ \left(1 + \frac{\delta R^{-1}}{\delta p}\right)$$

感度算出手法：
感度算出時のフィッティング範囲はTyp値の0.1%とした。

配線ばらつきパラメータ：　次の3変数をパラメータとし、下記の範囲でそれぞれ独立して変動させた。

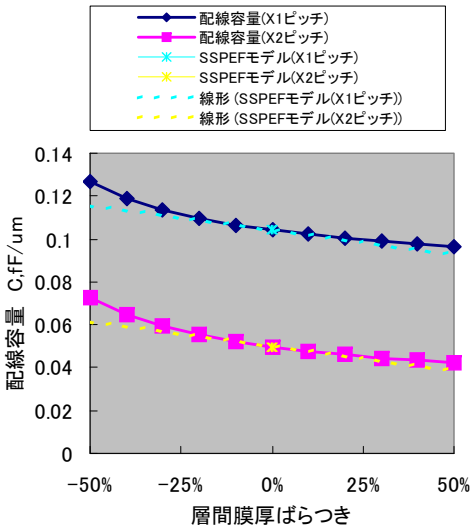
・層間膜厚(H)　　-50%～+50%、刻み幅10%
・配線幅(W)　　-50%～+50%、刻み幅10%
・配線厚(T)　　-50%～+50%、刻み幅10%

RC抽出結果①～層間膜厚ばらつき～

・今回の配線断面モデルでは、層間膜厚の変化に対して配線抵抗の変化はない。したがって、SSPEFモデルでの誤差は存在しない。

・配線容量は層間膜厚が薄くなる側でSSPEFモデルとの誤差が大きくなる。

・X1ピッチよりX2ピッチでの誤差が大きいが、絶対値としては同じである。



RC誤差評価結果①～層間膜厚ばらつき～

層間膜厚ばらつきまとめ

Intermediate(X1)	−50%	−40%	−30%	−20%	−10%	0%	10%	20%	30%	40%	50%
配線抵抗 [Ω/um]	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09
配線容量 [fF/um]	0.127	0.119	0.114	0.11	0.107	0.104	0.102	0.1	0.099	0.098	0.097
SSPEFモデル誤差(R)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SSPEFモデル誤差(C)	9%	5%	3%	1%	0%	0%	0%	1%	2%	3%	4%

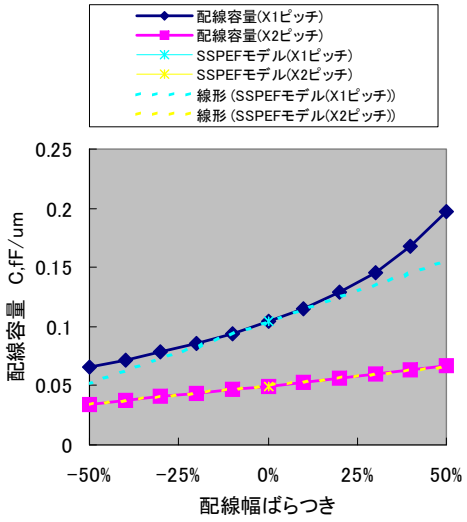
Intermediate(X2)	−50%	−40%	−30%	−20%	−10%	0%	10%	20%	30%	40%	50%
配線抵抗 [Ω/um]	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09	62.09
配線容量 [fF/um]	0.072	0.065	0.059	0.055	0.052	0.05	0.048	0.046	0.045	0.043	0.042
SSPEFモデル誤差(R)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SSPEFモデル誤差(C)	16%	9%	5%	2%	0%	0%	0%	2%	4%	6%	9%

RC誤差評価結果②　～配線幅ばらつき～

・配線幅の変化に対して、配線抵抗は反比例している。反比例で近似したSSPEFモデルと一致するため誤差は存在しない。（側壁の影響等を無視しているため）

・配線容量は配線幅に比例して大きくなるが、その割合は配線ピッチが狭い方がより大きい。カップリング容量依存が大きいと言える。

・配線容量におけるSSPEFモデルの誤差は配線ピッチが狭い方がより大きくなる。



RC誤差評価結果②　～配線幅ばらつき～

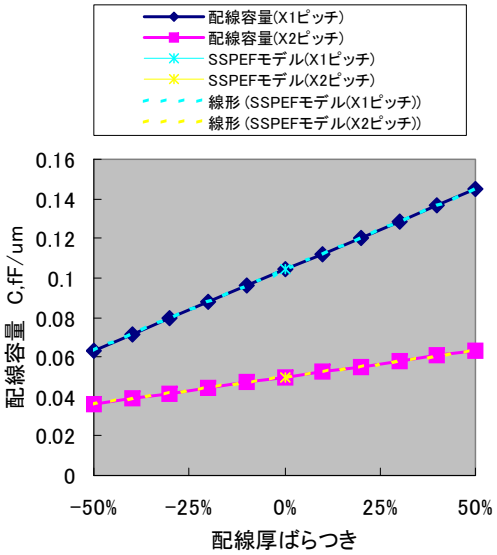
配線幅ばらつきまとめ

Intermediate(X1)	−50%	−40%	−30%	−20%	−10%	0%	10%	20%	30%	40%	50%
配線抵抗 [Ω/um]	124.2	103.5	88.7	77.61	68.99	62.09	56.44	51.74	47.76	44.35	41.39
配線容量 [fF/um]	0.066	0.072	0.078	0.086	0.094	0.104	0.115	0.129	0.146	0.167	0.197
SSPEFモデル誤差(R)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SSPEFモデル誤差(C)	21%	13%	7%	3%	1%	0%	1%	3%	7%	13%	21%
Intermediate(X2)	−50%	−40%	−30%	−20%	−10%	0%	10%	20%	30%	40%	50%
配線抵抗 [Ω/um]	124.2	103.5	88.7	77.61	68.99	62.09	56.44	51.74	47.76	44.35	41.39
配線容量 [fF/um]	0.035	0.038	0.041	0.044	0.047	0.05	0.053	0.056	0.06	0.063	0.067
SSPEFモデル誤差(R)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SSPEFモデル誤差(C)	2%	1%	1%	0%	0%	0%	0%	0%	1%	1%	1%

RC誤差評価結果③　～配線厚ばらつき～

・配線厚の変化に対して、配線抵抗は反比例している。反比例で近似したSSPEFモデルと一致するため誤差は存在しない。

・配線容量は配線厚に比例して大きくなる。比例近似したSSPEFモデルと一致するため誤差は存在しない。



RC抽出結果③　～配線厚ばらつき～

層間厚ばらつきまとめ

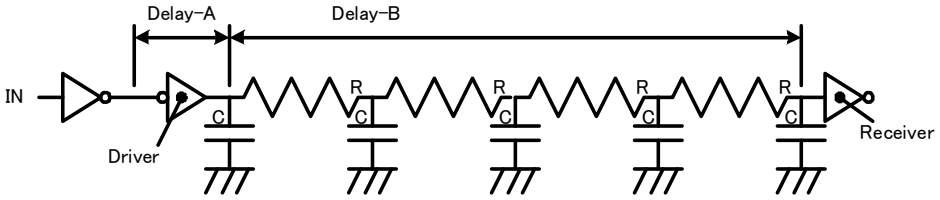
Intermediate(X1)	−50%	−40%	−30%	−20%	−10%	0%	10%	20%	30%	40%	50%
配線抵抗 [Ω/um]	124.2	103.5	88.7	77.61	68.99	62.09	56.44	51.74	47.76	44.35	41.39
配線容量 [fF/um]	0.063	0.072	0.08	0.088	0.096	0.104	0.112	0.12	0.129	0.137	0.145
SSPEFモデル誤差(R)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SSPEFモデル誤差(C)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

Intermediate(X2)	−50%	−40%	−30%	−20%	−10%	0%	10%	20%	30%	40%	50%
配線抵抗 [Ω/um]	124.2	103.5	88.7	77.61	68.99	62.09	56.44	51.74	47.76	44.35	41.39
配線容量 [fF/um]	0.036	0.039	0.042	0.044	0.047	0.05	0.052	0.055	0.058	0.061	0.063
SSPEFモデル誤差(R)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SSPEFモデル誤差(C)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

遅延に対するインパクト評価

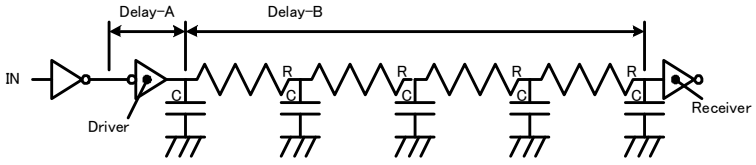
SPICEモデル　：PTM 22nm LP (Bulk)
SPICE電流モデル　：BSIM4
電源電圧(VDD)：0.95V
基本INV　：PMOS (W=135nm*22/45,L=22nm)
　：NMOS (W=90nm*22/45,L=22nm)
　～Nangate社45nmライブラリの基本INVより～

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を調査した。
　Delay-A：Driverでの遅延
　Delay-B：ターゲットの配線遅延



遅延に対するインパクト評価

評価条件



次の条件での評価を行った。

	Driver	Receiver	遅延比(Delay-A:Delay-B)
評価条件①	INV-X32	INV-X32	1:1
評価条件②	INV-X32	INV-X1	1:10
評価条件③	INV-X32	INV-X1	1:1
評価条件④	INV-X32	INV-X32	1:10

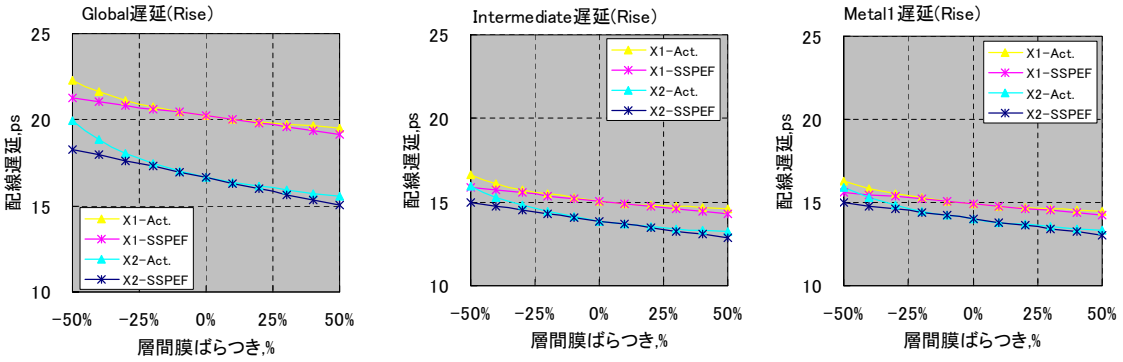
各評価条件につき、配線ピッチ2種類(X1,X2)、ばらつき成分(H,W,T)を±50%範囲で変動させ、シミュレーションを行った。

なお、RCを決めるための配線長(Lw)は、配線ばらつきのないTyp条件で“Delay-A＝Delay-B”となる配線長を用いた。

評価結果①

評価条件①で層間膜厚をばらつかせた場合のシミュレーション結果を示した。

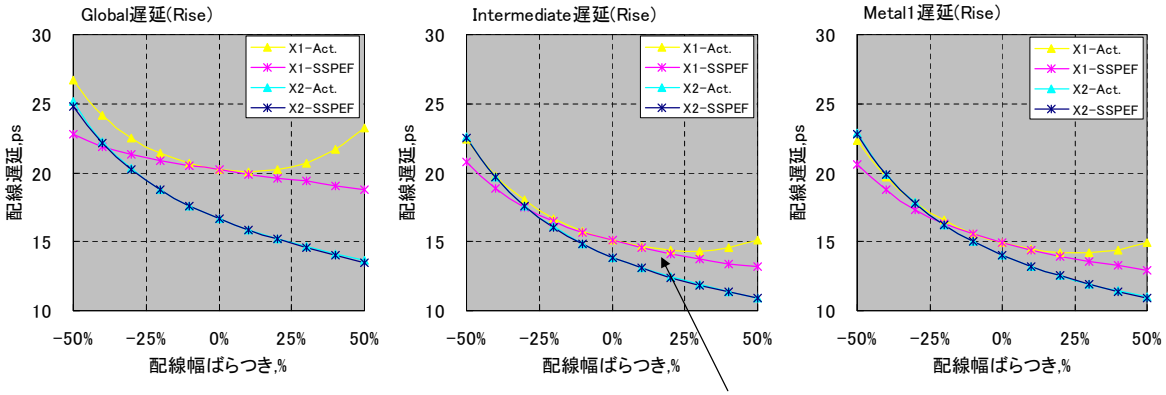
・ばらつきが大きい場合に多少誤差が生じるが、その割合は小さい。
・配線層に関わらず同様の傾向を示している。
・SSPEFモデルの感度誤差と同様の傾向を示している。



評価結果②

評価条件①で配線幅をばらつかせた場合のシミュレーション結果を示した。

・2倍ピッチの配線(X2)では誤差はほとんど生じていない。
・最小ピッチの配線(X1)では誤差が生じるが、Global配線で顕著である。
・SSPEFモデルの感度誤差と同様の傾向を示しているが、配線幅大の時の誤差がより大きくなっている。

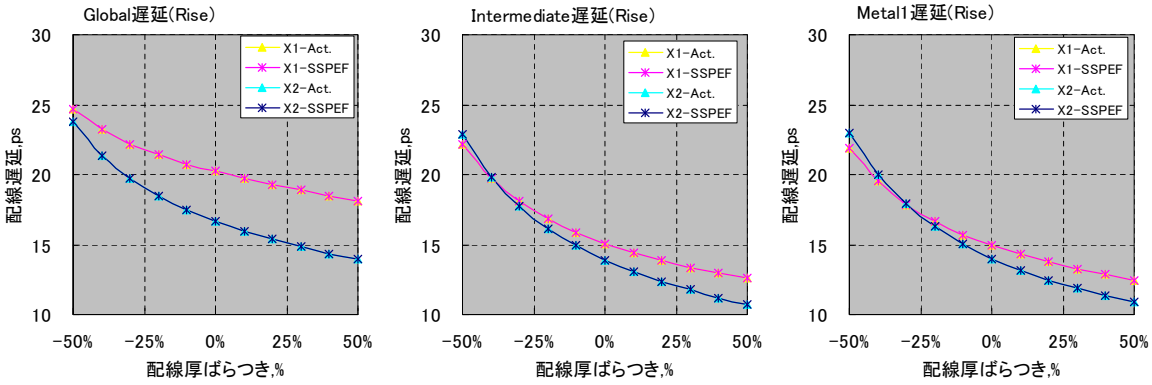


配線幅ばらつき20%→遅延誤差2%
30%→遅延誤差4%

評価結果③

評価条件①で配線厚をばらつかせた場合のシミュレーション結果を示した。

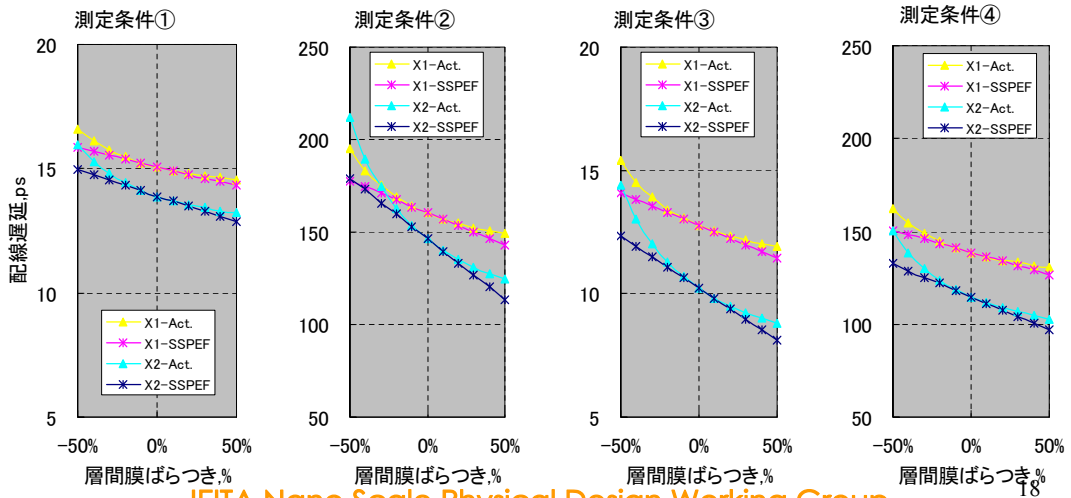
・SSPEFモデルの感度誤差がないため、実際の遅延時間と結果は一致する。



評価結果④

各評価条件でのIntermediateに対する層間膜厚の影響を示す。

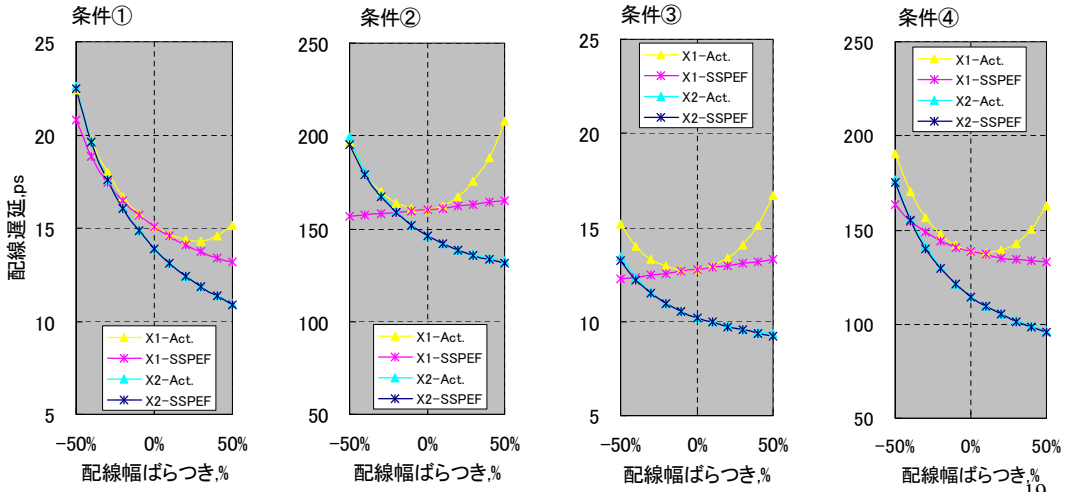
・いずれの測定条件においても同様の傾向を示す。
・上下層との配線容量が最も支配的な条件②のX2ピッチが最も誤差が大きくなっている。



評価結果⑤

各評価条件でのIntermediateに対する配線幅の影響を示す。

・カップリング容量の小さいX2ピッチの場合、SSPEFモデルに対する誤差はほとんど生じない。
・ X2ピッチの場合大きな誤差が生じている。また、Receiverのサイズにより傾斜が異なっている。



まとめ

- ITRSロードマップ値を用い、22nmプロセス世代におけるばらつき量と、配線ばらつき感度/SSPEFフォーマットの誤差の評価を行った。評価結果によると、容量の見積もり誤差は配線幅ばらつきの影響が最も大きく、配線幅ばらつき10%のときは誤差1%、20%のときは誤差3%、30%のときの誤差は7%に及ぶことが分かった。

- さらにPTMライブラリを用いてSSPEF誤差の遅延に与える影響を見積もった。見積もりの結果、配線幅ばらつき20%のときの遅延誤差は2%、30%のときの誤差は4%に及ぶことが分かった。

- また22nmプロセス世代においては、配線幅ばらつきの抑制が精度向上に必要であることが分かった。

残された検討項目

- 評価方法の改良
  - 次世代プロセスノードにおける実用的回路を想定した回路モデルの検討
  - Field Solverの使用による容量見積もり精度向上
  - Monte Carlo Simulation等による遅延分布としての誤差評価
- IEEE1481のSSPEFフォーマットにおいて、誤差を最小限とする感度モデル近似手法の提案。また現行IEEE1481のSSPEFフォーマット使用に際しての制約事項の検討
- さらなる誤差削減のためのSSPEFフォーマットの提案。および、モデル誤差とEDAコスト（CPU時間、データ量）とのトレードオフ検討

Appendix

- 1/29 JEITA-DASC informative meeting 資料
  日時　1月29日（金）　9:30－11:00
  場所　パシフィコ横浜内

Nano-scale Physical Design Working Group

- Activities
  - Pick out the problems on physical design and verification of LSI in the next generation technology node.
  - Make design rules and guidelines which specifies the library exchanged between a semiconductor vendor and its customer, design information, etc.
  - Standardize libraries, which improve accuracy, compatibility, and efficiency of physical design and verification.
  - Build benchmarking data set, which test accuracy of library verification.

Nano-scale Physical Design Working Group

Note :

First, I introduce Background and objective.

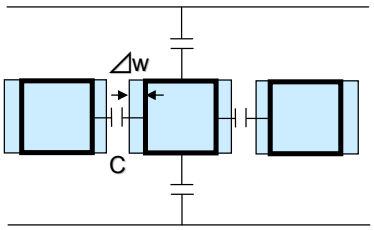
A new design issues has been appearing with evolution of a semiconductor device and interconnect technology. Moreover, the techniques or libraries, which each company developed to resolve these issues, need long time to be standardized even after the technology becomes mature. It obstructs cost reduction of developing and supporting the design environment, and then it makes communication between semiconductor vendors and their customers difficult.

Based on the background, this working group investigates and standardizes the following subjects for the purpose of contributing to realizing more efficient design environments.

- To pick out the problems on physical design and verification of LSI in the next generation technology node.
- To make design rules and guidelines which specifies the library exchanged between a semiconductor vendor and its customer, design information, etc.
- To standardize libraries, which improve accuracy, compatibility, and efficiency of physical design and verification.
- To build benchmarking data set, which test accuracy of library verification.

Nano-scale Physical Design Working Group

- Activities related to P1481-SSPEF
  - Concentrating on looking at the linear sensitivity model defined in SSPEF, to specify suitable range of the global process variability in upcoming 22nm era.



Interconnect cross-section

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

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

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

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

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

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

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

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

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

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

$$\Delta T=T-T_0$$

Nano-scale Physical Design Working Group

Note :

Actually, we have been waiting for the official approval of the revised P14 81 , date to be able to be referenced which includes SSPEF, abbreviated for Sensitivity SPEF, which includes new features for statistical timing analysis. Statistical timing analysis helps physical design of LSI a lot by reducing extra margins or costs from the variability point of view.

So far, we are concentrating on looking at the linear sensitivity model defined in SSPEF, to specify suitable range of the global process variability in upcoming 22nm era.

It is important to control our Fabs. to be suitable for this format, otherwise, we need to ask P14 81 to make additional enhancements to this model.

We will publish the results of our work through conferences and / or journal papers.

3. 三次元実装(3D-IC)に関する論文調査

動機と目的

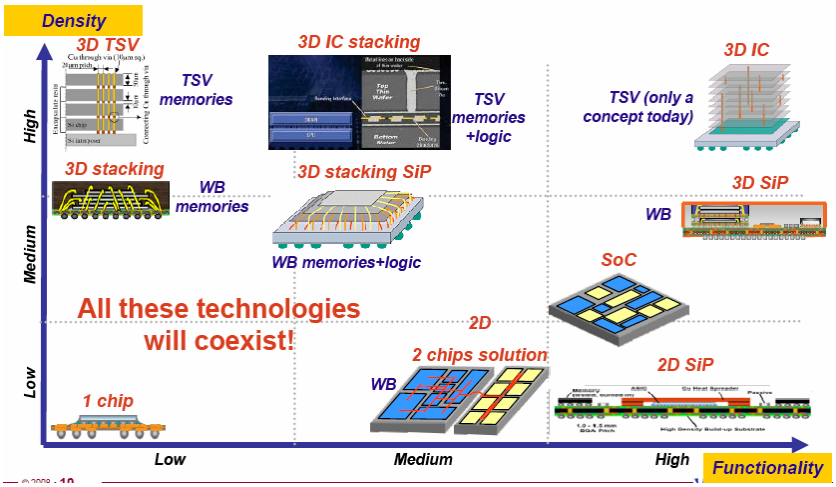
- 動機
  - 微細化の限界を打破する手段として、3次元の集積技術が注目を集めている。
  - アプリケーションとしては、Flashメモリ、DRAM、イメージセンサ搭載LSIの用途が主。
  - SoC/MCUの設計的観点で3次元化の設計的なメリット・デメリット、トレードオフが議論されている例は少ない。

参考：EE Times記事～ISSCC 2009】慶應大らがCPUとSRAMを誘導結合で3次元実装、「電力も面積も大幅減」

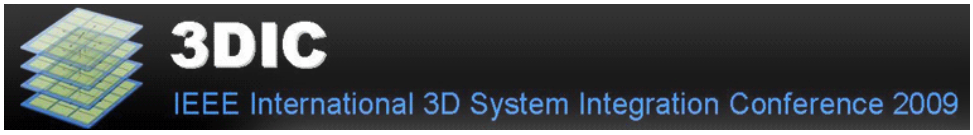
シリコン貫通電極について黒田氏は、「コストが高いという課題を抱えている。同技術の推進団体が掲げる目標投資額からざっと試算すると、2008年の時点でDRAMチップ1枚当たり25円程度のコスト増になる。これは、現実的には受け入れられないだろう」と指摘する。

動機と目的

- 目的
  - 3次元実装、とくにTSVに関するトレードオフ、SoC/MCUの分野での適用可能性を明らかにする。



文献調査対象



IEEE International
3D System Integration Conference 2009
Technical Program
September 28-30, 2009, San Francisco, California, USA

上記の国際会議予稿のなかから、面積、速度、パワーなどの設計的観点で3次元(3D)化のメリット、デメリットを議論されているものをピックアップした。

文献紹介

R.Egawa, J.Taday, H. Kobayashi, G. Gotoy, “Evaluation of fine grain 3-D integrated arithmetic units,” Proc. 3DIC 2009, pp.1 – 8

- 演算器にTSVを用いることで、性能面にどのような影響が表れるかを示した文献。
- 速度への影響を測定する手段として、チップ積層数(1～4)、TSVのRC、Critical-Path内のTSV数をパラメータとしている。
- Delayに対しては、TSVのRCによる影響が大きいという結果となっている。
  また、3D化による配線長短縮の効果も出ている。
- TSV遅延とLocal配線遅延との関係から2D,3Dを選択すれば良い。

表の2Dは1層。Criticalは2層でTSVを多くとったもの。　TSV1　:50Ω,120fF
TSV2　:100Ω,60fF
TSV3　:350Ω,3fF以下

Design	TSVs	Delay (ns)	FootPrint (μm^2)	No. of TSVs
2D	N/A	3.96	3710	0
2Layers	TSV1 TSV2 TSV3	3.98 3.83 3.67	1800	1
4Layers	TSV1 TSV2 TSV3	4.24 3.94 3.62	900	3
Critical	TSV1 TSV2 TSV3	10.17 7.07 3.61	2116	31

- 速度と電力、面積の関係に対しては、複数の回路構成を用いて評価している。
- 回路Aでは多層の3D化が良いという結果となっている。
- 回路Bでは電力が増加するため2Dの方が良いと思われる。
- 回路Cは2層3Dが最も遅くなるという特徴をもつ。
- このように最適解は回路構成により異なる。

回路A　CLA Adder

Design	TSVs	Delay (ns)	Power (mW)	FootPrint (μm^2)	No. of TSVs
2D	N/A	1.72	1.09	5540.1	0
2Layers	TSV1 TSV2 TSV3	1.89 1.84 1.69	1.06 1.06 1.04	2707	1
4Layers	TSV1 TSV2 TSV3	2.24 1.77 1.63	1.04 1.04 1.03	1421	3

回路B Kogge Stone Adder

Design	TSVs	Delay (ns)	Power (mW)	FootPrint (μm^2)	No. of TSVs
2D	N/A	1.57	1.41	9254	0
2Layers	TSV1 TSV2 TSV3	2.78 2.03 1.49	1.63 1.59 1.54	5399	62
4Layers	TSV1 TSV2 TSV3	7.27 2.03 1.43	1.84 1.59 1.52	2674.3	173

回路C 8-bit Array Multiplyer

Design	TSVs	Delay (ns)	Power (mW)	FootPrint (μm^2)	No. of TSVs
2D	N/A	12.79	48.7	207021	0
2Layers	TSV1 TSV2 TSV3	13.72 13.25 13.03	47 47.6 48.2	118674	62
4Layers	TSV1 TSV2 TSV3	13.09 12.66 11.95	38.7 39.6 40.8	54516	186

まとめ

- 3次元実装を行うことで、面積的なメリットは出せるが、速度や電力の面では回路構成により異なり2次元の方が良い場合もある。
- 回路構成を考慮した最適なフロアプランを行うことが、3次元実装を行う上での最も重要な要因となる。
- 3次元化する上では、TSVの構造が設計面に及ぼす重要な要因の一つとなる。
- 今後は最適なフロアプランニングの手法について、検討していく必要がある。

（参考:3D向け配置手法の提案）

石井, 齋藤, 松尾, 永井,「多層化半導体における回路長最短を目標とした配置手法の提案」,
情報処理学会第71回全国大会予稿集 pp.195-196

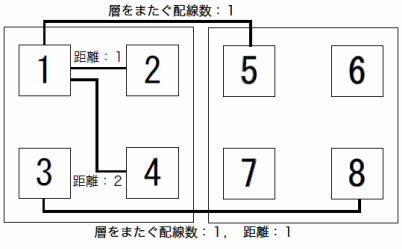


図 1: 配線の数え方

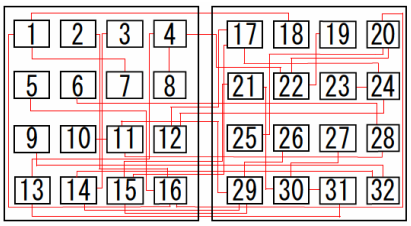


図 2: 4 × 4 の基盤が二層ある半導体回路の初期状態

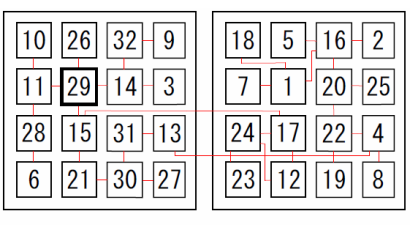


図 4: 提案手法による最適な配置

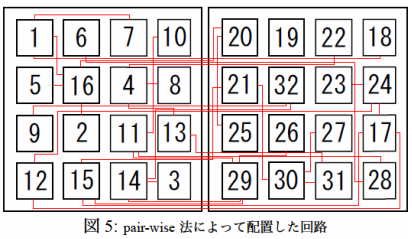


図 5: pair-wise 法によって配置した回路

4.2 SystemCワーキンググループ 2009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

 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	（NECエレクトロニクス）
委員	大島 良紀	（ルネサステクノロジ）
	西園寺 修	（日本シノプシス）
	清水 靖介	（ロームグループ・OKIセミコンダクタ）
	竹村 和祥	（パナソニック）
	立岡 真人	（富士通マイクロエレクトロニクス）
	旦木 秀和	（ソニー）
	長尾 文昭	（三洋半導体）
	牧野 潔	（メンター）
客員	今井 正治	（大阪大学）

（計11名）

 JEITA

目次

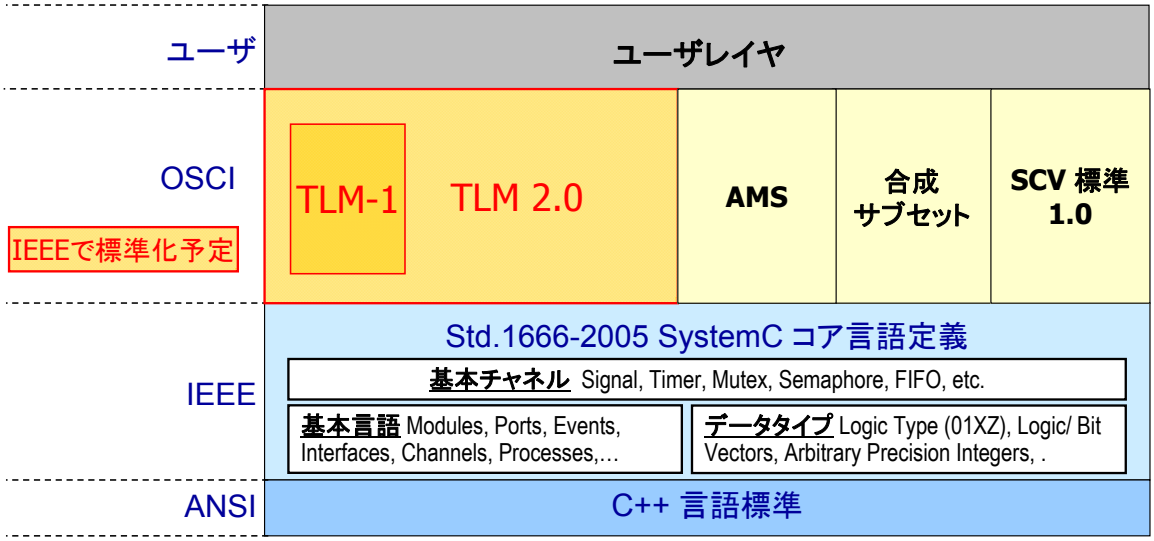
 JEITA

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
  - Transaction Level Modelingについては、TLM-2.0のIEEEでの標準化を2010年に予定
  - 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

 JEITA

SystemCの標準化の階層



OSCI資料(2007)に加筆

JEITA

4.2.1 SystemCワーキンググループ 概要

JEITA

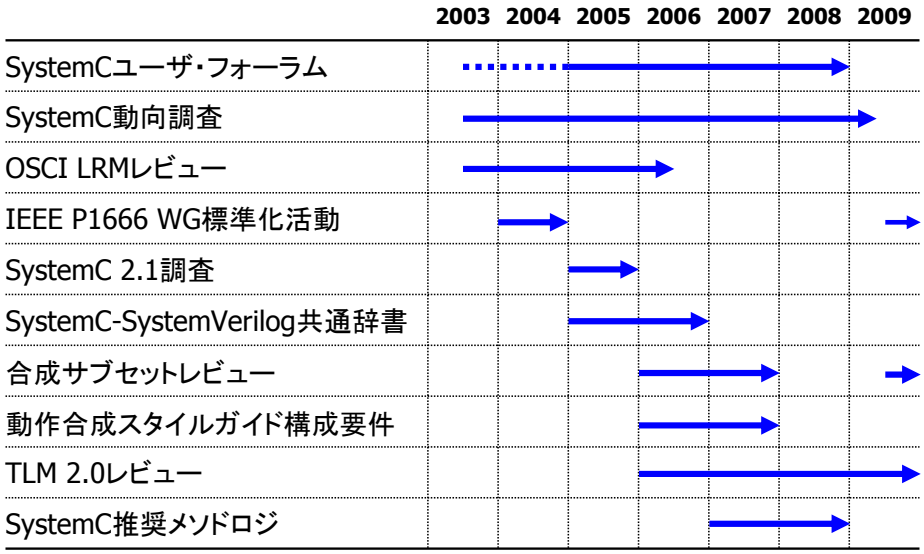
SystemCワーキンググループについて

- 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（SystemVerilogタスクグループと同時）
  - 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。

JEITA

SystemCワーキンググループの活動

- 活動の歩み



JEITA

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
- **SystemC動向・技術調査**
  - 2000年～2005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
  - 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。
- **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
  - 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

JEITA

2009年度報告資料について

4.2.2 TLM-2.0 LRMの日本語抄訳
- OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL (JA32)の日本語抄訳

4.2.3 TLM-2.0 LRM 差分まとめ
- User Manual (JA22)からの変更点について解説

4.2.4 SystemC Synthesizable Subset 1.3 draft 1章 日本語抄訳
- OSCI SystemC合成サブセット 1.3 ドラフト1章の日本語抄訳

4.2.5 JEITA ISSUE LIST for “SystemC Synthesizable Subset 1.3 draft”
- OSCI へ提出した合成サブセットにおける改善要望

4.2.6 SystemC Japan 2009 アンケート報告
- SystemC Japan 2009で実施したアンケート集計結果

4.2.7 ECSI HLS Workshop(ASP-DAC09)のまとめ
- 高位合成に関するワークショップ資料のまとめ

JEITA

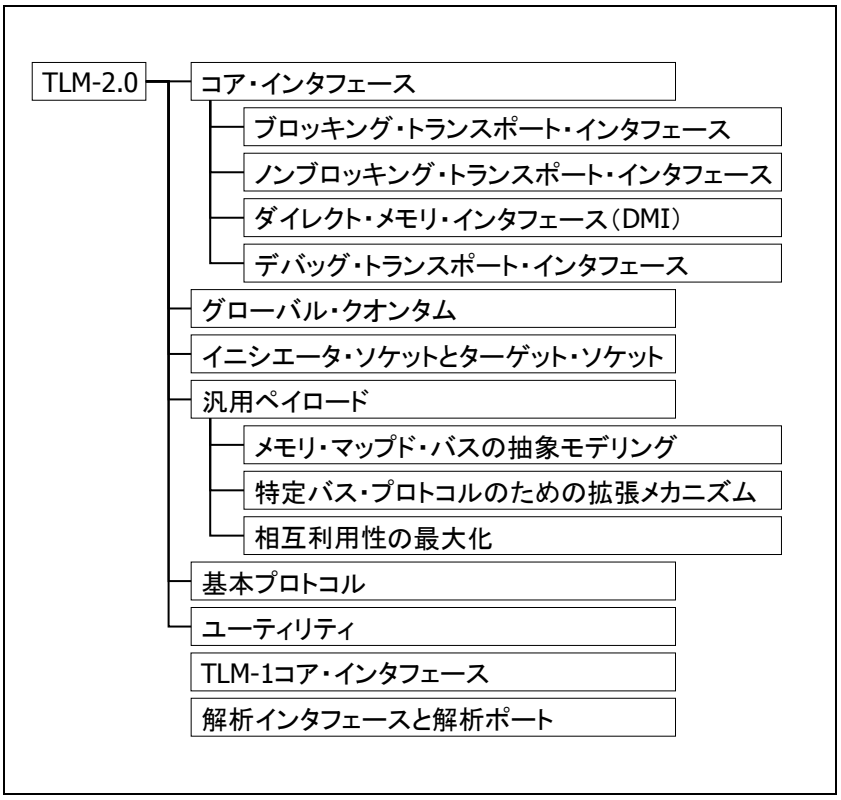
04-2-02 TLM-2.0 LRM の日本語抄訳

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。

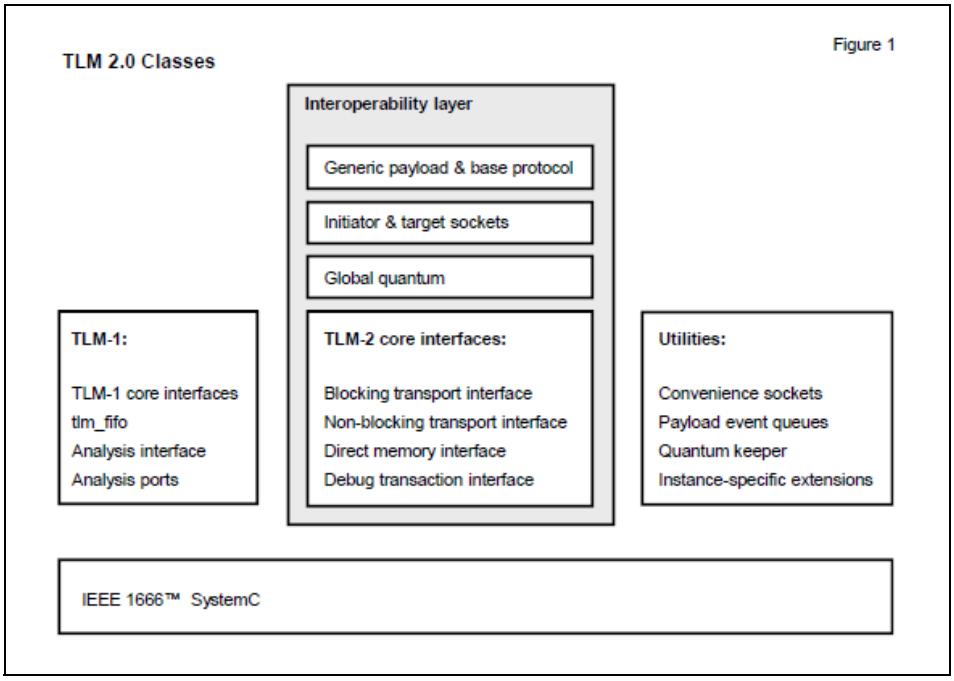
1. あらまし

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。これは、バージョン 2.0 ドラフト 1 とドラフト 2 を置換えるもので、これらと一般に互換性はない。標準のこのバージョンは、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、まとめて相互利用レイヤと呼ばれる。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・タイプとコア・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・タイプとコア・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を使う必要はまったくない。それにもかかわらず、スタイルの一貫性のため、可能ならばこれらのクラスを使うべきであり、TLM-2.0 標準の一部としてドキュメント化し、保守する。

- 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

  高速なルーズリィ・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。これら言及はコーディング・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。すべての裏づけとなるコード、例、ユニット・テストは記述しない。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相互利用レイヤ
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。相互利用性に本質的でない。
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 は値もしくはコンスト・リファレンス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り必要がある。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 文を呼ぶことでディレイを実装している。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値もしくはコンスト・リファレンス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 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。この方法は、新しいトランスポート・インタフェースでサポートされている。

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低レベルのプログラミングの仕組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。

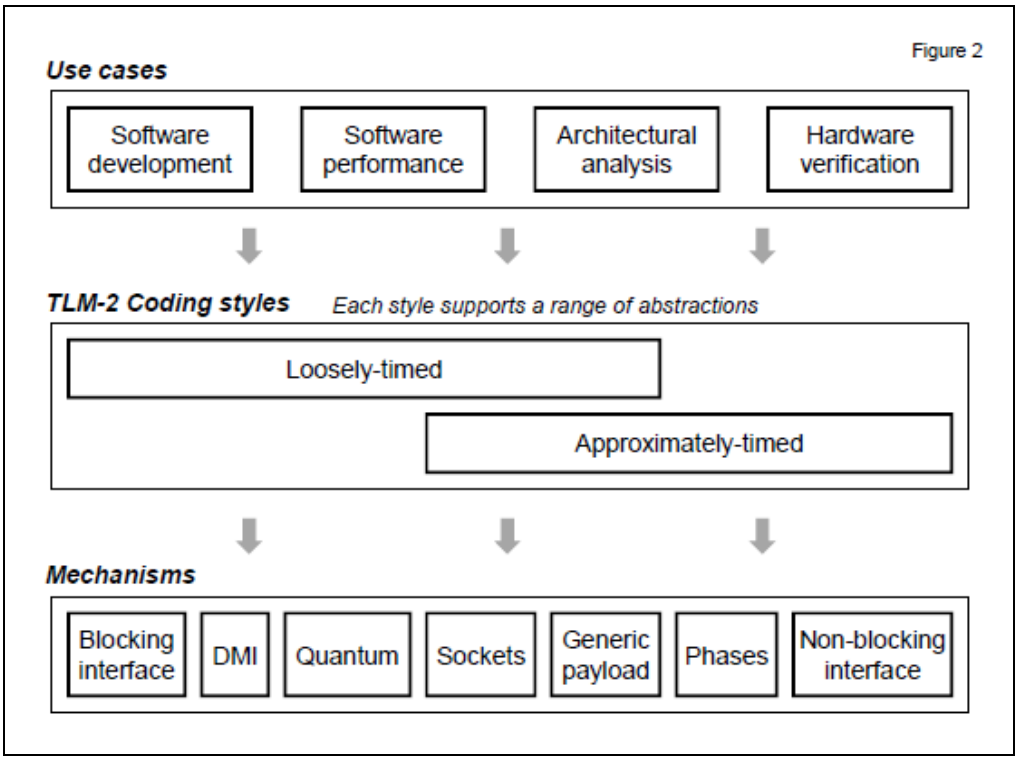
標準の TLM-2.0 インタフェースの定義はコーディング・スタイルの記述とは別物である。TLM-2.0 インタフェースはこの標準の規範を構成するもので、相互利用性を保証する。各コーディング・スタイルは機能、タイミング、コミュニケーションの様々な抽象化をサポートする。原理的には、ユーザは自分自身のコーディング・スタイルを作ることができる。

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。これはときどきアルゴリズム・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。なぜならが、定義により、トランザクションはコミュニケーションの抽象度で、シングル・スレッド・モデル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模倣するために複数の 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・インタフェースは、抽象的あるいは具体的プロトコル（タイミング情報に依存するしないに関わらず）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・イベントが個別プロセスの詳細なタイミングで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-20 の範囲外である。一方、複数の組込みソフトの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・モデルは強いあるいは弱い同期を使う。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ィ・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・モデルは、各トランザクションに対してプロトコルに特有な複数のタイミング・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、が必要かもしれない。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ー・タイムドと呼ぶ。



