

5. 標準システムレベル設計言語の動向と提案フローへの適用評価

5.1. 全体動向

5.1.1. 概要

システムレベル設計言語は、システムレベル設計フロー実現の鍵となる要素技術である。大規模なシステムLSIの設計では、仕様設計段階からシステムの機能、性能、消費電力、ノイズなどが最終的にどのようになるかあらかじめ見通しながら、最適な仕様を決定する必要がある。システムレベル設計言語は、システムの機能や制約を正しくモデル化し、かつ、それをハードウェアやソフトウェアを用いてどのように実現するかを決定するために用いられる。システムレベル設計言語の使用によって以下の利点が期待できる。

- 設計抽象度の向上：設計複雑度の減少、仕様設計からアーキテクチャ設計への設計成果物受渡し手段
- 厳密なモデル化：仕様誤りの早期発見、間違いのない情報伝達、ドキュメント化
- CAD 入力手段：システムレベル設計環境（ツール）構築の容易化、統合設計環境の実現

従来、電子回路やLSI設計ではVHDL、Verilog-HDLといったHDL（Hardware Description Language）、ソフトウェア開発ではC言語やアセンブリ言語といった標準的な設計言語が使用されてきた。システムLSIのシステムレベル設計が重要になるとともに、設計フローを実現するためのシステムレベル設計言語の重要性が増してきた。それに付随して、特に1999年9月ごろからSystemC、SpecCなどを初めとするC/C++言語ベースのシステムレベル設計言語の標準化推進団体が次々と発足し、標準化の動きが活発になってきた。2000年9月になってVHDL、Verilog-HDLといったハードウェア言語の標準化に深く関わってきたVI（VHDL International）^[7]とOVI（Open Verilog International）^[9]という2つの団体が統合してAccellera^[10]という組織を発足し活発な活動を開始した。その中のALC（Architectural Language Committee）では、C/C++での実現例を提供すると共に、システムレベル設計記述のリファレンスを定める標準化活動を行っている。