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、特定な抽象度、ソフトウェア・プログラミング・インタフェースではない。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、に制限する。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。なぜならは、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う。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使うのに適する。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をアクセスする。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ねばならない。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、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、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再実行する。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過程している。ソフトウェア・スタックがハードウェアのタイミングの低レベルの詳細に依存しない、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 文呼出しか、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・モデルは予測可能な実行を保証するため明示的同期の恩恵を受ける。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・ポイントがきっかり 4 つある。特定な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。プロセス相互作用にはディレイ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ディレイ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理ディレイと同じである。アノテートされたディレイは 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ちょうど 2 つのタイミング・ポイント、トランザクションの開始と終了を表す、を持つ。シミュレーション時間を使うが、プロセスはシミュレーション時間から一時的に分離されているかもしれない。各プロセスはある時間だけシミュレーション時間を先行するし、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ディレイはタイムアウト（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・コーディング・スタイル間の切換え

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、詳細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。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が。

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・モデルの存在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。

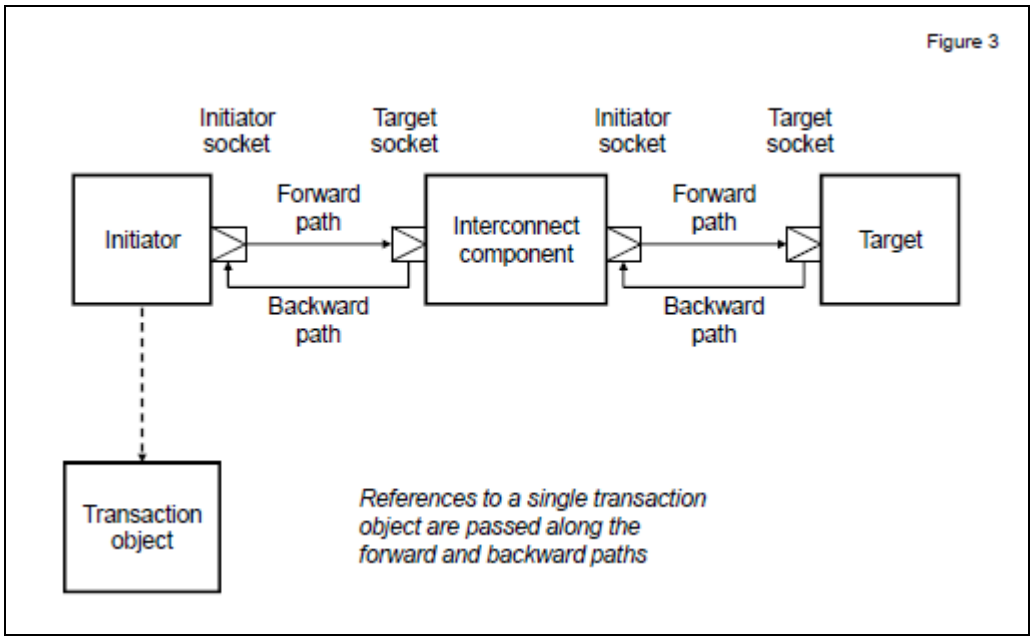
ブロック・トランスポート・インタフェースの強みは、1 回の関数コールでトランザクションを完了するモデルの単純なコーディング・スタイルを許すということである。ノンブロッキング・トランスポート・インタフェースの強みは、1 つのトランザクションに複数のタイミング・ポイントを結びつけることをサポートすることである。原理的に、どちらのインタフェースもテンポラル・デカップリングをサポートするが、アプロキシメイトリー・タイムド・モデルにおける複数タイミング・ポイントの存在がおそらくテンポラル・デカップリングの利点を無効化する。

3.3.9. ユースケースとコーディング・スタイル

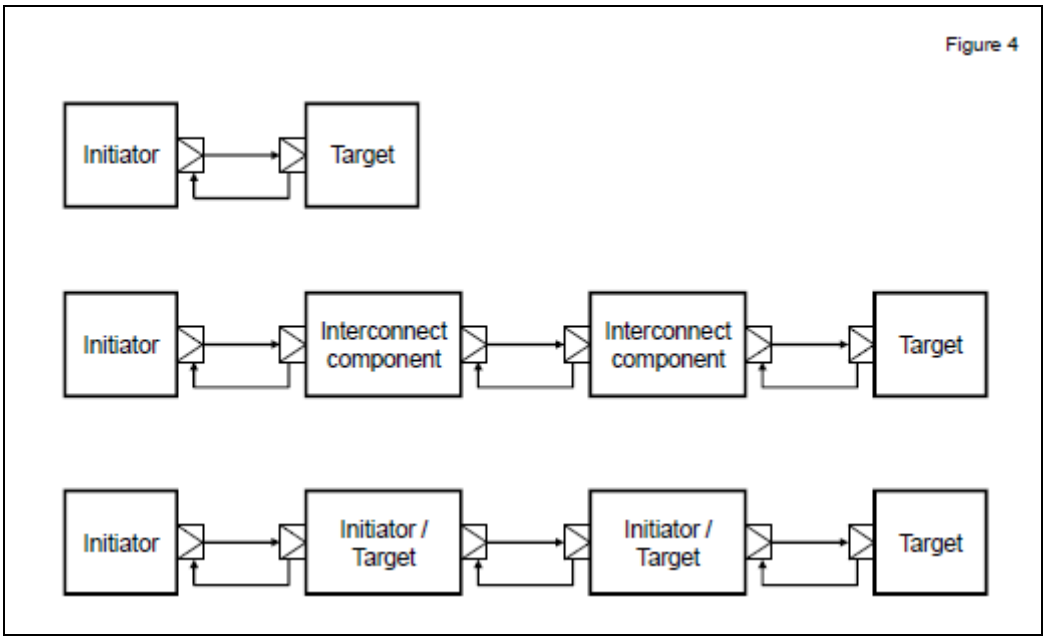
ユースケース	コーディング・スタイル
ソフトウェア・アプリケーション開発	ルーズリー・タイムド
ソフトウェア性能解析	ルーズリー・タイムド
ハードウェア・アーキテクチャ解析	ルーズリー・タイムド アプロキシメイトリー・タイムド
ハードウェア・パフォーマンス検証	アプロキシメイトリー・タイムド サイクル精度
ハードウェア機能検証	アンタイムド（検証環境） ルーズリー・タイムド アプロキシメイトリー・タイムド

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目的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をアクセスするが、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、アービタとラウターである。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。そのインターコネクト・コンポーネント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・メソッドはラウターのような 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、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 1 つのやり方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・パスに対応して 2 つのリターン・パスがある。しかし、意味は通常、内容から明らかである。）



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ねばならない。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・タイプ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ディレイ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・ソケットに結合されている。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）。これらの結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の選択の助けになる。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 の下で宣言されねばならない。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はず。

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ねばならない。

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

i) tlm_copyright メソッドは C string へ変換された copyright string を返さねばならない。

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る。

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で使われる為に設計された。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から単独に使用される事を許可するトランザクション・タイプと一緒に特殊化されるが、しかし相互利用性の利益の多くが失われるだろう。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・タイプに依存しており、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意図している。ブロッキング・トランスポート・インタフェースは、イニシエータが 1 回の関数呼び出しの経過の間、ターゲットと一緒にトランザクションを完了することをのぞむ所、トランザクションの開始と終了を記録する唯一の重要なタイミング・ポイントにおいて適切である。

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 の規格の一部としてまだ TLM-1 からのトランスポート・インタフェースとの計画的な類似性を持っているが、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、non-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、値が渡される（もしくは const の参照による）分離されたリクエストとレスポンスのオブジェクトを仮定するのに対し、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。この単一の引数は、現在のシミュレーション時間に相対的なトランザクションの開始及び終了の各々の時間を示すために、b_transport の呼び出しと b_transport からの応答の両方で使用される。

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・インタフェースがあらゆるタイプのトランザクションを転送するのに使用されてもよい事が意図されている。特別なトランザクション・タイプである 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・タイプである 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それら自身のトランザクション・タイプを代えることができる。異なるトランザクション・タイプ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どんな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通過するはずである。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・ソケットの為の 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かもしれない。

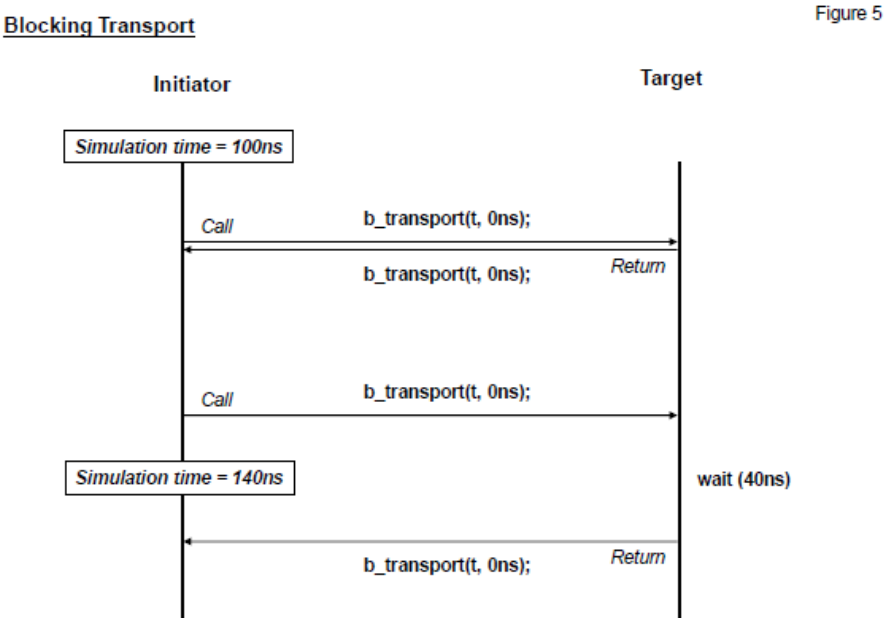
i) b_transport の実装が nb_transport_fw を呼び出す事を許可されるかどうかは、プロトコルと連携されたルールに依存する。基本プロトコルにおいて、便利ソケットの simple_target_socket が自動的にこの変換をする事が出来る。「9.1.2 シンプル・ソケット」を参照の事。

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

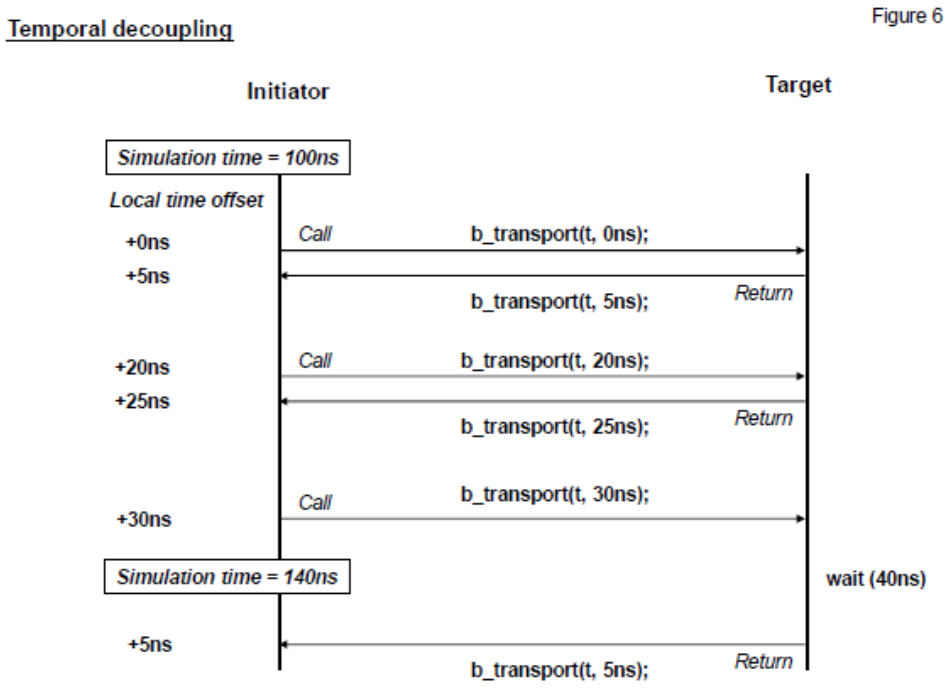
ブロッキング・トランスポート・メソッドは、すぐに応答するか（すなわち、現在の SystemC 評価フェーズで）、スケジューラに制御を譲渡し、シミュレーション時間より遅れたポイントでイニシエータに戻るかもしれない。イニシエータのスレッドはブロックされるかもしれないが、イニシエータの他のスレッドは、プロトコルに依存して、最初の関数呼び出しが応答する前に b_transport を呼び出すことが許可されるかもしれない。

Figure 5



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・タイムで走るかもしれないが、その場合以下に示すように、時間引数の為の非ゼロの値を 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抽象的な時間を与えるが、シミュレーション時間自身はイニシエータのスレッドが譲渡するまで進むことが出来ない。b_transport の本体は、それ自身で wait を呼び出すかもしれないが、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。

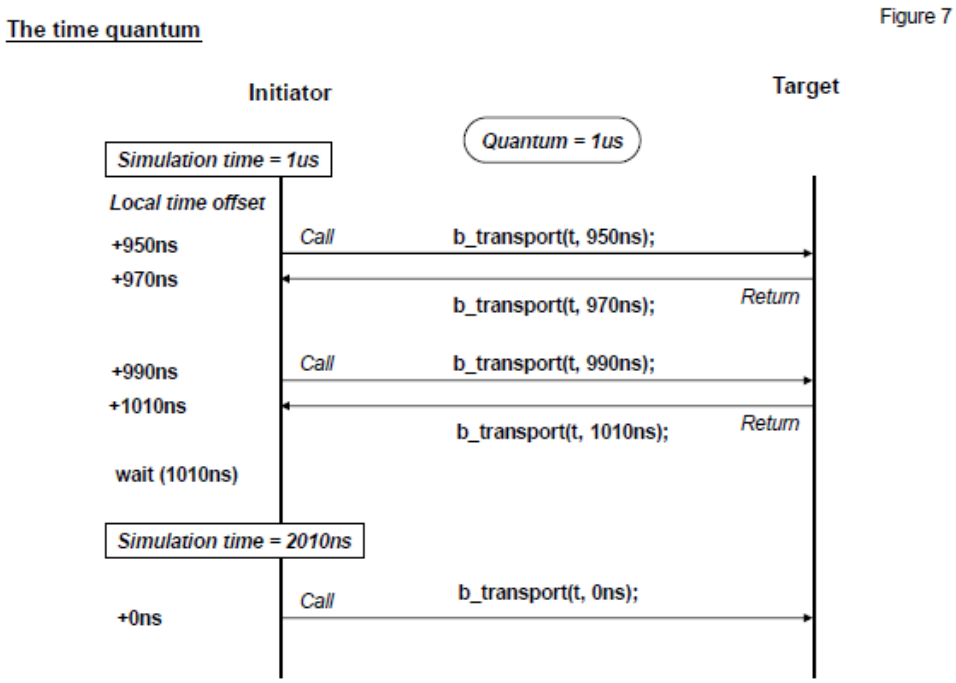


4.1.1.7. メッセージ・シーケンス図ータイム・クォンタム

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であろう。その時、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・タイムと一緒に wait メソッドを呼び出した時まで実行を停止し同期せざるをえないだろう。これはモデルの他のイニシエータが動作して、追いつくのを許す。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する、各々がいつそれ自身のローカル・タイムの経過を追うことによって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再び走るだけであるべきである。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



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、各々のフェーズがタイミング・ポイントに関連付けられた複数のフェーズにトランザクションを分解することである。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・トランザク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。オポジットパスで使用される 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コンスト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・トランザク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そ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する為の汎用ペイロードから分けて使用出来る。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・タイプとフェーズ・タイプを代用するかもしれない。異なったトランザクション・タイプ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「9.1.2 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・トランザクションには、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・トランザク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・トランザク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・トランザクションは、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・タイプ（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。「9.2.4 テンポラル・デカップリングの使い方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_quantum_keeper 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、トランザクション、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、トランザクション、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

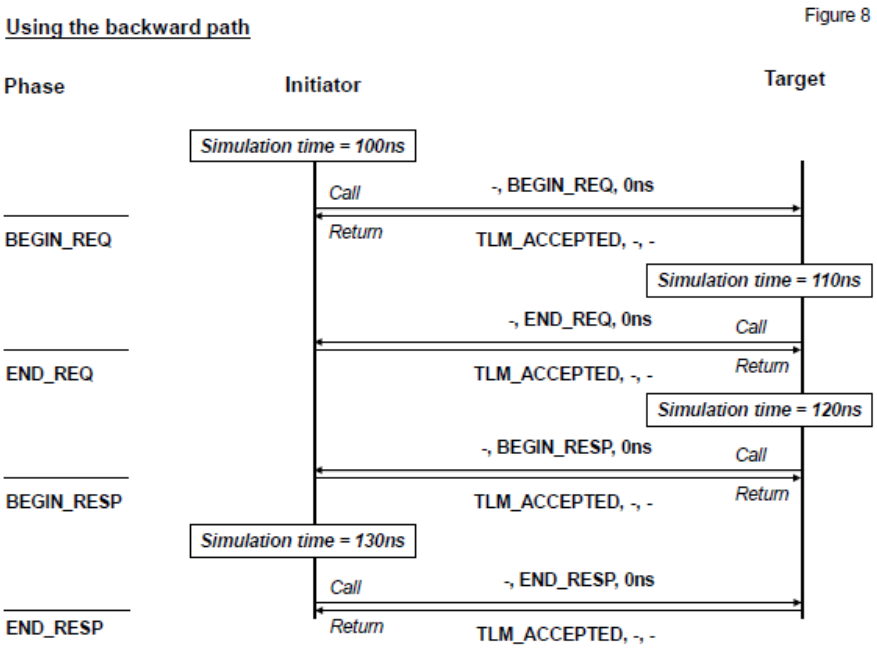
4.1.2.9. メッセージ・シーケンス・チャート – バックワード・パスを使用しているケース

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、“－”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・トランザク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。

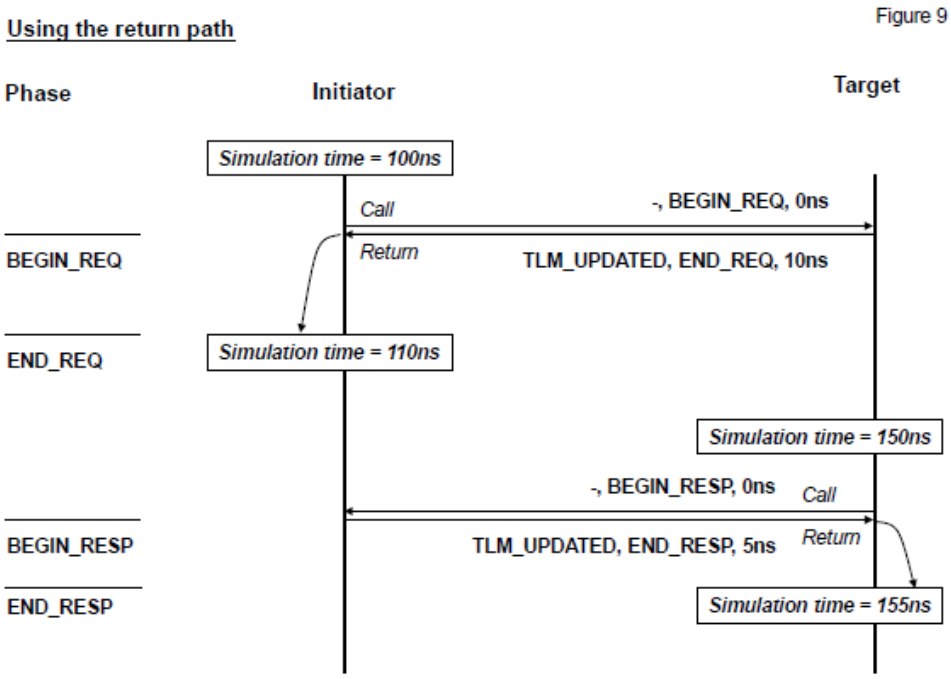
次のタイミング・ポイントがトランザクションの終わりを示すなら、受け取り側は TLM_UPDATED か TLM_COMPLETED のどちらかを返すことができる。

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。

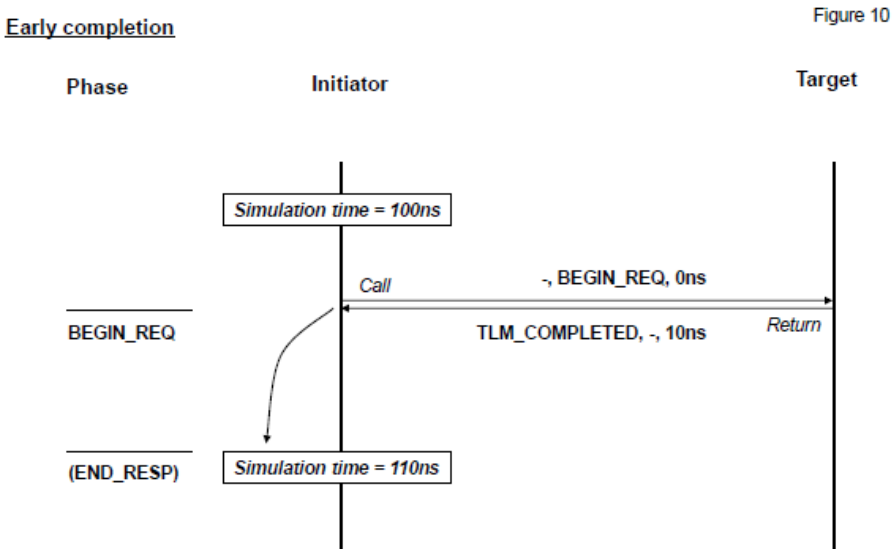
TLM_UPDATED と共に、呼び出し先関数はトランザクション、フェーズ、およびタイミング・アノテートを更新すべきである。

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・トランザク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

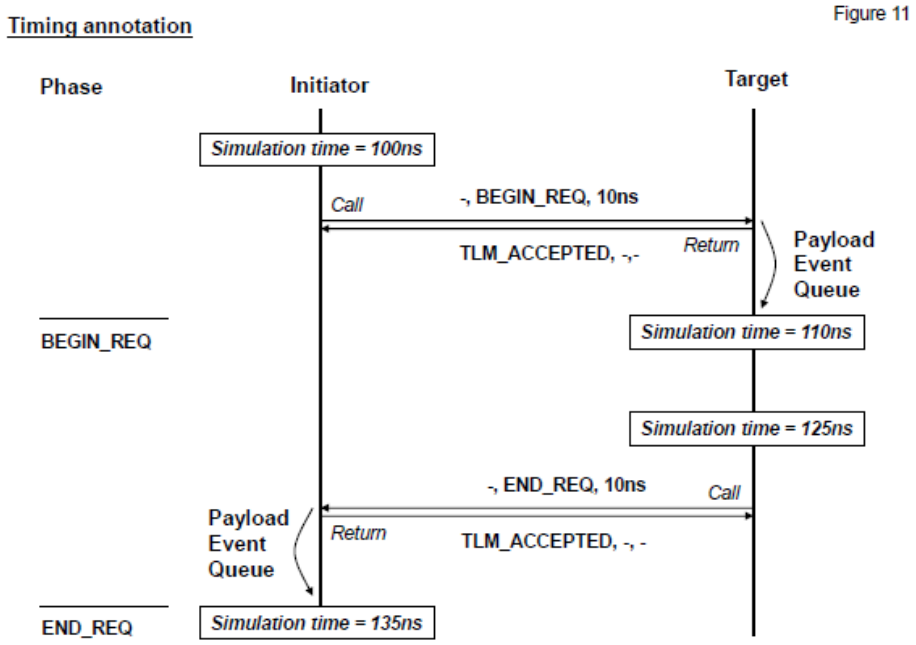
プロトコルに依存して、イニシエータやターゲットは早期にトランザクションを完了する為に nb_transport から TLM_COMPLETED を任意のタイミング・ポイントで返しても良い。イニシエータとターゲットのいずれも、このトランザクション・インスタンスの nb_transport 呼び出しを行わなくても良い。イニシエータやターゲットは、このようなトランザクションのショートカットを受け入れられるか否かは、その特定のプロトコルのルールに依存する。以下のダイアグラムのリターン・パスにおけるタイミング・アノテーションは、与えられた遅延後に最終タイミング・ポイントが起こる事をイニシエータに指示している。BEGIN_REQ から END_REQ 及び BEGIN_RESP を通し END_RESP までのフェーズ・トランザクションは、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「8.2.7 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（「9.3 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入って来るインタフェース・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、入って来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入って来る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、「9.2 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 記述子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ポインタが無効な場合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 の型宣言を含む新しいプロトコル・トレイツ・クラスの定義」を参照）

DMI 記述子は、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。

c) dmi_data 引数には、イニシエータで生成された DMI 記述子への参照値。

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・ソケットで get_direct_mem_ptr メソッドを呼び出すことかもしれない。

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に従う。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 は二つ目のトランザクション・オブジェクト、例えば重要な（ビット）幅変換のようなもの、を発生ようなコンポーネントを通るようなパス上では使うことができない。（もし DMI が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 引数を伝える必要がある。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引数値である。トランザクションと DMI 記述子のアドレス・アトリビュートは両方とも get_direct_mem_ptr メソッド終了後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の関数呼び出しを巻き出しているとき。

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 記述子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 記述子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 記述子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 記述子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。

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障する必要がある。

l) それぞれの get_direct_mem_ptr の呼び出しで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、メモリ状態を越えてターゲットの状態が変化する）が発生すると、ターゲットはメモリ領域への与えられたタイプの 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・アクセル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。

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によって相互利用性は制限される。

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属性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属性は、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すべきでない。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、コマンド属性の値に TLM_IGNORE_COMMAND は禁止されている。

h) トランザクション・オブジェクトのアドレス属性は、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・コンポーネントによって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属性をデコード及び、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属性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、他のすべての属性を無視してもよい。特に、応答ステータスの属性や許可されている DMI 属性は無視してもよい。許可されている DMI 属性は、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 呼び出しから次の呼び出し、さらには DMI やトランスポート・インタフェース、そしてデバッグ・トランスポート・インタフェースの呼び出しに渡ってトランザクション・オブジェクトは再利用されるかもしれない。

m) もしアプリケーションが、要求されている DMI アクセスの種類が決定したときターゲットによって利用される DMI トランザクション・オブジェクトにさらに属性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 記述子は tlm_dmi クラスのオブジェクトである。DMI descriptor 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 記述子は、次の属性を持つ：DMI ポインタ属性、許可されたアクセス型の属性、開始アドレス属性、終了アドレス属性、読み込みレイテンシ属性、そして書き込みレイテンシ属性。これら属性のデフォルト値：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、デフォルト値への DMI 拡張子のメンバを初期化しなければならない。

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 descriptor をフォワード・パスで修正してはいけないので、DMI descriptor 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ある。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 は偽の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 アクセスが許可された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を引数で渡された値で設定される。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、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、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あるほかのアクセスの種類を許可している（または拒否している）ことを示すために、DMI_ACCESS_NONE を許可されたアクセス型に設定すべきである。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しかできない。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 記述子におけいるアドレスのいかなる変換も、記述子が 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、同じアクセス型（例えば、読み込み及び読み込み／書き込みの両方や書き込み及び読み込み／書き込みの両方）で、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場合には、一つの領域が許可されもう一方が禁止される重なり合う DMI 領域は返さない。

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一部にとどめるべきである。さもなければ、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 記述子が 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に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 呼び出しをイニシエータからターゲットへのバックワード・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・パスと DMI のバックワード・パスでアドレスを変換するので、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インターコネクト･コンポーネントをバイパスして、直接アクセスすることが可能。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。

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・コンポーネントのように副作用のある場合でも、正しく動作するように注意が必要。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・コンポーネントトランスポート・インタフェースことが可能。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) イニシエータが、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は可能。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。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、特に注意が必要。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・インタフェース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 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、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。次の場合

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生成されるまで、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障される。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・メソッド呼び出しによって、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 領域を無効化することは可能。テンポラル・デカップリングされたイニシエータが、生成されずに実行される。

e) イニシエータが、他のコンポーネントに影響を与えない、生成しないで実行される間は、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生成された後で、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は可能である。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 ヒントつまり属性を許容された DMI 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属性を許容された DMI を確認することができる。

b) 汎用ペイロードによって属性を許容された DMI 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属性を許容された DMI の値を適切に設定すべきである。

c) 属性を許容された DMI 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属性を許容された DMI を自由に無視できる。

d) イニシエータが属性を許容された DMI 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属性を許容された DMI を確認する。

   iv. もし属性が 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。

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・インタフェースによって、遅延、待ち、イベント通知または通常のトランザクションに対する副作用を一切発生させないで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が可能である。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される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、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、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 の型宣言を含む新しいプロトコル・トレイツ・クラスの定義」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。

4.4. 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・アトリビュートはイニシエータによって要求されているデバッグ・アクセスの種類を示すために設定され、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 と等しいコマンド属性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属性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属性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する場合は、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、LT 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()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 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、「9.2 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（同期間隔）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・モデルは、おおよそこのクォンタム時間が来るごとに実際のシミュレーション時間との間で同期が図られる。ただし、ターゲットの要求によってはそれよりも頻繁に同期が実行される場合もある。

b) それぞれのイニシエータ・モデル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・モデルはグローバルなクォンタム時間を持つ。同期の回数が少なくてもすむ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、同期の回数が多く必要なモデルがもっともシミュレーション時間に大きく影響してしまうからである。

c) instance()メソッドは、唯一のグローバル・クォンタム・オブジェクトのリファレンスを返す。

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()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。グローバル・クォンタムの次の倍数から、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. 結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 I/F 、バックワード・トランスポート I/F 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。
- コア・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 I/F 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 I/F とデバッグ・トランスポート I/F はバックワード・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することは、技術的には可能であるが、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。
- 結合された I/F のテンプレートは、プロトコル・タイプ・クラスでパラメタライズされる。プロトコル・トレイツ・クラスは、フォワード I/F、バックワード I/F で使われるタイプ、すなわちペイロード・タイプとフェーズ・タイプを定義する。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・タイプは 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・ソケットとターゲット・ソケット

6.2.1. イントロダクション

- ソケット：port と export を組み合わせたもの
  - ソケットの種類
    - イニシエータ・ソケット：フォワード・パス用 port とバックワード・パス用 export
    - ターゲット・ソケット：フォワード・パス用 export とバックワード・パス用 port
  - ソケットは、port と export を接続先の port と export とをバインドする機能を持つ。
  - 階層をまたいでバインドを行う場合は、順序に注意が必要。
  - アプリケーションが通常使用するソケット
    - tlm_initiator_socket
    - tlm_target_socket
  - これらのソケットは、プロトコル・トレイツ・クラスでパラメタライズされる。
  - ソケットは、プロトコル・タイプが同じである場合にのみバインド可能となる。デフォルトのプロトコル・タイプは tlm_base_protocol_types である。
  - 新規にプロトコル・タイプを定義する場合は、汎用ペイロードを使用するかどうかに関わらず、新しいプロトコル・トレイツ・クラスを持つ結合された I/F テンプレートをインスタンス化するべきである。
- ソケットの利点
  - フォワード・パス、バックワード・パスに対するトランスポート、ダイレクト・メモリ I/F、デバッグ・トランスポート I/F を 1 つのオブジェクトにグループ化する。
  - フォワード・パスとバックワード・パスそれぞれの port と export を、 1 回の呼び出しでバインドするメソッドを提供。

- 互換性のないプロトコル・タイプでパラメタライズされたソケットに対する、強力なタイプ・チェック機能を提供。
- トランザクションを中断させるために利用可能なバス幅パラメタを持つ。

- 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便利ソケットとして知られる派生ソケットが定義されている。

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、複数の純粋仮想関数を備えている。これらはソケットと関連付けられた port、export、インタフェースオブジェクトを返すためのもので、派生クラスで上書きされる必要がある。

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 クラスでは、コンストラクタで指定されたキャラクタ文字列を、ベースクラスである sc_port のコンストラクタに対して引き渡し、インスタンス名として設定する。

また、同じ文字列に"_export"を付加した名前を、backward パスの sc_export に対してインスタンス名として設定する。これらは sc_gen_unique_name の呼び出しにより設定される。

e.g.) tlm_initiator_socket("foo")と指定
→ sc_port　　: foo
→ sc_export : foo_export
e.g.) デフォルト
→ sc_port　:　sc_gen_unique_name("tlm_base_initiator_socket")
→ sc_export: sc_gen_unique_name("tlm_base_initiator_socket_export")

b) tlm_target_socket クラスでは、コンストラクタで指定されたキャラクタ文字列を、ベースクラスである sc_export のコンストラクタに対して引き渡し、インスタンス名として設定する。また、同じ文字列に"_port"を付加した名前を、sc_port に対してインスタンス名として設定する。これらは sc_gen_unique_name により設定される。

e.g.) tlm_target_socket("foo")と指定
→ sc_export : foo
→ sc_port　　: foo_port
e.g.) デフォルト
→ sc_port　:　sc_gen_unique_name("tlm_base_target_socket")
→ sc_export: sc_gen_unique_name("tlm_base_target_socket_port")

c) kind メソッドは、クラス名を C ストリングとして返す。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、テンプレート引数 BUSWIDTH 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。本アトリビュートの厳密な解釈はトランザクション・タイプに依存する。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 値を持たなければならない。

g) bind、operator（）メソッドがソケットを引数とする場合、メソッドが属するソケット・インスタンスが、引数で渡されたソケット・インスタンスの当該メソッドにバインドされる。

h) bind、operator（）メソッドがインタフェースを引数とする場合、メソッドが属するソケット・インスタンスの export が、引数で渡されたチャネル･インスタンスにバインドされる（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・ソケットはターゲット・ソケットに対して、どちらかのソケットの bind メソッド、または operator()メソッドにより、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、それぞれのソケットの port と port をバインドする。また、それぞれのソケットの 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、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一致している。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相互にバインドする場合、フォワード・インタフェース・タイプ、バックワード・インタフェース・タイプ、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・ソケットとして振舞う。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。インデックスは、bind や operator()メソッドが呼ばれた順序に依存する。

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()メソッドがよびだされた順序に依存する。しかしながら、受け取り側のソケットは、呼び出し側を特定するメカニズムを持たないため、受け取り側のインタフェース・メソッド呼び出しは、anonymous となる。このメカニズムについては、便利ソケットの側で提供される。「**エラー! 参照元が見つかりません。エラー! 参照元が見つかりません。**」を参照。

s) 例えば、一つのソケットが、二つのターゲットにバインドされる場合を考える。二つの呼び出し、socket[0]->nb_transport_fw(...)、socket[1]->nb_transport_fw()は、ターゲットを区別することはできるが、これら二つのターゲットからの nb_transport_bw()は、呼び出し側は区別する

ことができない。

t) 仮想メソッド get_base_port と get_base_export は、ソケットの port と export のオブジェクトを返す。仮想メソッド get_base_interface 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ように実装するべきである。

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。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）は、通常アプリケーションが利用する。

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び出し、char*引数を渡す。

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・タイプ・クラス（デフォルトは tlm_base_protocol_types）、同じバス幅を持つ必要がある。新規プロトコル・タイプに対しても、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タイプ・チェックが可能である。これは汎用ペイロードをベースにし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 ストリングとして返す。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いる。2 つのレベルで向上する。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。一方、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を提供する。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。実装コスト削減で有利であり、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は自明なことであるが、シミュレーション速度を向上させる点で有利である。

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、バスバースかどうかに関わらず、については、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。プロトコル・オーナー、モデルの増加の問題、特別なプロトコルのガイドライン次第である。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橋渡し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い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。そのときは、一部のアトリビュートしか使えないが。

7.2. 拡張性と相互利用性

3 つの推奨する拡張手法がある。これらは blocking、non-blocking transport I/F のトランザクションテンプレート引数 TRANS と複数の I/F が組合されて定義された TYPES を置き換えることで実現するものである。

実現方法	TRANS トランザクション型	TYPES プロトコル型
a	tlm_generic_payload	tlm_generic_payload_types
b	tlm_generic_payload	ユーザ定義型
c	ユーザ定義型	ユーザ定義型

a→b→c の順に相互利用性が損なわれる。これら 3 つの方法で実現したモデルは、一つのシステムで混在されることも考えられている。

7.2.1. 無視可能な拡張を含む、汎用ペイロードの直接利用

- トランザクション型は tlm_generic_payload を、フェーズ型は tlm_phase を、プロトコル型は tlm_generic_payload_types を指定する。基本プロトコルを持つ標準イニシエータとターゲット・ソケットを利用することによってモデルの相互利用可能になる。
- 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ないものとして動作すること許容すること
- 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に対応する。一般的な原則として、ターゲットは汎用ペイロードのすべての機能をサポートすることを推奨。

7.2.2. tlm_generic_payload の型宣言を含む新しいプロトコル・トレイツ・クラスの定義

- 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、ソケットを特徴づけるプロトコル・トレイツ･クラスは、tlm_base_protocol_types を用いず、独自で定義した型を利用する
- 新しいプロトコル型は独自のルーツを持つが、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と引数の修正可能性を含む基本プロトコルのルールを拡張または矛盾していてもよい。
- 汎用ペイロード拡張メカニズムは制限なく、無視可能または無視不可能、必須拡張に利用される。すべての拡張のセマンティクスは徹底的に新しいプロトコル・トレイツ・クラスを用いてドキュメント化すべき。
- ユーザは拡張された汎用ペイロードのセマンティクスは十分注意する必要がある。
- プロトコル変換を行うブリッジを使えば、異なるプロトコルのトランザクションを使うことが可能
- 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拡張する場合は、ユーザは各拡張されたセマンティクスを十分考慮することが必要不可欠である。

7.2.3. 新しいプロトコル・トレイツ・クラス型と新しいトランザクション型の定義

- 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。
- 新しいプロトコル・トレイツ・クラスは接続されたインタフェースとソケットをパラメタライズするために定義する必要がある
- この手法は実現するプロトコルと汎用ペイロードが著しくことなる場合に利用する
- 相互利用性を考えるとこの手法よりは、前2つの節の方法を推奨する

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アクセスするパブリックなメソッドを持つ。各アトリビュートの変更は原則イニシエータが行うが、アドレスやDMI許可情報、レスポンス・ステータスなど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・コマンドにおけるデータ配列はターゲットが更新する。

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・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、オートマティック（stack）、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) 汎用ペイロードはメモリ管理への 2 つの異なったアプローチをサポートする。1 つは明白なメモリ・マネージャ、もう一つはイニシエータによる臨時のメモリ管理。混在は可能である。いずれのアプローチ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 CPU 時間のコストが大きい。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所有するメモリ・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よけ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、純粋仮想関数である 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、自動削除対象の全拡張を削除するために、tlm_generic_payload クラスの reset メソッドからもコールされる。

f) メソッド set_mm、acquire、release、get_ref_count、reset はメモリ管理に存在する。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臨時のメモリ管理の場合、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、いずれのアプローチも使用される。イニシエータによる臨時のメモリ管理は十分である。メモリ・マネージャが不在のとき b_transport、get_direct_mem_ptr、または transport_dbg メソッドが、トランザクション・オブジェクトとその拡張は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・マネージャは使用されるものとする。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ず適用される。

j) なにも既に存在していないなら、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・マネージャを設定しなければならない。

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、ヒープ（new か malloc で）からトランザクション・オブジェクトとその拡張オブジェクトも割り当てるものとする。

l) 臨時の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・マネージャを設定するものとする。引数として、アドレスが通過される。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・マネージャと参照が 0 以上を数えるトランザクション・オブジェクトのために set_mm を呼ばないものとする。

n) メモリ・マネージャが用意できていた場合にだけ、メソッド has_mm が true を返すものとする。nb_transport メソッドのボディーから呼ばれると、has_mm は true を返すはずである。

o) b_transport、get_direct_mem_ptr、または transport_dbg メソッドのボディーから呼ばれると、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 と呼んで、トランザクションにはメモリ・マネージャがいるかどうかによる適切な行動を取る。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）がいる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（例えば、acquire）の存在を要求するメソッドのいずれも呼ばないだろう。

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するものとする。メモリ・マネージャが不在のとき、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ものとする。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・マネージャが不在のとき、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、返された値は 0 であるだろう。

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、オブジェクトはもう必要でないときに、そのトランザクション・オブジェクトの release メソッドを呼ぶべきである。

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、オブジェクトはもう必要でないときに release メソッドに呼ぶ。

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・メソッド呼び出しやプロセスの中で 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ない限り、インタフェース・メソッド呼び出し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することはできない。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広げるのならば、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広げるのに参照カウントを使用するべきでない。

x) メモリ・マネージャで、参照カウントが、イニシエータ自体以外のどんなコンポーネントにもトランザクション・オブジェクトの参照がまだないのを示すまで、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使用されないものとするか、異なったインタフェースと共に再使用しないものとする。参照カウントが 1 と等しくなるまで、イニシエータは、トランザクション・オブジェクトのために acquire を呼ぶことを仮定する。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 は自動削除のためにマークされた拡張を削除して、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、削除されるものとする。ユーザが拡張オブジェクトのための明白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。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て、且つ拡張オブジェクトがプールになる場合、この後者の振舞いが必要である。

aa) メモリ・マネージャが不在の場合、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るあるいは与えられた拡張にセットするのいずれの場合でも、その同じ拡張を削除またはクリアする。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。（下流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、正式ではないが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・マネージャの存在の有無が分からない場合、拡張は set_extension コールで追加、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ることができる。（イニシエータの中では、常に、メモリ・マネージャの存在の有無は分かっている、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他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白なメモリ管理を提供したいなら、多分 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され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ものとする。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 non-null ならば、データとバイト・イネーブル配列は deep 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に大きいことを確認することを保証する。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。クローニングの場合で、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ものとする。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属性が 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 non-null かつ不一致ならば、データ配列は deep コピーされる。もしバイト・イネーブルのポインタが non-null で 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 deep 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 の一般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 deep 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属性と拡張情報をオリジナル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 deep コピーを行うか、あるいは滅多にないが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・タイプにも適用される。

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不可である。

d) 仮想のデストラクタ~tlm_generic_payload は、すべての拡張を削除するものとする（自動削除とマークされたもの等を含む）。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、これらの修正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・アトリビュートは下流の最終端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修正してはならない。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、何度でも修正することができる。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・タイプ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、標準的なエラーレスポンスを生成する。推奨するレスポンス・ステータスは TLM_COMMAND_ERROR_RESPONSE。

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拡張するために使うことができる。しかし、拡張を意図するルールは、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は書き換えない。

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。

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’

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、それぞれデータのデータ配列からの読み出し/書き込みを行う。

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アレイ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 Null（0）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 が返ったときはいつも、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）

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た、コピーされるデータ・アレイ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) 相互利用性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と同じか小さいとシングルワード転送を、それより大きいとバースト転送をトランザクション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’

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’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

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、バスの選ばれたバイト･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'

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。これらの無視されるバイトに対しては、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、データアレイ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が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が変更し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･タイプ･クラスを作ることを推奨する。「7.2.2 tlm_generic_payload の型宣言を含む新しいプロトコル・トレイツ・クラスの定義」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 ポインタ

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、拡張に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 null pointer 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、エラーを出す。推奨は、TLM_BYTE_ENABLE_ERROR_RESPONSE

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'

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、データアレイ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、オリジナルトランザクションと同じアドレスで、オリジナル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オリジナル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、オリジナル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、標準エラーを出す。推奨は、TLM_BURST_ERROR_RESPONSE

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、その場合通常は、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

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が実行できた場合は、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

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 response status attribute 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、ターゲットは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・コンポーネントにも上書きされない。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

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、ターゲットが無視できる拡張に関して従うルールが与える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される。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 のいずれも設定することは正しい。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) 汎用ペイロードのアトリビュート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・タイプに対しても、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・バスにおける慣例適用法に対応した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ステートを変更せず、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が、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. 受動的シミュレーション・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 Word 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 Word 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 Word 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 Byte 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 Byte 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 Byte 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 Helper 関数」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一打で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見よ）

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固まりは、アドレスが並んで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並ばなければならない。しかしながら、アドレス属性もデータ長属性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。ストリーミング幅属性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 LSBN を使うと、リトルエンディアンのホスト上ではバイトは{…, MSB0, LSB1, …, MSB1, LSB2, …}の順で蓄えられ、ビッグエンディアンのホスト上では{…, LSB0, MSB1, …, LSB1, MSB2, …}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。その規則は、それらの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 コードの一部で考えてみよう。そこでは、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) データ長属性とアドレス属性のどちらも 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で 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属性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。

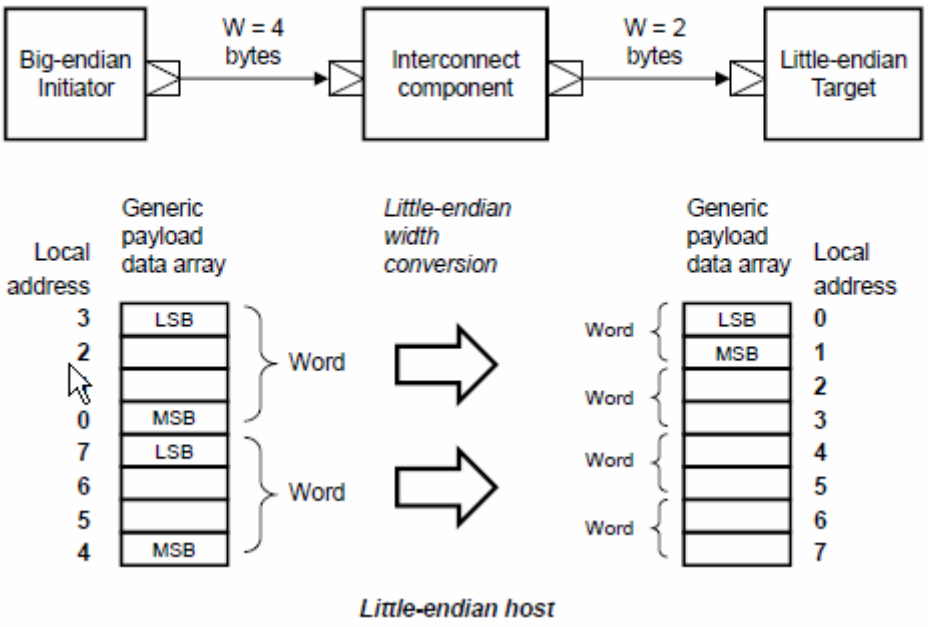
t) パートワードのアクセスのためにバイト・イネーブルを使うことの必要性はエンディアンに依存する。たとえば、最初のワードのすべてと第 2 のワードの LSB をアクセスする意図は、リトルエンディアンのホストでは次のように表現する。

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。イニシエータとた一別途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、出て行く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、ストリーミング幅属性がゼロでなく 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

Helper 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、そのときだけ 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、そのときだけ true の値を返す。

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 Helper 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 Word 幅を含む場合では極めて複雑になる。Helper 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。Helper 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で、ホストエンディアンにマッチするように変換されたとトランザクションを持つことである。各変換関数は、存在する汎用ペイロード・トランザクションを行いそして修正する。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。コスト面で_generic 関数の使用が推奨される。

変換関数には Arithmetic モードと Byte order モードがある。性能面で、Arithmetic モードが推奨される。変換関数は data word コンセプトが使われており、これは TLM2 ソケット幅と汎用ペイロード・データ配列とは独立である。

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 valid 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・メソッド呼び出し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、… 等を意味します。

- 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。
- ストリーミング幅属性はデータ・ワード幅の整数倍でなければならない。
- データ長属性はストリーミング幅属性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。
- ストリーミング幅属性はデータ幅属性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・イネーブル長属性はデータ長属性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。

- アドレス属性は、ソケット幅の整数倍でなければならない。
- データ長属性は、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。

- データ長属性はデータ・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したり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、新しいプロトコル特性（traits）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、adaption や bridging を使わず直接相互に接続することを許すような、同じプロトコル特性クラスで特化された類の汎用ペイロードのコア属性セットを運ぶ 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 adaption 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属性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、もし要求されたら入ってくるトランザクション・オブジェクトから出て行くトランザクション・オブジェクトへすべての拡張をコピーし出て行く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・オブジェクトの深いコピー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出て行く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・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。コンポーネントが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。「9.4 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) メモリ管理が存在するときの与えられた拡張のための 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。メモリ管理のないときに 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管理の 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管理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管理を持つかどうかに依存することに注意せよ。メモリ管理がある場合は、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。メモリ管理がない場合は、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。もしそうすると、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・フェーズ・タイプである。
- tlm_phase オブジェクトは unsigned int 値でフェーズを表す。
- unsigned int タイプ tlm_phase クラスは、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 の列挙体に割り当てる。
- DECLARE_EXTENDED_PHASE マクロを使用して tlm_phase_enum として提供された 4 つのフェーズのセットは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 メソッドを持つ tlm_phase から拡張された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では相互利用性を最大限に生かすには tlm_phase_enum の 4 つのフェーズのみを使用すべきである。もし、 更なるフェーズが特定のプロトコルの詳細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 タイプとの互換性を有す。
- 無視可能 vs.無視不可能または必須な拡張の本質は汎用ペイロード拡張と同様にフェーズにも適用される。 つまり、無視可能フェーズは基本プロトコルに許容される。
- 無視可能フェーズは、受信者と送信者の両方から無視可能にする必要がある。受信者の場合、受信者がフェーズ遷移を受け取っていないかのように単に動作できるという意味での無視可能であって、送信者では受信者からどんな応答がなくても送信者が実行を続けることができるという意味で無視可能でなければならない。
- この意味でフェーズを無視できないなら、新しいプロトコルの特性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になる。

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(arg)マクロは tlm_phase から拡張された tlm_phase_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 は public メソッドの get_phase を持っていて静的オブジェクトのリファレンスを返す。マクロの引数は operator<<によって書かれた文字列として、指示された対応するフェーズが使用される。

e) その目的は、 static const name_arg によって指示されたオブジェクトが、フェーズ引数として nb_transport に渡されるかもしれない拡張フェーズを表すためである。

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相互利用性を確実にする為のルールセットを持っている。

基本プロトコルはここに記載されたTLM-2.0相互利用性レイヤのクラスの使用とこの項で定義された規則を必要とする。:

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。

b) 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）。「6.2 イニシエータ・ソケットとターゲット・ソケット」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

もし、無視可能なフェーズ拡張であれば、基本プロトコル規則は汎用ペイロードと、フェーズの拡張を可能にする。 無視可能でない拡張は新しいプロトコルの特性クラスの定義を必要とする。「7.2.1 無視可能な拡張を含む、汎用ペイロードの直接利用」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 種類の型だけが定義される。 このクラス（ソケットへのテンプレート引数としての）を使用するすべてのコンポーネントは、基本プロトコルの規則を尊重する規約に従う必要がある。

新しいプロトコルの特性クラスを定義することが必要な場合に備えて、(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ど)、新しいプロトコルの特性クラスに関連している規則は基本プロトコルをオーバーライドすること。しかしながら、一貫性と相互利用性において、新しいプロトコル特性のクラスに関連する規則とコーディング・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 の型宣言を含む新しいプロトコル・トレイツ・クラスの定義」を参照。

TLM 規格のこのセクションは、明確に基本プロトコルに関連があるが、それにもかかわらず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されるかもしれない。他のプロトコルの特性クラスを示す特定のプロトコルは、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追加フェーズを含んだり、独自の規則を採用するかもしれない。その場合、それらは、基本プロトコルとの互換性がなくなるかもしれない。

8.2.2. 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
```

8.2.3. 基本プロトコルフェーズシーケンス

a) 基本プロトコルはブロッキング・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、または両方同時使用を可能にする。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。基本プロトコルと共に使用される場合、nb_transport を呼ぶ順番を管理する制約は b_transport を呼び出す順番を管理するものより強い。したがって、 nb_transport は AT 記述に最適で、b_transport は LT 記述に最適である。

b) 与えられたトランザクションの為のフェーズ遷移の完全なシーケンスは以下の通りである。

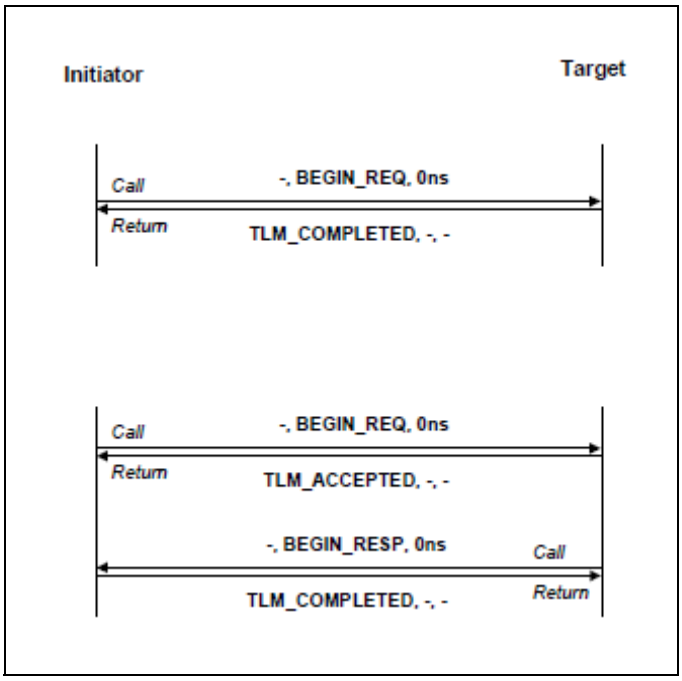
BEGIN_REQ → END_REQ → BEGIN_RESP → END_RESP

c) BEGIN_REQ と END_RESP はイニシエータ・ソケットのみ送信可能である。 END_REQ と BEGIN_RESP はターゲット・ソケットのみ送信可能である。

d) ブロッキング・トランスポート・インタフェースの場合、トランザクション・インスタンスは b_transport への一回の呼び出しと応答に関連づけられる。BEGIN_REQ の b_transport への呼び出しと、BEGIN_RESP の b_transport からの応答との対応は純粋に概念的である；つまり、b_transport には、どんな関連フェーズもない。

e) 基本プロトコルでは、TLM_UPDATED の値を持つ nb_transport への各コールと nb_transport からの各リターンは必ずフェーズ遷移しなければならない。 つまり、同じトランザクションのための nb_transport への 2 つの連続した呼び出しは、フェーズ引数に違う値を持たなければならない。 無視可能なフェーズ拡張は受入れられるが、その場合、拡張フェーズの挿入は、この規則の目的のために（フェーズが無視されても）フェーズ遷移とみなすものとする。

f) nb_transport に TLM_COMPLETED の値を返させることによって、フェーズシーケンスを短縮することができる。 しかし、その場合以下の方法の一つだけが使用可能である。フォワード・パスで BEGIN_REQ か END_RESP を受け取るとき、インターコネクト・コンポーネントまたはターゲットは TLM_COMPLETED を返すことができる。バックワード・パスで BEGIN_RESP を受けるとき、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 TLM_COMPLETED を返すことができる。TLM_COMPLETED のリターン値は特定のホップに関してトランザクションの終了を示す、その場合、フェーズ引数は呼び出し元関数（caller）によって無視されるべきである（「4.1.2.7 tlm_sync_enum の戻り値」を参照）。TLM_COMPLETED は成功した完了を示すわけではないので、イニシエータは成否の確認のためにトランザクションの応答ステータスをチェックするべきである。



Examples of early completion

g) また、フェーズ END_RESP への遷移は特定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（ callee）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 を返すとき、これは暗黙の END_REQ と暗黙の BEGIN_RESP を返すことを意味する。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チェックするべきであり、すぐに、次のトランザクションのために BEGIN_REQ を送ることができる。

i) BEGIN_REQ を受けとった後に返された TLM_COMPLETED は、それと一緒に暗黙の BEGIN_RESP を返すので、与えられたソケットを通して進行中の応答が既にあれば、この状況は応答除外規則によって禁じられている。この状況で、呼び出し先関数（callee）は、TLM_COMPLETED の代わりに TLM_ACCEPTED を返すべきであり、次のイニシエータ方向への応答を送る前に END_RESP を待つべきである。

j) BEGIN_REQ を受けとった後に返された TLM_COMPLETED が、トランザクションの終わりを示すとき、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その同じソケットを通して END_RESP を送る事は禁じられる。

k) BEGIN _RESP を受けとった後にコンポーネントで TLM_COMPLETED をターゲット側方向に返すとき、これはそれと一緒に暗黙の END_RESP を返す。

l) もし、コンポーネントが最初に END_REQ を受信せずにターゲット側方向のコンポーネントから BEGIN_RESP を受けとるなら、イニシエータはすぐに BEGIN_RESP に先行して暗黙の 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ための唯一のケースである ; BEGIN_RESP はいかなる他のトランザクションのためにも END_REQ を暗示しない、そして BEGIN_REQ を受けとるターゲットは、前のトランザクションのために END_RESP を推論できない。

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) どちらのパスで TLM_COMPLETED を返す時、またはフォワード・パスかバックワード・パスで END_RESP を渡す時、基本プロトコルのトランザクションは、(特定のホップに関して)終了する。

o) END_RESP がフォワード・パスで送られるケースでは、呼び出し先関数（callee）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ケースでも終了する。

p) 与えられたトランザクションは異なる時間に異なるところでホップを終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・マネージャを持つことを強いられ、汎用ペイロードのリファレンス・カウントがゼロに達するとき、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 メソッドを呼ぶどんなコンポーネント・オブジェクトもトランザクションの終了時、または、終了前に release メソッドを呼ぶべきである。「7.5 汎用ペイロード・メモリ管理」を参照すること。

q) コンポーネントがイリーガルか out-of-order フェーズ遷移を受け取った場合、これは送信側の誤りである。 受取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。

8.2.4. 許可されたフェーズの遷移

前項の規則の全ての説明をまとめて、基本プロトコルのために与えられたホップ上の許可されたフェーズ・トランザクションの集合は以下のテーブルのようになる。

Previous state	Calling path	Phase argument on call	Phase argument on return	Status on return	Response valid	End-of-life	Next state
//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 へ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テーブル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) 「previous state」列は、nb_transport を呼ぶ前の与えられたホップの状態を示す。

d) 「calling path」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）に呼ばれるかどうかを示す。

e) 「phase argument on call」列は、nb_transport を呼び出すフェーズ引数の値に与える。これは呼び出し先関数（callee）に渡されるフェーズになる。

f) 「phase argument on return」列は nb_transport からのリターン（return）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) 「status on return」列は nb_transport メソッド、Accepted（TLM_ACCEPTED）、Updated（TLM_UPDATED）、または Completed（TLM_COMPLETED）のリターン値を与える。

h) 「response valid」列は、トランザクションの応答ステータス・アトリビュートが nb_transport メソッドからのリターン時に有効であれば、チェックされる。

i) トランザクションがこのホップに関して生存期間の終わりに達したなら、すなわち、更なる nb_transport 呼び出しが与えられたホップ上の与えられたトランザクションのためにそれ以上許可されないなら「end-of-life」列はチェックされる。

j) 「next state」列は nb_transport からのリターンの後に与えられたホップの状態を示す。

k) フェーズ・トランザクションは、呼び出し元関数（caller）（phase argument on return 列の「-」で、示される）か、呼び出し先関数（callee）によって引き起こされる。

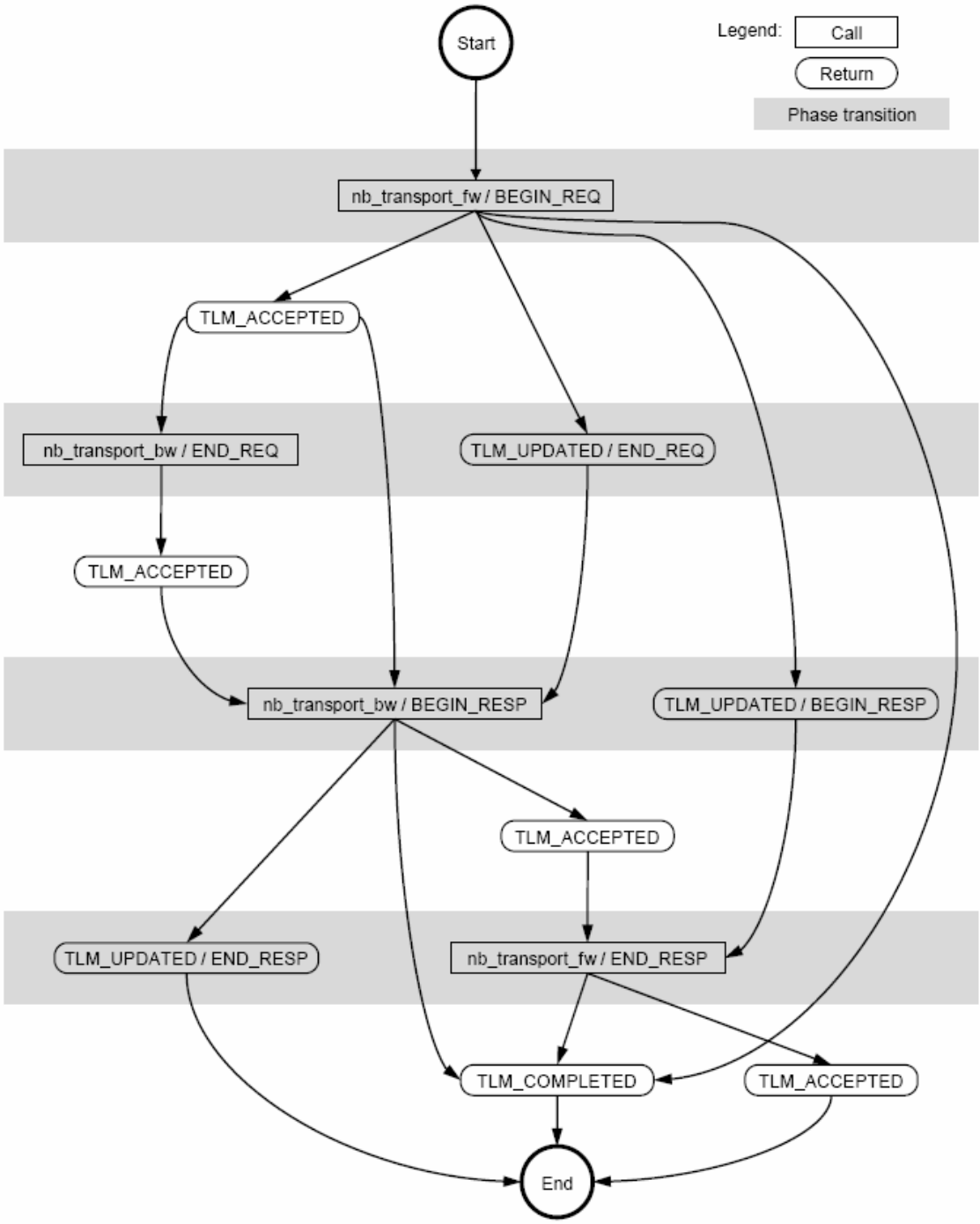
l) 無視可能なフェーズ拡張は BEGIN_REQ と END_RESP の間の任意な点で挿入することができる。

m) 有効な応答は、成功した終了を示さない。トランザクションは成功したかも知れないし、成功していないかも知れない。

n) 次のページのグラフィカルな形式の図 14 は、与えられたホップの上での基本プロトコルで許可された nb_transport 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の為の許可された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共に nb_transport の呼び出しを表す。丸い長方形は戻り値とリターン時の適切なフェーズ引数の値を示す。大きなグレーの長方形の帯はフェーズの変わり目を示す。

nb_transport Call Sequence for each Base Protocol Transaction

Figure 14



8.2.5. 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共に使用することができる。無視可能フェーズは受信側に無視されるかもしれない。

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お勧めの方法は、新しいプロトコルの特性クラスを定義することである。「7.2.2 tlm_generic_payload の型宣言を含む新しいプロトコル・トレイツ・クラスの定義」を参照。tlm_generic_payload のために typedef を含む新しいプロトコルの特性クラスを定義する。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ケースである。無視可能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ように余分なタイミング・ポイントを基本プロトコルに追加することを許可することである。たとえば、無視可能フェーズはイニシエータからターゲットへのデータ転送の始まりの時間をマークするかもしれない。

b) nb_transport への呼び出しに関するケースでは、もしそれが呼び出し先関数でフェーズを無視するのであるなら TLM_ACCEPTED の値を返すものとする。呼び出し先関数が TLM_UPDATED を返すケースでは、呼び出し元関数は、リターン・パスで通過するフェーズを無視するかもしれないが、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ことを強いられない。

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ケースと、呼び出し先関数が後で反対のパスを通して応答するケースとを見分け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ケースでも TLM_ACCEPTED を返すものとする。

d) 無視可能フェーズの存在は、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、いかなる基本プロトコルが壊れるルールをもたらすことも許されない。

e) 無視可能フェーズは、与えられたソケット上の与えられたトランザクションについて、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 の後の無視可能フェーズの存在は、基本プロトコル違反で誤りである。

f) 無視可能フェーズの存在は、汎用ペイロード・アトリビュートの正当性に関する規則、あるいはそれらのアトリビュートを変更するための規則を変えないものとする。例えば、無視可能フェーズを受け取り次第、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」を参照のこと。

g) 以下で定義されるように透過的なコンポーネントにおける例外のように、無視可能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把握する場合だけ、コンポーネントは引数のフェーズが nb_transport 呼び出しに渡すことが許可される。

h) 無視可能フェーズの受取側が、基本プロトコルに違反していないその無視可能なフェーズを認識するなら、そのコンポーネントの動きは、基本プロトコルのスコープ外で、基本プロト

コルでは未定義である。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うべきである。

i) 定義上、無視可能フェーズを送るコンポーネントは、TLM_ACCEPTED の値を返す nb_transport の最小量の応答以外に、そのフェーズが送られるコンポーネントからどんな種類の応答や要求もできない。応答を要求するフェーズは無視可能ではない。定義上、その場合、お勧めの手法は、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の特性クラスを定義することである。これは非互換プロトコルのソケットのバインドを防ぐ。

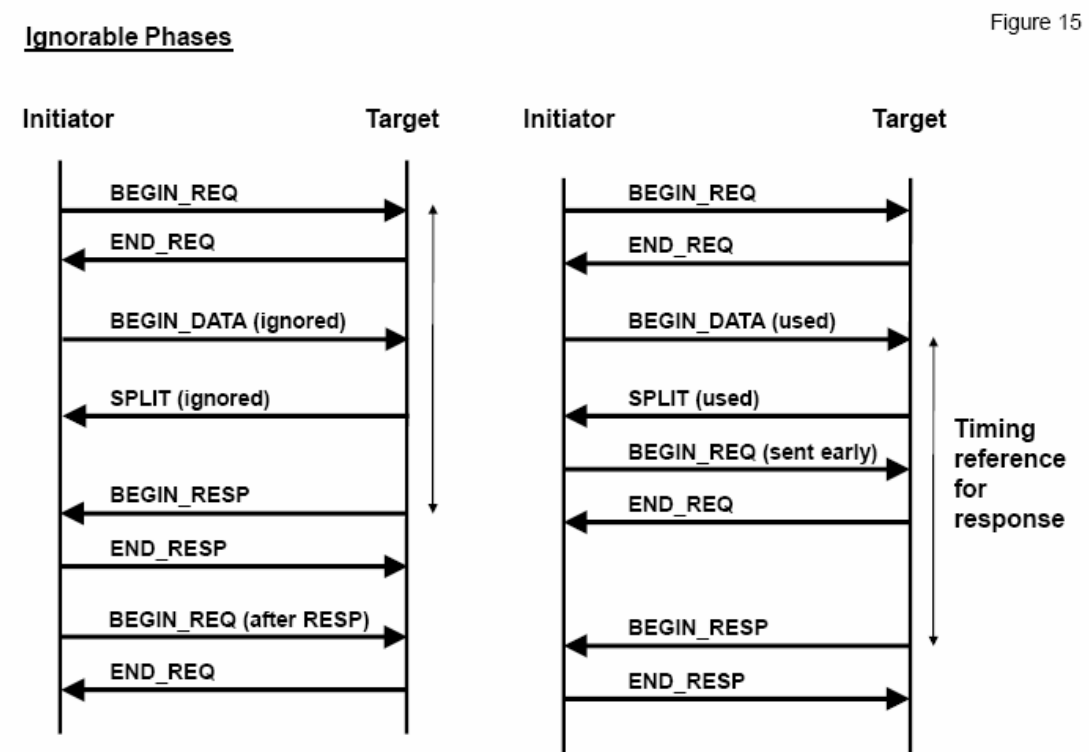
j) 他方では、到着する拡張フェーズを認識する基本プロトコルに準拠したコンポーネントは、反対のパスを通してあらかじめ合意された何らかの拡張プロトコルの規則に従った別の拡張フェーズを送ることによって返答することができる。この可能性は、基本プロトコルの規則が破られていない限り、TLM-2.0 規格によって受入れられる。例えば、そのような拡張プロトコルは汎用ペイロード拡張を利用することができる。

k) いわゆる透過的なインターコネクト・コンポーネント（すぐに、直接に、同じコンポーネントの中に含まれるターゲット・ソケットとイニシエータ・ソケット間のどんな TLM-2.0 のインタフェース・メソッドも通過させる）を作成することは可能である。この規格で透過的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 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、および変更なしですべてのトランザクションで両方の方向を通過する機能を通常持つ。）

l) 透過的なコンポーネントの中では、どんな TLM-2.0 コア・インタフェース・メソッドの実装も、少しの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ず、同一のインタフェース・メソッド・コールを、すべての引数をそのまま通過させながら、反対側のソケットを通して（イニシエータ・ソケットはターゲット・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）しなければならない。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の 1 つの例外として、どんな引数（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。そのような透過的なコンポーネントを通したルーティングは、固定されていて、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) 上記の規則の結果として、透過的なコンポーネントはどちらの方向へのどんな拡張フェーズや無視可能フェーズも通過させる。

例



イニシエータから発生された無視可能フェーズの例は、イニシエータからの書き込みコマンドに関するケースで、データ転送の最初のビートをマークするフェーズである。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、およびデータの入手のために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。

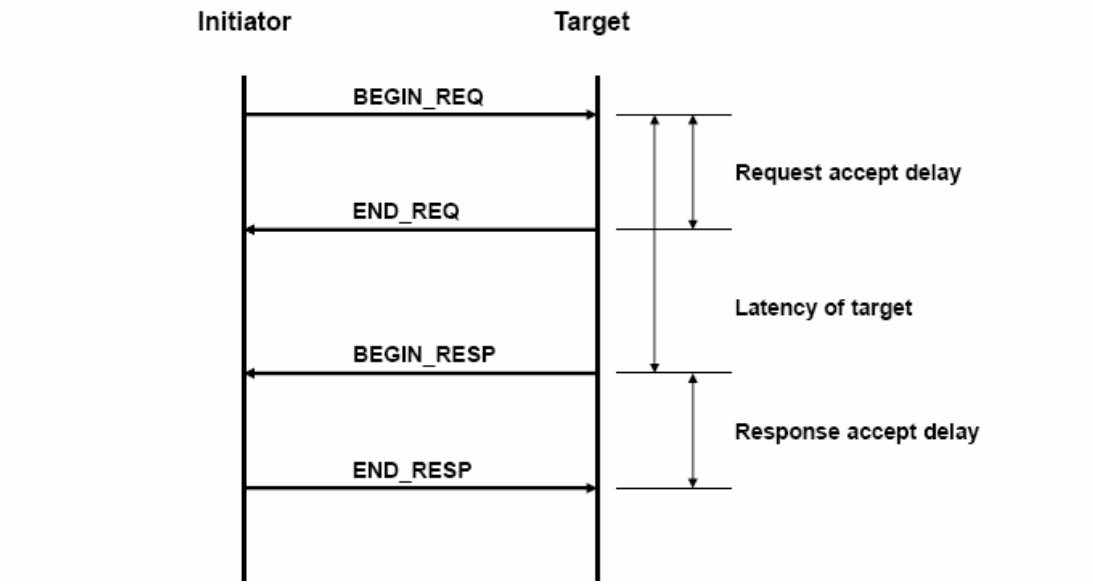
ターゲットで発生された無視可能フェーズの例は、スプリットトランザクションをマークするフェーズである。このフェーズを認識したイニシエータは、ターゲットがそれを処理する準備ができているのを知っていて、スプリットフェーズを受けしだい、すぐに次の要求を送るかもしれない。スプリットフェーズを無視したイニシエータは、2 番目の要求を送る前に最初の要求への応答を受けるまで待つかもしれない。

8.2.6. 基本プロトコルのタイミングパラメーターとフローの管理

a) 4 つのフェーズ、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付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。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) write コマンドのために、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。理論上、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している。ダウンストリーム・コンポーネントの義務は転送時間を計算し、次の転送を受け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・コンポーネントが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、あたりまえであるが、そうする義務はない。

c) read コマンドのために、BEGIN_RESP フェーズはインターコネクト・コンポーネントを通してターゲットからイニシエータへの転送のためにデータが利用可能になる時間をマークする。理論上、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している。アップストリーム・コンポーネントの義務は、転送時間を計算し、次の転送を受け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、あたりまえであるが、そうする義務はない。

d) READ 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 を戻して早く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のはアップストリーム・コンポーネントの義務である。(END_RESP を送ることは許可されていないので)。

e) 基本プロトコルの場合、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の END_REQ か 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終了するまで、イニシエータあるいはインターコネクト・コンポーネントは与えられたソケットを通して BEGIN_REQ フェーズを持つ新しいトランザクションを送らないものとする。これ

はリクエスト除外規則として知られている。

f) 基本プロトコルの場合、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の END_RESP を受信する、またはコンポーネントが TLM_COMPLETED でホップにより直前のトランザクションを終了するまで、ターゲットあるいはインターコネクト・コンポーネントは与えられたソケットを通して BEGIN_RESP フェーズを持つ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規則として知られている。

g) リクエストとレスポンス除外規則を含むフェーズ遷移を管理するすべての規則は、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（タイミング注釈）の値で影響を受けてはならないものとする。

h) 与えられたソケットを通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付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・オブジェクトが実行中（in flight）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）に返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) 与えられたソケットを通して 2 つの未処理のリクエストまたは応答が存在する可能性を除くこの規則は、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 への呼び出しに直接効果を全く持たないだろう。（b_transport 自身が nb_transport_fw を呼ぶ場合、この規則は b_transport への呼び出しに間接的影響があるかもしれない。）

j) 与えられた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、ゼロまたはそれ以上のインターコネクト・コンポーネントを通して伝播され、最終的にターゲットでそれを受け取るものとする。与えられた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 を受け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ることは許可されない。

k) 与えられたトランザクションにおいて、BEGIN_RESP はイニシエータがそれを受け取るまでいつもターゲットから開始され、ゼロまたはそれ以上のインターコネクト・コンポーネントを通して伝播されるものとする。与えられた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 を受け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ることは許可されない。これは、BEGIN_RESP が明示的か TLM_COMPLETED によって暗黙かに関係なく適用される。

l) 与えられたトランザクションのために、ダウンストリーム・コンポーネントから END_REQ を受ける前に、インターコネクト・コンポーネントは END_REQ をアップストリーム・コンポーネントに送ることができる。同様に、アップストリーム・コンポーネントから END_RESP を受ける前に、インターコネクト・コンポーネントは END_RESP をダウンストリーム・コンポーネントに送ることができる。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さない。これらの二つのフェーズが end-to-end での因果関係としては伝播されないため、それらは initiator-to-target や target-to-initiator 間の拡張の正当性を示すのに信頼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のにはそれらを使用できる。

n) 対応するフェーズを受け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規則による。

Figure 17

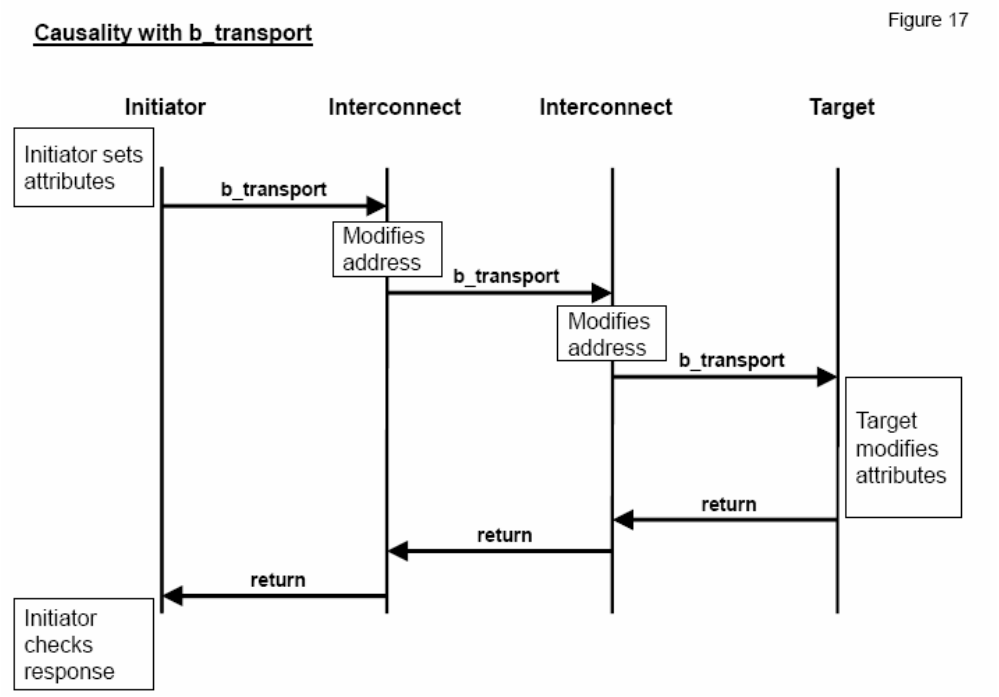
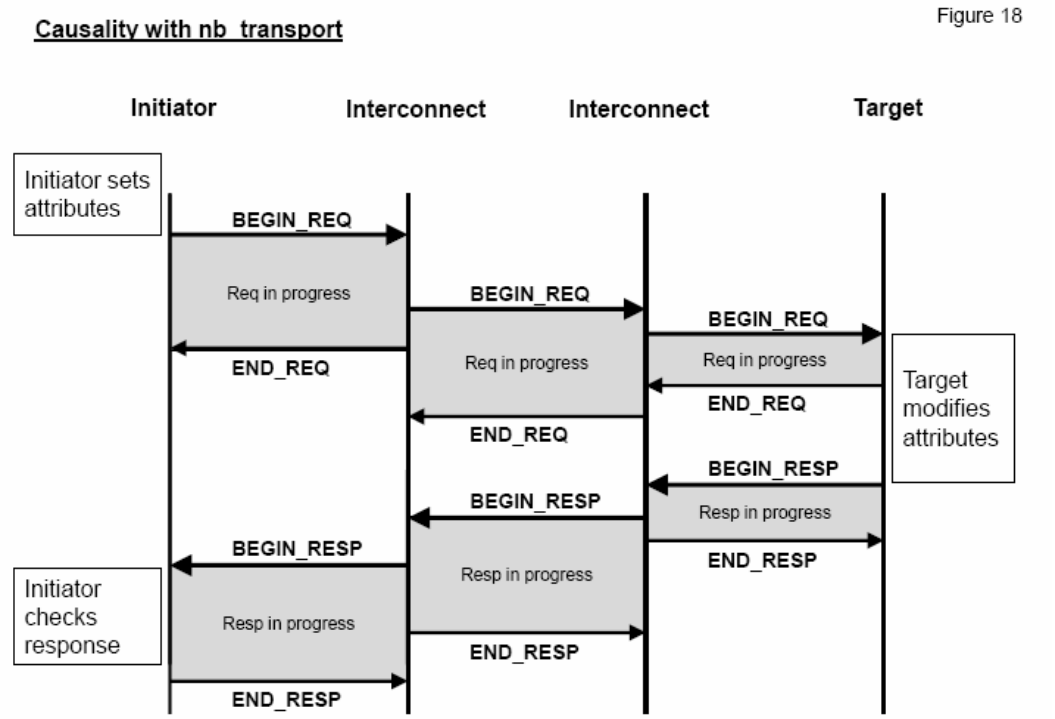


Figure 18



例

以下の擬似コードはタイミング・アノテーションと、リクエストと応答除外規則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(…);
}
```

8.2.7. 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) これらの規則は 4.1.3 Timing annotation with the transport interfaces に関連して読まれるべきである。

b) b_transport と nb_transport の実装に関して、時間引数 t を変更するのに有効なローカル時間 sc_time_stamp() + t がファンクションコールとリターンの間で非減算（加算のみ）であるという制約がある。「4.1.3.1 sc_time 引数」を参照のこと。

c) 与えられたトランザクションのための与えられたソケットを通して一連の nb_transport の呼び出しとリターンの為に、有効なローカル時間の結果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 という表現で与えられている。ここでは、t が nb_transport への時間引数である。このために、関数からの呼び出しとリターンの両方は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、フォワードとバックワード・パスで同じように適用される。この意図は、与えられたトランザクションに対し時間が逆行してはならないと言うことである。

d) 上記のルールはまた b_transport の呼び出しとリターン間にも適用される。「4.1.3.1 sc_time 引数」を再び参照のこと。

e) そのうえ、与えられたトランザクション・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、応答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リターンで与えられた有効なローカル時間のシーケンスは非減少になること。この意味で、リクエスト伝播は b_transport と BEGIN_REQ フェーズの呼び出しを含んでいる。応答伝播は b_transport からのリターン、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）の値を増加させることによって、あるいは SystemC シミュレーション時間（b_transport のみ）、または両方を進めることによって増加させることができる。

g) 異なったトランザクション・オブジェクトについては、b_transport と nb_transport への呼び出しの実効的なローカル時間は非減少になることの義務は全くない。それにもかかわらず、各イニシエータ・プロセスは一般に、非減少の有効なローカル時間で 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、out-of-order トランザクションが別々のイニシエータから発したと推論して、それらの特定のトランザクションが実行された順序を選ぶのは、自由である。しかしながら、異なった loosely-timed イニシエータからのトランザクションのストリームがどこで収束しても、out-of-order 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ることは可能である。

h) 与えられたソケットについて、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通して、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、イニシエータが異なったトランザクション・インスタンス（汎用ペイロードのメモリ管理規則に従ってすべて）に同じトランザクション・オブジェクトを再使用することが許可されている。「7.5 汎用ペイロード・メモリ管理」を参照。

8.2.8. 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 をコールできて、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 への呼び出しをすることができる。

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規則を遵守して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、同じイニシエータ・ソケットを通して、または、異なったソケットを通してそれらの呼び出しをかけるかどうかにかかわらず、同じモジュールの違うスレッドから作られる b_transport コールの順番に制約はない。

c) インターコネクトまたはターゲットは常に、異なったイニシエータスレッドや、異なったプロセスから、複数の並列の b_transport 呼び出しが任意の並列の nb_transport_fw 呼び出しに対し発生し、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 を用い、1 つのリクエストが別のリクエストを追いこすことができる。