本章は、このような背景のもと、以下を目的としてSLD研究会が行ってきたシステムレベル設計言語の調査結果をまとめている。

- システムレベル設計言語の標準化動向の把握
- SLD 研究会が提案したシステム設計フローへの適用を通じた評価
- 標準化活動へのフィードバック

本章では調査結果を2段階に分けてまとめた。

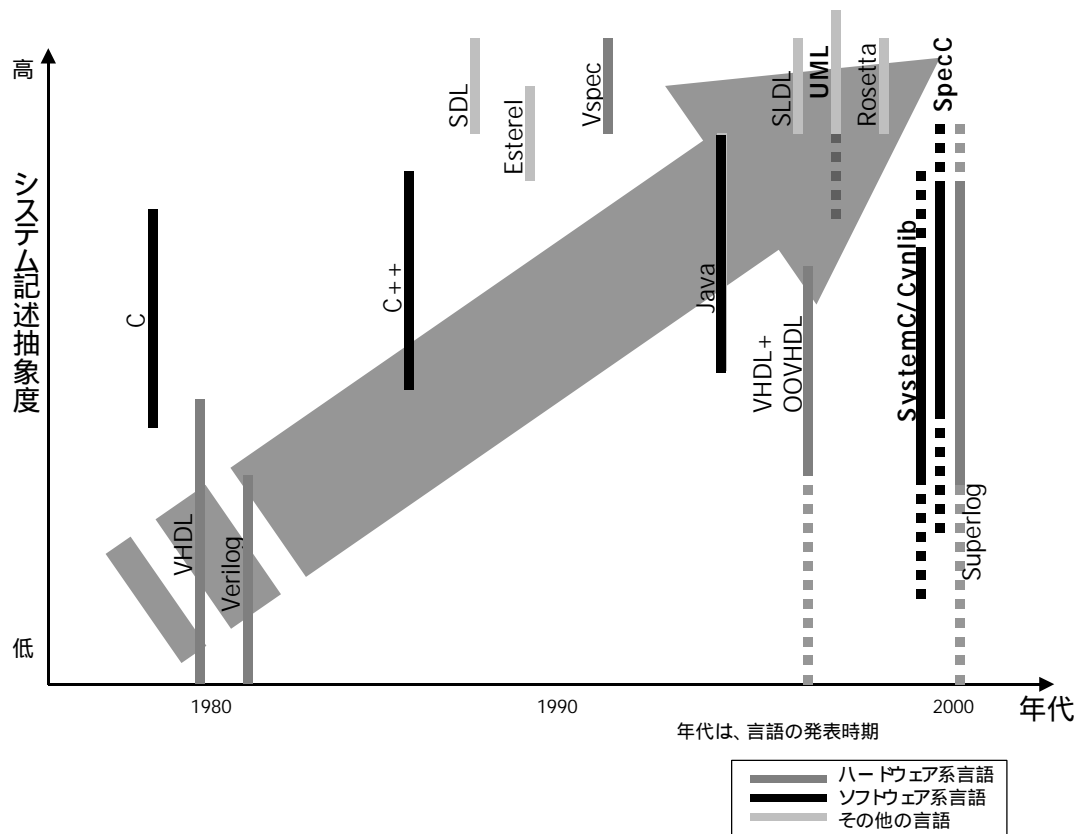


図 5.1-1 言語マップ

まず、第1段階ではシステムレベル設計への適用が試みられた既存のシステムレベル設計言語の動向を系統立てて言語マップとして整理した(図5.1-1)。図5.1-1では縦軸に言語の設計抽象度、横軸に言語の発表時期を表している。システムレベル設計言語はアプローチ的に見て、大まかに(1)ハードウェア系言語、(2)ソフトウェア系言語、(3)新規言語など上記以外、の3つの範疇に分類できる。言語マップではこれらの範疇がわかるように分類してマップした。図5.1-1からもわかる通り、全体的な傾向としては、ソフトウェア系言語、ハードウェア系言語ともに、年代とともに徐々に記述抽象度を広げ、適用範囲がシステム全体をカバーするように推移している。アプローチ別の全体的な言語動向の概要は5.1.2節から5.1.4節でまとめる。

第2段階では、特に注目すべき3つの言語を選び、詳細にまとめている。特に、1999年後半から、ソフトウェア設計言語であるC/C++をベースにシステムレベル設計言語

の標準化を推進しようとする動きが活発になり、SystemC、SpecCの標準化団体が相次いで発足した。さらに、ソフトウェア設計の世界では、メジャーな3つのオブジェクト指向設計手法を統合したUMLの標準化が活発に行われており、UMLをハードウェア含めたシステムレベルへの記述に適用しようとする研究事例などの動きが出てきている。そこで、第2段階では、標準化が活発なSystemC、SpecCと、将来の方向性の一つとして考えられるUMLを取り上げた。

他文献の引用箇所にはすべて注釈を置き出典を明確にした。参考文献は本章の最後にリストアップしているので、さらに詳細が必要な方々はそれらの文献を参照頂きたい。

5章の構成は以下の通りである。5.1節ではシステムレベル設計言語全体の動向をアプローチ別の言語に分類して概説する。5.2節から5.4節は注目すべき3つの言語SystemC、SpecC、UMLについて詳しく述べる。5.5節では本研究会が第4章で提案したシステム設計フローへの上記言語の適用を想定して評価と考察を行い、5.6節でまとめている。

5.1.2. ハードウェア設計言語系の言語

ハードウェア設計者がシステムレベルの記述を行いたいと考えた場合、VHDLやVerilog-HDLなど、既存のハードウェア設計言語を拡張してシステムレベルの記述を可能にしたいという考えはごく自然な発想である。

既存言語のシステムレベル拡張を行うアプローチとして、VHDLにもとづくシステムレベル拡張はIEEE/DASC内で1996年頃から議論され、以前からOOVHDL、VHDL+といった言語の標準化活動が行われてきた。Verilog-HDLに関しては、2000年に規格改訂がVerilog2000として行われ、上位の抽象レベルでのシステム記述のためにも仕様が一部拡張されたが、ハードウェア記述言語としての位置づけは変わっていない。Verilog系言語としては他にSuperlogが米国Co-Design Automation社からシステムレベル設計言語として提案されている。しかし、2000年10月にVHDL+の活動が停止するなど、VHDLベースの言語標準化の動きは以前ほどの活発さを失っており、さらに、長い間VHDL、Verilogの推進団体として標準化に関わってきたVIとOVIが、2000年9月に統合してAccelleraという組織を発足し、C/C++言語も含めたシステムレベル設計言語の標準化に関わる動きを打ち出してきた。

以下に主な言語の概要を示す。

(1) OOVHDL^[1] (Object-Oriented Extensions to VHDL)

IEEE/DASC の OOVHDL Study Group で 1998 年から議論が進められ、2001 年 3 月の時点でも標準化活動を行っている。OOVHDL では VHDL にオブジェクト指向性を

取り入れて設計再利用性やシステム抽象化を高め、テストベンチも含むシステムの抽象化やカプセル化を用いた容易なモデル化を実現することを目的としている。VHDL にオブジェクト指向性を取り入れる考え方は古くから SUAVE^[3]と Objective VHDL^[4]で議論されていたが、2000 年度に Objective VHDL が言語仕様として採択された。

(2) VHDL+^[2]

IEEE/DASC の SID(System and Interface based Design)中心に標準化が進められ、1999 年 6 月に PAR (Project Authorization Request : 1551) が IEEE Standard Board にアクセプトされた。VHDL+は英国 ICL 社が開発し、70 以上の企業・大学がユーザ会に登録している。OOVHDL と異なってオブジェクト指向の概念は導入せず、ポートを介さないメッセージ通信やデータタイプを指定しない通信など抽象度の高い通信モデルのサポートが特徴である。ツールも ICL 社を中心に開発されており開発環境は揃っている。しかし、2000 年 10 月に SID、ICL 社は活動を中断し、2001 年 3 月の時点では停止状態である。

(3) Superlog^[5]

Verilog 系システムレベル設計用言語として米国 Co-Design Automation 社から提案された新しい言語である。Superlog は Verilog-HDL と C をベースにしてさらに VHDL と Java のエッセンスを加えて提案された。システム LSI の設計に必要なハードウェア、ソフトウェア、システムの各設計分野について単一言語により設計手法の統一、設計効率改善、従来手法からの発展性などを実現する。Co-Design Automation 社は 1997 年に設立され、ツール開発では Magma Design Automation 社、Viewlogic 社、Sente 社がパートナーとなっている。2001 年 3 月の時点では、Co-Design Automation 社は Accellera に加盟して標準化活動に加わっている。

5.1.3. ソフトウェア設計言語系の言語

最近、最も注目されているシステムレベル設計言語は、C 言語や C++言語など、もともとソフトウェア設計用であったプログラム言語をシステムレベルに拡張した言語である。本節では、まず C/C++ベースのシステムレベル設計言語について説明し、その後で Java ベースのシステムレベル設計言語の動向について述べる。

【1】 C/C++ベース言語の動向

C 言語を拡張し、システムレベル設計言語として使用する研究は、1980 年代後半に始まった。スタンフォード大学の DeMicheli 教授のグループにより開発された Hardware-C^[13]が有名である。1990 年代後半になり、EDA ベンダーからも C/C++をベースにしたシステムレベル設計言語が提案されてきた。カリフォルニア大学アーバイン校の Gajski 教授らにより開発された SpecC^[17,18]は ANSI-C 言語の仕様を拡張したも

のであり、米国 Synopsys 社が中心になって開発した SystemC^[20] と米国 CynApps 社(現 Forte Design Systems 社)が開発した Cynlib^[19]はともに C++をベースにしている。また、米国 C Level Design 社^[16]の様に ANSI-C/C++の両方をサポートする動きもある。SpecC と SystemC は、それぞれ 1999 年後半に推進団体である SpecC Technology Open Consortium (STOC)^[18]、Open SystemC Initiative (OSCI)^[20]を発足させ、2001 年 3 月の時点では最も活発に標準化活動を行っている。また、標準化の動きはないものの、シャープが開発した Bach-C^[11,12]や NEC が開発した BDL^[14,15]のように社内使用を目的とした C/C++ベースのシステムレベル設計言語も研究され、動作合成ツールの入力言語として社内での実績を出しているものも多い。