d) 同じソケットを通して同じトランザクション・オブジェクトのために b_transport にリエントラントコールをすることは禁じられる。

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;
}
```

8.2.9. request と response 順番関連の基本プロトコル・ルール

以下の規則の意図は、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るとき、イニシエータからそれらを送った順番どおりに、ターゲットでそれらのリクエストを実行するのを保証することである。汎用ペイロード・トランザクションがイニシエータの同一性もターゲットの同一性も格納しないので、イニシエータは入って来るソケットの同一性から推定できるだけである。そして、ターゲットは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。オーバラップしていないアドレスに送ら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通して入ってくるリクエスト、または、到着する応答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・ソケットを通して到着するリクエストに、または特定のデータの長さを持っているリクエストに対し、低い優先度リクエストに追いつくのを許容して、より高い優先度を割り当てることができる。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けられた順序に合わせるために異なったイニシエータ・ソケットを通して返って来る応答を並べ替えるかもしれない。

b) リクエストルーティングは、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。（これらはトランスポート、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唯一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・コンポーネントがオーバラップするアドレスに複数

の同時要求をフォワード・パスに送るなら、それらのリクエストは同じイニシエータ・ソケットを通して送られるものとする。複数の同時要求の意味は、対応する応答がターゲットからまだ受けられていない要求を意味する。オーバラップするアドレスの意味は、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 Read と Write の要求は、それらが同時発生でないなら、異なったソケットを通して送られることができる。

d) 同じターゲット・ソケットを通して 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）が複数の同時要求を受け取るなら、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フォワードに送る（または、それぞれ実行）するものとする。同じ順番の意味は、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・メソッド呼び出し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け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してもこの規則が成立することに注意すること。）

e) 前の規則は、入ってくる b_transport 呼び出しには当てはまらない。そのため、複数の同時要求の順序に制約はない。他方では、入って来る nb_transport_fw 呼び出しを出ていく b_transport 呼び出しに変換するなら、b_transport 呼び出しは nb_transport 呼び出しの順序の規則を実施するために順番に並べなければならない。

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、異なったターゲット・ソケットを通して受け取ったか、異なったイニシエータ・ソケットを通して送ったか、アドレスが重ならないか、入って来る b_transport 呼び出しのためのものは、複数の同時要求を並べ替えることが許可されている

g) 応答は並べ替えられるかもしれない。対応するリクエストを送ったとき、応答はイニシエータに同じ順番で帰って来るという保証が全くない。

h) インターコネクト・コンポーネントに複数の同時要求を並べ替えることを許す無視可能な拡張を使用することは、基本プロトコルで技術的に可能である。その場合、拡張を加えたイニシエータは、ターゲットによるアウト・オブ・オーダー実行を許容できなければならない。他方では、リクエストを送ったのと同じ順序での応答を強いる拡張は無視可能な拡張でないであろう。したがって、基本プロトコルでは、受入れられないであろう。

8.2.10. b_transport と nb_transport のスイッチングの基本プロトコル・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が異なるトランザクション・オブジェクトのために b_transport と nb_transport_fw の呼び出し間で切り換えることが許可されている。意図は、イニシエータが loosely-timed と approximately-timed コーディング・スタイル間を切り替えることを許可することである。b_transport と nb_transport_fw を交互に呼ぶイニシエータは、タイミング精度に関して低い期待を持つべきである。

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とノン・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することを強いられる。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な内部状態情報も保守することが強いられる。

これは、同じソケットを通して、または、異なったソケットを通して受信した、入って来るインタフェース・メソッド呼び出しに適用される。

c) イニシエータまたはインターコネクト・コンポーネントのスレッドは、同じトランザクション・インスタンスのために b_transport と nb_transport_fw の両方を呼ばないものとする。スレッドは、その時々異なったトランザクション・インスタンスを示す同じトランザクション・オブジェクトのために b_transport と nb_transport_fw の両方を呼ぶことができる。

d) その同じスレッドから前の nb_transport_fw 呼び出しによる進行中のトランザクションがまだあれば、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が、b_transport を呼ばないことを推奨する。すなわち、非ゼロ・リファレンス・カウントの先行のトランザクションがあるときである。そうでなければ、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来たと誤って推論するかもしれない。

e) コンビニエンス・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 のどちらか一つを実装することを要求されながら、どう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両方をサポートできるかの例を提供する。「9.1.2 シンプル・ソケット」を参照。

8.2.11. その他の基本プロトコル・ルール

a) 複数の平行なソケットを通して、または、同時に複数の平行な経路に沿って与えられたトランザクション・オブジェクトを送っ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、トランザクション・インスタンスの生存期間が残っていて、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に共通のコンポーネントとソケットを通してユニークな明確な経路を取るものとする。もちろん、与えられたソケットを通して送られた異なるトランザクションは、異なった経路を取るかもしれない、すなわち、それらは異なって送られるかもしれない。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、または、ターゲットとして機能するかどうかをダイナミックに選ぶかもしれな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) Write トランザクション（TLM_WRITE_COMMAND）のために、応答ステータスの TLM_OK_RESPONSE は、Write コマンドがターゲットでうまく完了したことを示す。ターゲットは b_transport から戻る前か、バックワードまたはリターン・パスに従って BEGIN_RESP を送る前か、TLM_COMPLETED を返す前に応答ステータスを設定することが強いられる。言い換えれば、インターコネクト・コンポーネントは、ターゲットからうまく完了したという確認することなく、Write トランザクションの完了の合図することが許可されない。この規則の意図は、ターゲットシミュレーションモデルの中でストレージのコヒーレンシーを保証することである。

d) Read 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、Read 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ものである。ターゲットは、b_transport から戻る前か、バックワードまたはリターン・パスに従って BEGIN_RESP を送る前か、TLM_COMPLETED を返す前に応答ステータスを設定することが強いられる。

8.2.12. トランザクション順番に関する基本プロトコル・ルールのまとめ

以下のテーブルは基本プロトコルのために、トランザクション順番規則の概要を示す。規則の完全な解説について、上の項を参照すること。

基本プロトコル順番規則は 3 つのカテゴリの規則の合体である: 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規則、因果関係とフェーズに関する基本プロトコルの特定の規則、そして、パイプライン化したリクエストが、ターゲットでイニシエータによって予想された順序に実行されることが保障されるという基本プロトコルの特定の規則。

状況	順番規則
インタフェース・メソッド呼び出し順序と異なった有効なローカル時間の順序	受取側は、順不同なトランザクションを実行、または送ることができる。他のすべての規則より優先する。
同じソケットを通した同じトランザクションのための連続したトランスポート・メソッド呼び出しとリターン	実効的なローカル時間の順序は非減少でなければならない。
同じイニシエータ・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。
同じイニシエータ・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きているなら b_transport を呼ばないことを推奨する。
同じソケットを通した異なったトランザクションのため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、もし入って来るトランザクションのストリームが非減少であるなら、非減少を推奨する。
nb_transport だけ、同じソケットを通しての 2 つのリクエストまたは 2 つの応答のアウトスタンディング	禁止する
異なったソケットを通したトランザクションの入力	実行される、または送られる順序の義務は無い。
重複アドレスによる複数の同時要求	もし、前方に送るなら、同じソケットを通して送らなければならない
nb_transport の同じソケットを通した重複アドレスの入力の複数の同時要求	それらを受け取ったのと同じ順番で実行、または前方に送られなければならない。
b_transport を使用する複数の同時要求入力	実行、または前方に送られる順序の義務は無い。
複数の同時発生の応答	実行、または送って戻される順序の義務は無い。

8.2.13. Guidelines for creating base-protocol-compliant components

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規則の簡潔な言い換えであり、利便性を提供する。

8.2.13.1. Guidelines for creating a base protocol initiator

a) 各接続のためにクラス tlm_initiator_socket（または、派生しているクラス）の 1 個のイニシエータ・ソケットをメモリ・マップド・バスにインスタンスする。

b) tlm_initiator_socket にテンプレートの TYPES 引数のためのデフォルト値 tlm_base_protocol_types を取ることを許容する。

c) メソッドの nb_transport_bw と invalidate_direct_mem_ptr を実装する。（イニシエータはコンビニエンス・ソケット simple_initiator_socket をインスタンスすることによって明らかにこれらのメソッドを実装する必要性を避けることができる。）

d) 引数として b_transport か nb_transport_fw にそれをパスする前にそれぞれの汎用ペイロード・トランザクション・オブジェクトのあらゆるアトリビュートを設定すること、具体的には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。（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 であるケースにおいて設定される必要はなくて、拡張は使用する必要はない。）

e) 汎用ペイロード拡張メカニズムを使用するときには、どんな拡張も確実にターゲットとインターコネクト・コンポーネントでも無視可能になるようにすること。

f) フェーズ順序、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番に関する基本プロトコル規則に従うこと。

g) トランザクション（または BEGIN_RESP を受けた後に）の完了のときに、応答ステータス・アトリビュートの値をチェックすること。

8.2.13.2. Guidelines for creating an initiator that calls nb_transport

a) 引数としてトランザクションを nb_transport_fw にパスする前に、トランザクション・オブジェクトのためにメモリ・マネージャを設定すること、そして、トランザクションの acquire メソッドをコールする。トランザクションが完了であるときに release メソッドをコールする。

b) nb_transport_fw を呼ぶときには、トランザクションの状態に従って、BEGIN_REQ か END_RESP にフェーズ引数を設定する。前のトランザクションから END_REQ を受ける（または、推論される）前に、BEGIN_REQ を送ってはならない。

c) 与えられたトランザクションのために、一連の呼び出しを nb_transport_fw にするときには、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）が値の非減少であるシーケンスを形成することを確実にする。

d) バックワード・パス（nb_transport_bw への呼び出し）、リターン・パス（nb_transport_fw から戻った TLM_UPDATED）、または暗黙に受け取るか否かに関係なく、適切に入って来るフェーズ値の END_REQ と BEGIN_RESP で応答すること。（例えば、nb_transport_fw から戻った

TLM_COMPLETED)。BEGIN_REQ と END_RESP で入って来るフェーズの値は不正である。他のすべての入って来るフェーズの値は無視可能であるとして扱うこと。

8.2.13.3. Guidelines for creating a base protocol target

a) 各接続のために、クラス tlm_target_socket（または、派生しているクラス）の 1 個のターゲット・ソケットをメモリ・マップド・バスにインスタンスする。

b) tlm_target_socket にテンプレートの TYPES 引数のためのデフォルト値 tlm_base_protocol_types を取らせること。

c) b_transport、nb_transport_fw、get_direct_mem_ptr と transport_dbg メソッドを実装する。（ターゲットはコンビニエンス・ソケット simple_target_socket を使用することによって、明らかにあらゆるメソッドを実装する必要性を避けることができる。）

d) b_transport と nb_transport_fw メソッドの実装では、応答ステータス、DMI ヒント、およびどんな拡張の例外のある汎用ペイロードのあらゆるアトリビュートの値を点検し、値に応じ動作すること。汎用ペイロードの完全な機能性を実装するよりむしろ、ターゲットは、誤り応答を発生させることによって与えられたアトリビュートに応答することを選ぶことができる。応答ステータス・アトリビュートの値にトランザクションの成否を示すように設定すること。

e) フェーズ順序、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番に関する基本プロトコル規則に従うこと。

f) get_direct_mem_ptr の実装では、値の false を返すか、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。そして、リターン値と適切に DMI 記述子（tlm_dmi クラス）のすべてのアトリビュートを設定する。

g) transport_dbg の実装では、値 0 を返すか、コマンドの値とアドレス、データ長と汎用ペイロードのデータ・ポインタ・アトリビュートの値を点検し、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、汎用ペイロードにおけるどんな無視可能拡張も点検し、動作することができるが、そうすることが強いられている訳ではない。

8.2.13.4. Guidelines for creating a target that calls nb_transport

a) nb_transport_bw を呼ぶときには、トランザクションの状態に従って、END_REQ か BEGIN_RESP をフェーズ引数に設定すること。前のトランザクションから END_RESP を受け取る（または、推論する）前に、BEGIN_RESP を送らないこと。

b) 与えられたトランザクションのために、一連の呼び出しを nb_transport_bw にするときには、実効的なローカル時間（シミュレーション時間+タイミング・アノテーション）が値の非減少であるシーケンスを形成することを確実にする。

c) フォワード・パス（nb_transport_fw への呼び出し）、リターン・パス（nb_transport_bw から戻った TLM_UPDATED）、または暗黙に受け取るか否かに関係なく、入って来るフェーズ値の BEGIN_REQ と END_RESP に適切に応答すること。（例えば、nb_transport_bw から戻った TLM_COMPLETED）。入って来るフェーズ値の END_REQ と BEGIN_RESP は不正である。

他のすべての入って来るフェーズの値は無視可能であるとして扱うこと。

d) nb_transport_fw の実装では、メソッドからリターンを超えてポインタかリファレンスをトランザクション・オブジェクトに保つのが必要であるときには、トランザクションの acquire メソッドをコールすること。トランザクション・オブジェクトが使い終わったら、release メソッドをコールすること。

8.2.13.5. Guidelines for creating a base protocol interconnect component

a) 各接続のために tlm_initiator_socket クラスか tlm_target_socket クラス（または、クラスを派生させる）の 1 個のイニシエータかターゲット・ソケットをメモリ・マップド・バスにインスタンスする。

b) 各ソケットにテンプレートの TYPES 引数のためのデフォルト値 tlm_base_protocol_types を取らせること。

c) それぞれのイニシエータ・ソケットのために nb_transport_bw と invalidate_direct_mem_ptr メソッドを実装し、それぞれのターゲット・ソケットのために b_transport、nb_transport_fw、get_direct_mem_ptr、および transport_dbg メソッドを実装する。（コンビニエンス・ソケットを使用することによって、明らかにあらゆるメソッドを実装する必要性を避けることができる。）

d) 両方のフォワードとバックワード・パスに必要に応じて入ってくるインタフェース・メソッド呼び出しを TLM_COMPLETED に続いて、リクエスト、応答除外規則、トランザクション順番規則、および更なる呼び出しが全く許されていないという規則を守ってパスすること。無視可能フェーズはパスしないこと。トランザクション・オブジェクトを進めないで、get_direct_mem_ptr と transport_dbg メソッドの実装は、それぞれ false と 0 値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では、インターコネクト・コンポーネントで修正できる唯一の汎用ペイロード・アトリビュートは、アドレスと、DMI ヒントと、拡張である。他のアトリビュートは変更しないこと。いかなる他のアトリビュートも変更する必要があ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て、その結果、それ自体で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して、そして、必要なら、システム・メモリ・マップのターゲットのロケーションに従って、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、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に適用される。

g) トランスポート・インタフェースの実装では、フェーズの順番、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番に関する基本プロトコル規則に従うこと。

h) get_direct_mem_ptr の実装では、フォワード・パスの DMI 記述子のアトリビュートを変更してはならない。DMI ポインタ、DMI 開始アドレス、および終わりのアドレスを変更すること。そうすれば、DMI はリターン・パスで適切にアトリビュートにアクセスする。

i) invalidate_direct_mem_ptr の実装では、バックワード・パスに沿って呼び出しをパスする前に、アドレスレンジ引数を変更すること。

j) nb_transport_fw の実装では、関数からリターンを超えてポインタかリファレンスをトランザクション・オブジェクトに保つのが必要であるときには、トランザクションの acquire メソッドをコールする。トランザクション・オブジェクトが使い終わられたら release メソッドをコールする。

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、汎用ペイロードにおけるどんな無視可能な拡張も点検し、実効することができる。但しそうすることは絶対条件ではない。トランザクションが、さらに拡張される必要があるなら、どんな拡張も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・メモリ管理規則を守ること。

9. ユーティリティ

ユーティリティは、一貫したコーディング・スタイルを保つことを助ける利便性の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、これを使用することはインタオペラビリティを保つために必須なものではない。

9.1. 便利ソケット

9.1.1. 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､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 の相互利用性レイヤ（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。

9.1.1.1. 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

  対応するインタフェースの関数を実装しているオブジェクトに対するソケットをバインドする(通常の)方法ではなく、呼ばれるべきインタフェース・メソッド・コールの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、その逆をすることができる。(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

9.1.1.2. 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許されるバインディングは以下の表にて要約される。From と To をソケットタイプとすると、バインディングは、From（To） 又は From.bind（To）の形式になる。

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つの区域に分割されている。

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。

9.1.2. シンプル・ソケット

9.1.2.1. 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、それらのタイプを持つソケットに直接バインドできる。

シンプル・ソケットは、双方向に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・メソッド呼び出し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それぞれの関数の登録をおこなうために使用することができる。

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・メソッドを登録してもいいが、必ずしもそうする必要はない。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定義するだけでよい。この場合、登録されていない方のインタフェース・メソッドが来た場合には、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が基本プロトコルの規則に遵守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・インプリメンテーションとして、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は、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、 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。

9.1.2.2. 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
```

9.1.2.3. 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。

9.1.2.4. 規則

a) コンストラクタに引数が一つ有った場合には、char *型の引数としてそのまま引き渡す形式にて、該当する基底クラスのコンストラクタを呼び出す。（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基底クラスのコンストラクタの引数として使用する。

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。なおこの二つは同じ効果である。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 にも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、呼ばれるインタフェース・メソッド・コールを実装する。それに対して、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し、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 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。入力された 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。（「9.1.2.5 シンプル・ターゲット・ソケット b/nb 変換」を参照）

f) b_transport 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。入力された nb_transport_fw 呼び出しは、自動的に登録されている b_transport 呼び出しに変換される。

g) ターゲットのシンプル・ターゲット・ソケットに b_transport も nb_transport_fw も登録されていない場合、インタフェース・メソッドの応答は、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、インタフェース・メソッドの応答時に、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・ソケットに対しては、transport_dbg 関数を登録する必要は必ずしも無い。登録しなかった場合に、**transport_dbg** が呼びれた場合には、0 の値をリターンする。

j) ターゲットのシンプル・ターゲット・ソケットに対しては、**get_direct_mem_ptr** 関数を登録する必要は必ずしも無い。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、**invalidate_direct__mem_ptr**

関数を登録する必要は必ずしも必要ない。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。

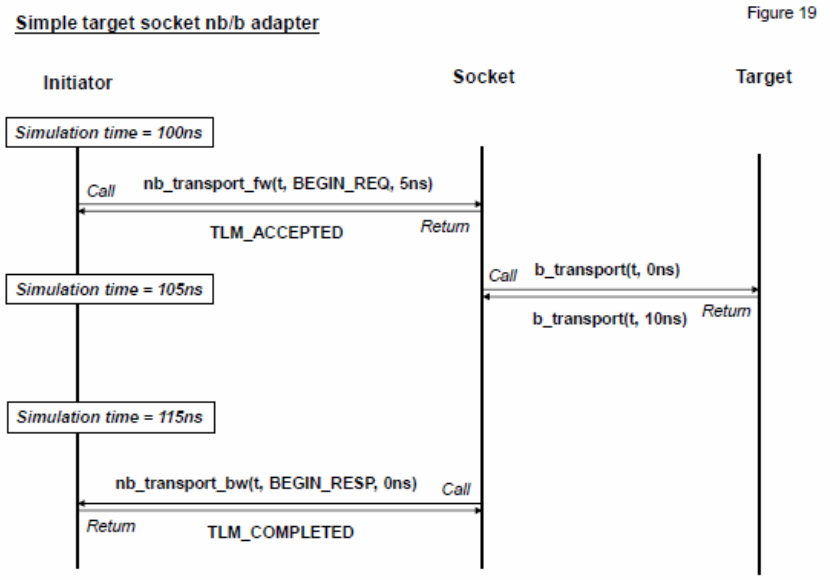
9.1.2.5. シンプル・ターゲット・ソケット b/nb 変換

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て、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 メソッドをインプリメントした観点から、基本プロトコルの規則を遵守している。

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・オブジェクトが新規に作られたりはしない。

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 メソッドがまだ進行中に、イニシエータが、nb_transport_fw と呼ぶことが許可されない。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。



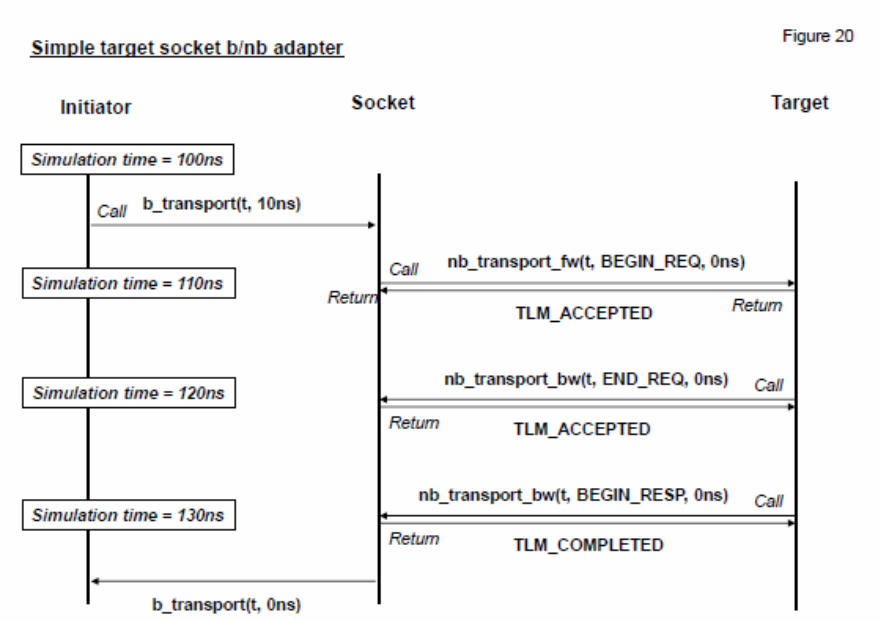
e) 図 19 は、イニシエータがどう nb_transport_fw メソッドをコールする様子を示している。イニシエータは nb_transport_fw メソッドをコールするが、ターゲットは、シンプル・ターゲットケットに b_transport コールバック関数が登録されているだけである。イニシエータは BEGIN_REQ を送る。そして、ソケットは TLM_ACCEPTED をイニシエータにリターンする。ソケットは、次に、b_transport をコールして、リターンされたら、BEGIN_RESP をイニシエータにリターンする。最後は、イニシエータは TLM_COMPLETED をソケットにリターンする。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・メソッドを呼ぶのが許されない。したがって、ソケットは、nb_transport_fw メソッドから 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。他の例は、図の一番最後の遷移で、イニシエータは TLM_COMPLETED ではなく、TLM_ACCEPTED をリターンした場合である。その場合、ソケットはイニシエータからこの後に END_RESP を受け取ると予想される。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、イニシエータは、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 コールバック関数が登録されている。イニシエータは、b_transport メソッドをコールする。そして、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_RESP を送るかもしれない。あるいは、いきなり 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 であるのを見て 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
  }
};
```

9.1.3. タグ付きシンプル・ソケット

9.1.3.1. 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、コールバックが呼ばれるとき、どのソケットに到着し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。コールバック関数を登録したときに使用した id 番号が、コールバック関数が呼ばれるときに、その初めの引数として追加される。

9.1.3.2. ヘッダファイル

タグ付きシンプル・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 の中に定義される。

9.1.3.3. 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
```

9.1.3.4. 規則

a) コンストラクタに引数が一つ有った場合には、char *型の引数としてそのまま引き渡す形式にて、該当する基底クラスのコンストラクタを呼び出す。(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") の戻り値をそれぞれ基底クラスのコンストラクタの引数として使用する。

b) タグ付きシンプル・ソケットは、int id タグが別な引数として追加されるだけで、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) 与えられたコールバック・メソッドは、異なったタグを使用する複数のソケットとして登録できる。

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、マルチ・ソケットや他のコンポーネントにバインドすることができない。(「**エラー! 参照元が見つかりません。エラー! 参照元が見つかりません。**」を参照)

9.1.4. マルチ・ソケット

9.1.4.1. 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、単一のソケットが他のコンポーネントの複数のソケットに割り当てられることを許可する。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・メソッド呼び出しが別のコンポーネントのどのソケットから到着するかを特定できる。また、マルチ・ソ

ケットは、他の便利ソケットと異なり、イニシエータ、ターゲット双方で、子から親へのソケット階層バインドをサポートする。

9.1.4.2. 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。

9.1.4.3. 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
```

9.1.4.4. 規則

a) multi_init_base と multi_target_base ベースクラス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) コンストラクタに引数が一つ有った場合には、char *型の引数としてそのまま引き渡す形式にて、該当する基底クラスのコンストラクタを呼び出す。(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基底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つクラステンプレートは、バインド数とバインドポリシーを定義するテンプレート・パラメタを持つ。このパラメタには、sc_port テンプレートインスタンスをパラメタライズするクラスインプリメンテーションが使われている。

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 とバインドできる。 複数の multi_passthrough_target_socket、simple_initiator_socket、multi_passthrough_target_socket 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 ソケットに階層バインドできる。multi_passthrough_target_socket は、ただ一つの multi_passthrough_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 initiator-socket から target-socket の方向のみで使える。言い換えれば、multi_passthrough_target_socket には、tlm_target_socket や simple_target_socket クラスと異なって、ターゲット・ソケットをイニシエータ・ソケットに割り当てる（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、下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する。これは、上記のイニシエータ-ターゲットのバインド方向のルールと一貫性がある。

h) もし、multi_passthrough_initiator_socket か multi_passthrough_target_socket が複数回バインドされていた場合、メソッド operator [] を、 ソケットにバインドしている応答オブジェクトにあてはめることができる。 インデックス値は、bind メソッドか operator()がソケットをバインドする為に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 タグを決めるのに使われる。

i) 例えば、multi_passthrough_initiator_socket が２つの別なターゲットにバインドされている場合を考えてみる。

socket[0]->nb_transport_fw(...)

socket[1]->nb_transport_fw()

は２つのターゲットに割り当てられる。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こととなってしまう（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) もし、マルチ・ソケットから下階層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。それもしなかった場合には、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) もし、マルチ・ソケットから下階層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) もし、マルチ・ソケットから下階層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) もし、マルチ・ソケットから下階層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。それもしなかった場合には、**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) もし、マルチ・ソケットから下階層への階層バインディングが無い場合には、イニシエータ

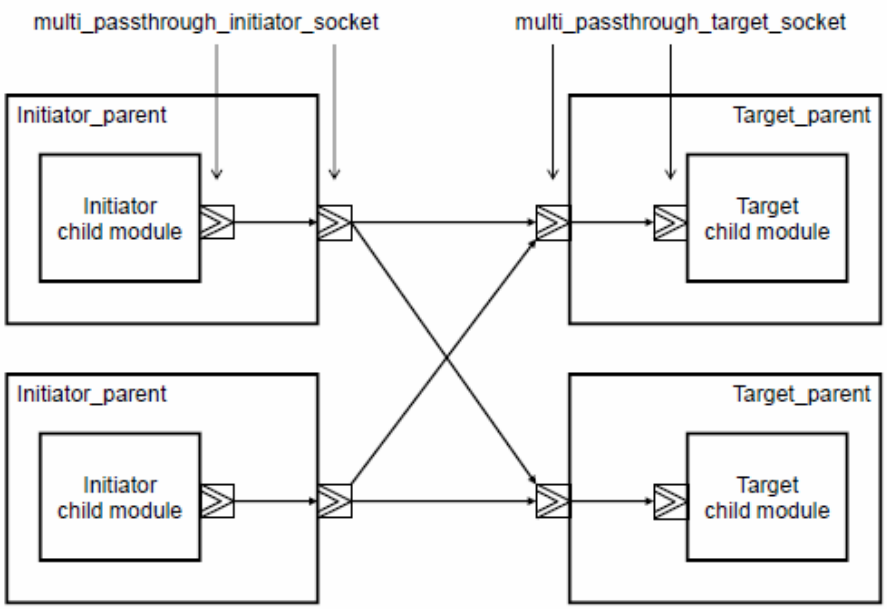
は、マルチ・ソケットに対して、**invalidate_direct_mem_ptr** コールバックを登録しなくても良い。この場合に **invalidate_direct_mem_ptr** が呼ばれると無視される。

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 );
  }
};
```

Hierarchical Binding of Multi-sockets

Figure 21




```
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 );
 }
};
```

9.2. クォンタム・キーパー

9.2.1. 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。

tlm_quantumkeeper クラスには、クォンタム時間に対して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 記述の中にこの機能を実装するということも可能である。tlm_quantumkeeper クラスの使用有無に関わらず、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。

9.2.2. 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。

9.2.3. 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
```

9.2.4. テンポラル・デカップリングの使い方に関する一般ルール

a) 最大のシミュレーション速度を実現するためには、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 プロセスは全く存在しないか、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・デカップリングを持つイニシエータ・モデルのみに、SystemC プロセスがあり、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・カーネルに戻る（Yield）ようにする。

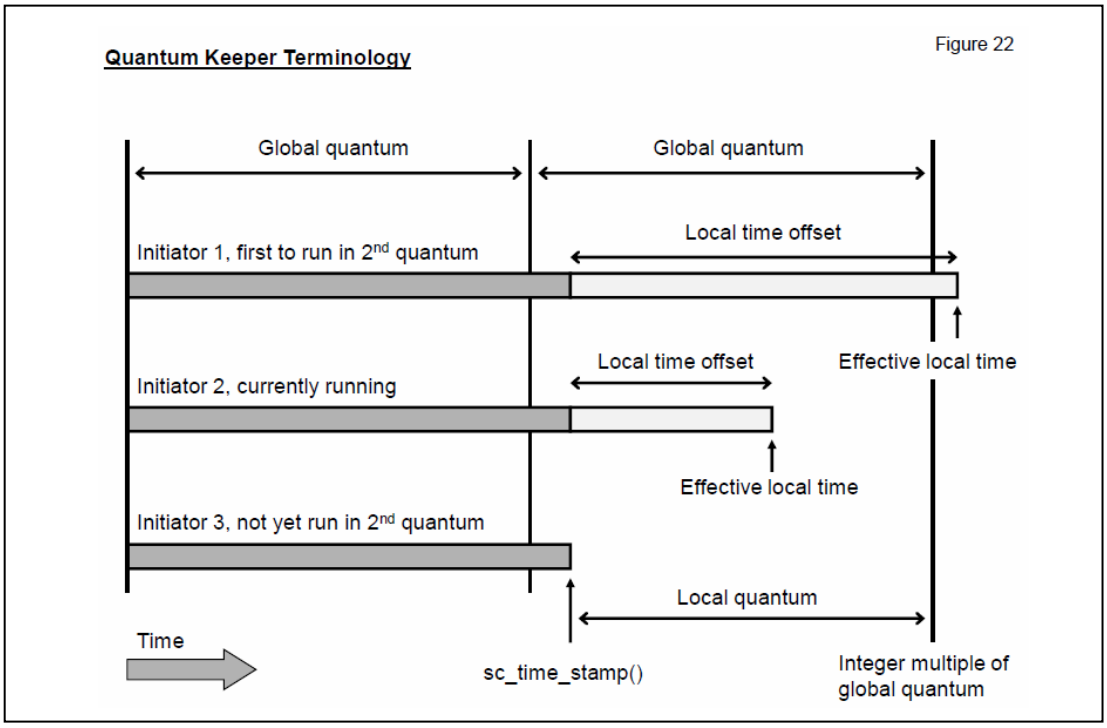
c) テンポラル・デカップリングを持つイニシエータは、他のプロセスとの通信において明示的な同期が取れていればクォンタム時間を用いる必要はない。クォンタム時間を用いる場合は、イニシエータ間の一般的な通信間隔よりも小さくなるよう設定すべきである。そうしないと重要なプロセス間通信が失われた不完全なモデル動作になるかもしれない。

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・デカップリングといえども、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、LT コーディング・スタイルは他のコーディング・スタイルと混在できる。

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。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、そうすると、シミュレーション速度が落ちてしまう。

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・オフセット時間変数を用いて、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・オフセット時間を管理することを推奨する。



h) sc_time_stamp()を呼んだときにはあくまでも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・オフセット時間は、SystemC スケジューラからは不明である。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・オフセット時間を渡すべきである。

j) nb_transport()関数でテンポラル・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、ルールの範囲外である。しかし、AT 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無効にな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。そのため明示的な同期機構がない場合、変数があるプロセスで書き換えられて別のプロセスで読まれた時、その読んだ値は不確定である。新しい値は現在のクォンタム、あるいは次のクォンタムで有効になるかもしれない。クォンタム長と比較して相対的に値が変化するだけならば、アプリケーションは新しい値が有効になるタイミングに寛容な造りにしておく必要がある。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、sc_signal の値はクォンタム開始時点から変更されない。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。

9.2.5. tlm_quantumkeeper クラス

a) コンストラクタはローカル・タイムのオフセットを SC_TIME_ZERO に設定するが、仮想関数の compute_local_quantum()はコールしない。これは、コンストラクタでローカル・クォンタムを計算せずに、アプリケーションがクォンタム・キーパー・オブジェクトを生成直後に、reset()メソッドをコールからである。

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せずに、コンストラクタが 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、グローバル・クォンタム時間を引数によって設定する。しかしローカルのクォンタム時間は変更されない。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・オフセット時間を返す。

e) get_current_time()メソッドは、ローカル時間、すなわち、sc_time_stamp()+ローカル・オフセット時間を返す。

f) inc()メソッドは、ローカル・オフセット時間を引数によって渡しローカル時間を実時間よりも先に進める。

g) set()メソッドは、引数で渡された値をローカル・オフセット時間に設定する。

h) need_sync()メソッドは、現在のローカル・オフセット時間がローカル・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・オフセット時間)を呼び出し、実際のシミュレーション時間とローカルの時間との同期をとり、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。このメソッドをオーバーライドしてはいけない。

k) reset()メソッドは、compute_local_quantum()を呼び出し、ローカル・オフセット時間を 0 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。

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なり、この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、普通ではない。

n) ローカル・オフセット時間がローカル・クォンタム以上になった場合には、プロセスを yield させる必要があるが、自分で wait()を呼び出さないで、sync()メソッドを呼び出すことを強く推奨する。

o) ローカル・クォンタム量を超えた場合に同期を自動的に実行するようなしかけはない。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、そのコール前後でグローバル時間が変化していた場合に、自身で yield するかもしれない。ローカル・オフセット時間やタイミング・アノテーションの値は、常にグローバル時刻からの相対値で表現される。b_transport()あるいは nb_transport_fw()からの戻りでは、イニシエータが set()メソッドのコールによって、クォンタム・キーパーのローカル・オフセット時間を設定し、need_sync()メソッドのコールによって同期チェックを行う必要がある。

q) イニシエータがローカル・クォンタム量を超えるよりも前に実行をサスペンドして、実時間を先に進んでしまっているローカル時間に追いつかせ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 させてもいい。途中で同期させるこの方法を、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
      }
    }
  ...
};
```

9.3. ペイロード・イベント・キュー

9.3.1. イントロダクション

ペイロード・イベント・キュー（PEQ）は、トランザクション・オブジェクトに関連付けられたSystemC のイベント通知用のキューである。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、AT 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る時、PEQ への格納が適している。PEQ は、正確なシミュレーション時間に発生する nb_transport()コールに関するタイミング・ポイントをスケジューリングする。

PEQ の notify()メソッドを、引数にディレイ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。すぐにスケジューラへ通知を行う notify()メソッドも存在している。

遅延時間は現在のシミュレーション時間（sc_time_stamp）との加算に使われ、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、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取り出し方法が異なる。peq_with_get の場合は、get_event()メソッドをコールすると、取り出しが可能な状態のトランザクションに関連するイベントを返す。get_next_transaction()メソッドは、あるタイミングで有効になっているトランザクションの 1 つを取り出すために、繰り返しコールされるメソッドである。

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に登録することで、各トランザクションを取り出す際にそのコールバック・メソッドがコールされる。この PEQ を用いる場合には、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。

9.3.2. 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 ヘッダファイルにある。

9.3.3. 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
```

9.3.4. ルール

a) notify メソッドは PEQ にトランザクションを挿入する。そのトランザクションは t1+t2 の時間で PEQ から取り出せる。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。すぐに通知を行う場合には、トランザクションを SystemC スケジューラの現在の評価フェーズで取り出せる。

b) 上記ルールによって、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) 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ある場合、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を取り出せる。

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初期状態に戻す。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 クラスのテンプレート引数である PAYLOAD には、PEQ で用いるトランザクション・タイプを指定する。

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す。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する。評価フェーズに

おいてイベント通知が発生した時に PEQ からトランザクションを取り出すことができない場合には、その後のタイミングで再度 get_next_taransaction()コールによって取り出すまでトランザクションは有効なまま継続する。

i) 現在の評価フェーズにおいて受け取れ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 ポインタを返す。

j) peq_with_cb_and_phase クラスのテンプレート引数である TYPES には、PEQ で用いるトランザクションとフェーズ・タイプを含むプロトコル・トレイツ・クラスを指定する。

k) peq_with_cb_and_phase クラスのテンプレート引数である OWNER には、PEQ のコールバック関数をメンバとして持つクラス名を指定する。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*には、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を指定する。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 には、PEQ のコールバック関数を指定し、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ると、常に PEQ のコールバック関数を呼び出す。コールバック関数の引数は、対応する notify メソッドの引数として渡される変数への参照であり、第一引数はトランザクション・オブジェクトへの参照、第二引数はフェーズ・オブジェクトへの参照である。

o) peq_with_cb_and_phase クラスは、SystemC の SC_METHOD プロセスから PEQ のコールバック関数を呼び出すため、コールバック関数はノンブロッキングである必要がある。

p) peq_with_cb_and_phase クラスは、トランザクション毎に 1 回ずつ PEQ のコールバック関数を呼び出す。コールバック関数を呼び出した後で、PEQ からトランザクション・オブジェクトを削除する。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。

9.4. インスタンス固有の拡張

9.4.1. 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 1 つまでのインスタンスを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たなければならない。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はできない。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、単体のトランザクション・オブジェクトに加えられた同じ型の複数の拡張について述べる。

インスタンス固有の拡張の型は、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。そのアクセス方法は set_extension と 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。だから、set_extension を呼ぶコンポーネントは clear_extension も呼び出さなければならない。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なければならない。

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間の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アプリケーションによって直接使われてはならない。それは実装において定義される名前である。

9.4.2. 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ある。

9.4.3. 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;                                          ser-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 );
...
```

10. 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・インタフェースとアナリシス・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に入れられているが、TLM-2.0 標準の主体からは切り離されている。ここでは詳しく触れない。

10.1. TLM-1 コア・インタフェース

シグネチャ付きのトランスポート・メソッドである transport(const REQ& , RSP&) 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
```

10.2. 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
```

10.3. 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;
};
```

10.4. 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、複数のコンポーネントにまたがるトランザクションに対する解析機能、例えば、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を実現するために使用される。

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り、アナリシス・ポートに対して write()をコールしたときにそれぞれのサブスクライバーに対して同一の write()をコールさせるようにすることである。アナリシス・ポートは 0 以上複数のサブスクライバーに対してバインドすることができ、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 const reference によってトランザクションに渡され、サブスクライバーはただちに処理を実行する。もし､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（generic_payload）に対する deep_copy()メソッドを行い、サブスクライバー自身がコピーして作成したトランザクションの持ち主になり自分でメモリ管理を行うことになる。

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てはならず、その他の解析等の側面的な解析機能のみで使用しなければならない。 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、他の目的にも使用可能である。例えば、「9.3 ペイロード・イベント・キュー」においても用いられている。

tlm_fifo は、tlm_analysis_triple というトランザクションと、開始時間、終了時間を含むものもサポートしている。

10.5. 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
```

10.6. ルール

a) tlm_write_if と tlm_analysis_if 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引数として使用する。これは、ベースクラスの sc_object に渡される。

c) bind メソッドは、サブスクライバーのアナリシス・ポートのインスタンスを登録し、write メソッドが呼ばれたときに登録されたサブスクライバー内の write メソッドが呼ばれる。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可能。

d) operator() は、bind()メソッドと同一。

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()の逆であり、サブスクライバーのリストから削除される。

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に対して、write()メソッドが const リファレンスの引数と共に呼ばれる。

h) write()メソッドはノンブロッキングであり、その実装の中に wait()を含んではいけない。

i) write()メソッドは、const reference によって渡されたトランザクションの内容や、その中のポインタ先のデータ（data、バイトイネーブル配列）を書き換えてはいけない。

j) もし、write()メソッドが return する前にそのトランザクション・オブジェクトに対しての処理を完了することができない場合には、新しいトランザクション・オブジェクトを deep_copy()メソッドによってコピーしてから実行する。コピーされたオブジェクトのメモリ管理はサブスクライバーが責任を負う。