C/C++ベース言語は、C や C++のシンタックスを利用し、ハードウェアを記述するために必要なセマンティクスを拡張しているため、以下のような特徴を持つ。

- ハードウェアを記述するために、ソフトウェアプログラミング言語にはない機能を追加している。例えば、ハードウェアを表現するデータ型、並列性、同期 / 非同期、クロックなど時間的な概念、信号接続やプロトコルなど通信の表現などを追加している。
- 多くのソフトウェア設計者、システム設計者 / アルゴリズム開発者に使用されている言語をベースとしており、容易に習得することができる。
- C/C++のソフトウェアプログラム開発環境を利用できる。
- HDL など他のシステムレベル設計言語にくらべ、検証スピードが速い(ただし、スピード差は言語や記述レベルに依存する)。
- C/C++ベース言語から、RTL-HDL を自動生成できる動作合成ツール(動作合成ツールとも呼ぶ)の開発が他のシステムレベル設計言語に比べ進んでいる。

以下に、代表的な言語の概要を示す。

(1) SystemC^[20]

米 Synopsys 社が中心になって開発した言語で、推進団体 OSCI により標準化活動が進められている。SystemC については第 5.2 節で詳しく説明する。

(2) SpecC^[17,18]

カリフォルニア大学アーバイン校の Gajski 教授が提唱した言語であり、推進団体 STOC により標準化活動が進められている。SpecC については第 5.3 節で詳しく説明する。

(3) Cynlib^[19]

米 CynApps 社が開発したハードウェア記述可能な C++ライブラリである。検証可能なシミュレーションカーネル、ハードウェア記述用のクラスライブラリ、モデル記述方法を提供している。ハードウェア記述のためのデータ型、並列性、階

層記述、同期 / 非同期などの機能を含むほか、Verilog-HDL とのインタフェースを用意している。CynApps 社では、設計レベルを Pure C アルゴリズムレベル、Bit 精度レベル、Cycle 精度レベルの 3 つにわけ、設計者が段階的に記述レベルを下げる設計手法を提案している。検証環境のほか、Cycle 精度レベルの記述を Verilog-HDL 変換するツール Cynthesis がサポートされている。CynApps 社は Accellera に加入している。また、2001 年 3 月に Chronology 社と合併し、Forte Design Systems 社に社名変更した。

(4) C Level Design 社の ANSI-C/C++ ^[16]

C Level Design 社は、ANSI-C と C++の両方の言語をそのまま使用できる設計検証環境を提供している。言語拡張は行っていないが、ビット幅などハードウェアを記述するための工夫を行っている。C Level Design 社では、ファンクショナル C と RTL - C との 2 つの記述レベルを定義している。CSim と呼ばれる検証ツールと System Compiler とよばれる C/C++モデルを RT レベルの HDL に変換するツールを提供している。また、OVI の Architectural Committee による “Semantics Reference Manual Standard for C++ class library” に従った System C++と呼ばれる C++クラスライブラリも提供し、2001 年 3 月の時点では Accellera メンバーとして活動を行っている。

(5) Hardware-C ^[13]

動作合成を備えた設計検証システム (Olympus Synthesis System) における動作レベルのハードウェア記述用言語としてスタンフォード大学で開発された。C 言語に対して、ハードウェアを記述するために必要な並列動作や構造の記述、タイミングなどの拡張を行っている。設計制約を記述できる点が、他の C/C++ベース言語にはない特徴である。すでに研究は終了し実設計に使用できる言語ではないが、研究レベルでは、CD や DAT のバスとオーディオバスとのインタフェース回路や画像処理などの設計に適用された例が報告されている。

(6) Bach-C ^[11,12]

シャープとシャープヨーロッパ研究所が共同開発した言語である。C 言語に、並列動作、同期 / 非同期通信、ビット幅、合成用のプラグマなどハードウェア・アルゴリズムを記述する上で必要な拡張がなされている。動作合成ツール、検証ツールを含む Bach システムが用意されている。標準化活動は行っていないが、シャープ社内では画像処理、通信関係の設計を中心に実設計で適用されている。

(7) BDL ^[14,15]

NEC が開発した言語である。C 言語に、ビット幅、並列実行、割込み、ハンドシェイク、クロック・サイクル指定、レジスタやメモリの指定などハードウェアを記述する上で必要な機能を追加している。Cyber とよばれる動作合成ツールや

ClassMate とよばれる C++シミュレータ環境とリンクし、動作レベルから RT レベルまでの設計検証環境が整っている。標準化活動は行っていないが、NEC 社内では、デジタル家電、ネットワーク、コンピュータなどデータパス回路、制御回路をとわず実回路の設計に使用されている。次世代の携帯電話のシステム開発にも適用されている。

【2】 Java ベース言語

Java ベースのアプローチは、C++ベースのアプローチに酷似している。すなわち、オブジェクト指向言語の特徴であるクラスライブラリを用いて、本来、ソフトウェア言語である Java 言語が持たないハードウェア的な性質を拡張しようとしている。これにより、ソフトウェアだけでなく、ハードウェアをも表現できるようなシステムレベル設計言語を確立しようとするアプローチである。このアプローチを採るグループは、システムレベル設計言語として用いるとき、Java 言語は、C++言語に比べて以下の点で優れた性質を持っていると主張している。

(a) 高い生産性

- ◇ オブジェクト指向性を利用した高いプログラム生産性
- ◇ JavaVM とバイトコードによる標準的なプラットフォーム

(b) 効率的なハードウェアの生成

- ◇ オブジェクト指向による高い構造化
- ◇ ポインタの禁止による暗黙の並列性の識別と抽出の容易化
- ◇ マルチスレッドによる明確な並列制のサポート

欠点として実行速度が遅いことがあげられるが、この問題が解決されれば、Java 言語ベースのアプローチはシステムレベル設計言語としての有力な候補になる可能性を持っている。Java 言語そのものを用いた実際の設計ツールもすでにいくつか発表されており、米 LavaLogic 社 (2001 年 3 月の時点では Xilinx 社に吸収) のハードウェア動作合成システム Forge-J^[21]などがその結果を発表している。

代表的な Java ベース言語として以下が存在する。

(1) JavaTime^[25]

JavaTime は言語名というよりは方法論にもとづくシステムの名前である。カリフォルニア大バークレイ校の Newton 教授により提案された異種多様なエンベデッドシステム設計のための実験システムである。システムの仕様、モデル化、実装に対して、コンポーネントベースのアプローチを適用している。Java 構文として、時間の概念の組込みは行わず、マッピング形式で時間概念を導入しているのが特徴である。また、アーキテクチャ自動生成は行わず、

- 1) 最初に制限のない汎用プログラム言語で記述し、
 - 2) 言語解析を通じて段階的に目的の制限された計算モデルへ変換する、
- SFR (Successive Formal Refinement) という仕様開発の方法論を導入している。

5.1.4. その他の言語

本節では、その他のいくつかのシステムレベル設計言語について概説する。

(1) UML (Unified Modeling Language)

UML は、3 つのオブジェクト指向設計手法を統合した標準化言語であり、もともとはソフトウェアシステム設計のための言語であるが、近年になって UML をハードウェア含めたシステムレベルへの記述に適用しようとする動きが出てきている。UML に関しては 5.4 節で詳しく説明する。

(2) SLDL (Systems Level Design Language)^[24]

SLDL はこれまでに発表された数少ない多言語系システムレベル設計言語の 1 つである。多言語系言語とは、C や VHDL などという 1 つの言語をベースにするのではなく、言語自体が他の言語で記述されたシステムを取り込み統合できるハーネスのような仕組みを持った言語である。