k) tlm_analysys_fifo クラスのコンストラクタは、bound されてない tlm_fifo を構成する。

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す。

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
  }
};
```

11. 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、異なる抽象度のトランザクション・レベル・インタフェース間を接続するモジュール。異なるプロトコル・タイプで特定された 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 artifacts of the simulation model (a timestamp, for example).	アトリビュート	トランザクションで運ばれる、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。モデル化された BUS もしくはプロトコルで継承されたアトリビュート、タイムスタンプ等とシミュレーション・モデルの成果物も含む。
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	自動削除のために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 タイプ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・モデルの間の最大限度の相互利用性を確実にするため関連セット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・アトリビュートはトランザクション存続期間中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セグメントを接続するコンポーネント。バス・ブリッジは 2 つの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	ある関数を呼び出している関数、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イニシエータを指すときによく使われる。
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	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ターゲットを指すときによく使われる。
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薦される。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。
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: ブロッキングとノンブロッキング、フォワードとバックワード（を統合した）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	ある SystemC モジュールの一インスタンス。ある SystemC モジュールの一インスタンス。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 種類コンポーネントを区別している。

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・インタフェースで、他のいかなるインタフェースも継承しておらず、トランザクション・タイプをテンプレート・パラメタとして持つ。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 とは違う。
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、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 1 対 1 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1対1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、機能的状態とトランザクション境界で定義されるキーとなるタイミング・ポイントで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、関数のパラメタと関数本体を規定する。(「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と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 タイプ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、すなわち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 attributes are partitioned into those applicable for all models, and those only applicable for approximately timed models	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のセマンティクス。アトリビュートは、全てのモデルに使えるものとアプロキシメイトリー・タイムド・モデルにのみ使えるものとに分けられる。
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、規定のタイム・クォンタム。意図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	親モジュールのソケットへ子モジュールのソケットをバインドする、あるいは子モジュールのソケットに親モジュールのソケットのバインドする事。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)。ホップはターゲット・ソケット方向のものとイニシエータ・ソケット方向のものがある。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使用可能である。
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や、いかなるレスポンスも必要としないコンポーネントからは無視される。無視されるフェーズは基本プロトコルで使用可能である。
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のライフタイムの終わりに、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、そして、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を含むクラス、及び、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。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作用するモジュール。ペイロードのルールに従って、インターコネクト・コンポーネントがトランザクション・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。
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 を継承した抽象クラス（ただし、sc_object は継承しない）。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メモリ・マップド・バスプロトコルを実装したモデルアダプター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、相互利用のための手間を削減を意味する。
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、イニシエータとターゲットのソケット、汎用ペイロード、tlm_global_quatum と tlm_phase を構成する。 密接に基本プロトコルと関係する。
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、ストレージがある場合アロケートされ、コンストラクタ呼び出しが完了した時に開始され、そしてストレージがリリースされた時、もしくは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、無効になるまでの時間の周期を指す。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為、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 インタフェースの呼び出し時のように、複数の引数スタックされる事もある。

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前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に必要とされる機能をサポートする為に、最適で十分な最小タイミング情報を与えたモデリング・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、または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、しかし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	この用語には明確な標準の技術定義は無いが、バス・トラフィックを開始する為にメモリ・マップド・バス制御を行う事の出来るモジュール、またはポート、また自律的なソフトウェア・スレッドを実行する事が出来るものや、その他システムを起動するコンポーネントを指す。一般的には、バス・マスタはイニシエータになりうる。

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じ事を表している。SystemC でメソッドとは、インタフェース・メソッド呼び出しの文脈の中で用いられる。一般的にメンバ関数は C++クラス定義（C++標準に準拠した）される時に使用され、メソッドはより略式的な文脈か、またはインタフェース・メソッド呼び出しを論じている時に使用される。（SystemC 用語）
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。
multi-socket	One of a family of convenience sockets that can be bound to multiple sockets belonging to other components. A 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	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便利ソケットのひとつ。イニシエータ・マルチ・ソケットに複数のターゲット・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・ソケットに複数のイニシエータ・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・コールをする際、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	ストレージの領域。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 inverse relationship to *child*. Module A is the *parent* of module B if module B is a *child* of module A. (SystemC term)	オポジットパス	逆方向のパス。 フォワード・パスでは逆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 A がモジュール B の親であるならば、B は A の子である。（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、各通知が関連トランザクション・オブジェクトを運ぶ。トランザクションは遅延時間をアノテートしてキューに積まれる。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の後ろから出てくる。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、便利である。

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	トランザクションが有効な期間。 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推移はタイミング・ポイントと関係する。 タイミング・ポイントは nb_transport の時間引数で与えられた合計で遅延される。
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・メソッドにおけるフェーズ引数の値の変化であり、nb_transport へのコールと nb_transport からのリターンの際 TLM_UPDATED の値を伴う。
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）	機能要求に忠実なソフトウェア・プログラマ向けの使用法であり、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を実施するハードウェアのルーズリー・タイムド・モデルを指す。
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のタイプのための typedef を含むクラスであり、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タイプ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さらに進めることが許されている処理の量。
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・クラスであり、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際のトランザクション・ライフタイムにおけるステージのひとつ。 基本的には、リクエストはコマンド、アドレス、ライト・コマンドのためのデータアレイなどの汎用ペイロード・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値参照、データアレイの場合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際の トランザクション・ライフタイムにおけるステージのひとつ。基本的には、レスポンスはステータス、リード・コマンドのためのデータアレイなどの汎用ペイロード・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値参照、データアレイの場合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	一組のインタフェース・メソッド・コールのスタックが呼ぶコントロールパスであり、フォワード・パスかバックワード・パスのどちらかに沿って巻き戻される。フォワード・パスへのリターン・パスはターゲットからイニシエータまで情報を運ぶことができ、そして、バックワード・パスへのリターン・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ができる。

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	直接制限されなければならないソケットよりむしろ必要なインタフェースを実装する別のオブジェクトへのソケット・オブジェクトに登録されるべきコールバック・メソッドを許容するので使用するのが簡単な便利ソケットのひとつ。シンプル・ターゲット・ソケットは、自動変換を 2 つの間に提供することによってブロッキングとノンブロッキングの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排除する。
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 an OSCI target.	スレーブ	この用語は、この標準の中では明確な技術的定義を持たないが、バス・マスタからのコマンドに応答する事が出来る反応的なメモリ・マップド・バス上のモジュールまたはポートを意味する際に用いられる。しかしそれ自身は、バス・トラフィックを起こす事が出来ない。一般的にスレーブは、OSCI ターゲットのようにモデルされる。
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コールする。

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・オブジェクトの終了時に（C++で自動的及び明示的に）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・オブジェクトは残される。 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	他のプロセスが走るのを待つ。あるいは、テンポラル・デカップリングを使用する際、または現在のタイム・クォンタムの終わりまで待つ。
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。加えて、各クォンタムの終わりが来るたびに同期ポイントが取られる。
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	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）を特定するためにあらゆる入って来るインタフェース・メソッド・コールに int id tag を加える便利ソケットのひとつ。

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。書き込み操作においては、データがイニシエータから 1 個以上のターゲットまでコピーされる。読み出し操作においては、データはあるターゲットからイニシエータまでコピーされる。 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を含むクラス、及び、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。 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 masters 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を進めるマスタを一つ以上許容する機能を指す。
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間。ルーズリータイムド・トランザクションには b_transport からのコールとリターンに関係する 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・タイムドのベース・プロトコル・トランザクションはフェーズ・トランジションに関する 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, write, or make themselves sensitive to attributes of the transaction. A transaction may be an atomic transaction or a complex transaction.	トランザクション	二つ以上の並列プロセスにおける、相互作用や通信を行う為の抽象化。トランザクションは時間範囲とアトリビュートのセットを伝える。アトリビュートは、特定の時間範囲内で有効である事を意味する。トランザクションにおける時間調整は、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。

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てもよい。
transaction object	"The object that stores the attributes associated with an OSCI 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	トランザクションに含まれるアトリビュートを保持するオブジェクト。
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として抽出される際の抽象度。この用語には、時間、構造、動作の抽象化についての詳細な粒度は含まない。
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・レベルによるモデルと、そのようなモデルを作成する行動を表す。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（一般的なワード･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・インタフェースの集合による接続がなされるような、異なる抽象度を扱う際に頻繁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が、一つのトランザクタには、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可能性がある。「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	すべての受け取ったインタフェース・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）の変更なく、遅延なしでコンポーネントを通して直ちに反映されるというプロパティ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 標準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リクエスト・トランザクション・オブジェクトを返す。 TLM-2.0 標準ではさらにブロッキング・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・インタフェースの２つに分割される。"

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	トランザクションのアトリビュートがトランザクション・ライフタイムの最初のタイミング・ポイントから最後までの期間で厳密に読み出しのみに限定された 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によって表される。T valid get(T& t)の場合では、この関数からのリターンによって表される。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 つ目のトランザクション・オブジェクトの終わりと 2 つ目のトランザクション・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言及しないモデリング・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。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、複数並列スレッド間のオペレーションの順序性を実現しなければならない。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、この方法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はあるが、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な状態である期間。（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にインスタンスとモジュールの間に存在する関係。あるいは、入れ子にされたモジュールの中にインスタンスがなければ、提供される。（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）譲渡する	SystemC スケジューラにコントロールを返すこと。スレッド・プロセスにおいて、yield は、call wait である。 メソッド・プロセスにおいては、yield は、関数から戻ることである。

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 環境で 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

04-2-03 TLM-2.0 LRM 差分まとめ

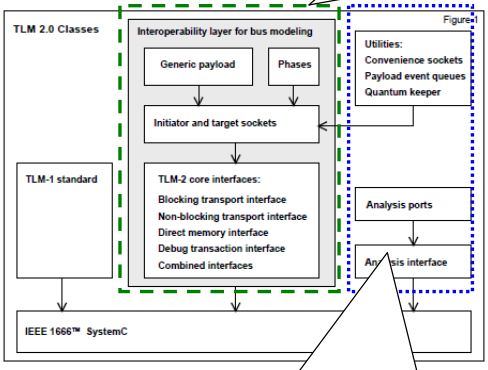
修正内容

- power pointの文書校正/スペルチェックで出てくるtypoの修正
(それ自体がLRMの修正ポイントになっている部分は除く)
- 各ページのタイトルの統一(章番号＋章の見出し)
- フォントの統一
- 小さなフォントは、可能な限り大きめに変更(Glossaryで複数
項目が1ページになっているもので、フォントを大きくして1枚に
収まらないものは、ページを分けたものがあります)
- フィードバックは最後のページにまとめました。
- 用語集で、約の”・”の打ち方等、本文と異なるところは合わせました(本文で表記ゆれチェックの結果統一したものも含む)。

1 Overview

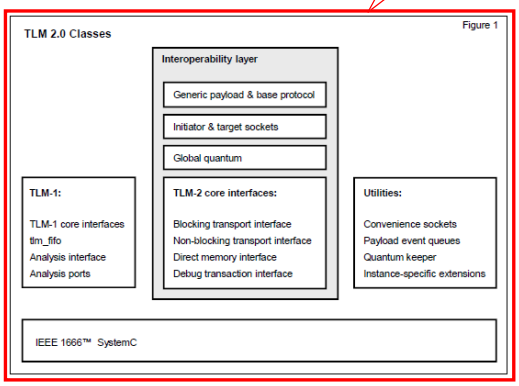
TLM-2クラス

相互利用性を最大にするため、このレイヤーのクラスを使用することを推薦



ユーティリティ、解析ポート/インタフェースの使用は厳密には必要ではないが、スタイルの一貫性を保つためTLM-2.0標準に入れる。

図変更



- バスモデル間の相互利用性のためユーティリティを使う必要はない。
- スタイルの一貫性を保つためTLM-2.0標準の一部とする。
- 解析ポート/インタフェースについての
- 言及が無くなった。

JEITA

1 Overview

変更なし。

- 汎用ペイロード
  - メモリ・マップ・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）
  - アトリビュート、フェーズは拡張可能（特定なプロトコルをモデリングするため）
    - 相互利用性を阻害するので使用には注意が必要
- コーディング・スタイル
  - TLM-2標準の範囲外（示唆、提案のみ）
  - モデルとスタイルの関係
    - ルーズリー・タイムド・モデル ⇒ ブロッキング・トランスポート・インタフェース、DMI、テンポラル・デカップリング
    - アプロキシメイトリー・タイムド・モデル ⇒ ノンブロッキング・トランスポート・インタフェース、ペイロード・イベント・キュー

JEITA

1.1 Scope

- このドキュメントの焦点
  - TLM-2コア・インタフェースとクラスのキー・コンセプトとセマンティクス。

TLM-2.0クラス（ユーティリティ含む）のコンセプトとセマンティクスにフォーカス。

- TLM-1コア・インタフェースをリストアップするがそのセマンティクスを定義はしない。

変更なし。

- 言語リファレンス・マニュアルではない。

まず最初に、このドキュメントは「TLM-2.0標準のリファレンスマニュアル」であると明記。

- TLM-2.0を如何に使うかのより実践的なガイドラインを追加していくつもりである。

この文なし。

1.2 Source code and documentation

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_req_rsp	:TLM-1標準
include/tlm/tlm_h/tlm_req_rsp/tlm_1_interfaces	:TLM-1コア・インタフェース
include/tlm/tlm_h/tlm_req_rsp/tlm_channels	:TLM-1 fifo、req-rspチャネル
include/tlm/tlm_h/tlm_req_rsp/tlm_ports	:TLM-1ノンブロッキング・ポート、イベント・ファインダー
include/tlm/tlm_h/tlm_req_rsp/tlm_adapters	:TLM-1スレーブ-トランスポート、トランスポート-マスタ・アダプタ
include/tlm/tlm_h/tlm_trans	:TLM-2相互利用性クラス
include/tlm/tlm_h/tlm_trans/tlm_2_interfaces	:TLM-2コア・インタフェース
include/tlm/tlm_h/tlm_trans/tlm_generic_payload	:TLM-2汎用ペイロード
include/tlm/tlm_h/tlm_trans/tlm_sockets	:TLM-2ソケット
include/tlm/tlm_h/tlm_quantum	:TLM-2グローバル・クオンタム
include/tlm/tlm_h/tlm_analysis	:TLM-2解析インタフェース、ポート
include/tlm/tlm_utils	:TLM-2ユーティリティ・クラス
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	:コード例
unit_test	:リグレッション・テスト

1.2 Source code and documentation

以下のように変更。

TLM-2.0リリースのディレクトリ構造

include/tlm	: TLM-2.0標準のソースコード、readmeファイル、リリースノート
include/tlm/tlm_h	: TLM-2.0相互利用レイヤー
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・クオンタム
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、req-rspチャネル
include/tlm/tlm_1/tlm_req_rsp/tlm_ports	: TLM-1ノンブロッキング・ポート（イベント・ファインダーを持つ）
include/tlm/tlm_1/tlm_req_rsp/tlm_adapters	: TLM-1 slave-to-transport、transport-to-master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 （相互運用性には本質的ではない）
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	: アプリケーション指向の例題（ドキュメント含む）
unit_test	: リグレッション・テスト

JEITA

2 References

変更なし。

- 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

JEITA

2.1 Bibliography

変更なし。

- 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 Introduction
3.1 Background

変更なし。

- TLM-1標準は、メモリ・マップ・バス、オンチップ・コミュニケーション・ネットワークをモデリングする上で3つの欠点がある

欠点	TLM2での解決
トランザクション・クラスの標準が無い。	汎用ペイロード
タイミング・アノテーションをサポートしない。 （モデル間でタイミング情報を交換する標準的方法がない。wait文で遅延を実装する。）	ブロッキング/ノンブロッキング・トランスポート・インタフェースでのタイミング・アノテーション
トランザクションを値もしくは参照渡しする。	トランザクション・オブジェクトが幾つかのトランスポート・コールに渡る仕組み（新しいトランスポート・インタフェースがサポート）

3.2 Transaction-level modeling, use cases and abstraction

変更なし。

- 分類法を変更
  - ユースケースに応じた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アプローチをとる。
  - 様々なユースケースに適するコーディング・スタイルを説明する。
- TLM-2インタフェースは標準の一部で、相互利用性を保障する。
- 各コーディング・スタイルは様々な抽象度をサポートできる。

JEITA

3.2 Transaction-level modeling, use cases and abstraction

変更なし。

- アンタイムド機能モデル（アルゴリズム・モデル）
  - 唯一つのSystemCプロセスからなる。
  - トランザクション・レベルではない（プロセス間の通信がないから）。
- トランザクション・レベル
  - 複数のプロセスから成る。
  - 他のプロセスへ制御を譲渡する仕組み（wait文）。
  - 強い同期と弱い同期
    - 強い同期：一連のコミュニケーションがあらかじめ正確に決まっている。
      - FIFO、セマフォで実現可能。これは完全なアンタイムド・モデリング・スタイルなので、TLM-2.0の範囲外。
    - 弱い同期：一連のコミュニケーションが個別のプロセスの詳細タイミングで部分的に決まる。

JEITA

3.2 Transaction-level modeling, use cases and abstraction

変更なし。

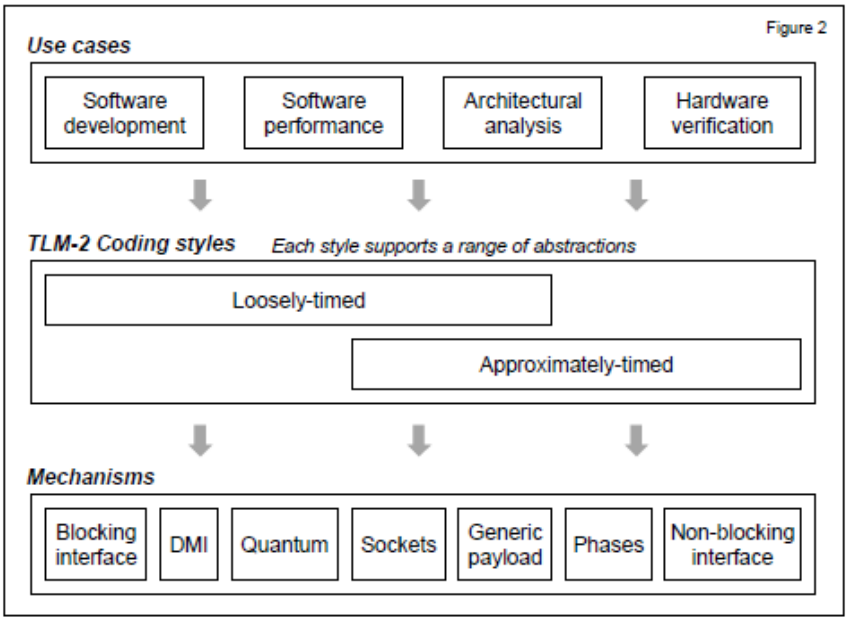
- ルーズリー・タイムド・コーディング・スタイル
  - 複数の組込みソフトウェア・スレッドの並列実行を許すヴァーチャル・プラットフォーム・モデルは、強い、もしくは、弱い同期を使う。このモデルに適したスタイル。
- アプロキシメイトリー・タイムド・コーディング・スタイル
  - プロトコル固有の複数のタイミング・ポイント（プロトコルの各フェーズの開始・終了のタイミング等）をトランザクションに関連付けるためには、より詳細なトランザクション・レベル・モデルが必要なケースがある。適切なタイミング・ポイントを選ぶことにより、コンポーネント・モデルをクロック精度で動作させることなくタイミング精度の高いコミュニケーションをモデリング可能。このようなコーディング・スタイル。

 JEITA

3.2 Transaction-level modeling, use cases and abstraction

変更なし。

Figure 2



 JEITA

3.3 Coding styles

- 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
  - アプロキシメイトリー・タイムド

  についてのみ詳しく述べる。
- コーディング・スタイルは正確に定義されるわけではない。
- TLM-2コア・インタフェースを律するルールはコーディング・スタイルとは別に定義される。

TLM-2.0

3.3.1 Untimed coding style

変更なし。

- アンタイムド・コーディング・スタイルは規定しない。
  - 現代のバス・ベース・システムは組込みプロセッサ上で動作するソフトウェアをモデリングするため、何らかのタイミングの概念が要求される。
- 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。

3.3.2 Loosely-timed coding style and temporal decoupling

- **ブロッキング・トランスポート・インタフェース**を使用。
  - 基本プロトコルでは、2つのタイミング・ポイント（トランスポート関数コールと戻り）
    - リクエスト開始とレスポンス開始にゆるく対応。
    - 同じシミュレーション時刻でも、異なる時刻でも良い。
- ユース・ケース：SW開発（マルチ・プロセッサのヴァーチャル・プラットフォーム・モデル）
  - タイマーと割込みのモデリングをサポート（OSブートとコード実行に十分）。
- 特徴：**テンポラル・デカップリング**

「ゆるく」がなくなった。

3.3.2 Loosely-timed coding style and temporal decoupling

変更なし。

- テンポラル・デカップリングをサポート
  - モデルは他モデルとの同期ポイントに到達するまで自分のローカル時間を先行して実行可能。
    - 各プロセッサは次のプロセッサが実行される前に、ある時間（**クォンタム**）だけ実行

⇒ **高速なシミュレーション**

  - スケジューリングのオーバーヘッド低減

3.3.2 Loosely-timed coding style and temporal decoupling

変更なし。

- SystemCスケジューラからの考察
  - イベントが発生する時刻まで時間を進め、その時刻で動作すべきプロセスを実行。
  - プロセスの実行が終了すると、シミュレーション・カーネルに制御が返る。
  - モデルが詳細だと、イベント・スケジューリングとプロセスのコンテキスト・スイッチのオーバーヘッドがシミュレーション速度の支配的要因となる。

  ⇒ **シミュレーション高速化の一つの方法**：現在時刻に先行してプロセスを実行すること。
    - **テンポラル・デカップリング**

3.3.2 Loosely-timed coding style and temporal decoupling

変更なし。

- テンポラル・デカップリングの実装
  - プロセスは、以下の2つの外部依存のケースに出会うまで時刻を先行し実行可能。このとき、現状の値を受け入れて実行継続か、シミュレーション・カーネルに制御を返す。
    - 他プロセスにより値を書きかえられる変数に依存
    - 他プロセスと情報交換
  - 2つの対処法
    - 同期を強制（シミュレーション時間に追いつくまで他のプロセスに制御を譲渡 = 標準のSystemCシミュレーション・セマンティクス）。
    - 現在の値を受入れ、実行継続（値の早期サンプリングがダメージを与えない、以降の値変化は以降のプロセス実行中で取込まれると仮定。ソフトウェア・スタックがハードウェアの詳細タイミングに依存しないヴァーチャル・プラットフォーム・モデルではこの仮定が成り立つ）。

3.3.2 Loosely-timed coding style and temporal decoupling

変更なし。

- クォンタム
  - 制限なしにシミュレーション時刻を先行してプロセスが実行 ⇒ SystemCスケジュール動作しない ⇒ 他プロセスは決して動作しない。

  ⇒ **グローバル・クォンタム**で制限。
    - 先行実行の上限を設定。
    - アプリケーションが設定。
    - シミュレーション速度と精度のトレードオフ
      - 小さすぎる値 ⇒ 低速。制御の譲渡と同期が頻繁。
      - 大きすぎる値 ⇒ システム機能停止。タイミングの一貫性が保てない。

JEITA

3.3.2 Loosely-timed coding style and temporal decoupling

- 例：プロセッサ、メモリ、タイマー、低速なペリフェラルから構成されるシステム。
  - ソフトウェアはその実行時間の大半をメモリからの命令のフェッチと実行に費やし、システムの残りとはタイマーからの割込み（例えば、1 ms毎）が発生したときのみ情報交換。
  - ISSモデルは1 msのクオンタムまで先行実行可能（メモリ・モデルには直接アクセスし、タイマー割込みでペリフェラルとのみ同期）。この間は他のハードウェア・モデルのクロックにロックされる必要なし。
  - 1000倍のスピード・アップが可能。

DMIを使って10倍か100倍

JEITA

3.3.2 Loosely-timed coding style and temporal decoupling

- テンポラル・デカップリングしたモデル、しないモデルが混在する場合、テンポラル・デカップリングしないモデルがシミュレーション・スピードのボトルネックになる可能性が高い。
- TLM-2でのサポート

TLM-2.0

  - テンポラル・デカップリング
    - ↑ tlm_global_quantumクラス、ブロッキング/ノンブロッキング・トランスポートのタイミング・アノテーション
  - グローバル・クォンタム
    - ↑ tlm_quantumkeeperユーティリティ・クラスによるアクセス

 JEITA

3.3.3 Synchronization in loosely-timed models

変更なし。

- アンタイムド・モデル
  - 明示的同期ポイント(wait文、ブロッキング・メソッドのコール)
- ルーズリー・タイムド・モデル
  - 明示的同期 ⇒ 予測可能な実行を保証
  - イニシエータはクォンタムが終了するまで先行実行可能(明示的同期ポイントなし)
  - 要求ベース同期(クォンタム終了前にスケジューラへ制御を譲渡) ⇒ タイミング精度向上
- タイム・クォンタム・メカニズム
  - 正確なシステム同期を保証するためのものではない。
  - スケジューラの仕組みに基づいたシミュレーション環境で、複数イニシエータが動作する仕組みを提供。
  - システム・レベルでの明示的同期スキームを設計するための仕組みではない。

 JEITA

3.3.4 Approximately-timed coding sty

変更なし。

- 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**
  - タイミング・アノテーション
  - 複数フェーズ
  - トランザクションのライフタイム中のタイミング・ポイント
- ユースケース：**アーキテクチャ探索、性能解析**
- トランザクションのフェーズ
  - **基本プロトコルでは、4つのタイミング・ポイント**
    - リクエストの開始と終了、レスポンスの開始と終了
  - 特定プロトコルでは、タイミング・ポイントの追加が必要なケースあり ⇒ 汎用ペイロードとの相互利用性が失われる。
  - 2つのタイミング・ポイント（トランザクションの開始と終了）も可能だが、ルーズリー・タイムド・モデルでは、ブロッキング・トランスポート・インタフェースの使用が望ましい。

3.3.4 Approximately-timed coding style

- アプロキシメイトリー・タイムド・モデルの特徴
  - テンポラル・デカップリングを使わない（要求されるタイミング精度のため）。
  - プロセスの情報交換に2種類の**ディレイをアノテート**。
    - ターゲットのレイテンシ。
    - ターゲットの開始間隔（アクセプト・ディレイ）
    - wait(delay)もしくはnotify(delay)で実装。

・データ転送時間
・ターゲットのレイテンシ
・基本プロトコルでは、データ転送時間は2つの連続したリクエストもしくは2つの連続したリクエスト間のインターバル/アクセプト・ディレイと同じ。
・wait(delay)もしくはnotify(delay)で実装。

3.3.5 Characterization of loosely-timed and approximately-timed coding style

変更なし。

- 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付けられる。

	タイミング・ポイント	テンポラル・デカップリング（タイミング制御）
ルーズリー・タイムド	2つ （トランザクション開始と終了）	使用。 プロセスは先行実行した時間情報を持つ。 明示的同期ポイント、クォンタムを使いきったら制御を譲渡。
アプロキシメイトリー・タイムド	複数 （基本プロトコルでは、4つ）	SystemCスケジューリング・セマンティクスに従う。 プロセス間の情報交換にアノテートされたディレイは、wait(delay)もしくはnotify(delay)で実装。
アンタイムド	なし	明示的に予め決定されている同期ポイントで制御を譲渡。

3.3.6 Switching between loosely-timed and approximately-timed modeling

変更なし。

- シミュレーション中にモデルのルーズリー・タイムド・コーディング・スタイルとアプロキシメイトリー・タイムド・コーディング・スタイル間のスイッチが可能。
- 利用例
  - ルーズリー・タイムド・レベルでリセットとブートを高速に実行し、興味のあるステージに到達したら、より詳細な解析をするためアプロキシメイトリー・タイムド・モデリングへスイッチする。

3.3.7 Cycle-accurate modeling

TLM-2.0

- 現時点ではTLM-2の範囲外
  - TLM-1でモデリング可能。
  - 標準化は未解決の課題（将来のOSCI標準ではカバーされるかもしれない）
- 原理的には、アプロキシメイトリー・タイムド・コーディング・スタイルに含まれるように拡張されかもしれない。
  - 適切なフェーズとルールの定義により。
  - TLM-2.0リリースはこれに十分な機構が含まれるが、詳細はまだうまくいっていない。

3.3.8 Blocking versus non-blocking transport interfaces

変更なし。

- 異なるタイミング・レベルをサポートする別個のインタフェース

ブロッキング	• トランザクションの開始と終了をモデリング可能。 • 1回の関数コールでトランザクションを完了。
ノンブロッキング	• トランザクションを複数のタイミング・ポイントへ分割。 • 1回のトランザクションを完了するのに複数の関数コールが必要。

3.3.8 Blocking versus non-blocking transport interfaces

- 相互利用性：1つのインタフェースにまとまっている。
  - コーディング・スタイルに従って、ブロッキング、ノンブロッキングのいづれか、もしくは、両方を使用。
  - TLM-2トランスポート・インタフェースを提供するモデルは、相互利用性を最大にするため、**ブロッキングとノンブロッキングの両方を提供**すること（実装はかならずしも必要ない）。

TLM-2.0

JEITA

3.3.8 Blocking versus non-blocking transport interfaces

変更なし。

- 便利ソケット
  - ブロッキング → ノンブロッキング、ノンブロッキング → ブロッキングのトランスポート・コール変換を提供。
  - 変換にコストがかかる（特に、ノンブロッキング → ブロッキング変換）
    - アプロキシメイトリー・タイムド・モデルがシミュレーション時間を支配しているので、このコスト・オーバーヘッドはおそらく緩和される。

JEITA

3.3.8 Blocking versus non-blocking transport interfaces

変更なし。

- トランスポート・コールの混在と順序のルール
  - イニシエータは動的にブロッキング、ノンブロッキングのどちらをコールするか選べる。
  - **本標準には、同じターゲットへのブロッキング/ノンブロッキング・トランスポート・コールの混在と順序のルールが含まれる。**

3.3.8 Blocking versus non-blocking transport interfaces

変更なし。

- 各トランスポート・インタフェースの強み

ブロッキング	・コーディング・スタイルが単純（1回の関数コールでトランザクションが完了）。 ・テンポラル・デカップリングを有効に利用可能。
ノンブロッキング	・複数タイミング・ポイントのサポート。 ・（テンポラル・デカップリングも可能だが、複数タイミング・ポイントのサポートがテンポラル・デカップリングの利点を無効化。）

3.3.9 Use cases and coding styles

変更なし。

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・パフォーマンス解析	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 Initiators, targets, sockets, and bridges

- コンポーネントの種類

イニシエータ	トランザクションを開始 ・トランザクション・オブジェクトを生成。 ・コア・インタフェースの1つのメソッドをコールしてトランザクション・オブジェクトを渡す。
ターゲット	トランザクションの最終目的地 ・ライト・トランザクションでは、イニシエータ（プロセッサ）がターゲット（メモリ）にデータを書き込む。 ・リード・トランザクションでは、イニシエータがターゲットからデータを読む。
インターコネクト・コンポーネント	イニシエータもしくはターゲットとして振舞う ・例：アービタ、ラウター

イニシエータ、ターゲットとして振舞わない

追加：
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れる。
例えば、あるトランザクションではインターコネクトの役割のコンポーネントがあるトランザクションではターゲットの役割をする。

3.4 Initiators, targets, sockets, and bridges

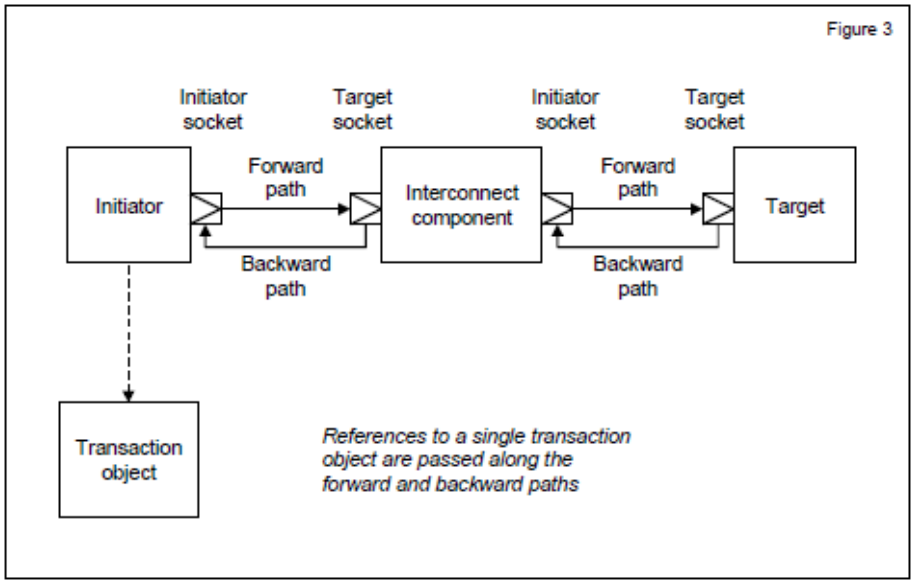
変更なし。

- トランザクションのライフタイム
  - ① イニシエータがトランザクション・オブジェクトを生成。
  - ② **フォワード・パス**
    - アービタのようなインターコネクト・コンポーネントが実装するトランスポート・インタフェース・メソッドに渡される。
    - インターコネクト・コンポーネントで、さらにトランスポート・メソッド（ラウターのような2番目のインターコネクト・コンポーネントで実装）に渡される。
    - 2番目のインターコネクト・コンポーネントで、さらにトランスポート・メソッド（メモリにようなターゲットで実装）に渡される。
  - ③ ターゲットでトランザクションが処理される。
  - ④ イニシエータへトランザクション・オブジェクトを返す（以下の2種類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り決まる）。
    （1）メソッド・コールの返り値 ＝ リターン・パス
    （2）逆方向のメソッド・コール ＝ **バックワード・パス**

3.4 Initiators, targets, sockets, and bridges

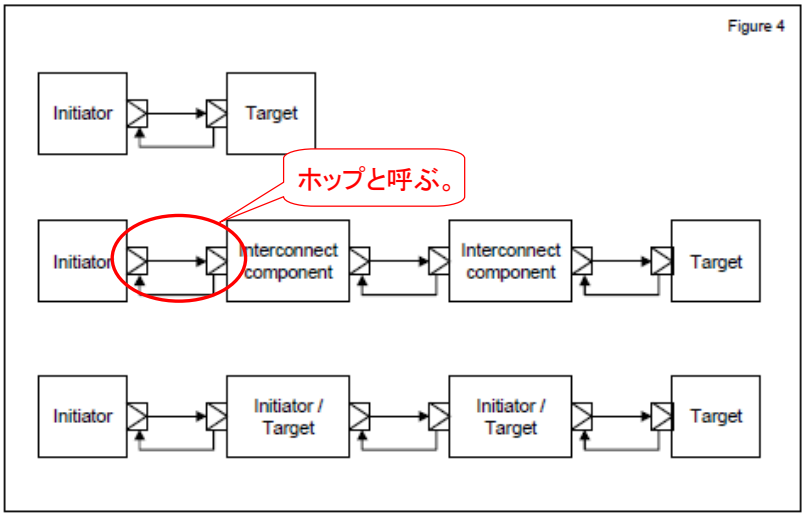
変更なし。

- フォワード・パスとバックワード・パス



3.4 Initiators, targets, sockets, and bridges

- 汎用ペイロードを使用するとき、フォワード・パスとバックワード・パスはコンポーネントとソケットの同じ組（もしくは逆向き）を通る。



JEITA

3.4 Initiators, targets, sockets, and bridges

- ソケット
  - フォワード・パスとバックワード・パスをサポートするため、ポートとエクスポートが要求される。

ソケット	インタフェース・メソッド・コール	
	フォワード・パス	バックワード・パス
イニシエータ	ポート	エクスポート
ターゲット	エクスポート	ポート

  - DMIとデバッグ・トランスポート・インタフェースも持つ。
  - イニシエータは、少なくとも1つのイニシエータ・ソケットを持つ。
  - ターゲットは、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・コンポーネントは少なくとも1つずつ持つ。
  - 異なるトランザクション・タイプを転送するコンポーネントはいくつかのソケットを持つ。
    - イニシエータとターゲットの役割。
    - TLM-2トランザクション間のブリッジ。

この記述がなくなった。

JEITA

3.4 Initiators, targets, sockets, and bridges

- バス・ブリッジのモデリング方法
  - （1）インターコネクト・コンポーネント
    - 1つのトランザクション・オブジェクトのポインタを渡す。⇒ シミュレーション高速化に適する。
  - （2）TLM-2トランザクションのブリッジ

TLM-2.0

    - トランザクションをコピー。
    - 別のアトリビュートを付加できる（より柔軟）。

3.4 Initiators, targets, sockets, and bridges

- TLM-2ソケットの使用を推薦

TLM-2.0

  - 相互利用性を最大化
  - 便利
  - 一貫性を持つコーディング・スタイル
  - ポートとエクスポートの使用は非推薦

3.5 DMI and debug transport interfaces

- ターゲットの持つメモリへの直接アクセス、デバッグ・アクセスを提供するインタフェース。
- インターコネクト・コンポーネントを通る通常のパスをバイパス。

追加：DMIリクエストが許可されたら、

インタフェース	目的	提供するパス
DMI	ルーズリー・タイムド・シミュレーションでの通常のメモリ・トランザクションの高速化。	・フォワード・パス（イニシエータ→ターゲット） ・バックワード・パス（ターゲット→イニシエータ）
デバッグ・トランスポート・インタフェース	通常のトランザクションに付随するディレイ、副作用を引き起こさないデバッグ・アクセスを提供。	・フォワード・パス

3.6 Combined interfaces and sockets

ソケット	提供するインタフェース
イニシエータ ターゲット	・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 ・DMI ・デバッグ・トランスポート・インタフェース

- 4つのインタフェースを1つのターゲットへのアクセスに並行に使用可能
- 4つのインタフェースと汎用ペイロード
  ⇒ TLM-2標準を使ったモデルの相互利用性を保証

TLM-2.0

ターゲット・ソケット
- 4つのインタフェースを提供 ⇒ 事実上、すべてを実装
  - ブロッキング⇄ノンブロッキング変換の用意があれば、両方実装する必要なし。
  - モデルのスピードと精度の要求に応じてどちらかを実装してもよい。
- イニシエータは、スピードと精度の要求に応じて、コア・インタフェースのどのメソッドをコールしてもよい。

3.6 Combined interfaces and soc

変更なし。

- コーディング・スタイルは適切なインタフェースの選択のガイドとなる
- 典型例

コーディング・スタイル（イニシエータ）	インタフェース
ルーズリー・タイムド	・ブロッキング・トランスポート ・DMI ・デバッグ
アプロキシメイトリー・タイムド	・ノンブロッキング・トランスポート ・デバッグ

3.7 Namespaces

namespace	クラス
tlm	メモリ・マップド・バスのモデリング用の相互利用インタフェースのクラス
tlm_utils	ユーティリティ・クラス （相互利用性には必ずしも必要ではないが、TLM-2標準の一部）

TLM-2.0

3.8 Header files and version numbers

- アプリケーションは、以下のヘッダー・ファイルをインクルードすること。
  - include/tlm下のtlm.h
  - include/tlm/tlm_utils下のヘッダー・ファイル（ユーティリティ・クラスを使用する場合）
- 最新版のOSCIシミュレータを使用する場合は、をSystemCヘッダー・ファイルをインクルードする前に、SC_INCLUDE_DYNAMIC_PROCESSESマクロを定義すること。
- include/tlm/tlm_h/tlm_version.hはマクロとOSCI TLM-2ソース・コードのバージョン・ナンバーの定数を含んでいる。アプリケーションはこれらを使ってもよい。
  - tlm_releaseメソッドは、sc_core::sc_releaseメソッドが返すのと同じフォーマットの文字列を返す。

TLM-2.0

この文なくなった。

JEITA

3.8.1.1 Definitions（あらたに追加）

```
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
```


JEITA

3.8.1.2 Rules（あらたに追加）

a) 実装定義の数は[0-9]からなる十進数。quotation markで囲わない。
b) originatorとpre-releaseストリングはquotation markで囲われた[A-Z][a-z][0-9]_から成る文字列。
c) version release dateはISO 8601 basicフォーマットの日付（YYYYMMDD、各値は十進数）で、quotation marksで囲われている。
d) IS_PRERELEASEフラグはFALSEまたはTRUE。quotation marksで囲わない。
e) version stringは、”major.minor.patch_prerelease-originator”または”major.minor.patch-originator”。Major、minor、patch、prerelease、originatorはそれに対応したストリングの値（quotation markで囲わない）。prereleaseストリングはがあるかないかはIS_PRERELEASEフラグの値に依存する。
f) copyright stringはcopyright noticeを設定する。
g) 各定数は、対応したマクロで定義された値を適切はデータ型に変換された値で初期化される。
h) tlm_releaseメソッドとtlm_versionメソッドはC stringへ変換された値を返す。
i) tlm_copyrightメソッドはcopyright stringの値をC stringへ変換されたものを返す。

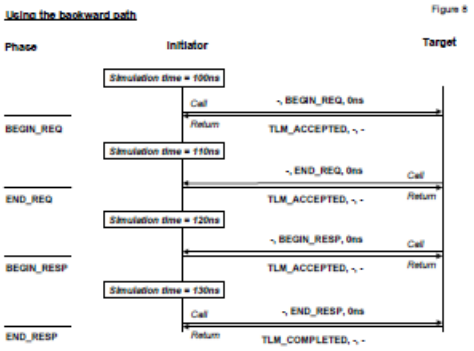
 JEITA

4. TLM-2.0 Core Interfaces：差分サマリ

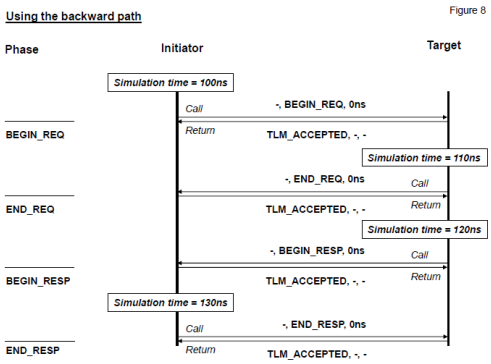
- 章の構成は変更無し
  - 他章の構成変更の関係上、本文中の参照章が一部変更
  - その他、説明文の追加、文言の変更、Typo修正あり
- 用語の変更
  - TLM-2をTLM-2.0に変更
  - ホップ（hop）という単語を使用
    - 以前「caller/callee connections」等の箇所に適用
  - 実行ローカルタイム（effective local time）という単語を使用
- 図の変更
  - タイミングダイアグラムで消費時間を表す箱が一部Target側へ移動し、見やすくなった。

 JEITA

4. TLM-2.0 Core Interfaces：図の変更例



JA22



JA32

JEITA

4. ～ 4.1「トランスポート・インタフェース」 4.1.1「ブロッキング・トランスポート・インタフェース」

- 4. ～ 4.1「トランスポート・インタフェース」
  - 変更点は、細かい説明文の変更レベルで、大きな変更無し
- 4.1.1「ブロッキング・トランスポート・インタフェース」
  - 4.1.1.4「規則」の追加
    - h) 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通過するはずである。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・ソケットの為の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かもしれない。
    - j) b_transportの実装はnb_transport_bwを呼び出してはならない。

JEITA

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・トランスポート・メソッドの呼び出しと戻りは、フェーズ・トランザクションと一致する旨の文章を追加
- 4.1.2.5「trans引数」
  - 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。
- 4.1.2.6「phase引数」
  - b)の削除
  - 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・トランザク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。

 JEITA

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」

- 4.1.2.7「tlm_sync_enumの戻り値」
  - 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。
- 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」
  - タイミングチャート図の説明文を追加。
- 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」
  - 説明文が詳細化された。
    - (追加)「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、ペイロード・イベント・キューに登録されたコールバックのどちらかによって起こる。」
    - (追加)「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。」

 JEITA

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」

- （追加）「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。」

- 4.1.3.1「sc_time引数」
  - 時間引数の減少が定義された。
    - 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。

 JEITA

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」

- 4.1.3.1「sc_time引数」
  - ルールとして詳細な記載が追加された。
    - 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入って来るインタフェース・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、入って来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入って来る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。)

 JEITA

コード例の追加

```
// ----------------------------------------------------
// 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()
  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_tim
{
  // 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」

- 変更なし

4.2.1 ダイレクト・メモリ・インタフェース

全体的にメソッドの仕様で不明確・わかりにくい部分を追加説明
初期化について文章中でも説明追加
DMI/トランスポート比較の節とDMIとテンポラル・デカップリングの節が追加

- protocol typesからprotocol traitsへ名称変更
- 無視可能拡張がnon-ignorable extensionから non-ignorable or mandatory extensionに説明文が変更
  - 前回OSCIフィードバックが反映されている

4.2.2 クラス定義

変更なし

4.2.3 get_direct_mem_ptrメソッド

i),j),m),n),o)の5つが追加
その他詳細な以下の説明が追加

- g) ターゲットがDMIアクセスを許可した場合は，DMI記述子は許可されたアクセスの詳細を示すために利用
- h) ターゲットがDMIアクセスを許可しない場合は，DMI記述子は許可されないアクセスの詳細を示すために利用
- 以下i),j)の条件追加
- i)ターゲットは、メモリ領域の全領域や一部の領域に対するDMIアクセスを許可しても許可しなくてもよい
- j)ターゲットがDMIを許可して、trueを返した場合でも、インターコネクト・コンポーネントがfalseを返してもよい
  しかしターゲットがDMIを許可していない場合、インターコネクト・コンポーネントも許可してはいけない
- m) DMIアクセスによる書き込みによって、ターゲットの動作に悪影響を及ぼす場合は書き込みをしてはいけない。
  悪影響を及ぼすのは書き込みで、読み込みは問題ない
- n) get_direct_mem_ptr関数の実装はinvalidate_direct_mem_ptrを呼び出してもよい
- o) get_direct_mem_ptr関数の実装ではwaitを直接的・間接的問わず呼び出してはいけない

 JEITA

4.2.4 テンプレート引数と tlm_generic_payloadクラス

TLM_IGNORE_COMMANDの説明が追加

- f)の説明をわかりやすく説明
- g)追加
  - 基本プロトコルでは、コマンド属性がTLM_IGNORE_COMMANDの値を持つことを禁止。ただしこの値はその他のプロトコルで使ってもよい。

 JEITA

4.2.4 テンプレート引数と tlm_generic_payloadクラス

初期化に対する説明が追加

- a) イニシエータがDMI拡張子を再利用してもよいとの記述が削除
- B) DMI関連の属性(DMIポインタなど)の初期値の定義が文章で記載された
- c) 追加：initメソッドがDMI記述子のメンバーをデフォルト値で初期化する
- d) 追加：DMI記述子がget_direct_mem_ptrへの引数の時はいつもデフォルト状態. デフォルト状態にリセットする場合はinit関数を利用する
- h) organisation → organization （英国綴り→米国綴り）
- j) get_granted_accessの初期値の定義(DMI_ACCESS_NONE)が追加
- m) ターゲットが許可されているアクセスの属性だけではなく、拒否されているアクセスについても設定する必要があることが追加
- n) 追加：DMI領域の読み込み・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_ACCESS_READ_WRITEに許可アクセスを設定するべきという記載が追加
- r) ターゲットがすべてのメモリ領域にういてアクセスを拒否したい場合は、スタート・アドレスを0. エンド・アドレスをsc_dt uint64の最大値に設定するという具体例が追加
- w) 文法間違い get_direct_mem_ptr return → get_direct_mem_ptr returns
- cc)読み込みと書き込みレイテンシ属性が単にメモリ・トランザクションのレイテンシではなく、1バイトごとの平均レイテンシに記述変更。具体例を追加

4.2.5 tlm_dmiクラス

変更なし

4.2.6 invalidate_direct_mem_ptrメソッド

小変更，下記2点

- j)追加
  - すべてのTLM-2.0コア・インターフェースの実装では、invalidate_direct_mem_ptr関数を呼び出す。
- l)追加
  - invalidate_direct_mem_ptr関数の実装は、wait関数を直接または間接的に呼び出してはいけない。

 JEITA

4.2.7 DMI対transport （追加された節）

この節まるまる追加.
DMIとトランスポート・インタフェースの違いの説明

- a) 定義では、ダイレクト・メモリ・インターフェースによってインターコネクトをバイパスして、イニシエータとターゲット間を直接インターフェースことができる。
一方トランスポート・インターフェースはインターコネクト・コンポーネントをバイパスすることができない

- b) インターコネクト・コンポーネントが状態を保持、またバッファ化されたインターコネクトやキャッシュメモリをモデル化したインターコネクトのような副作用を持つときでも正しい動作を行うように注意する必要がある。
ダイレクト・メモリ・インターフェースがインターコネクト・コンポーネントをバイパスする一方で、トランスポート・インターフェースはインターコネクト・コンポーネントにアクセスして、状態を更新するかもしれない。最も安全な方法は、インターコネクト・コンポーネントがDMIアクセスをいつも拒否することである。

- c) イニシエータはトランスポート・インターフェースとダイレクト・メモリ・ポインタを交互に切り替えることが可能またーつのイニシエータがDMIを使い、もう一方のイニシエータがトランスポートインターフェースを使うことも可能。トランスポート呼び出しはタイミング・アノテーションを行うので、正しく動作させるために特に注意する必要がある。
これはアプリケーション側の責任である。例えばあるターゲットDMIとトランスポートを同時にサポートすることができるし、またトランスポートを呼ぶ度にすべてのDMIポインタを無効化させることも可能

 JEITA

4.2.8 DMIとテンポラル・デカップリング（追加された節）

この節まるまる追加.
テンポラル・デカップリングを利用する際にDMIで注意すべきことの説明

- a) ターゲットかインターコネクト・コンポーネントがinvalidate_direct_mem_ptrを呼ぶ出すことのみでDMI領域を無効化できる
- b)DMIポインタを使う前にそのDMI領域が有効化をイニシエータはチェックする必要がある。
- c) SystemCのコルーチン・セマンティクスでは、一度イニシエータが実行開始されると、他のSystemCプロセスはそのイニシエーターがなくなるまで実行できない。特にSystemCプロセスがDMIPポインタを向こうかすることはできない。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DMI領域が有効かを確認する必要は必ずしもない。
- d) イニシエータから呼び出されたインターフェース・メソッドによって、他のコンポーネントがinvalidate_direct_mem_ptrを呼び出すことになるかもしれない。
- e) イニシエータがその他のインターコネクト・コンポーネントに影響を及ぼさないで実行されている間はすべての有効なDMI領域はずっと有効のままである。
- f) DMIを利用するテンポラル・デカップリングされたイニシエータの後に、他のテンポラル・デカップリグされたイニシエータが同じDMI領域を現状のクオンタム（デカップリング単位）内であれば、無効化することができる。
  これはテンポラル・デカップリングが本質的に内在する不正確性を表している。

 JEITA

4.2.9 DMI hintを利用した最適化（旧4.2.7）

内容的な変更はあまりなし

- **フィードバック反映：DMI Hint → DMI hintに統一**
  - **大文字と小文字の混在のフィードバックを反映済み**
- **文中のDMI hintがすべて DMI allowed attributeに変更されている**

 JEITA

4.3.1 デバッグ・トランスポート・インターフェース

全体的にtlm_phaseが追加された
それ以外は用語変更・説明追加で大きな変更なし

- 全体的に
  - protocol typeからprotocol traitsに用語変更
  - non-ignorableとmandatoryを両方併記
  - tlm_phaseの利用
- 異なるプロトコルを接続するものとしてブリッジの他にアダプタという言葉が追加された

 JEITA

4.3.2 クラス定義

変更なし

 JEITA

4.3.3 TRANSテンプレート引数と tlm_generic_payloadクラス

変更なし

4.3.4 ルール

e)とf)のルール2個追加，その他小変更

- d) 文章を分かりやすく修正と、TLM_IGNORE_COMMANDが追加された（内容は変更なし）
- e) 追加
  TLM_IGNORE_COMMANDと同等のコマンド属性をもつトランザクションを受け取った時に、ターゲットは読み込みまたは書き込みの実行を中止すべきだが、拡張されたデバッグ・トランザクションを実行するときに拡張を含む汎用ペイロードの属性の値を利用するかもしれない。
- f) 追加
  トランスポート・インターフェースの場合のように、無視可能な汎用ペイロードの拡張（デバッグ・トランスポート・インターフェース）を使う場合は、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
- k) データ長の属性は、インターコネクトやターゲットが変更できないことが追加された
- l) データポインタは、インターコネクトやターゲットが変更できないことが追加されたデータ長の属性が0の時、データポインタ属性はNULLポインタであり配列は確保されないことが追加された
- m) transport_dbgの実装について、書き込みコマンドの場合、ターゲットはデータ配列の中身を修正することはできないことが追加
- n) organisation → organization （英国綴り→米国綴り）
- q) num_bytes(ボールド体)→データ長属性の値 に変更
- r) 詳細に定義する文章に変更.
  直接または間接的に,transport_dbgはwaitを呼んではいけない、またデバッグ用の書き込みコマンドを即座に実行してくれる効果があるが、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変化させてはいけない

5.グローバル・クォンタム

- 構成
  - 前版「8章 その他のクラス」の「8.1 グローバル・クォンタムとクォンタム・キーパー」にあった、「8.1.1 イントロダクション」および、「8.1.4 tlm_global_quantumクラス」部分が、独立した章として切り出された
  - 内容は、前版とほぼ同じ
- 解説の変更
  - 補足的に、「テンポラル・デカップリングを行うプロセス間の同期で、クォンタム時間の使用は必須ではないが、クォンタム時間を使うプロセスは、グローバル・クォンタムを使わなければいけない」との明文化がされている。

JEITA

6.結合インタフェースとソケット

- 章の構成
  - 前版で同じ章に入っていた便利ソケットの章は、全てUtilityの章へ移動
- 用語の変更
  - プロトコル・タイプ・クラス(protocol type class)
    →プロトコル・トレイツ・クラス(protocol traits class)
    - ここでの意味：フォワードI/F、バックワードI/Fで使われるペイロード・タイプとフェーズ・タイプを定義したもの
- コードの追加/変更
  - tlm_base_initiator_socketクラス、tlm_base_target_socketクラス、tlm_initiator_socketクラス、tlm_target_socketクラスに対して、以下のAPIを追加
    virtual const char* kind() const;
    - 目的：クラス名をCストリングとして返すためのメソッド。例えば、tlm_base_initiator_socketクラスのkind()を呼び出した場合は、“tlm_base_initiator_socket”が返る。
  - tlm_base_target_socketクラスでinitiator_socket_typeのtypedefを削除(もともと使われていなかった)

あとは、細かい説明の追加・変更、typoの修正のみ。

JEITA

7 Generic payload 7.1 Introduction（旧6.1）

これら3つが新たに追加

- 汎用ペイロードは
  - TLM-2.0標準が提供するトランザクション・オブジェクトのためのクラス
  - 基本プロトコルに密接に関係
  - 汎用ペイロードを使ったときの相互利用性を確保するためのルールを定義する
- MMBモデルの相互利用性を改善することが目的
  - （1）既成の汎用目的のペイロード。バスプロトコルの詳細が重要でない場合に有効。無視可能なアトリビュートで拡張メカニズムも提供。
  - （2）特定バスプロトコルの詳細モデルのベースになる。実装コスト削減とブリッジが必要なときのシミュレーション速度アップ。

7 Generic payload 7.1 Introduction（旧6.1）

変更なし。

- 汎用ペイロードはMMBモデリングを目的
  - アトリビュート
    - 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、レスポンスステータス
  - MMB以外のプロトコルのモデリングの基礎にもなる
- 特殊なアトリビュートを持たせる拡張メカニズムを提供

7 Generic payload 7.1 Introduction（旧6.1）

変更なし。

- モデリングと相互利用性はプロトコル・オーナーの責任で、OSCIのスコープの範囲外
  - 汎用ペイロードはモデル作成の基礎となりえる
- 汎用ペイロードをイニシエータ・ソケット/ターゲット・ソケットと一緒に使うことを推薦
  - Strong type checkingができる
- 汎用ペイロードは、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、DMI、デバッグ・インタフェースで使用可能。

7.2　拡張性と相互利用性（旧6.2）

全体的にtlm_phaseが追加された
それ以外は用語変更・説明追加で大きな変更なし

- 全体的に
  - protocol typeからprotocol traitsに用語変更
  - non-ignorableとmandatoryを両方併記
  - tlm_phaseの利用
- 異なるプロトコルを接続するものとしてブリッジの他にアダプタという言葉が追加された

7.2.1 無視可能拡張を用いた汎用ペイロード利用の場合（旧6.2.1）

拡張部分に関する振る舞いについて，
具体例が削除され記述がシンプルになっている

- トランザクション型として，tlm_generic_payloadとprotocol traitsにフェーズ型としてtlm_phaseが追加
- 相互利用可能なモデルとして，
  - “汎用ペイロードに特化したコア・トランスポート・インターフェースを使う”
    →“基本プロトコルを持つ標準イニシエータとターゲット・ソケットを使う”に変更
- 拡張部分を無視する振る舞いとして，具体的な記述（ターゲットやインターコネクト・コンポーネントがフェイルしないエラーが出ない）が削除されてシンプルな説明になっている
- 無視可能拡張を無視する場合のデフォルト値やどういったデータを転送するかについての一例文が削除