SLDL は、EDA Industry Council のもとで 1996 年 10 月に活動を開始した SLDL Committee で検討が重ねられ、1999 年より VI 傘下の SLDL Initiative のもとに活動の場を移した。1999 年 6 月にシステムの設計制約記述を中心とした Rosetta (Phase I) を発表した。この Rosetta は、設計制約記述に関して Vspec 言語を参考にしている。2000 年 3 月に Rosetta の Java ベースのパーサを WWW ベースで公開した¹が、2000 年 9 月に VI が Accellera に統合されて C/C++言語も含めた標準化活動を開始したのに伴い、少し先の将来的な活動として位置づけられ、活動が鈍くなっている。

SLDL は以下のような言語的特徴を持つ。

- ◇ ハードウェア / ソフトウェア分割前のシステム記述が可能
- ◇ 複数の言語によるシステム記述を統合
- ◇ Phase-1 で要求と制約記述を実現 (Rosetta)
- ◇ Phase-2 で Bridging Semantics により多言語記述を取り込み、システムシミュレーションとコシンセシスの実現を図る。

(3) SDL^[28]

遠距離通信のプロトコル仕様を記述するために開発された言語で、ITU-T の標準

¹ <http://www.sldl.org/> から入手できる。

規格となっている。テキスト及びグラフィカルなエントリ言語が定義されており、主な特徴として、動作の階層記述、構造の階層記述、状態遷移記述、メッセージ送受信記述、時間記述を扱える。ツール環境としては、SDL から C、もしくは機能レベル VHDL を生成する COSMOS、SDL に UML のグラフィカル記法を組み込み、解析、設計、テストまでをサポートする Telelogic 社の Tau が知られている。

(4) Esterel^[29]

1980 年代初期にリアクティブシステム（リアルタイムシステムや制御装置等の様々な計算機システム）のプログラム言語として Sophia-Antipolis で開発され、1990 年代初期にハードウェア設計のための変更が行われた。カルフォルニア大バークレイ校で開発された POLIS システムのエントリ言語として採用されている。主な特徴として、動作の階層記述、同時発生動作記述、時間記述を扱える。

(5) その他

上記以外にも、並列システム（マルチプロセッサマシン上のプログラム）をモデリング、検証するために開発された CSP（Communicating Sequential Processes）、リアクティブシステムの仕様言語として開発されたペトリネット、StateChart や後に SpecC へと展開された SpecChart、ISO の FDT（Formal Description Technique）グループにより 1980 年代に開発された LOTOS 等の言語が知られている。

5.2. SystemC

5.2.1. SystemC とは

SystemC は、Synopsys 社、CoWare 社、Frontier Design 社の研究成果をもとに開発された C++ベースのシステムレベル設計言語である。1999 年 9 月に Version0.9 (以下、SystemC0.9)、2000 年 4 月に Version1.0 (SystemC1.0)、2000 年 7 月には Version1.1 の版 (SystemC1.1) が公開され、2001 年 2 月には、Version2.0 (SystemC2.0) の仕様²と Version1.2 の版 (SystemC1.2) がリリースされている。全てのバージョンのソースコードはオープン・ソース・ライセンス形態による無償提供である。SystemC2.0 に関しては、Synopsys 社、CoWare 社、Cadence 社、Motorola 社、STMicroelectronics 社、富士通などが中心となって仕様策定が行われた。