7.2.2 tlm_generic_payloadの型宣言を含む新しいプロトコル・トレイツ・クラスの定義（旧6.2.2）

ユーザー定義のプロトコル型は
基本プロトコルと矛盾してもよいことが記載
LRMにはあまり必要性がない推奨例が削除

- “新しいプロトコル型の独自ルールは汎用ペイロードのメモリ・マネージメント・ルールと引数の修正可能性を含む基本プロトコルのルールを拡張または矛盾していてもよい”ことが記載
- “プロトコル変換を行うブリッジを使えば，異なるプロトコルのトランザクションを使うことが可能”という文が追加
- 推奨される下記2つの利用パターンが削除
  - イニシエータ/インターコネクト/ターゲット全てが共通に同じ新しいプロトコル型を使用
  - 末端だけが新しいプトロコルで、その間のモデルはgeneric_base_protocol_typesを使用
- sockets parameterized with different protocol types
  →sockets specialized using different protocol traits classes
- 拡張が汎用ペイロードを通して転送
  →拡張が拡張されたことを知らないコンポーネントを通して転送

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の定義（旧6.2.3）

変更なし

7.3 - 7.5 汎用ペイロードのアトリビュート・メソッド・メモリ管理

- 構成
  - 前版「6章 汎用ペイロード」の、6.3 - 6.5に対応
  - 内容は、前版とほぼ同じ
- 「7.4 クラス定義」で、メソッドの追加あり
  - max_num_extensions()インライン関数の追加
    旧ライブラリでも本関数は存在しているので、前版での記載漏れ？
  - tlm_generic_payload::update_original_from()関数の追加
    ディープコピーにより複製された汎用ペイロードを、再度アップデートする関数。対象は、レスポンスステータス、DMI許可情報、リードデータ
- 「7.5 汎用ペイロード・メモリ管理」で、ルールの追加あり
  - 汎用ペイロードオブジェクトの連続的な生成・消滅は、CPU時間のコストが大きいため避けるべき。代替としてメモリマネージャの活用や、b_transport使用時は同一のオブジェクトの再利用を推奨としている
  - deep_copy_from関数、update_original_from関数、update_extension_from関数の使いどころとして、トランザクションブリッジでの利用を示唆している
  - その他メンバ関数についても、いくつか補足説明を追加 (仕様変更はない)

7.6 - 7.7 汎用ペイロード

- 追加
  - 「7.7 アトリビュートのデフォルト値と変更可否」で、アトリビュートの値をいつ変更できるかに関して、詳細な説明を追加(従来はテーブルのみで説明なし)。
- その他
  - 内容的には同じだが、「7.7 アトリビュートのデフォルト値と変更可否」で、データ配列、バイト・イネーブル配列を別テーブルに分けた。

7.8 コマンド・アトリビュート

- "ignore command"の説明が追加
  - コマンド・アトリビュートがTLM_IGNORE_COMMANDだと，"ignore command". [c]
  - リード・ライトを実行しなくても使え，汎用ペイロードの拡張して使える. [g]
  - "ignore command"を受け取った際の応答方法. [i, j]

7.10 データ・ポインタ・アトリビュート

- Readコマンド時の説明追加
  - 上書きされるのは，ターゲットが応答を返す前のみ．ターゲットが応答を返すタイミング
    - 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
    - BEGIN_RESPフェーズがnb_transportの引数として渡ったとき
    - nb_transportからTLM_COMPLETEDが返ったとき

  も言及．[i]

7.16 応答ステータス・アトリビュート

- ターゲットが上書きしない場合の説明が追加
  - デフォルト値”TLM_INCOMPLETE_RESPONSE”をインターコネクトは上書きしないし，ターゲットがコマンドを実行しなかった場合も同様．[e]
  - エラー応答の一つとして，上書きされなかったときの”TLM_INCOMPLETE_RESPONSE”を追加
- “ignore command”への応答の説明が追加
  - TLM_OK_RESPONSEか，適当なエラー応答を返してよい．[i, j]
- エラー応答の選択方法の説明が追加
  - ターゲットがエラー応答を決める際に自由度を持つ．コマンドとアドレスがエラーだったら，TLM_ADDRESS_ERROR_RESPONSEか，TLM_COMMAND_ERROR_RESPONSEか，TLM_GENERIC_ERROR_RESPONSEのどれも正しい[m]

7.17 エンディアン

- 前版は、6.17章であった。
- 追加
  - 「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。」の1文

7.18 ホストエンディアンを決定するHelper関数

- 前版は、6.18章であった。
- 変更なし。

7.19 エンディアン変換用Helper関数

- 前版は、6.19章であった。
- 細かい表記以外の変更なし。
  - 例）hostendian → host-endian

7.20 汎用ペイロードの拡張

- 前版は、6.20章であった。
- 変更
  - Ignorable or mandatoryであったものが、ignorable or non-ignorable, mandatory or non-mandatoryに変更された。これに合わせて「7.20.1 イントロダクション」の文が差し替えられた。
  - 「7.20.2 原理」の文も全面的に差し替えられている。
  - ブリッジがトランザクション・ブリッジに言い換えられている。
  - Update_original_fromメソッドが追加された。
  - 「7.20.4 規則」にはいくつかの変更がある。
    - 旧規則dが削除。旧規則eからnは、dからmへ移動。新規則nが追加
  - Protocol types class → protocol traits class
- その他
  - 「7.20.1.1 Ignorable extensions 」と 「7.20.1.2 Non-ignorable and mandatory extensions」の2つの表題には一貫性がないように思える。

8. Base protocol and phases

- 構成
  - 前版「7章 **Phases and base protocol**」が
    「8章 **Base protocol and phases** 」と**protocolとphases**が逆になった
  - 内容は、前版とほぼ同じ
- 解説の変更
  - 補足的に、状態遷移図が追加されていて理解しやすくする工夫が見られる。

8.1.3 Rules

【旧】
f) If an extended phase cannot be ignored by any component that receives it, the application should define a new protocol types class and use the name of that class as a template argument when instantiating associated sockets. This is in order to prevent the binding of sockets that represent incompatible protocols.
g) A transition to an ignorable phase may simply be ignored by any recipient. In the case of a call to *nb_transport, if the callee is ignoring the phase transition is should return the value TLM_ACCEPTED.*

【新】
削除

8.2.1 Introduction

【旧】
The base protocol consist of a set of rules to ensure maximal interoperability between transaction level models of components that interface to memory-mapped buses. The base protocol requires the use of:

【新】
The base protocol consists of a set of rules to ensure maximal interoperability between transaction level models of components that interface to memory-mapped buses. The base protocol requires the use of the classes of the TLM-2.0 interoperability layer listed here, together with the rules defined in this clause:

注記：文言が丁寧に説明されている。上記文面後のリストに参照すべき章を案内している。

 JEITA

8.2.1 Introduction

【旧】
なし

【新】
In cases where it is necessary to define a new protocol traits class (e.g. because the features of the base protocol are insufficient to model a particular protocol), the rules associated with the new protocol traits class override those of the base protocol. However, for consistency and interoperability it is recommended that the rules and coding style associated with any new protocol traits class should follow those of the base protocol as far as possible. See 7.2.2 Define a new protocol traits class containing a typedef for tlm_generic_payload

This section of the standard specifically concerns the base protocol, but nonetheless may be used as a guide when modeling other protocols. Specific protocols represented by other protocol traits classes may include additional phases and may adopt their own rules for timing annotation, transaction ordering, and so forth. In doing so, they may cease to be compatible with the base protocol.

注記：上記文言の説明が追加されている。

 JEITA

8.2.3 Base protocol phase sequences

【旧】
a) This clause is specific to the base protocol, but may be used as a guide when using the non-blocking transport interface to model other protocols. In order to model other protocols it may be necessary to define other phases, but doing so may result in a loss of interoperability with the base protocol.

【新】
削除

 JEITA

8.2.3 Base protocol phase sequences

【旧】
e) In the case of the blocking transport interface, a single call to and return from **b_transport shall describe** the entire lifetime of one transaction instance. Any correspondence between the call to **b_transport and** BEGIN_REQ, or the return from **b_transport and BEGIN_RESP, is purely notional.**

【新】
d) In the case of the blocking transport interface, a transaction instance is associated with a single call to and return from **b_transport. The correspondence between the call to b_transport and BEGIN_REQ, and the return from b_transport and BEGIN_RESP, is purely notional; b_transport has no associated phases.**

注記：文言が変更されている。

 JEITA

8.2.3 Base protocol phase sequences

【旧】
g) The phase sequence can be cut short by having *nb_transport return a value of TLM_COMPLETED. A* return value of TLM_COMPLETED indicates the end of the transaction, in which case the phase argument should be ignored (see clause 4.1.2.7 The tlm_sync_enum return value). TLM_COMPLETED does not imply successful completion, so the initiator should check the response status of the transaction for success or failure. A transition to the phase END_RESP shall also indicate the end of the transaction, in which case the callee is not obliged to return a value of TLM_COMPLETED.

【新】
f) The phase sequence can be cut short by having *nb_transport return a value of TLM_COMPLETED, but only in one of the following ways. An interconnect component or target may return TLM_COMPLETED when it receives BEGIN_REQ or END_RESP on the forward path. An interconnect component or initiator may return TLM_COMPLETED when it receives BEGIN_RESP on the backward path. A return value of TLM_COMPLETED indicates the end of the transaction with respect to a particular hop, in which case the phase argument should be ignored by the caller (see 4.1.2.7 The tlm_sync_enum return value). TLM_COMPLETED does not imply successful completion, so the initiator should check the response status of the transaction for success or failure.*

注記：文言が変更されている。

 JEITA

8.2.3 Base protocol phase sequences

【旧】
なし

【新】
h) When TLM_COMPLETED is returned in an upstream direction after having received BEGIN_REQ, this carries with it an implicit END_REQ and an implicit BEGIN_RESP. Hence the initiator should check the response status of the generic payload, and may send BEGIN_REQ for the next transaction immediately.
i) Since TLM_COMPLETED returned after having received BEGIN_REQ carries with it an implicit BEGIN_RESP, this situation is forbidden by the response exclusion rule if there is already a response in progress through a given socket. In this situation the callee should have returned TLM_ACCEPTED instead of TLM_COMPLETED and should wait for END_RESP before sending the next response upstream.
j) Since TLM_COMPLETED returned after having received BEGIN_REQ indicates the end of the transaction, an interconnect component or initiator is forbidden from then sending END_RESP for that same transaction through that same socket.
k) When TLM_COMPLETED is returned in a downstream direction by a component after having received BEGIN_RESP, this carries with it an implicit END_RESP.

注記：TLM_COMPLETED を受信したときの動作を規定している。

 JEITA

8.2.3 Base protocol phase sequences

【旧】
h) If an initiator receives a BEGIN_RESP from a target without having first received an END_REQ, the initiator shall assume an implicit END_REQ immediately preceding the BEGIN_RESP.

【新】
l) If a component receives a BEGIN_RESP from a downstream component without having first received an END_REQ for that same transaction, the initiator shall assume an implicit END_REQ immediately preceding the BEGIN_RESP. This is only the case for the same transaction; a BEGIN_RESP does not imply an END_REQ for any other transaction, and a target that receives a BEGIN_REQ cannot infer an END_RESP for the previous transaction.

m) The above points hold regardless of the value of the timing annotation argument to *nb_transport.*

n) A base protocol transaction is complete (with respect to a particular hop) when TLM_COMPLETED is returned on either path, or when END_RESP is sent on the forward path or the return path.

o) In the case where END_RESP is sent on the forward path, the callee may return TLM_ACCEPTED or TLM_COMPLETED. The transaction is complete in either case.

p) A given transaction may complete at different times on different hops. A transaction object passed to *nb_transport is obliged to have a memory manager, and the lifetime of the transaction object ends when the reference count of the generic payload reaches zero. Any component that calls the* ***acquire method of a generic payload transaction object should also call the release method at or before the completion of the transaction. See 7.5 Generic payload memory management***

q) If a component receives an illegal or out-of-order phase transition, this is an error on the part of the sender. The behavior of the recipient is undefined, meaning that a run-time error may be caused.

注記：implicit END_REQ を受信したときの動作を規定している。

JEITA

8.2.4 Permitted phase transitions

【旧】
旧7.2.3の
i) Taking all the previous rules into account, the set of permitted phase transition sequences is as follows, where the path (forward, backward or return) is shown in parenthesis. Ignorable phase extensions may be inserted at any point. In each case the transaction may or may not have been successful.
BEGIN_REQ(fw) (target returns TLM_COMPLETED)
BEGIN_REQ(fw) → END_REQ(bw) (initiator returns TLM_COMPLETED)
BEGIN_REQ(fw) → BEGIN_RESP(bw) (initiator returns TLM_COMPLETED)
BEGIN_REQ(fw) → END_REQ(rtn/bw) → BEGIN_RESP(bw) (initiator returns TLM_COMPLETED)
BEGIN_REQ(fw) → BEGIN_RESP(rtn/bw) → END_RESP(rtn/fw)
BEGIN_REQ(fw) → END_REQ(rtn/bw) → BEGIN_RESP(bw) → END_RESP(rtn/fw)
j) If a component receives an illegal or out-of-order phase transition, this is an error on the part of the sender. The behavior of the recipient is undefined, meaning that a run-time error may be caused.

【新】
8.2.4 Permitted phase transitionsと
8.2.5 Ignorable phases
として詳細に説明されている。（次ページへ）

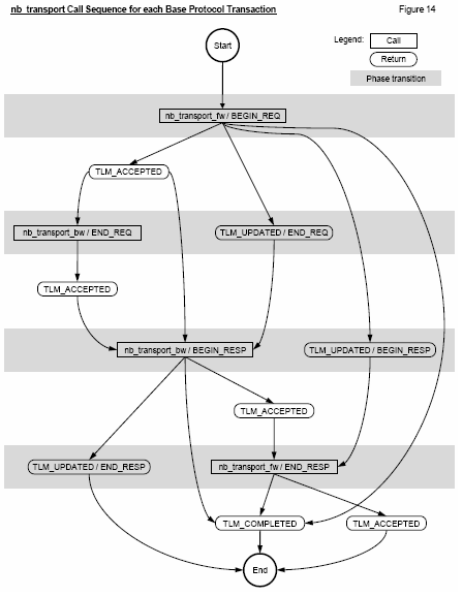
JEITA

8.2.4 Permitted phase transitions

【旧】
なし

【新】
3ページにわたって表と状態遷移図を用いて詳しく規定動作を説明している。

Previous state		Calling path	Phase argument on call	Phase argument on return	Status on return	Response valid	End-of-life		Next state
//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



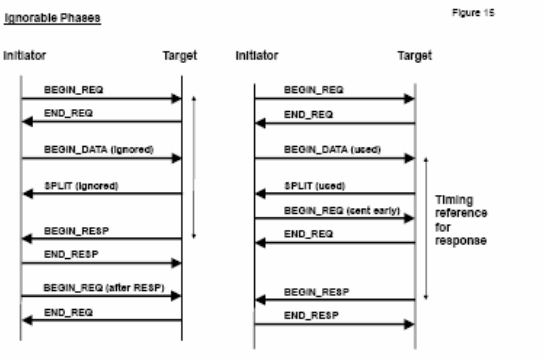
 JEITA

8.2.5 Ignorable phases

【旧】
なし

【新】
2ページちょっとを割き、例をあげて事例を説明している。

Example



注記：TLM_COMPLETED を受信したときの動作を規定している。

 JEITA

8.2.6 Base protocol timing parameters and flow control

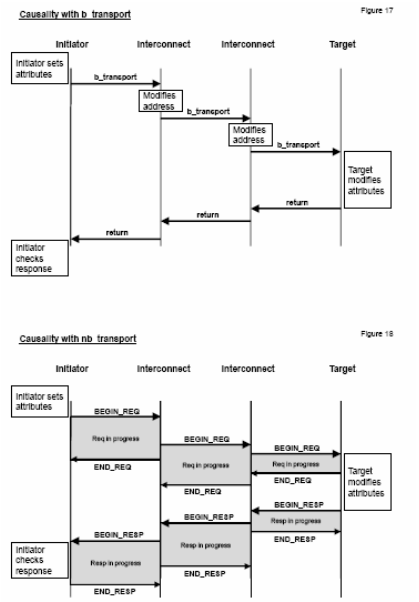
【旧】

【新】
同様に図やコード例を交えて詳細に説明されている。

Example

The following pseudo-code illustrates the interaction between timing annotation and the request and response exclusion rules:

```
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(…);
}
```



8.2.7 Base protocol rules concerning timing annotation

【旧】
旧7.2.5 Base protocol transaction ordering rulesを6つに分割し、詳細に説明する。

【新】
8.2.7 Base protocol rules concerning timing annotation 、
8.2.8 Base protocol rules concerning b_transport 、
8.2.9 Base protocol rules concerning request and response ordering 、
8.2.10 Base protocol rules for switching between b_transport and nb_transport 、
8.2.11 Other base protocol rules 、
8.2.12 Summary of base protocol transaction ordering rules

8.2.7 Base protocol rules concerning timing annotation

【旧】
旧7.2.5 Base protocol transaction ordering rulesを6つに分割し、詳細に説明する。

【新】
8.2.7 Base protocol rules concerning timing annotation 、
8.2.8 Base protocol rules concerning b_transport 、
8.2.9 Base protocol rules concerning request and response ordering 、
8.2.10 Base protocol rules for switching between b_transport and nb_transport 、
8.2.11 Other base protocol rules 、
8.2.12 Summary of base protocol transaction ordering rules

 JEITA

8.2.7 Base protocol rules concerning timing annotation

【旧】
旧7.2.5 Base protocol transaction ordering rulesのa), b), c)を、詳細に説明する。

【新】
8.2.7 Base protocol rules concerning timing annotation

 JEITA

8.2.8 Base protocol rules concerning b_transport

【旧】
旧**7.2.5 Base protocol transaction ordering rules**のd)を、詳細に説明する。

【新】
8.2.8 Base protocol rules concerning b_transport

 JEITA

8.2.9 Base protocol rules concerning request and response ordering

【旧】
旧**7.2.5 Base protocol transaction ordering rules**を、詳細に説明する。

【新】
8.2.9 Base protocol rules concerning request and response ordering

 JEITA

8.2.10 Base protocol rules for switching between b_transport and nb_transport

【旧】
旧7.2.5 Base protocol transaction ordering rulesのi), j)を、詳細に説明する。

【新】
8.2.10 Base protocol rules for switching between b_transport and nb_transport

8.2.11 Other base protocol rules

【旧】
旧7.2.5 Base protocol transaction ordering rulesのl), m), n)を、詳細に説明する。

【新】
8.2.11 Other base protocol rules

8.2.12 Summary of base protocol transaction ordering rules

【旧】
旧7.2.5 Base protocol transaction ordering rulesのまとめ。

【新】
8.2.12 Summary of base protocol transaction ordering rules
（表にしている。）

 JEITA

8.2.13 Guidelines for creating base-protocol-compliant components

【旧】
旧7.2.6 Summary of obligations on base protocol components
7.2.6.1 Obligations on an initiator
7.2.6.2 Obligations on an initiator using *nb_transport*
7.2.6.3 Obligations on a target
7.2.6.4 Obligations on a target using *nb_transport*
7.2.6.5 Obligations on an interconnect component

【新】
8.2.13 Guidelines for creating base-protocol-compliant componentsを若干詳細に説明。
8.2.13.1 Guidelines for creating a base protocol initiator
8.2.13.2 Guidelines for creating an initiator that calls *nb_transport*
8.2.13.3 Guidelines for creating a base protocol target
8.2.13.4 Guidelines for creating a target that calls *nb_transport*
8.2.13.5 Guidelines for creating a base protocol interconnect component

 JEITA

9.1 Utility Socket (1)

- 便利ソケットのコンストラクタにてソケット名称を与え無かった時のデフォルト名称が自動生成されるように変更された
  - 一つのモジュールに複数ソケットが有る場合でソケット名称を明示的に与えない場合、同じ名称となってしまう問題があった。
  - TLM2.0.1では、sc_gen_unique_name()関数を用いて必ずユニークな名称になるように変更された。
  - 常に明示的に名称を与えておけば、ソースコード上の互換性は2.0.0<->2.0.1で保たれる。(変更の必要はない)

9.1 Utility Socket (2)

- マルチソケット使用時に、boostパッケージを必要としなくなった。
  - 2.0.0では、別途boostパッケージをダウンロードして使用する必要があった。
  - 2.0.1では、boostパッケージを使用する必要が無くなった。
  - 2.0.0で動作していたモデルが2.0.1で動かなくなる事はない。
  - 今まで動作しないモデルが、TLM2.0.1によって動作するケースはある。(別途別バージョンのBoostを使用している場合など)

9.1 Utility Socket (3)

- 階層バインドルールの明確化
  - TLM2.0.0では階層バインドのルールについて明確に定義されていない部分があった。
    - 実際のソースを見ないとルールがわからなかった。
  - TLM2.0.1では、階層バインドのルールがすべて明確に定義された。
    - 実際のソースと一致しているルールになっている。

9.1 Utility Socket (4)

- Utilityソケットごとのヘッダーファイル名の定義
  - TLM2.0.0ではそれぞれのユーティリティソケットがどのヘッダーファイルに定義されているかの記述が無かった。
  - TLM2.0.1では、それぞれのユーティリティソケットに必要なヘッダーファイルが明確に定義された。

9.1 Utility Socket (5)

- マルチソケットにおける未バインドの処理
  - マルチソケットにおいて、階層バインドを行わなかったときに、どうなるのかがTLM2.0.0では、定義されていなかった。
  - コールバックも無かった場合
    - b_transport,nb_transport_* ランタイムエラー(必須)
    - transport_dbg return 0
    - get_direct_mem_ptr return false
    - invalidate_direct_mem_ptr 無視する

 JEITA

9.1 Utility Socket (5)

- マルチソケットのExample追加
  - TLM2.0.0ではマルチソケットに関しては一切Exampleの記述が無かった。
  - TLM2.0.1ではマルチソケットに関する簡単なExampleソースがLRM中に記載された。

 JEITA

9.2 クォンタム・キーパー

- 構成
  - 前版「8章 その他のクラス」の、「8.1 グローバル・クォンタムとクォンタム・キーパー」に対応 (グローバル・クォンタム部分は5章に移動されている)
  - 内容は前版とほぼ同じ。クォンタム・キーパーの動作を説明する図が追加されている（図を使った説明は特になし）
- 「9.2.3 クラス定義」で、メソッドの追加あり
  - tlm_quantumkeeper::set_and_sync()メソッドの追加
    既存のset()メソッド、sync()メソッドを連続でコールするだけ
- 「9.2.4 テンポラル・デカップリングの使い方に関する一般的ルール」で、ルールの修正・追加あり
  - クォンタム管理下の複数プロセスの実行順序は不確定であるため、明示的な同期機構がない場合、ある変数がプロセスAで更新されプロセスBでリードされる場合、その値は不確定。そのため、適切な同期機構を用いるなどして、変数へのアクセスをガードする必要があると明記
- 「9.2.5 tlm_quantumkeeperクラス」は大きな変更なし
  - 追加されたset_and_sync()メソッドの説明が追加された程度

9.3 ペイロード・イベント・キュー

- 構成
  - 前版「8章 その他のクラス」の、「8.2 ペイロード・イベント・キュー」に対応
  - 内容は大きく加筆修正されている。前版ではクラス定義のコードを示して終わっていたが、ルールの追加による詳細な説明あり
- 「9.3.1 イントロダクション」、「9.3.2 ヘッダファイル」
  - peq_with_cb_and_phaseクラス使用時は、
    SC_INCLUDE_DYNAMIC_PRCESSESのマクロ定義が必要
  - peq_with_getクラス、peg_with_cb_and_phaseクラス使用時は、
    tlm_utils/peq_with_get.h、あるいはpeq_with_cb_and_pahse.hのインクルードが必要
- 「9.3.3 クラス定義」で、メソッドの追加あり
  - peq_with_get::cancel_all()メソッド、
    peq_with_cb_and_phase::cancel_all()メソッドが追加。これらは、PEQにキューイングされている全トランザクションを削除する機能
- 「9.3.4 ルール」
  - 全16ルール
  - peg_with_cb_and_phaseに関連付けるコールバック関数は、ノンブロッキングでなければいけない(SC_METHODプロセスから呼ばれるため) 等

9.4 インスタンス固有の拡張

- 前版は、6.21章であった。
- 追加
  - 「9.4.2 ヘッダファイル」として以下の1文を追加。

    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ある。

- その他
  - 例中の関数において引数名が削除された。

 JEITA

10章 TLM-1 AND ANALYSIS PORT

(1) 章の番号が変更
　旧9章から10章へ
 analysisについて
　旧8.3　章を10.4章に移動 ： ”10.4　analysis interface and analysis ports”

【旧】
The latter interface is not specific to analysis, and may be used for other purposes. For example, see clause 8.2 Payload event queue.
The TLM-2 kit includes　the tlm_analysis_fifo, which is simply an infinite tlm_fifo that implements the tlm_analysis_if to write a transaction to the fifo. The tlm_fifo also supports the tlm_analysis_triple, which consists of a transaction together with explicit start and end times.
【新】
The latter interface is not specific to analysis, and may be used for other purposes. For example, see 9.3 Payload event queue.
The tlm_analysis_fifo is simply an infinite tlm_fifo that implements the tlm_analysis_if to write a transaction to the fifo. The tlm_fifo also supports the tlm_analysis_triple, which consists of a transaction together with explicit start and end times.

(2) Rulesの章番号変更 ： 8.3.2 → 10.4.2
　文章変更なし

 JEITA

11. Glossary: adapter

【旧】
adapter: 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. Typically, an adapter is used to convert between two transaction-level interfaces of different types. See transactor.

【新】
adapter: 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.

旧では同じことを言っていた「異なるトランザクションレベルのインターフェイス」文言から「異なるプロトコルタイプのSocket」の変換に利用してよいに変更

 JEITA

11. Glossary: base protocol

【旧】
base protocol: A protocol types 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

【新】
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

プロトコルのType（型）クラスからtraitsクラスへ変更
新規にtraitsクラスを利用。

【参照】 traits クラス：何らかのオブジェクトの特性を別のテンプレートで表現する手法。

 JEITA

11. Glossary：bridge

【旧】
bridge: A module that connects together two similar or dissimilar transaction-level interfaces, each representing a memory-mapped bus or other protocol, usually at the same abstraction level. A bus bridge is a device that connects two similar or dissimilar buses together. A communication bridge is a device that connects network segments on the data link layer of a network. In TLM-2, a bridge is a component that acts as a target for an incoming transaction and an initiator for an outgoing transaction. See transactor.

【新】
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.

修正後の文言
2つのネットワークの通信セグメントを接続するコンポーネント。バス・ブリッジは2つのバスを接続するデバイス。
＃余分な説明が削除された。

 JEITA

11. Glossary：component/effective local time

【旧】
なし

【新】
component: An instance of a SystemC module. This standard recognizes three kinds of component; the initiator, interconnect component, and target.

effective local time: The current time within a temporally decoupled initiator.
effective_local_time = sc_time_stamp() + local_time_offset

コンポーネント：あるSystemCモジュールのーインスタンス。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種類コンポーネントを区別している。

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

 JEITA

11. Glossary：exclusion rule

【旧】
なし

【新】
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と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。

 JEITA

11. Glossary：extension

【旧】
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 mandatory extension requires the definition of a new protocol types class.

【新】
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.

non-ignorableとmandatoryは別物？
同じもの？→定義必要

修正：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タイプ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、無視できないすなわち必須である拡張は新しいプロトコルのtraitsクラス定義が必要である。

 JEITA

11. Glossary：generic payload

【旧】
generic payload: A specific set of transaction attributes and their semantics together defining a transaction level payload which may be used to achieve a degree of interoperability between untimed, loosely timed and approximately timed models for components communicating over a memory-mapped bus. The same transaction class is used for all modeling styles.

【新】
generic payload: 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.

Untimedを削除した理由は？

修正：トランザクション・アトリビュートのセットとメモリ・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するためのセマンティクス。アトリビュートは、全てのモデルに使えるものとアプロキシメイトリー・タイムド・モデルにのみ使えるものとに分けられる。

 JEITA

11. Glossary：hierarchical binding/hop

【旧】
なし

【新】
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.
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.

階層バインディング：親モジュールのソケットへ子モジュールのソケットをバインドする、あるいは子モジュールのソケットに親モジュールのソケットのバインドする事。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。
ホップ：イニシエータからターゲット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ーフェイスメソッドコール(のパス)。ホップはターゲットソケット方向のものとイニシエータソケット方向のものがある。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。

 JEITA

11. Glossary：ignorable extension/ ignorable phase

【旧】
なし

【新】
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.
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.

無視可能な拡張：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や、いかなるレスポンスも必要としないコンポーネントからは無視される。無視されるフェーズは基本プロトコルで使用可能である。

無視可能なフェーズ：マクロのDECLARE_EXTENDED PHASE　によって作られ、どのレスポンスも需要（利用？）できないコンポーネントとフェーズを受けるコンポーネントからも無視されるフェーズ。無視されるフェーズは基本プロトコルで許容されている。

 JEITA

11. Glossary：initiator

【旧】
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 1.0 interfaces, the term initiator as defined here may not be strictly applicable, so the terms caller and callee may be used instead for clarity.

【新】
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.

修正：TLM 1.0->TLM-1

 JEITA

11. Glossary：initiator socket

【旧】
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 also overloads the SystemC binding operators to bind both port and export.

【新】
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 the port and the export.

修正：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を含むクラス、及び、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。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。

 JEITA

11. Glossary：local time offset

【旧】
なし

【新】
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

 JEITA

11. Glossary：multi-socket

【旧】
multi-socket: One of a family of convenience sockets that can be bound to multiple sockets belonging to other components. A multi-initiator 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 multisockets, the destinations are distinguished using the subscript operator.

【新】
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 target multi-socket. When calling interface methods through multi-sockets, the destinations are distinguished using the subscript operator.

修正：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便利ソケットのひとつ。イニシエータ・マルチソケットに複数のターゲット・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・ソケットに複数のイニシエータ・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・コールをする際、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。

 JEITA

11. Glossary：nb_transport

【旧】
nb_transport: The nb_transport_fw and nb_transport_bw methods. In this document, the italicised term nb_transport is used to describe both methods in situations where there is no need to distinguish between them.

【新】
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.

修正：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。

 JEITA

11. Glossary：opposite path

【旧】
なし

【新】
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.

逆のパス：逆方向のパス。
フォワードパスでは逆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。

11. Glossary：phase

【旧】
phase: The period in the lifetime of a transaction occurring between successive timing points. The phase is passed as an argument to the non-blocking transport method.

【新】
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.

修正：トランザクションが有効な期間。
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推移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の時間引数で与えられた合計で遅延される。

11. Glossary：phase transition

【旧】
なし

【新】
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・メソッドにおけるフェーズ引数の値の変化であり、nb_transportへのコールとnb_transportからのリターンの際TLM_UPDATEDの値を伴う。

11. Glossary：protocol traits class

【旧】
protocol types 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.

【新】
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.

修正：プロトコルタイプのクラス/Type（型）クラス？からtraitsクラスへ変更

11. Glossary：request

【旧】
なし

【新】
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際のトランザクション・ライフタイムにおけるステージのひとつ。基本的には、リクエストはコマンド、アドレス、ライトコマンドのためのデータアレイなどの汎用ペイロード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値参照、データアレイの場合はポインター渡しにより行われる。）

11. Glossary：response

【旧】
なし

【新】
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際のトランザクション・ライフタイムにおけるステージのひとつ。基本的には、レスポンスはステータス、リードコマンドのためのデータアレイなどの汎用ペイロード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値参照、データアレイの場合はポインター渡しにより行われる。）

11. Glossary: sticky extension

【旧】
sticky extension: A generic payload extension object that will not be automatically deleted when the reference count of the transaction object reaches 0. Sticky extensions are not deleted by the memory manager.

【新】
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終了時に(C++で自動及び明示的に)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アクションオブジェクトは残される。
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。

11. Glossary: synchronization-on-demand

【旧】
synchronization-on-demand: An indication from the nb_transport method back to its caller that it was unwilling or unable to fulfill a request to effectively execute a transaction at a future time (temporal decoupling), and therefore that the caller must yield control back to the SystemC scheduler so that simulation time may advance and other processes run.

【新】
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.

修正: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。加えて、各クォンタムの終わりが来るたびに同期ポイントが取られる。

11. Glossary: timing annotation

【旧】
なし

【新】
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。

 JEITA

11. Glossary: timing point

【旧】
timing point: A point in time at which the processes that are interacting through a transaction either transfer control or are synchronized. Certain timing points are implemented as function calls or returns, others as event notifications. Timing points mark the boundaries between the phases of a transaction. Consecutive timing points could occur in different delta cycles at the same simulation time.

【新】
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.

トランザクションのライフタイム内の重要な時間。ルーズリータイムド・トランザクションにはb_transportからのコールとリターンに関係する2つのタイミング・ポイントがある。アプロキシメートリー・タイムドのベース・プロトコル・トランザクションはフェーズ・トランジションに関する4つのタイミングポイントがある。

 JEITA

11. Glossary:TLM-1/TLM-2.0

【旧】
TLM-1: The first major version of the OSCI Transaction Level Modeling standard. TLM-1.0 was released in 2005.
TLM-2: The second major version of the OSCI Transaction Level Modeling standard. This document describes TLM-2.0.

【新】
TLM-1: The first major version of the OSCI Transaction Level Modeling standard. TLM-1 was released in 2005.
TLM-2.0: The second major version of the OSCI Transaction Level Modeling standard. This document describes TLM-2.0.

1.0→1への修正?
2→2.0への修正

 JEITA

11. Glossary: traits class/transaction bridge

【旧】
なし

【新】
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.

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。

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。

 JEITA

11. Glossary: instance/transparent component

【旧】
なし

不二？

【新】
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.
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てもよい。

トランスペアレント・コンポーネント:すべての受け取ったインターフェイ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)の変更なく、遅延なしでコンポーネントを通して直ちに反映されるというプロパティ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。

11. Glossary: transport interface

【旧】
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 adds separate blocking and non-blocking transport interfaces.

【新】
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.

"TLM-1標準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リクエスト・トランザクション・オブジェクトを返す。
TLM-2.0標準ではさらにブロッキング・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・インタフェースの2つに分割される。"

11. Glossary: unidirectional interface

【旧】
unidirectional interface: A TLM 1 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.
以下省略

【新】
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.

1→1.0への修正

 JEITA

11. Glossary: utilities

【旧】
なし

【新】
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はあるが、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。

 JEITA

04-2-04 SystemC Synthesizable Subset 1.3 draft 1章 日本語抄訳

本節は、OSCI の SystemC Synthesizable Subset 1.3 draft の第 1 章の日本語抄訳である。

1 概要

1.1 範囲

Open SystemC Initiative(OSCI)の Synthesis Working Group (SWG)は Synthesizable-SystemC (SSC)の定義を開発した。これはハードウェア設計者が SystemC を使ったモデル開発やデザイン・プロセスを迅速化するのに有用なだけでなく、SystemC コンプライアンスの合成ツールを開発する EDA ツール開発者のためにも有用である。SSC は、C++と IEEE1666 SystemC 仕様の中で定義される。ただし、SWG は将来の効率的なシンセシスのための可能性のために SystemC の拡張と追加ライブラリを提案するかもしれない。

1.2 目的

ユーザーはどのような抽象レベルのどのようなシステムでも SystemC で開発して、それを OSCI ウェブサイト上の有効な許諾契約に基づいて利用可能なリファレンスインプリメンテーションで検証することが出来るであろう。現在、ユーザーガイド、ファンクショナル・スペシフィケーションと言語リファレンス・マニュアルという言語記述の情報が入手可能である。また、SystemC モデリングのための本もいくつか発行された。しかしながら、現在 SystemC の合成可能な記述の規格は定義されていない。私たちはここで、これを「Synthesizable-SystemC(SSC)」と呼ぶつもりである。

合成技術はハードウェア設計に必要な時間を劇的に短縮する。そして、それは設計フローで最も重要なフェーズの 1 つであることが広く知られている。強力な SystemC ベースの設計環境を活用するために、SCC は SystemC トップダウン設計のために不可欠である。

SCC はコーディング・ガイドラインを含む SystemC 言語の合成サブセットの定義から成り立っている。合成サブセットは、SystemC のどの構文の要素が合成ツールで合成されるべきであるかを定義する。それは少なくとも RT レベルとビヘビア・レベルを含んでいる。また、もっと多くの抽象レベルが次のステップとしてこのワーキンググループで議論されている。このサブセットを完全にサポートする合成ツールは Synthesizable-SystemC 準拠であるとして認めることができる。

コーディング・ガイドラインはハードウェア設計者が、合成可能な SystemC コードを効率的に記述する事を助ける。論理合成のための RTL と高位合成のためのビヘビア・レベルのコーディング・ガイドラインは各設計段階での適用を容易にするために個別に説明されるかも知れない。

次のページの図 1.1 は論理合成と高位合成の両方に関わる設計フローの抽象度の概念図を示す。

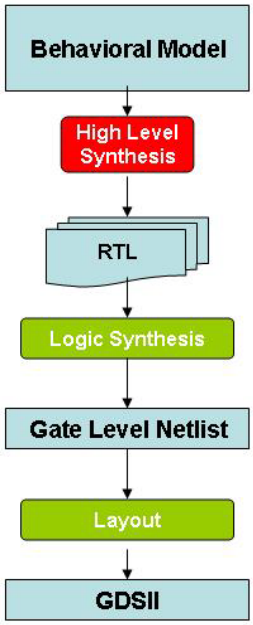


図1.1 論理合成と高位合成の両方に関わる設計フローの抽象度の概念図

このドキュメントはSynthesis作業部会によって議論され合意の上のSystemCの合成サブセットの定義である。
このドキュメントの目的は、ツールでサポートできる最低限の最初のサブセットについて説明することである。
このサブセットを超える構文の合成サポートを制限する意図ではない。

1.3 用語

"shall" (必須)は、この標準を満たすために厳密に守らなければならない要求事項を意味し、例外は許されない。
"shall" (必須)は、"required to" (要求される)と同義である。

"should" (推奨) は、その事項が推奨されることを意味するが、必ずしも要求されないことを意味する。
また、否定形の意味においては、その事項がサポートされなくなるが、禁止されるわけではないことを意味する。
"should"は、"recomended" (推奨される)と同義である。

"may" (許可) は、その事項が標準の制限の範囲において許されることを意味する。
"may" は、"permitted" (許される) と同義である。

"合成ツール"とは、SystemC 構文を正しい(Legal)入力であることを条件の元に受け付けるものである。
"合成ツール"とは、その SystemC 構文を解釈するか、その構文の解釈方法を提供するかにより、ある構文を表現する何かを生成するものである。

"合成ツール"は、許される全ての構文の解釈方法を提供する必要はなく、この標準で定義される解釈方法のみ提供すればよい。

この標準でいう構文は次の項目にて分類される。

Supported: (サポート) "合成ツール" は、その構文を解釈することが必須である。つまり等価なハードウェア表

現に関連づけられるということである。

Extended: (拡張サポート) "合成ツール" は、元々のC++シンタックスの意味を拡張することによりその構文を解釈することが必須である。

Ignored：(無視) "合成ツール"は、その構文を無視することが必須である。その構文があった場合に、合成ツールは失敗してはいけないが、合成結果は、シミュレーション結果と合わないことを"許可"する。
そういった構文が見つかったときに、どのように合成ツールがユーザーに警告するかといったメカニズムに関してはこの標準では定義されない。 "Ignored" 構文は、サポートされていない構文を含むことがあっても良い。

Not Supported：(サポート無し) 合成ツールはこの構文をサポートしない。合成ツールは、その構文があることを期待しておらず、その構文があった場合に、どのようにツールが失敗するかについては定義されない。合成ツールは、その構文があった場合に失敗するかもしれない。しかし、失敗しなければならないというわけではなく、それらの構文を無視するように取り扱うことが許される。

1.4 表記規則

本書では以下の表記規則を用いる。

a) 本標準におけるテキスト本文ではSystemCあるいはC++の予約語を示すのに**ボールド体**を用いる。(例えば、**sensitive**)
b) SystemCサンプルやコードの断片は固定幅フォントで表わされる。
c) 取り消し線の構文テキストはサポートされない構文を示す。（例えば、~~テキスト~~）
d) 下線付きの構文テキストは無視される構文を示す。（例えば、テキスト）
e) 影付きの構文テキストは拡張される構文を示す。（例えば、テキスト）
f) ハイフンを含むローマン体フォントの小文字は統語的な範疇を示すために用いられる。例えば、入れ子にされた-名前空間-指示子
g) 生成規則は左辺、“::=”のシンボル(これは「置き換えることができる」と読む)、右辺で構成される。生成規則の左辺は常に統語的な範疇となる。右辺は置き換え規則となる。生成規則の意味は文字の置き換えである。左辺に出現したものは右辺のインスタンスに置き換えられる。
h) 垂直バー(|)は以下のように置き換える場合に開き括弧の後直後に置かれていない場合を除いて生成規則の右辺を2者択一のアイテムを区切っている。クラスあるいはネームスペースの名前はクラス名かネームスペース名である。式リストは割り当てた式あるいは式リスト、割り当てた式である。このインスタンスでは“クラスあるいは名前空間の名前”の存在はクラス名あるいは名前空間名のどちらかで置き換えられる。２つ目のケースでは、式リストはカンマ(,)で区切られた割り当て式のリストで置き換えられる。
i) 四角ブレース[ ]はオプションアイテムを囲んでいます。このアイテムはゼロ回あるいは1回だけ現れる。よって、以下の2つの生成規則は等しい：

    init-declarator ::= declarator [initializer]

    init-declarator ::= declarator | declarator initializer

j) ”NOTE—“で始まる段落は標準ではなく、参考情報を提供する。
k) “例” の文書で示される例は合成向けのSystemCの構文と記号を明示するためのものである。これは等価なハードウェア表現を生成するのに、より効率である（かあるいはあまり効率がない）コーディングス

タイルの表現や推奨あるいは強調するという基準を意図していない。 加えて、たとえこれらの例が（他の点で注意される場合を除いて）この規範に従っていたとしても、コンプライアンス・テスト・スイート、あるいはパフォーマンスベンチマークに相当する例を示すための基準を意味しているものではない。

1.5 抽象レベル

設計をサポートする抽象レベルについて

システムレベル設計手法のゴールは、第１に設計コストと設計期間を短縮することであるが、最近のシステムは複雑であり、直接的に実装を記述できない。さらに、再利用目的で実装記述の中から機能やアーキテクチャを容易に切り出すことができないため、異なる機能や異なるアーキテクチャを持つ派生実装を行うことが困難である。したがって、システムを設計する場合には、機能やアーキテクチャ、実装を分離することが必要である。また、設計コストと設計期間を短縮するためには、機能とアーキテクチャを結合的に実装する設計作業が必要となる。システム設計フローの中で、機能、アーキテクチャ、実装の分離を可能にするためには、抽象レベルを定義する必要がある。抽象レベルの考え方は、タイミングやデータ粒度のような、実装を徐々に詳細化していく考えに基づく。我々は主な３つの抽象レベル、すなわち機能レベル、アーキテクチャレベル、実装レベルを定義した。各レベルはモデリングビューを持ち、以降のセクションで詳細を説明する。図 1.2 にシステム設計フローにおける抽象レベルを示す。

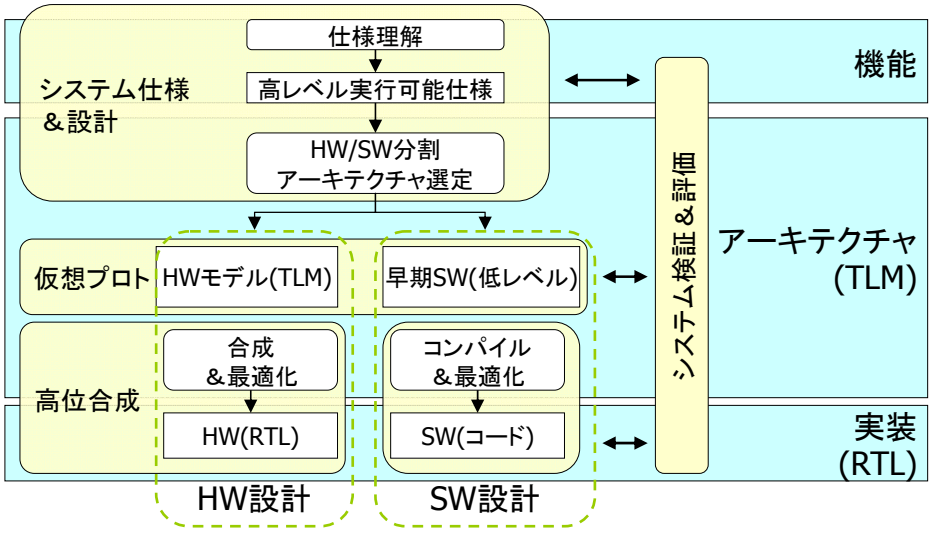


図1.2 システム設計フローにおける抽象レベル

1.5.1 機能レベル

この抽象レベルは、システムに求められている機能を、アーキテクチャを想定することなく素早く決定する目的で用いる。したがって、派生機能を作成したり、さまざまなアーキテクチャを想定して合成を行うなど、機能を再利用する可能性がある。このレベルの例としては、シミュレーションにより解析可能な YAPI[1]内のプロセスネットワークとしてモデリングされる機能がある。

機能レベルには２つのデザインステップがある。

[1] YAPI(Y-chart API)：Kahn Process Network を表現可能な C++ライブラリ。Philips Research が開発し、アプリケーションの実行が可能。

- アルゴリズム仕様
- 通信タスク分割

1.5.1.1 機能レベル:アルゴリズム仕様

この設計ステップでは、アルゴリズムの実行可能な機能仕様が作られる(例えば、C/C++/Matlab のコード)。この実行可能仕様は、アルゴリズムの妥当性判定に用いられる。この設計ステップでのシミュレーションはシーケンシャルで、タイミング情報はなく、スレッド制御は１つだけである。タイミングと詳細なアーキテクチャを持たないので、シミュレーション速度は高速である。
プロファイリング技術は、異なる機能の計算負荷や、機能間の転送データ量の初期評価を得るために用いられる。コードインスペクションは、各機能で要求される柔軟性を評価するために用いられる。コードインスペクションとプロファリングの両方の結果は、タスク分割や後の HW/SW 分割のための入力として用いられる。
このステップで作られる実行可能な機能仕様は、次のアルゴリズム検証以降、フロー全体を通してゴールデンリファレンスモデルとして用いられる。
設計の制約や要求、適したアーキテクチャテンプレート、前のアルゴリズム設計ステップでの結果などから、システムはタスクへと分割される。このタスクは、タスク間でやりとりされるデータを通じて、機能やチャネルの処理を実行する。
そのようなネットワーク内のプロセスは並列であり、通信チャネルにより接続される。プロセスはデータ要素を生成し、双方向の通信チャネルを通して送信される。通信チャネルは送信先プロセスが受け取れるまで、FIFO によりデータ要素を蓄える。このようなネットワークは Kahn プロセスネットワーク(KPN)とも呼ばれる。
KPN では、並列性と通信とが明示的にモデリングされ、これはマルチプロセッサシステム上へマッピングするには不可欠である。KPN のもう１つの特性は、アプリケーション開発者が実行順を定義することなしにネットワーク内へプロセスを結合可能な点である。
この特性のおかげで、既存の資産を用いた新たなアプリケーション生成が容易となり、モジュール生成やアプリケーション(機能 IP)の再利用が進むことになる。
マルチスレッドシミュレーションを用いることで、チャネルにおける通信負荷、タスクにおける計算負荷が解析される。必要であれば、制約や要求を満足するためにシステムを再構成することもできる。また、分割の機能的正確性は、マルチスレッドシミュレーションによりチェックされる。
Kahn プロセスネットワークとして信号処理アプリケーションをモデリングするために、YAPI を用いることができる。YAPI の目的は、信号処理アプリケーションの再利用、およびハードとソフトを含むヘテロシステム上への信号処理アプリケーションのマッピングを可能とすることにある。
YAPI はまた SystemC にも実装されている。SystemC に実装された YAPI は、プロセスネットワークとしてストリーム処理アプリケーションをモデリングするために用いられるルールセットと一緒に、SystemC クラスライブラリとして提供されている。上記の通り、YAPI での計算モデルは KPN に基づいている。

1.5.2 アーキテクチャレベル

アーキテクチャレベルの目的は、早急に効果的な実装方法（電力、タイミング、面積他）を見つけること。
トランザクション・レベル・モデリング(TLM)はシステム(SoC)をアーキテクチャレベルで抽象的にモデリングし、効果的なアーキテクチャを探索するために開発された。
合成サブセットでは、OSCI TLM での定義・用語を用い、OSCI TLM WG で開発されたライブラリのみを利用する。

1.5.2.1 トランザクション・レベル・モデリング

TLM の計算(computation)と通信(communication)のうち、OSCI TLM1.0 と 2.0 で定義している通信部分について、トランザクションは、モジュール間を通るデータ構造（ペイロード）と定義する。モデリング・ビュー（またはモデリングスタイル）は、TLM WG の定義は以下

1. Un-Timed (UT): 時間またはサイクルの明確な記述がないが、並列性と処理順序を記述するモデリングスタイル．イベントやミューテックス、ブロッキング FIFO のような同期機構のような記述は含まない。
2. Loosely Timed (LT): OS のブート処理や、高精度なスレッド間の同期を必要としない場合に、必要最小限のタイミング情報で記述するモデリングスタイル。LT では、タイマーモデルや抽象的なアービトレーション間隔、実行時間単位を含む場合がある。
3. Approximately Timed (AT): モデルと対応するリファレンスとを比べた場合、双方の状態遷移の順序は正しいが正確なタイミング精度では合わないモデリングスタイル。タイミング精度は定義されていない。
4. Cycle Accurate (CA): モデルの実行状態がすべてのサイクルで予測することが可能なモデリングスタイル。すべてのサイクルでモデルと対応する RTL の 1 対 1 対応が可能だが、すべてのサイクルでモデル全体の状態が明確に再評価されたり、すべての外部ピンや内部レジスタの状態を明確に表現される必要はない。サイクルのみを表現するモデル。

UT モジュールは合成可能。LT モジュールは絶対時間を持つため合成不可能。例えば、ある関数のレイテンシを sc_time(10,SC_NS)で表現した場合は、合成不可能。同様に AT モジュールも、状態と状態遷移をタイミングを正確に定義できないため合成不可能。CA モジュールについては現時点で TLM2.0 で定義されていないため、わからない。

1.5.3 実装レベル

この抽象レベルはインタフェースの詳細およびタイミングの完全もしくは部分的な記述/モデリングを含む IO 機能を表す。ブロックの間のコミュニケーションは信号レベルで渡される。インタフェースの記述はピン精度で、トップレベルモジュールにおいて合成後も維持される。実装レベルはレジスタトランスファーレベル(RTL)、ゲートレベルおよび動作レベルを含む。RTL およびゲートレベルは広く利用されており、Verilog/VHDL および SystemVerilog のような HDL 言語で記述されている。ゲートレベルより下の抽象レベルは GDSII フォーマットで表現される。SystemC はこの抽象レベルには適さない。

ゲートレベルはテクノロジーのリーフセルのインスタンスの相互接続から成る。記述は構造である。各セルの動作はシミュレーション用に書かれ、一般にかなり単純である。例えば、組み合わせゲートは通常並行式の形で書かれる。順序ゲートはレジスタおよびメモリを含み、通常 RTL レベルと同じ形で書かれる。

RTL レベルは構造およびさらなる動作記述、両方の記述が可能である。

- ビット精度の論理に加えて、a*bのようなwordレベルの演算の記述と合成が可能である。
- 繰り返しが定数のループを記述できる。そのようなループは完全に展開される。
- 合成がFSMとして認識し、ステートのエンコード等のような最適化できる方法で、有限状態機械(FSM)を記述できる。次のステートおよび出力はif-then-elseやcase文といった動作記述で計算される。
- 有限状態機械で、状態および遷移は複雑な論理および(出力への単純な定数代入ではない)演算動作を含むことができる。これはexplicit-state machineと呼ばれる。レジスタはexplicit-state machineの中でも外でも記述できる。

RTL サブブロックのインタフェースは合成によって変わることがある(境界最適化)、しかしトップレベルのインタフェースは保持される。クロック、リセット、そして enable 動作は明確に指定される。オペレーションの内部サイクルタイミングはユーザー制御の下、限られた方法(retiming)で変えることがある。

リファレンス RTL の記述に対する RTL 合成結果の検証方法は、組み合わせおよび順序回路の両方に対し十分に定義されている。例えば IEEE Standard 1076-2004 は VHDL のこれを定義している、SystemC RTL 記述でも同じ方法は適用できる。

動作レベルは、部分的なサイクル毎の IO 動作制約のみから、オペレーションと IO がどのようにスケジュール

されるかにおいてある程度の自由度を持つ。　レジスタは明示的に定義されない代りに合成で定められる。　記憶素子の必要性はオペレーションがどのようにスケジュールされるか依存する: レジスタは生成されたサイクルより 1 つ以上後のサイクルで使用される値の保持に使用される。　配列の記憶素子はメモリまたはレジスタにマップされる。動作の記述は一般には RTL で用いる明示的なステートマシンよりむしろ暗黙のステートマシンの形になる。暗黙のステートマシンでは、次にどの動作が実行されるかの選択に使われる明確な状態変数がない。その代り、動作はクロックおよびリセット信号(非同期リセットのための)にセンシティブなプロセスから成る。プロセスはループのような言語構造、continue やループ exit の構造、条件付き動作(if-then-else や case 構造)それに出力代入のセット間のサイクルタイミングを記述する wait 文が使われる。
動作合成からの出力は合成可能な RTL 記述かつ/もしくはゲートレベルの記述である。動作の記述(ソース)に対し、生成された記述の検証手法は合成によってサイクル毎の動作が変えられるため(ゲートレベル記述に対する RTL レベルよりも)複雑である。
実装レベルの議論を導くため、式 Y=P(X)でターゲットプロセスを表す、ここで P はターゲットプロセスの機能である。X={x1, x2, …, xn, v1, v2, …, vm}が入力信号のセットで、xi, 1≦i＜n はセンシティビティリストの信号、vj, 1≦j≦m がセンシティビティリストでない信号である。そして、Y = {y1, y2, …, yl}は出力信号のセットである。

. *例*:

```
SC_MODULE( AddMul_1 ) {
    sc_in< sc_clock > clk;
    sc_in< sc_uint<16> > a, b, c;
    sc_out< sc_uint<32> > result;
    void addmul_1() {
        result = a + (b * c);
    }
    SC_CTOR( AddMul_1 ) {
        SC_METHOD( addmul_1 );
        sensitive << clk.pos();
    }
};
```

上記の例で P は関数 void addmul_1(); X = {x1, v1, v2, v3}, x1 = clk, v1 = a, v2 = b and v3 = c; Y = {y1}, y1 = result である。

1.5.3.1 動作レベル

動作レベルは、[De Micheli 94、Knapp 96]で呼ばれたように機能レベルまたは動作アーキテクチャレベルとしても知られる。動作レベルでは、式 Y = P(X)も X や Y も時間を含まない。X の xi の値が変化したらすぐに P は Y を計算する。UT と違って、X の空でないサブセットだけがセンシティビティリストになりうる。仮想クロックが唯一、もしくは X 中の xi のどれか一つとなりうる。仮想クロックがプロセスをトリガーした時、P は同時に X に基づき Y を計算する。

. *例*:

```
SC_MODULE( AddMul_2 ) {
    sc_in< sc_uint<16> > a, b, c;
    sc_out< sc_uint<32> > result;
    void addmul_2() {
        result = a + (b * c);
```

```
    }
    SC_CTOR( AddMul_2 ) {
        SC_METHOD( addmul_2 );
        sensitive << a << b << c;
    }
};
```

上記の例でプロセス関数 P は addmul_2 (); 入力セットは X = {x1, x2, x3 },x1 = a, x2 = b, x3 = c; 出力セットは Y = {y1}, y1 = result である。

. *例*:

```
SC_MODULE( AddMul_3 ) {
    sc_in< sc_clock > clk;
    sc_in< bool > rst;
    sc_in< sc_uint<16> > a, b, c;
    sc_out< sc_uint<32> > result;

    void addmul_3() {
        result = 0;
        wait();
        while (1) {
            result = a + (b * c);
            wait();
        }
    }
    SC_CTOR( AddMul_3 ) {
        SC_CTHREAD( addmul_3, clk.pos() );
        reset_signal_is(rst, false);
    }
};
```

上記の例でプロセス関数 P は void addmul_3()と SC_CTHREAD の意味するところと reset_signal_is ()の組合せである。なお clk と rst ポートはセンシティブ・イベントとして、clk はクロックポート、rst はリセットポートして認識される。入力セットは X = {x1, x 2, v1, v2, v3}, x1 = clk, x2 = rst, v1 = a, v2 = b, v3 = c である。出力セットは Y = {y1}, y1 = result である。 SC_CTHREAD および reset_signal_is()の意味論は後のセクション 9.4 に記載される。

1.5.3.2 レジスタトランスファーレベル

レジスタトランスファーレベル(RTL)は、名前が示唆するように、レジスタからレジスタへの関数や信号を記述する。このレベルの基本要素は組み合わせおよび順次関数/論理ユニット、レジスタおよび信号である。レジスタを含まない物理的な回路は RTL ではモデリングできないことに注意したい。それはゲートレベルもしくはそれ以下でのみモデリングできる。

RTL では:

1. 時間は必ずしもサイクル単位でなくてもよい。場合によっては、ピコ秒またはナノ秒のような時間になる。

2. 時間はY = P(X)の暗黙の要因である。 時間は入力とも出力信号ともみなされない。その代り時間はYを計算するPに組み込まれる。Pは加算、乗算、減算、and、orといった各単位オペレーションがどれくらいの時間を使い、Yを計算するのにどれくらいの時間を使うかの評価することを暗に記述する。

3. 物理的なクロックポートはセンシティビティのリストにある。

RTL モジュールはターゲットモジュールのサイクル毎の動作を記述する有限状態機械(FSM)備えている。各ステートのファンクション動作は FSM の中で記述することができる。この種類の FSM を FSM with Datapath(FSMD)と呼ぶ。またはファンクション動作は FSM によって制御される別の datapath で記述される。

. *Example*:

```
SC_MODULE( AddMul_4 ) {
    sc_in< sc_clock > clk;
    sc_in< bool > rst;
    sc_in< sc_uint<16> > a, b, c;
    sc_out< sc_uint<32> > result;

    void addmul_4() {
        sc_signal<sc_uint<32> > tmp1;

        tmp1 = 0;
        result = 0;
        wait();
        while (1) {
            tmp1 = b * c;
            wait();
            result = a + tmp1;
            wait();
        }
    }

    SC_CTOR( AddMul_4 ) {
        SC_CTHREAD( addmul_4, clk.pos() );
        reset_signal_is(rst, false);
    }
};
```

上記の例でプロセス関数 P = void addmul_4 ()が if(rst==false){}ブロックといったリセットステートを持ったサイクル毎の FSMD であり、while(l){}ブロックといった 2 -ステートのレジスタ毎の計算ユニットである。そしてレジスタはステートレジスタと tmpl を受け継ぐ。サイクル時間は SC_CTHREAD、reset_signal_is()そして wait()文の結合された意味論で暗黙に記述される。クロックポートｃｌｋは SC_CTHREAD()によりセンシティブ入力として記述されている。

1.5.3.3 ゲートレベル

このレベルで基本的な要素は AND、OR、NOT、XOR などといった論理ゲートと、ゲートに接続された信号である。このレベルでモデリングするには、最初にゲートをインスタンスできるよう論理ゲートのライブラリが必要になる。そして論理ゲートをつなげる適切な信号を宣言する。

. *Example*:

```
SC_MODULE( AND2 ) {
    sc_in< sc_uint<1> > a, b;
    sc_out< sc_uint<1> > c;

    void and2() {
        c = a & b;
    }
    SC_CTOR( AND2 ) {
        SC_METHOD( and2 );
        sensitive << a << b;
    }
    };
    SC_MODULE( OR2 ) {
        sc_in< sc_uint<1> > a, b;
        sc_out< sc_uint<1> > c;

        void or2() {
            c = a | b;
        }

        SC_CTOR( OR2 ) {
            SC_METHOD( or2 );
            Sensitive << a << b;
        }
    };
    SC_MODULE( ANDOR ) {
        sc_in< sc_uint<1> > a, b, c;
        sc_out< sc_uint<1> > d;
        void andor() {
            AND2 *andgate;
            R2 *orgate;
            sc_signal<sc_uint<1> > wire1, wire2, wire3, wire4;
            andgate = new AND2("andgate");
            orgate = new OR2("orgate");
            a(wire1);
            b(wire2);
            andgate->a(wire1);
            andgate->b(wire2);
            andgate->c(wire3);
            orgate->a(wire3);
            orgate->b(wire4);
```

```
        d(wire4);
        while (1) {
            wait(); // forever loop
        }
    }
    SC_CTOR( ANDOR ) {
        SC_THREAD( andor );
        sensitive << a << b << c;
    }
};
```

上記の例で、最初に ANDOR モジュールでライブラリの部品として使えるよう、AND2 と OR2 ゲートを作る。興味深いのは、ANDOR のモデリングスタイルが AND2 と OR 2 のそれと異なっている点である。ANDOR ではゲートワイヤがインスタンスされ接続されている。void andor()には計算や制御の記述はない。しかしながら AND2 と OR2 で、モデリングスタイルは動作レベルと合っている。そのようなプリミティブな部品のために、動作の記述が実際にゲートレベルにもある。

1.6 ESL 合成

1.6.1 イントロダクション

この章で述べられる申し合わせは非プログラマブル RTL コアの合成ソリューションに焦点を当てている。
高位合成（動作合成）すなわち第 2 レベルの ESL 合成は、動作記述もしくは高レベル・モデルの RTL 実装への変換と最適化の自動化により、高抽象度での設計を可能にする。これはアンタイムドもしくは部分的にタイムドな機能モデルを全タイムドな RTL 実装へ変換する。
マイクロ・アーキテクチャは自動生成されるので、設計者はモジュールの機能の設計と検証に集中することができる。既存の RTL 設計メソッドのような回路リソースの完全なスケジュールと割当ての必要がなくなるので、設計チームはより短期間で設計・検証できる。この動作レベル設計フローは設計効率を向上し、エラーを削減し、検証をスピードアップする。図 1.3 に論理合成と高位合成が関わる設計フローの概略を示す。

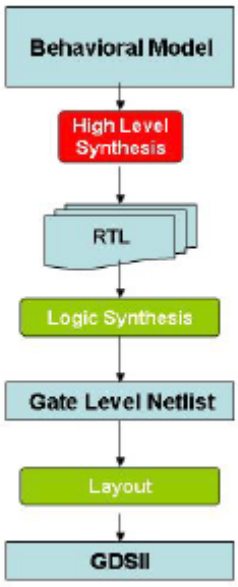


図 1.3 概略フロー

典型的な高位合成プロセスは幾つかの複雑なステージから成る。このプロセスは I/O 動作と計算機能を含んだモ

ジュール動作の高位言語記述からスタートする。結果の複雑さを低減するため幾つかのアルゴリズム最適化が実行され、記述は基本的な演算と演算間のデータフロー依存関係について解析される。
高位合成プロセスへの入力は他に、ターゲットのテクノロジ・ライブラリ（選択した製造プロセスのターゲット・クロック周期でキャラクトライズされる）、結果として得られるアーキテクチャに影響するディレクティブのセットが含まれる。 ディレクティブは、例えば、ツールのアルゴリズムで使われるタイミング制約で、これは演算をサイクル毎にスケジュールするために使われる。キャラクトライズされたライブラリは、演算を加算器・乗算器・比較器等の機能ユニットへ割当てるため、アロケーションとバインディングに使われる。
最後に、所望の機能を実装するデータパスを制御するステートマシンが生成される。データパスとステートマシンは論理合成ツールまたはフィジカル合成ツール用に最適化された、RTL コードとして出力される。

1.6.2 ビジョン

プロジェクト・チームが ESL 設計へ移行する幾つかの理由は、システムの複雑さのより容易な管理、設計の実装と検証の加速、設計再利用の機会増加、実装の選択肢の広さ。しかし、高抽象度への移行によって ESL と RTL の設計ギャップが出現してきた。
抽象的なレベルでモデル化されたアルゴリズムをハードウェアで実現する方法は数多くある。しかしながら、厳しいスケジュールと複雑さの増加により、アルゴリズムとアーキテクチャが決まった後に 2 つ以上の RTL 実装を手設計する十分な時間はない。性能・回路規模・消費電力に重大なインパクトを与えるかもしれないハードウェア実装の代案はめったに作られないし評価されない。
SystemC のような C++をベースとして汎用言語が使えるようになり、また、高位合成ツールと高位検証ツールの成熟により、高位レベル・モデルが今まではできなかったアーキテクチャとアルゴリズムのトレードオフの評価の助けになるようになってきた。まとめると、高位合成が ESL と RTL フローのギャップを埋めることで、高位設計への移行を可能にしている。

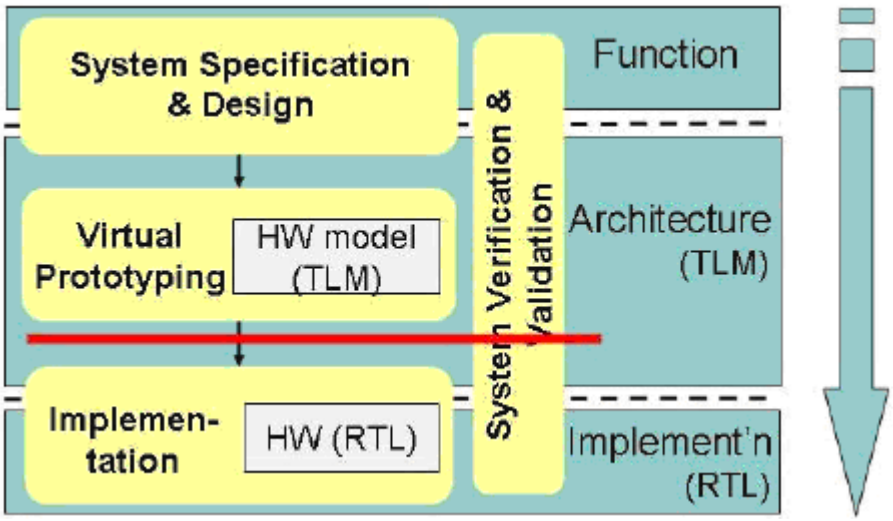


図 1.4 アーキテクチャ（TLM）から実装（RTL）までのビジョン

図 1.4 にアーキテクチャから RTL への合成にビジョンを示す。アーキテクチャのコンポーネントは合成可能な SystemC で記述され、TLM ライブラリを使ってインタフェースが書かれ、接続されている。前述したように、HLS は ESL 合成の第 2 階層であり、第 1 階層はアーキテクチャ合成である。アーキテクチャ合成は TLM ライブラリを使って記述されたアーキテクチャを読み込み、メモリ構造、キャッシュ構造、DMA 構造、バス階層、I/O デバイス、CPU 選択等を決める。アーキテクチャ合成と HLS がいっしょになって ESL 合成を構成する。TLM セマンティクスを合成することがアーキテクチャ合成への第一歩であるが、TLM ライブラリの合成サブセットを定義することは合成 WG の将来の作業である。

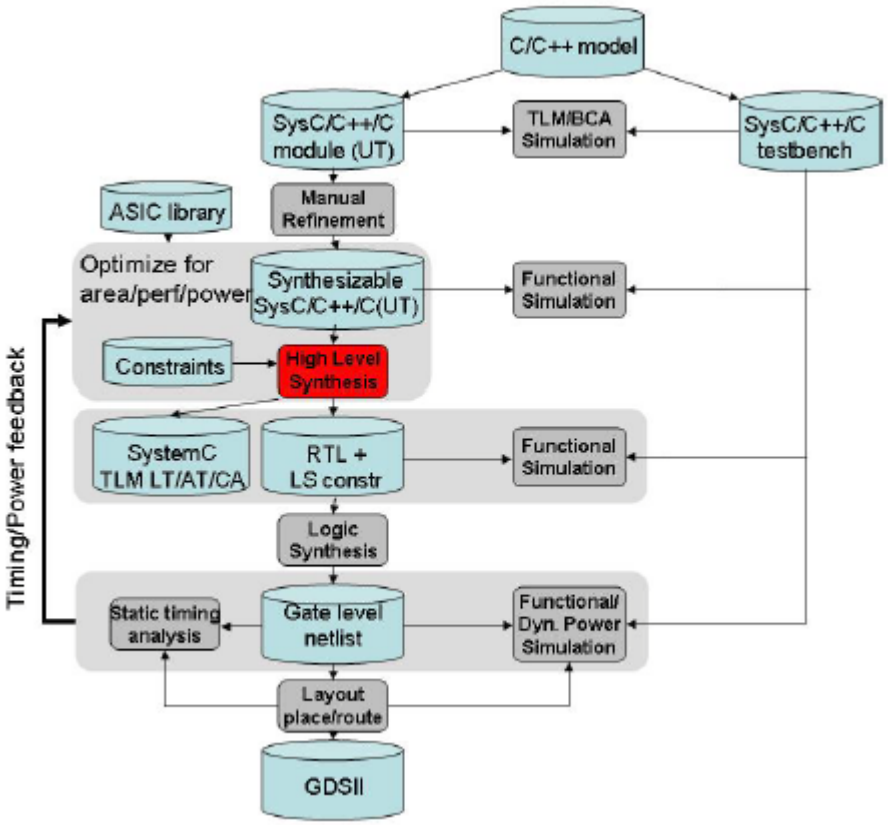


図 1.5 ESL から GDS II までの可能性のある実装フローの抽象ビュー

図 1.5 に ESL 合成を組み入れた ESL から GDS II までの可能性のある実装フローを示す。この図をベースとした ESL 合成ソリューションへの欲しいものリスト、C/C++/SystemC をベース、を以下のように定義できる：

- 動作 C/C++/SystemC モデル（TLM）からの RTL 実装自動化
- 速い設計時間（実装と検証）のサポート
  - 手設計と比較して少なくとも 2 倍改善
- 性能、回路規模、消費電力の最適化のサポート
  - 性能要求を満たして回路規模は人手設計と比較して同等か改善。
  - バック・アノテーション型静的タイミング解析に基づいた自動タイミング・クロージャー
  - 動的消費電力シミュレーションをベースにした自動消費電力最適化
- タイミング・アノテーションを持った動作 SystemC TLM モデル（LT/AT）と SystemC CA モデル生成のサポート
- RTL 実装とネットリストの検証への高位テスト環境と生成された SystemC TLM ビューの再利用の自動化
  - アサーションのサポート
  - ファンクショナル等価性確認のサポート
- TLM 2.0 準拠モデルからの合成とモデル生成のサポート
- システムレベル設計環境への高位合成手法の完全な統合

4.2.5 JEITA ISSUE LIST for “SystemC Synthesizable Subset 1.3 draft”

以下の表は、OSCI Synthesis Forum へ 2010 年 1 月 29 日に提出された“SystemC Synthesizable Subset 1.3 draft”のレビュー結果である。

JEITA Issue #	Clause/ Subclause	Title	Type of Comment (Technical/Editorial	Comments(Justification)	Proposed change
1	Contents		Editorial	There is a numberring miss for Annex.	Change “ANNNEX C GLOSSARY” to “ANNEX B GLOSSARY”.
2	1	Overview	Editorial	The terms of ”high level synthesis” and ”behavioral synthesis” are used. The defference is not clear. There is only ”behavioral systheis” at Annex B Glossary.	
3	1.3	Terminology	Editorial	The words ”shall”, ”should” and ”may” is defined. But the word ”must” is used in section 2.	Change the word ”must” to ”shall”
4	1.4	Conventions	Editorial	Boldfase is used for the reserved words at section 2, 3, 5, 8, 13, 15, 17, 18 and AnnexA, but boldface is not used for the one at section 1, 4, 7, 9. Examples at section 3.2 and 18 also have boldfase words. The paragraph from c) to k) is useful only for Annex A.	Change the font for the reserved words at section 1, 4, 7, 9. and examples, to boldfase. Add some explanations before the paragraph from c) to k) like ”The conventions from c) to k) are used at Annex A.”

5	1.5.3.1	Behavioral Level	Editorial	The first example makes the image of a combinational logic like the following code:
				
				module AddMul_3(a,b,c,result);
				input unsigned wire[15:0] a, b, c;
				output unsigned wire[31:0] result;
				
				always @(a or b or c)begin
				result = a + (b * c);
				end	

6	1.5.3.2	Register Transfer Level	Technical	First example of RTL is better to use clearly the state machine and the registers.
				
				The register tmp1 does not need to be sc_signal in the example.
				The example get a result each two clocks. It is not efficient.
					Change the process and the constroctor in the example:
					
					sc_signal<sc_uint<32> > tmp1;
					void addmul_4(){
					if(!rst.read()){
					tmp1 = 0;
					result = 0;
					}else{
					tmp1 = b.read() + c.read();
					result = a.read() + tmp1.read(); // The value of tmp1.read() is the one set before one cycyle.
					}
					}
					SC_CTOR(AddMul_4){
					SC_METHOD(addmul_4);
					sensitive << rst.neg() << clk.pos();
					}
					
					or
					
					sc_signal<sc_uint<32> > tmp1;
					void addmul_4(){
					tmp1 = 0;
					result = 0;
					while(1){
					wait();
					tmp1 = b.read() + c.read();
					result = a.read() + tmp1.read(); // The value of tmp1.read() is the one set before one cycyle.
					}
					}
					SC_CTOR(AddMul4){
					SC_CTHREAD(clk.pos());
					reset_signal_is(rst, false);
					}

7	1.5.3.3	Gate Level	Technical	It is better for gate level coding style to follow section 10.	
8	3.1.2.2	Signals	Editorial	There is no definition for sc_signal.	Add the sentence ”Signals can be declared using sc_signal
9	3.1.2.4	Module constructor	Technical	Better to use "int" rather than "unsigned char", in template part of example 2.	Change to ”template< int N = 2 >”.
10	3.2	Deriving	Technical	There is watching sentence in the	Change to ”reset_signal_is(reset, true)”.
11	4.1.1	Integer Types	Editorial	The types of ”short”, ”long”, ”long long” are missed.	
12	4.4.1	Integer Types	Editorial	Inadequate note about 'compile flag _32BIT_'.	Remove the sentence of "The compile flag _32BIT_ is not supported as it is not even mentioned in the LRM.".
13	4.4.1	Integer Types	Editorial	Inadequate note about 'compile flag	Remove the sentence of "(note: LRM does not mention this limit).".
14	4.4.1.4	Shift Operators	Technical	Regarding the 'negative shift value', there is no consistency between integer type and fixed point data type. (even not described in LRM) integer : ?? fixed point : negative shift value is allowed (in reference simulator)	IEEE 1666 has no description about the behavior with negative shift value. Need to revise LRM first. Then synthesizable subset follows it.
15	4.4.1.4	Shift Operators	Technical	Also, the negative shift value in sc_bigint/sc_biguint is not defined in IEEE 1666. (Inconsistency to sc_int/sc_uint.)	IEEE 1666 has no description about the behavior with negative shift value for sc_bigint/sc_biguint.. Need to revise LRM first. Then synthesizable subset follows it.
16	4.4.1.6	Bit Select Operator	Technical	Inconsistent behaviors when out of range value is specified. ([0,W-1]) sc_bigint/sc_biguint : accept out of range value (in simulator) sc_int/sc_uint : not accept	IEEE 1666 defines out of range value is not allowed. 7.2.5 Bit-select It shall be an error if the specified bit position is outside the bounds of its numeric type or vector object. So that, synthesizable subset should follow it.
17	4.4.1.7	Part Select Operator	Technical	Inconsistent behaviors when reverse order value is specified. sc_bigint/sc_biguint : accept reverse order (in simulator) sc_int/sc_uint : doesn't accept	”IEEE 1666 defines reverse order is not allowed. 7.2.6 Part Select It shall be an error if the left-hand index position or right-hand index position lies outside the bounds of the object. NOTE 1-A part-select cannot be used to reverse the bit-order of a limited-precision integer type. So that, synthesizable subset should follow it.”

JEITA Issue #	Clause/ Subclause	Title	Type of Comment (Technical/Editorial	Comments(Justification)	Proposed change
18	5.1.1.9	SystemC type specifiers	Technical	'sc_bit' is deprecated in IEEE 1666.	Delete 'sc_bit'.
19	5.1.1.9	SystemC type specifiers	Technical	Unnecessary description (...) in the bottom of Page 30.	Remove (…) in the bottom of Page 30.
20	5.2.5	Initializers	Technical	It is unclear whether the other classes than module class are supported or not.	Clearly state whether the other classes than module class are supported or not.
21	5.2.5.2	Lexical conventions	Technical	'string' should be treated as an array of 'char' type.	Remove 'struck-through' on 'string'. (Make 'string' usable in literal.)
22	8.3	Wait statement	Technical	Thread process SC_THREAD is not supported, as described in 9.3 SC_THREAD.	Remove description.
23	8.7	Jump statement	Technical	Missing 'goto' clause.	Add 'goto' clause.
24	12.1.4.1	Static members	Editorial	Should refer chapter 7, static member	Add reference to chapter 7, static member function.
25	12.4.1	Constructors		There is watching sentence in the	Change to "reset_signal_is(rst, true)".
26	12.4.7	Construction and destruction	Technical	Unclear description for 'destruction'.	Improve description for 'destruction'.
27	A.2	Lexical conventions	Technical	All the sc_object should accept the 'string' as a name. In other way, 'string' can be converted to the constant.	Remove 'struck-through' on 'string'. (Make 'string' usable in literal.)
28	A.3	Basic concepts	Editorial	Missing 'Shadow' for 'sc-main-definition'.	Add 'Shadow' for 'sc-main-definition'.
29	A.4	Expressions	Technical	Destructor should be supported.	TBD
30	A.6-1	SystemC Type Specifiers	Technical	'sc_bit' is deprecated in IEEE 1666.	Delete 'sc_bit'.
31	A.7	Declarators	Technical	Declaration of the pointer for static memory should be supported.	Add : (Make these usable) * [cv-qualifier-seq] [::] [nested-name-specifier] * [cv-qualifier-seq]
32	A.8.1	Module declaration	Technical	The definition of sc-signal-declaration is not correct. There is the definition for sensitive_pos and sensitive_neg. There is the definition for watching.	Change the definition of sc-signal-declaration to the following: sc-signal-declaratin ::= sc-signal-key < type-specifier > signal-declarator-list ; | sc-clock-key signal-declarator-list ; | sc-resolved-key signal-declarator-list ; | sc-resolved-vector-key < constant-expression > signal-declarator-list; Add the following definition: sc-signal-key ::= sc_signal | sc_in | sc_out | sc_inout sc-clock-key ::= sc_in_clk | sc_out_clk | sc_inout_clk Remove sensitive_pos and sensitive_neg. Replace the definition of sc-watching-statement to the following definition of sc-reset-signal-statement: sc-reset-signal-statement ::= reset_signal_is(sc-edge-event , boolian-literal); sc-edge-event ::= identifier.pos() | identifier.neg()

4.2.6 SystemC Japan 2009 アンケート報告

SystemC Working Group,
EDA-TC/JEITA
July 10, 2009

 JEITA

SystemC Japan 2009の概要

- 主催：
  - コーウエア㈱
  - 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㈱
- 協賛：
  - アーム㈱
  - NECシステムテクノロジー㈱
  - ㈱礎デザインオートメーション
  - ㈱エッチ・ディー・ラボ
  - ㈱沖ネットワークエルエスアイ
  - ㈱プライムゲート
  - コ・フルエントデザイン
  - 大日本印刷㈱
  - 日本イヴ㈱
- 日時：2009年7月10日　12:30～19:20
- 会場：新横浜国際ホテル(定員300名)
- 参加者数：293名(申し込み者数：400名)

 JEITA

SystemC Japan 2009の概要

- 講演内容：
  - 開会のご挨拶「SystemC Community Update」
    - Open SystemC Initiative Executive Director and Board Member Patrick Sheridan
  - 基調講演 「TCTモデルによるマルチコアSoC統合最適化設計プラットフォーム」
    - 東京工業大学 大学院理工学研究科 集積システム専攻 准教授 一色 剛
  - 「富士通マイクロエレクトロニクスにおける上流設計手法の適用事例」
    - 富士通マイクロエレクトロニクス㈱ 共通技術本部 設計共通技術統括部 第一設計部 プロジェクト課長 中村 和正
  - 「SystemC TLM2.0モデルの効率的な設計手法」
    - メンター・グラフィックス・ジャパン㈱ テクニカル・セールス本部 シニアアプリケーションエンジニア 牧野 真
  - 「Cynthesizerによる設計事例と新機能」
    - フォルテ・デザイン・システムズ㈱ シニア・アプリケーション・エンジニア 桜井 至
  - 「画像処理システムのPlatform Architectへの展開」
    - 東芝情報システム㈱ 第二LSIソリューション事業部 第九LSI設計センター主任 小片 亮
  - 「マルチコアHW,SW向け先進SystemCソリューション」
    - コーウエア㈱ 技術本部 スタッフ・アプリケーション・エンジニア 内田 憲法
  - 「仮想プラットフォームを用いた通信用SoC開発事例」
    - ㈱リコー 研究開発本部 基盤技術研究センター 第三研究室 研究主任 木村 貞弘

 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把握
  - 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。

 JEITA

SystemC Japan 2009　サマリー

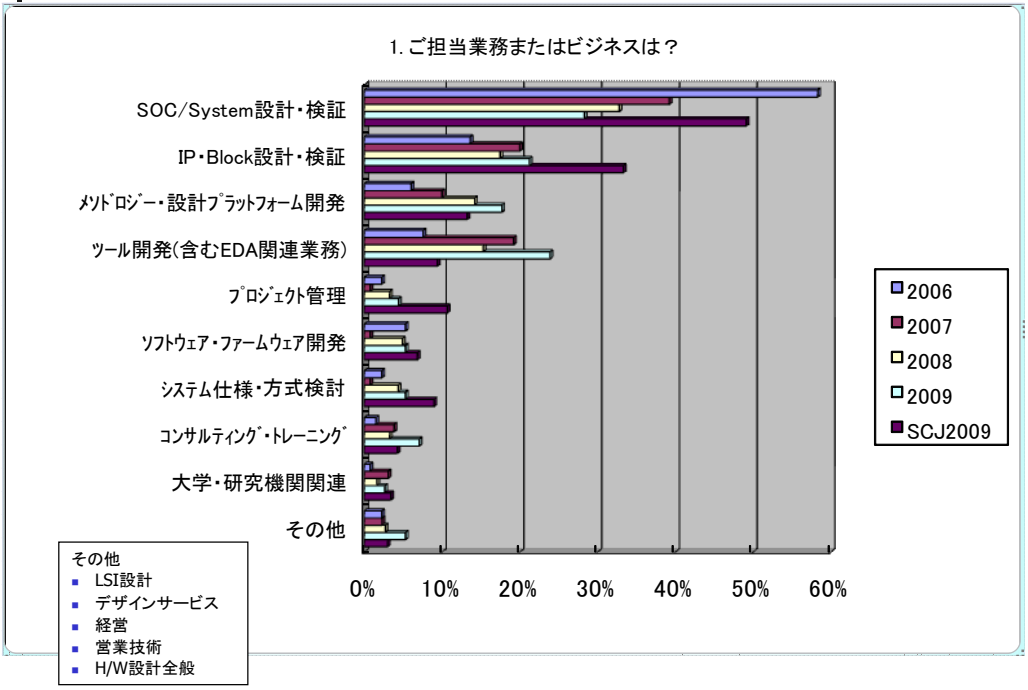
- 講演内容について：
  - SystemC Community Update　としてPatrick Sheridan 様からはTLM2.0 LRMなどOSCIの活動状況について説明がなされた。
  - 基調講演はSW/HWの並列化に関する”Tightly-Coupled Thread Model”の提案について説明されたが、DAC２００９で発表とのことで概要的な説明のみになった。
  - ユーザ事例としては富士通マイクロエレクトロニクス（FML)の中村様よりTLM2.0の適用事例、東芝情報システムの小片様及びリコーの木村様よりベンダーツールを利用した事例の講演がなされた。TLM2.0の利用は3件中1件（FML)であった。
  - ESLツールベンダーからはメンターの牧野様、フォルテの桜井様、コーウエアの内田様よりTLM2.0や合成のベンダーソリューションの講演がなされた。

アンケート調査集計結果

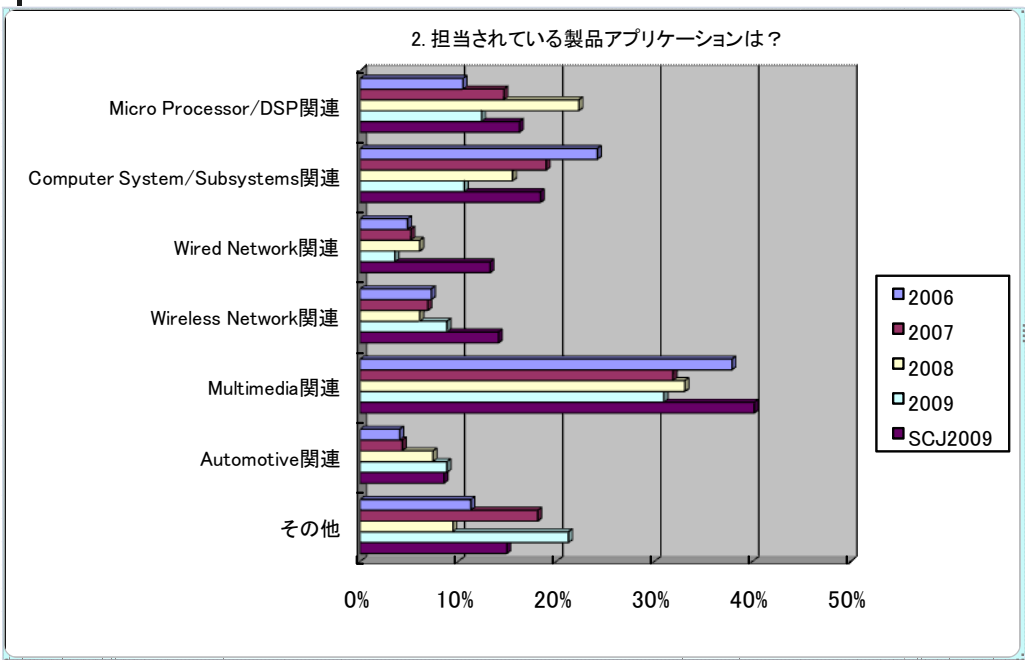
- SystemCユーザフォーラムでのアンケートとほぼ同様の内容（但し一部削除）でアンケートを実施し、今までの結果と合わせて聴講者の動向を分析しました。
  - SystemCユーザフォーラム（有料）とSystemC Japan（無料）のアンケートの結果、有料無料に関係なくほぼ同じ傾向が伺える。
  - 受講者はシステム及びIPの設計検証関係者が多いことには変わらないが、ネットワーク関係LSI開発者が増加する傾向にある。
  - e言語は減少傾向にあり、SystemVerilogが増加傾向にあるが、SystemC/C/C++のユーザが半数を占める。
  - SysmteCのハードウエア設計への利用傾向が増加している。
  - 高位合成の要求は多い。
  - LTはまだ浸透していない。
  - OSCI　APIの利用が増えている。特にTLM2.0利用が増加している。（ベンダAPIが14%、TLM1.0が9.5%、TLM2.0が25%の割合）
  - 事例調査の目的の方が多く、SystemCの標準化への注目は減っている。TLM2.0により標準化は定着しつつある。

今後もSystemCに関する調査は継続的に実施する必要がある。

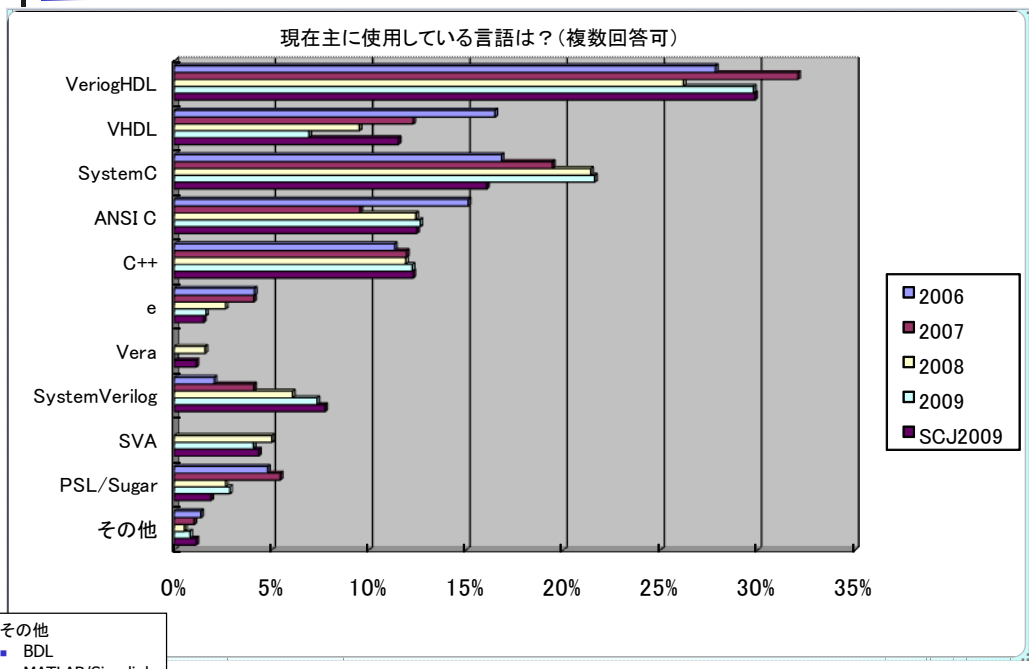
①　ご担当業務またはビジネスは?



②　担当されている製品アプリケーションは?

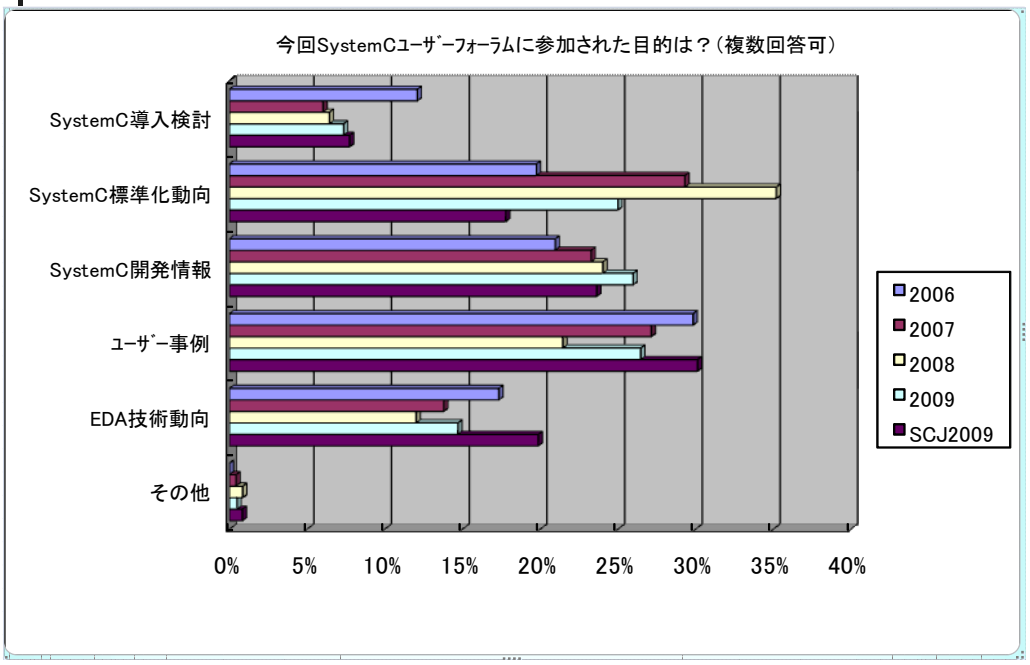


③　現在主に使用している言語は?



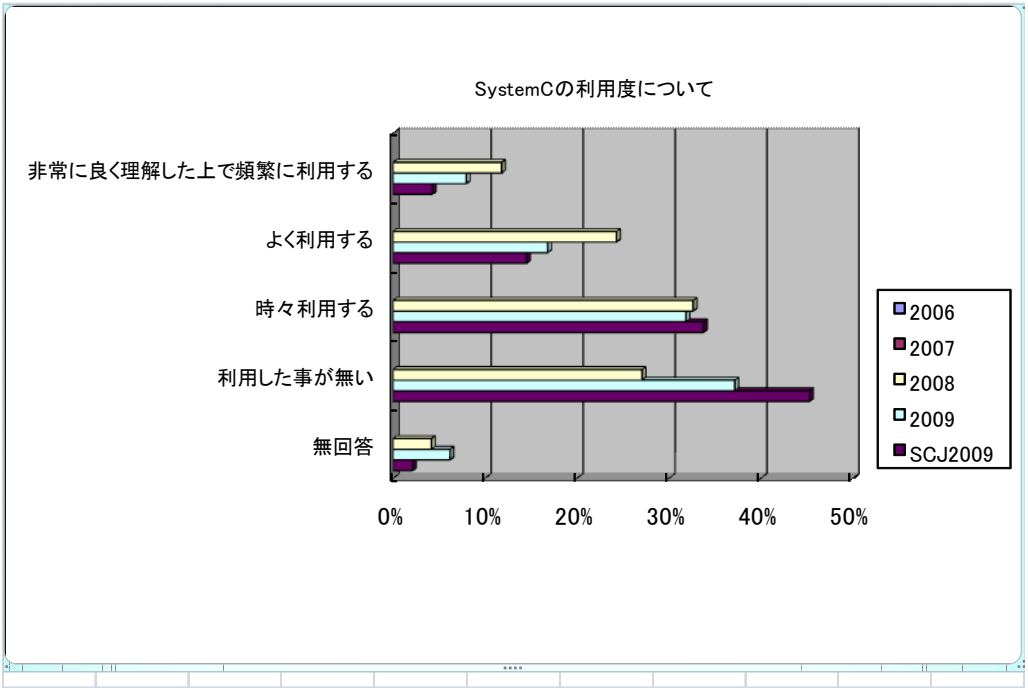
JEITA

④SystemCユーザフォーラムに参加された目的は?



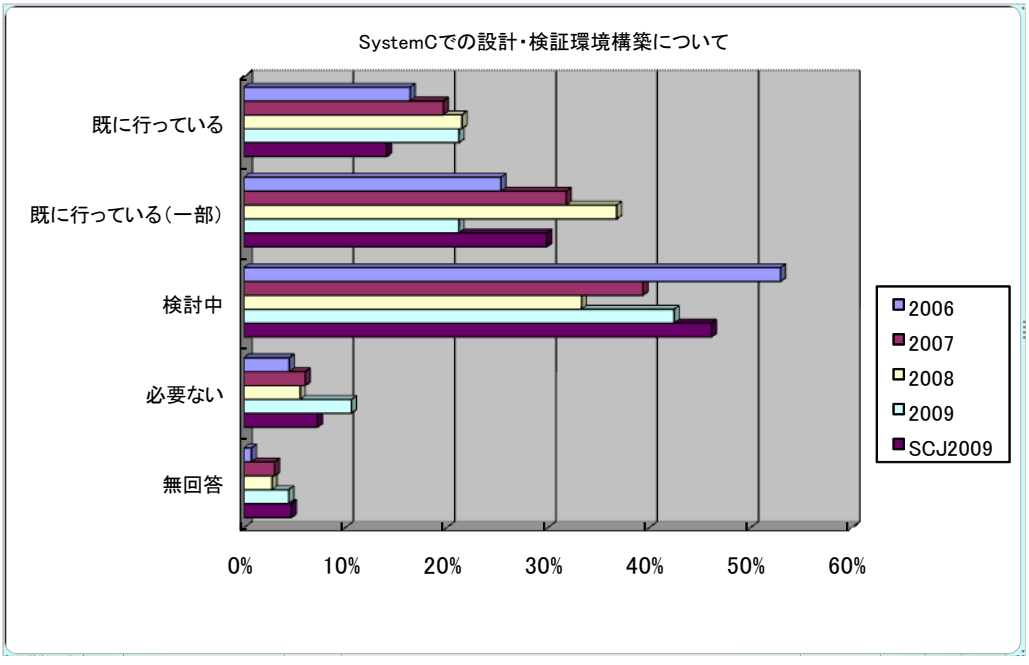
JEITA

⑤　SystemCの利用度について



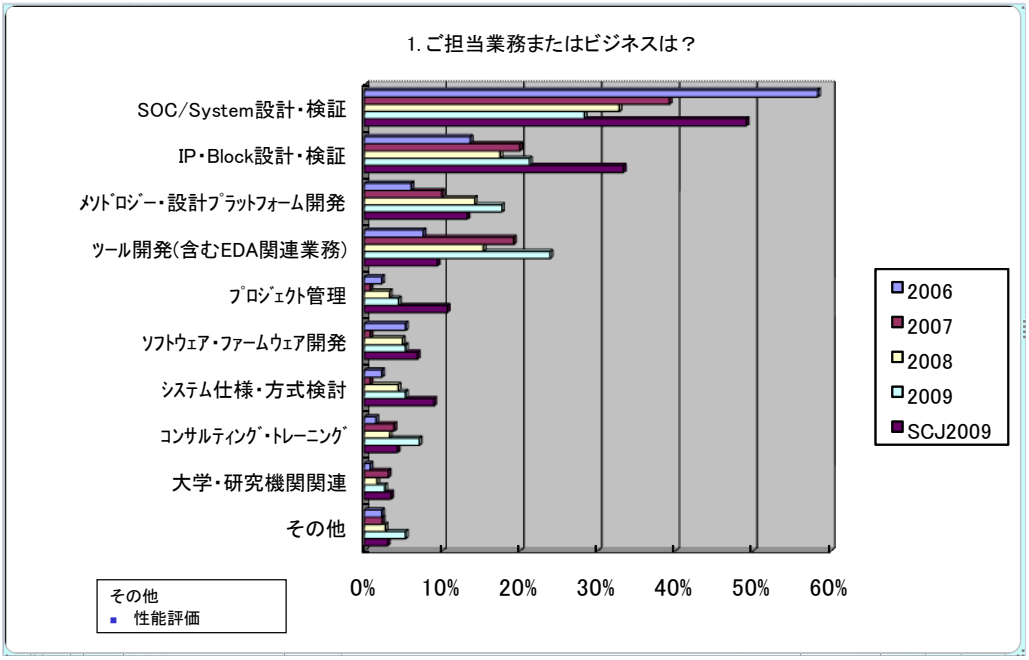
JEITA

⑥　SystemCでの設計･検証環境構築について



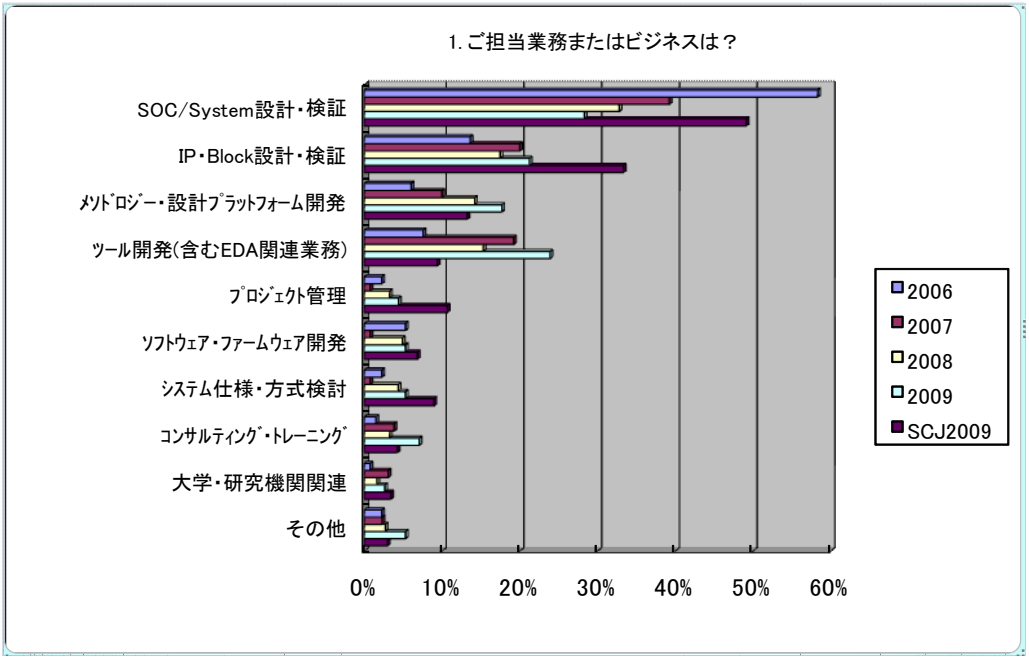
JEITA

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?



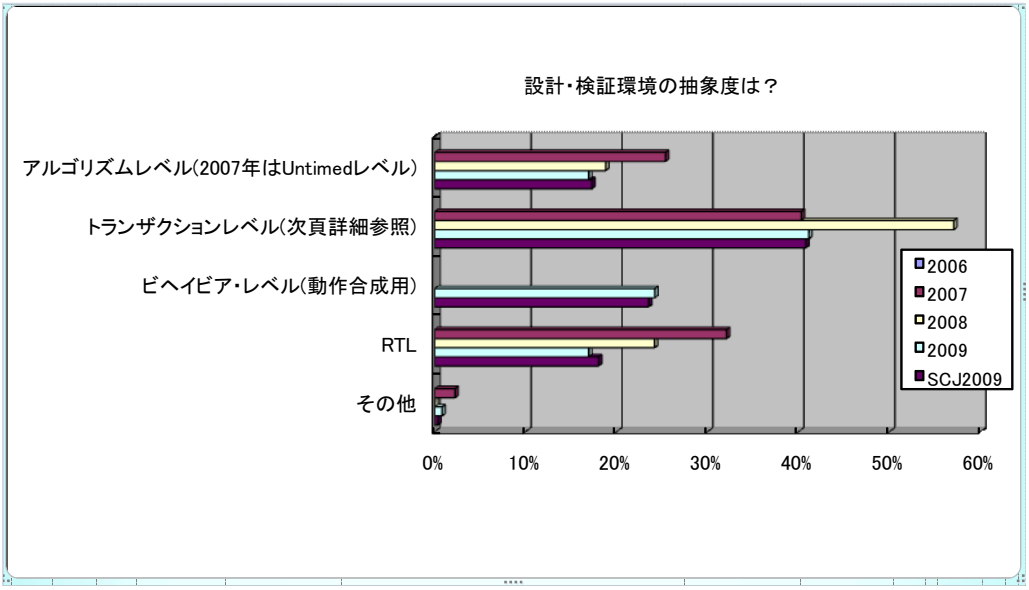
JEITA

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?

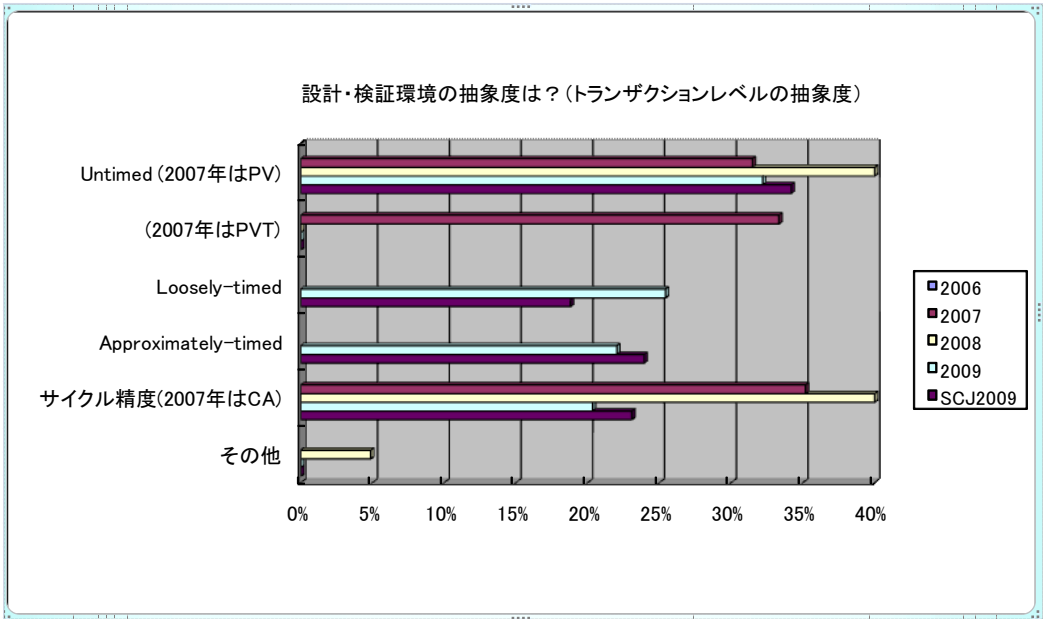


JEITA

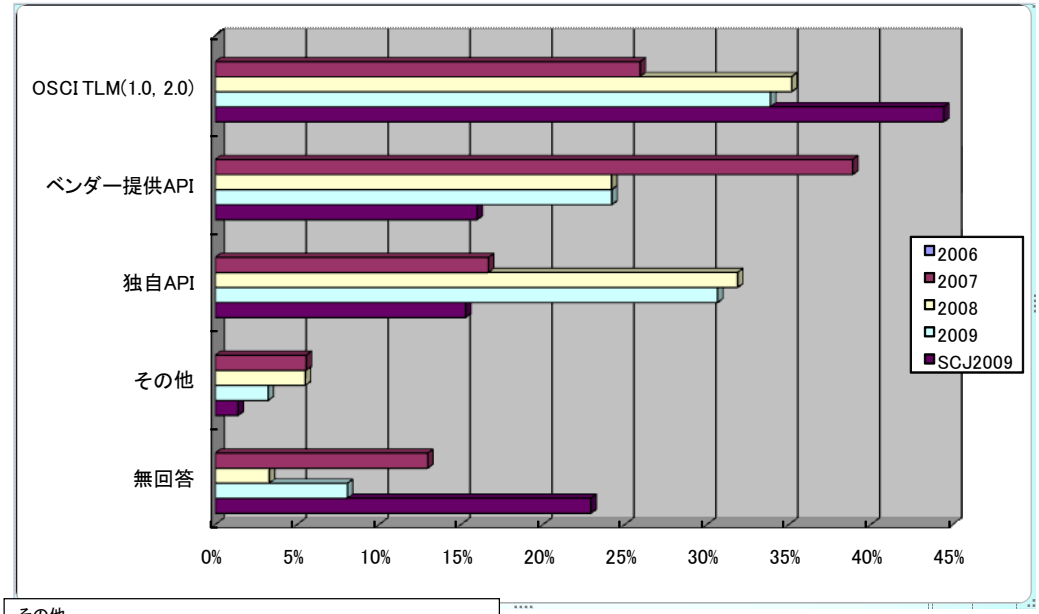
⑦　⑥で「既に行っている」または「検討中」と回答された方
c）設計・検証の抽象度は？



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）？



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？

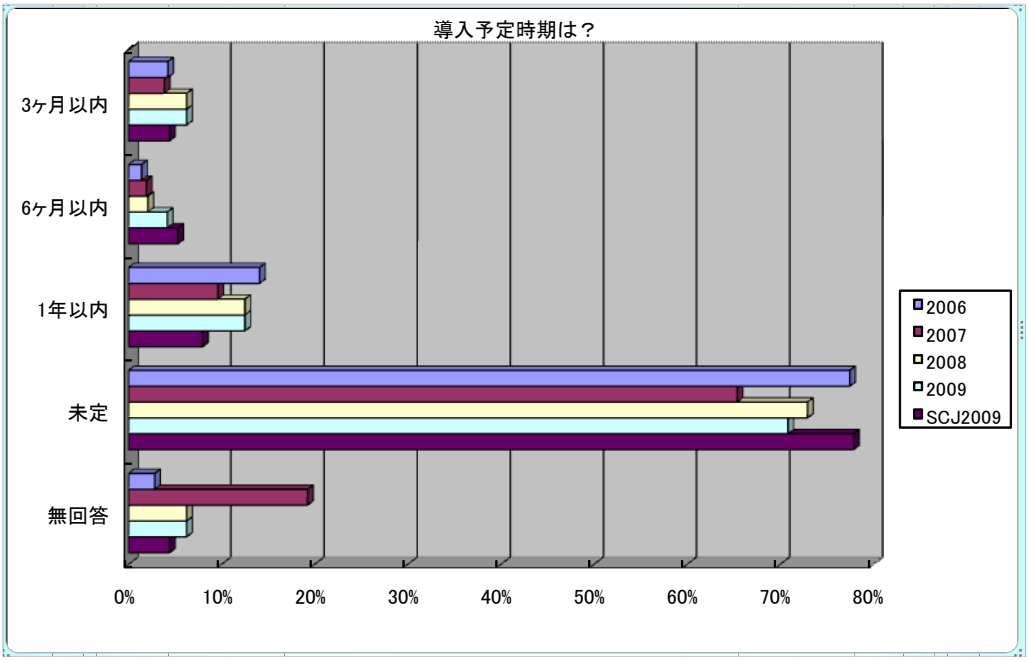


その他
- SystemCをツール間のインターフェースとして使用C←→Verilog

※ OSCI TLMは、TLM1.0が9.5%、TLM2.0が25%の割合となっています。

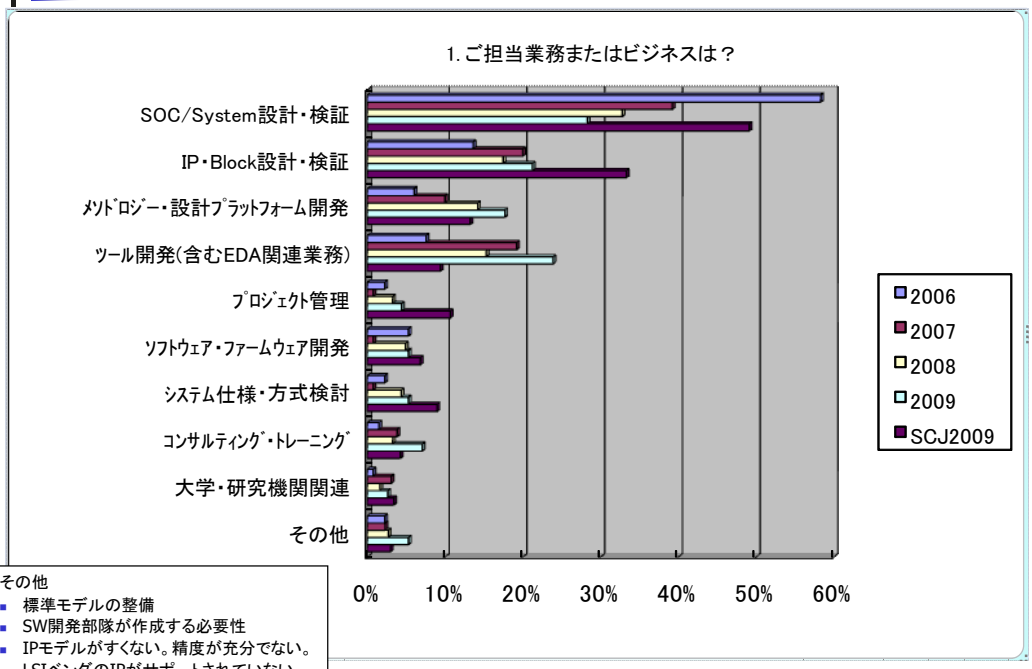
JEITA

⑧ ⑥で「検討中」と回答された方へ
導入予定時期は?



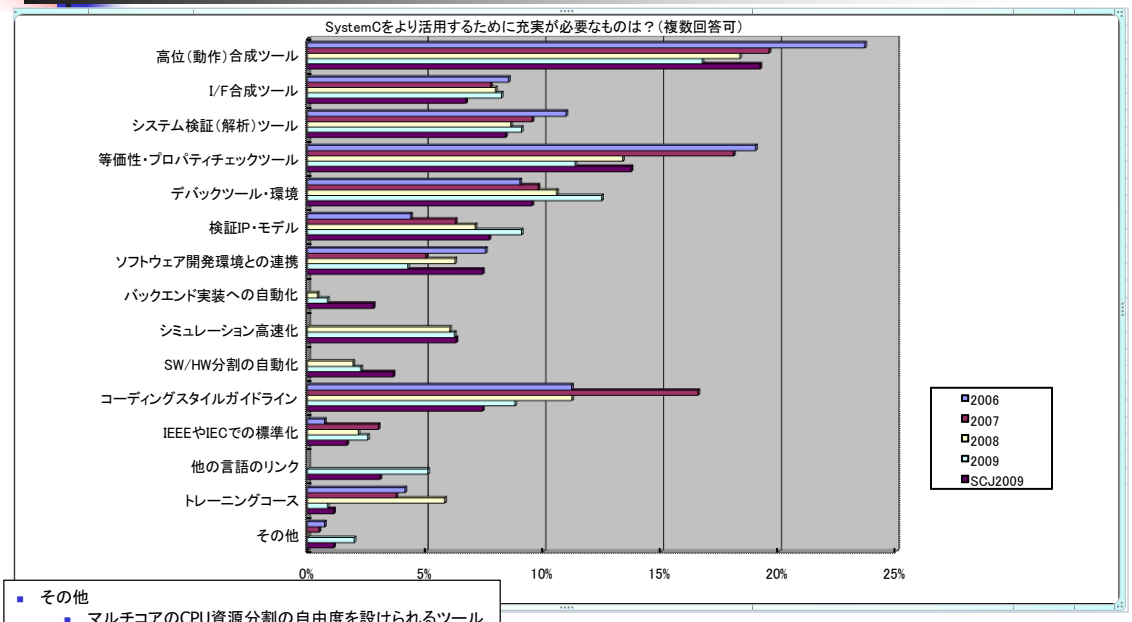
JEITA

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?



その他
- 標準モデルの整備
- SW開発部隊が作成する必要性
- IPモデルがすくない。精度が充分でない。
- LSIベンダのIPがサポートされていない
- 学習中

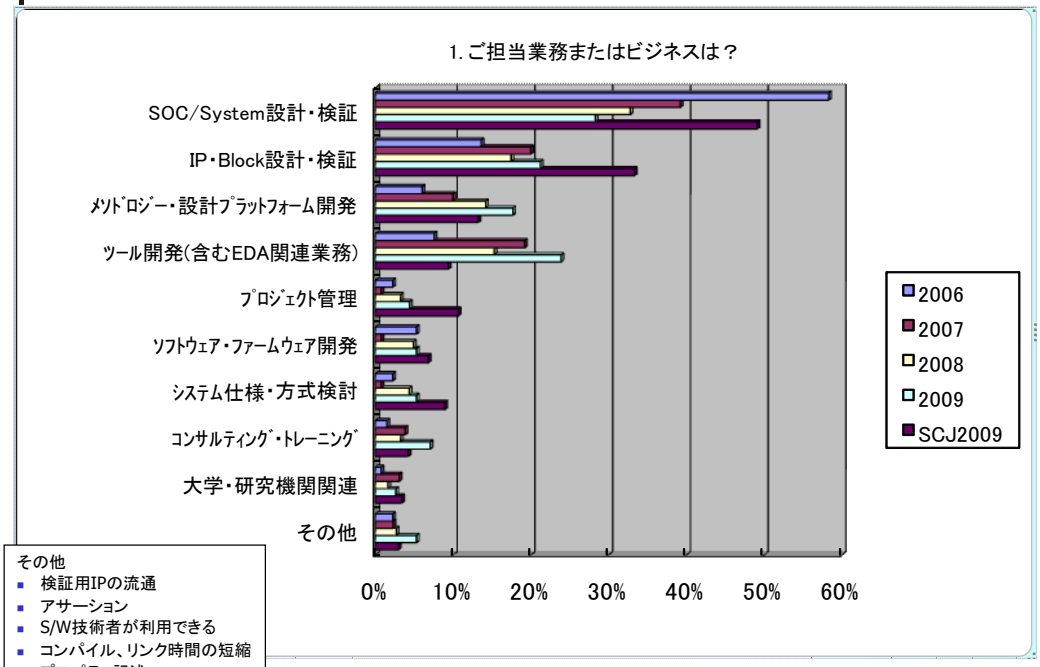
⑩ SystemCをより活用する為に充実が必要なものは？



- その他
  - マルチコアのCPU資源分割の自由度を設けられるツール
  - モデルフリーのツール
  - FPGAへの対応
  - ISS
  - 検証機能
  - 抽象度混在HWモデリング構造

他の言語のリンク
- VHDL、Verilog、
- MATLAB,C/C++

⑪　今後SystemCの言語拡張･標準化で期待することは?



その他
- 検証用IPの流通
- アサーション
- S/W技術者が利用できる
- コンパイル、リンク時間の短縮
- プロパティ記述

JEITA

4.2.7 ECSI HLS Workshop ASP-DAC09 まとめ

ECSI : European Electronic Chips & Systems Design Initiative
http://www.ecsi.org/

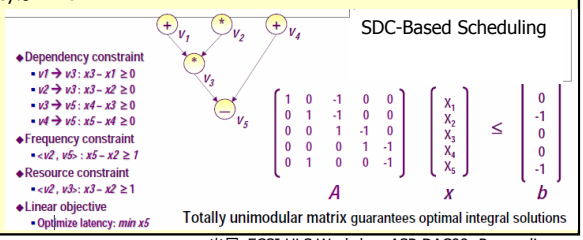
JEITA

調査内容一覧

- Session 1: Architecture and Design Flow I
  - [S1.1] Algorithmic Foundation for ESL 2.0
  - [S1.2] BSV: a radically different approach to HLS
  - [S1.3] User Guided High Level Synthesis
- Session 2: Architecture and Design Flow II
  - [S2.1] C-Based SoC Design Flow with CyberWorkBench
  - [S2.2] PICO: Advanced Algorithmic Synthesis for SoCs and FPGAs
  - [S2.3] Equivalence Checking in Context of HLS
- Session 3: Architecture and Design Flow III
  - [S3.1] The CHStone Benchmark Suite for Practical CBased High-Level Synthesis
  - [S3.2] Applying SystemC Synthesis To All Your Design Challenges
  - [S3.3] Benefits of Model-Based High Level Synthesis with Synplify DSP
- Session 4: Low-Power HLS
  - [S4.2] New Binding Algorithms for Glitch and Inter-transition Power Reduction
  - [S4.3] High-level Synthesis: Optimizing for Low Power Design

JEITA

Session 1: Architecture and Design Flow I

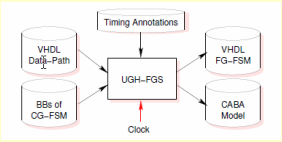
タイトル	[S1.1] Algorithmic Foundation for ESL 2.0
著者	Jason Cong
所属	UCLA Computer Science Department
概要	ESL Synthesisの基盤となるアルゴリズムと実現例
内容	■ 高位合成の基盤となる新アルゴリズム • SDC-Based Scheduling：各種設計制約をsystem of integer difference constraints (SDC)で表すと共に、設計対象を線形関数で表現する手法 効果：平均で16%のレイテンシ削減 • Simultaneous FU and Register Binding (SFR)：リソースシェアリングを行うステップにおいて、FUとレジスタのバインディングを交互に繰り返しながら徐々に最適化を進める手法 効果：MUXを9.8~13.8%削減、周波数8.4~11.3%削減 • Pattern-based Synthesis：データパスから規則性を抽出し、最適化の際に考慮する手法 効果：平均20%の面積削減、Latencyは+7% ■ xPilot：プラットフォームベース合成システム • 入力：SystemC/C/C++、プラットフォーム記述、制約情報 • 出力：プロセッサコア＋executables、ドライバSW、カスタムロジック • 特長：プロセッサ&アーキテクチャ合成機能、I/F合成機能、動作合成機能 ■ AutoPilot：xPilotをベースに、AutoESL社で製品化した動作合成システム 

Session 1: Architecture and Design Flow I

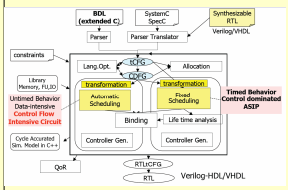
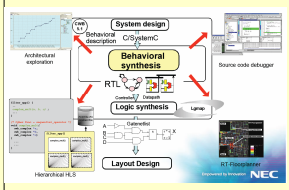
タイトル	[S1.2] BSV: a radically different approach to HLS
著者	Rishiyur S. Nikhil, Ph.D.
所属	Bluespec, Inc.
概要	Bluespec SystemVerilog がHLSに対して根本的に違うアプローチを取る訳

内容	■ Bluespecと旧来HLS手法が違う理由
	• C/C++は並列アルゴリズムやアーキテクチャを表現するには能力不足
	•SoC設計者は少しのIPブロックではなくシステム全体を設計・最適化する必要がある
	■ 逐次アルゴリズムは、より良い実装の可能性を除外するかもしれません
	•逐次アルゴリズムから並列アルゴリズムにシフト
	■ では、なぜC/C++に並列性を追加しないのか？
	•ヘテロジーニアスやきめ細かい並列性のモデル化には向かないことが認知済み
	•CベースのHLSフローはきれいに纏まっているがSoCデザインの一部にしか対応しない
	■ 「ルール」ベースの記述を提案
	•複雑な制御論理を自動化
	•コントロールの適応性
	•アーキテクチャをパラメーター化
	■ 実績と実例
	`rule r1 (x1 > x2);`
	`  x1 <= x2; x2 <= x1;  // swap`
	`endrule`
	出展：ECSI HLS Workshop ASP-DAC09 Proceeding

Session 1: Architecture and Design Flow I

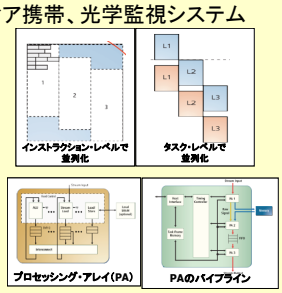
タイトル	[S1.3] User Guided High Level Synthesis
著者	Ivan Augey / Frederic Petrot
所属	LIP6, Universite Pierre et Marie Curie / TIMA, INP Grenoble
概要	2段階に分けたスケジューリングで設計容易で最適化の効果の高い手法を実現
内容	始点：ソフトウエア向けとIC設計向けで最適なアルゴリズムは異なる。 Coarse Grain Scheduling：概略スケジューリング  入力：データパス案(リソース)の指示、 動作記述のCソース 出力：マクロセルからなるデータパス記述、FSM Fine Grain Scheduling：詳細スケジューリング  FSMを動作周波数とデータパスの特性に応じて最適化 出力：クロックで動作し最適化されたFSM、 高速ICシミュレーションのためのCABAモデル Cycle-Accurate, Bit-Accurate

Session 2: Architecture and Design Flow II

タイトル	[S2.1] C-Based SoC Design Flow with CyberWorkBench
著者	Kazutoshi Wakabayashi & Benjamin Carrion Schafer
所属	EDA R&D Center, NEC Corp. Japan
概要	全てのアプリケーションにおいて，Cで設計・検証することが実現できている
内容	 ■全てのアプリケーションに対応するため，2つのエンジンを持つ •“Automatic scheduling”：UntimeのBehavior記述からの合成 •“Fixed Scheduling”：Timed Behavior記述からの合成． サイクル毎の振る舞いをCで記述可能 ■合成エンジンの他に，アーキテクチャ探索機能，階層合成，ソースコードデバッガー，RT-FloowPlannerとの連携など充実した環境 •合成オプションなどを変更しながら，マイクロアーキテクチャの自動探索 •配線性レポートや，グローバルワイヤの共有，RT-FoorPlannerの結果からの再スケジュールで，配線性の考慮 •その他，ソースコードデバッガ，動作IPの提供，動作モデルの暗号化も実現  出展：ECSI HLS Workshop ASP-DAC09 Proceeding

Session 2: Architecture and Design Flow II

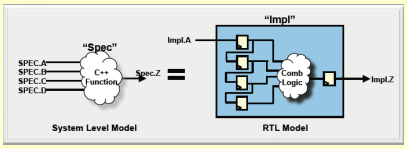
タイトル	[S2.2] PICO: Advanced Algorithmic Synthesis for SoCs and FPGAs
著者	Vinod Kathail
所属	Synfora
概要	ループ文の合成手法に特徴のあるC入力の動作合成
内容	• 設計ターゲット：複雑なアプリケーション・エンジン（コーデック） – アルゴリズム・レベルでの再利用の機会あり。製品毎に実装が違う。 • 入力：Cアルゴリズム、実装制約（クロック周期、ライブラリ、性能要求）、テストベンチ・データ • 出力：RTL、テストベンチ・ベクタ、SystemC TLM、SWドライバ • メリット：設計期間短縮、アルゴリズム・実装をいろいろ試せる、様々な製品・プロセスに渡った再利用が可能 • 適用例：セットトップボックス、ビデオカメラ、ビデオアルゴリズム、マルチメディア携帯、光学監視システム • PICOの特徴 – 様々なループ文に対する並列化 – 柔軟性の高いアーキテクチャテンプレート使用 – 階層合成（関数を部品として合成、パイプライン・ループ・ストールを持つことも可） – 消費電力に関するアーキテクチャ・トレードオフ可とクロックゲーティング – RTLテストベンチ生成、コーナーケース検証のための疑似ランダム追加

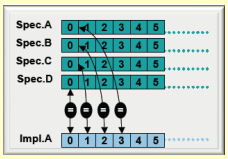


JEITA
出展：ECSI HLS Workshop ASP-DAC09 Proceeding

Session 2: Architecture and Design Flow II

タイトル	[S2.3] Equivalence Checking in Context of HLS
著者	Venkat Krishnaswamy
所属	Calypto
概要	高抽象度等価性チェックで解決すべき問題の概要と実フローでの要件
内容	•HLSで求められる等価性検証技術 - 高抽象度言語対応（C. C++. SystemC） - シーケンシャルな制約が定義できる能力 （レイテンシ／スループット） - シーケンシャルな推論エンジン ／高キャパシティな演算器検証エンジン／帰納的な証拠エンジン •Sequential Logic Equivalence Checker (SLEC)とは？ - 時系列的インターフェース差異をもつデザインの等価性チェック （シリアル⇔パラレル、TLM⇔ピン） - 内部状態の差異を持つデザインの等価性チェック （リタイミング、パイプライン化等） •実フローでの等価性チェックの要件 （デバッグ容易化やECO対応） - 階層情報の保持 ： RTLやゲートレベル信号を階層境界のCの信号へ逆マッピングできる能力 - ループ情報の保持 ： 入力変数、出力変数、ループ長、繰り返し数、カウンタ変数等 - 合成結果RTLへワードレベル演算器情報の保持 ： ビットレベルの演算器は下流合成ツールに任せる - C変数とRTLレジスタのマッピング情報 ： Cモデル中の全ての静的変数からFFへのマッピングを提供すること - メモリ ： C配列変数とRTLのRAMとのマッピング情報





JEITA
出展：ECSI HLS Workshop ASP-DAC09 Proceeding

Session 3: Architecture and Design Flow III

タイトル	[S3.1] The CHStone Benchmark Suite for Practical CBased High-Level Synthesis
著者	Hara Yuko, Hiryoyuki Tomiyama, Shinya Honda, Hiroaki Takada
所属	Graduate School of Information Science, Nagoya University, Japan
概要	C言語からの動作合成用のベンチマーク用データ12種類を作成して公開した
内容	C言語ベースの動作合成ツール研究用途で一般的に使用できる標準ベンチマークが今までなかったため、12種類開発して公開した。2%以内のソースコード変更にて実際の商用動作合成ツールで合成可能であることを確認ずみ。現状の動作合成に不向きな、Compositeデータタイプ、再帰呼び出し、動的メモリアロケーションを排除してある。 http://www.ertl.jp/chstone/

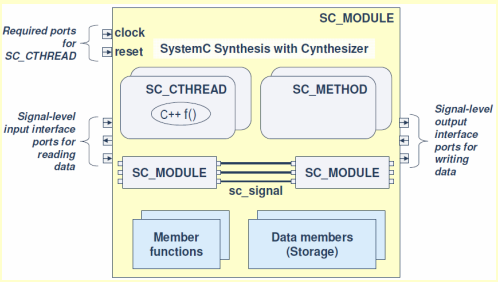
Application domain	Name	Design description	Source
Arithmetic	DFADD	Double-precision floating-point addition	SoftFloat
	DFMUL	Double-precision floating-point division	SoftFloat
	DFDIV	Double-precision floating-point multiplication	SoftFloat
	DFSIN	*Sine* function for double-precision floating-point numbers	SoftFloat, Authors' group
Micro-processor	MIPS	Simplified MIPS processor	Authors' group
Media processing	ADPCM	Adaptive differential pulse code modulation decoder and encoder	SNU
	GSM	Linear predictive coding analysis of global system for mobile communications	MediaBench
	JPEG	JPEG image decompression	Authors' group, The Portable Video Research Group
	MOTION	Motion vector decoding of the MPEG-2	MediaBench
Security	AES	Advanced Encryption Standard	AILab
	BLOWFISH	Data Encryption Standard	MiBench
	SHA	Secure hash algorithm	MiBench

出展：ECSI HLS Workshop ASP-DAC09 Proceeding

JEITA

Session 3: Architecture and Design Flow III

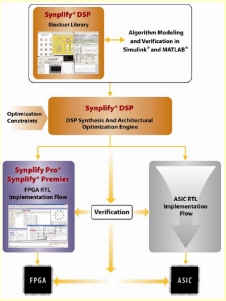
タイトル	[S3.2] Applying SystemC Synthesis To All Your Design Challenges
著者	不明
所属	Forte Design Systems
概要	Cynthesizerを用いた動作合成は、制御系に適用してもメリットあり
内容	・制御系に適用するメリット - 入力を抽象化して、コード量の削減、抽象レベル切り替え(TLM、PIN、RTL)が可能 - コントロールとアルゴリズムで、同一言語を使用した統一的な検証が可能 - シミュレーション対象をそのまま合成可能 ・抽象レベルを上げて設計効率を向上 - 高精度なクロックエッジ制御が不要であれば、自動スケジューリングが使える - インタフェース(ポート・プロトコル等のコード)のカプセル化により、WriteとVerifyの同時実行、再利用性の向上が可能 - 暗黙的なステートマシンを自動生成 (ループ構造、メモリへの配列アクセス等)



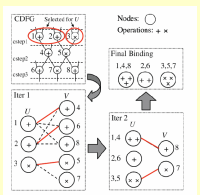
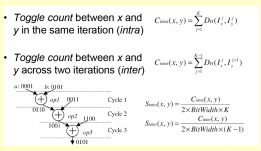
出展：ECSI HLS Workshop ASP-DAC09 Proceeding

JEITA

Session 3: Architecture and Design Flow III

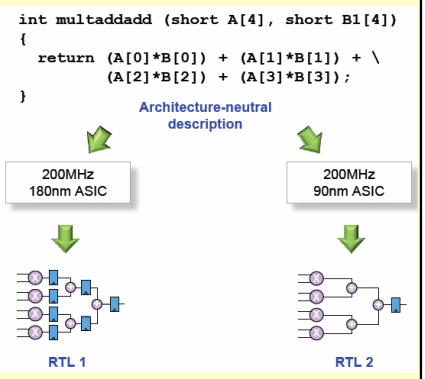
タイトル	[S3.3] Benefits of Model-Based High Level Synthesis with Synplify DSP
著者	Chris Eddington
所属	Synopsys
概要	DSPアルゴリズムの回路実装を容易にする
内容	**以下の特徴と機能を持つSnplify DSPよりアルゴリズム開発はMATLAB/Simulink上のモデルからターゲットFPGA/ASICに最適化したRTLの自動生成が可能となる。設計者はアルゴリズム およ び回路仕様検討のみに集中できる。** **モデルライブラリ(DSP Blockset:MATLAB/Simulinkと連携)** -高位IP:Simulink上で利用できる高位IPライブラリ、パラメータ化、最適化可能なIP作成可能。 -数学関数のサポート:固定小数点、マルチレート、ベクトル表記など -メモリやレイテンシーの管理:FPGAへメモリやレジスタマッピングが可能 -制御用言語:M言語のサブセットを用いてステートマシーンや制御ロジック用に言語エントリが可能 **High Level Synthesis Engine** -システムアーキテクチャの最適化:リソースシェアリング/スケジューリング/パイプライン/リタイミング/エリアなどを最適化する動作合成エンジン -IP毎にその内容と制約に従い最適な内部アーキテクチャを選択 -マルチレート、マルチクロックドメインサポート -FPGA.ASICのサポート:ベンダーやデバイスを超えたデザインポータビリティ(移植性)  出展:ECSI HLS Workshop ASP-DAC09 Proceeding

Session 4: Low-Power HLS

タイトル	[S4.2] New Binding Algorithms for Glitch and Inter-transition Power Reduction
著者	Deming Chen, Scott Cromar
所属	Department of Electrical and Computer Engineering University of Illinois
概要	ダイナミックな消費電力を削減する高位合成のバインディング手法の提案
内容	下記のスイッチングアクティビティ(SA)を削減することによってダイナミックな消費電力を削減するバインディングアルゴリズムの提案 ■グリッチ削減アルゴリズム ・FPGAへのマッピング情報からSAスイッチングを推定して,その結果からMUXのsizeが小さくなるようにバインディング <結果> 従来手法(LOPASS)に比べて19%の低消費電力化  ■機能的遷移の削減 ・同一イタレーションでのデータ遷移(intra transition)と2つのイタレーション間でのデータ遷移(inter transition)の両方を加味して,遷移(SA)が少なくなるバインディングを実施 <結果> 従来手法に比べて4.1%~6.7%のSAを削減  出展:ECSI HLS Workshop ASP-DAC09 Proceeding

Session 4: Low-Power HLS

タイトル	[S4.3] High-level Synthesis: Optimizing for Low Power Design
著者	Andres Takash
所属	Mentor Graphics
概要	カタパルトC合成により、高位レベルで低消費電力化を図ることが可能
内容	・カタパルトによる電力最適化 - アーキにとらわれない記述から、周波数等のパラメータにより最適なRTLコードを生成 - 消費電力・面積・パフォーマンスのバランスから、インタフェースの最適化(ビット幅の調整等)と、マイクロアーキテクチャの最適化を行う - 多くのアルゴリズムにとって、消費電力・面積・パフォーマンスは、メモリアーキテクチャに依存。様々なメモリアーキの探索が可能 ・TELEGENT社のCOFDMレシーバ開発事例 - C++コーディングとRTL生成は一ヶ月。(カタパルトなしなら4ヶ月は必要) - 新仕様追加に検証済CからRTLを1日で作成。(カタパルトなしなら1週間必要)



出展:ECSI HLS Workshop ASP-DAC09 Proceeding

EDAアニュアルレポート　2009

2010年5月発行

発　　行　　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1-1-3
大手センタービル5F
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