現在も活発に改良が進んでおり、今後は、ソフトウェア (RTOS など) のモデル化や、アナログ・ミックスドシグナルのモデル化などの機能拡張が計画されている。

以下、SystemC の特徴を列挙する。

- SystemC は、C++言語の文法を拡張せずに専用のクラスライブラリを追加することによって、ハードウェアおよびソフトウェアを記述することを可能にしている。具体的には、クロックや並列動作、信号、ハードウェア用データ型、割込みなどの記述が可能になっている。このように C++言語の文法を拡張しないことによって、既存の C++コンパイラやデバッガなどの標準的な開発環境をそのまま利用することが可能である。
- SystemC を用いてシステムの仕様を記述することによって、シミュレーション可能な仕様書 (実行可能な仕様書) として利用することが可能である。これによって従来、紙媒体を用いることによるシステム設計者とハードウェア・ソフトウェア設計者間のコミュニケーションの壁を低くすることができる。
- 単一の言語でハードウェアおよびソフトウェアのモデルを記述することができ、統一的に扱うことが可能になる。これによって高速な協調シミュレーションや統一的なデバッグが容易になる。
- ハードウェアを記述・検証するための機能や環境が整っている。例えば、任意の精度で固定小数点を扱うデータ型や、シミュレーション結果の視覚化 (波形表示ツールへのリンク) などが用意されている。このため、既存の HDL 設計環境からの移行が比較的容易である。
- システムブロックを機能 (ビヘイビア) 部分とブロック間の通信部分 (コミ

² 2001 年 2 月 1 日現在、SystemC2.0 仕様の正式な言語仕様および製品は 2001 年第 3 四半期にリリース予定。

コミュニケーション)を分離して設計することによって、ブロック間の通信プロトコルの変更が容易になり、モデル再利用性が向上する。SystemC2.0 では、コミュニケーション・リファインメントと呼ぶ手法によって、システムレベルの抽象的なブロック間コミュニケーション記述から、実装レベルの記述まで段階的に詳細化していくトップダウン設計が容易になる³。

5.2.2. SystemC1.0 の言語仕様

本項では、SystemC1.0 の言語仕様(文法と実行セマンティック)の概要を説明しているが、SystemC2.0 で新たに追加・変更された言語仕様については次項で説明する。

また、言語仕様の詳細については、SystemC1.0 および SystemC2.0 の仕様書^[33,34]を参照のこと。

【1】 SystemC モデルの基本構造

SystemC モデルの基本構造を図 5.2-1 に示す。

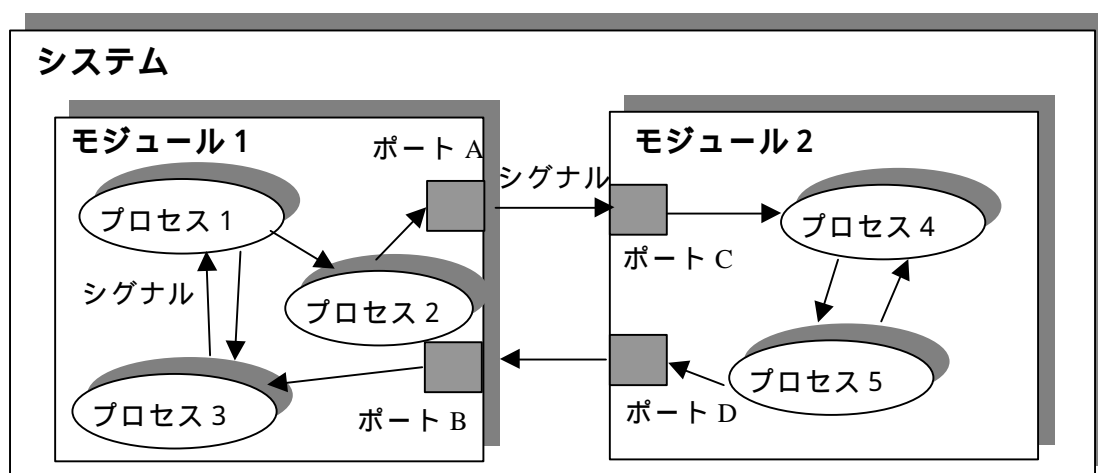


図 5.2-1 SystemC モデルの基本構造

システムは、並列に実行される複数のプロセスの組合せで構成されており、プロセスには機能を記述する。プロセス間の通信は、シグナルを通して行われ、モジュール記述を用いて階層的に記述することも可能。モジュールには、入出力のインターフェースとして入出力ポートを持ち、モジュール間の通信は、ポート間にシグナルを接続して行う。

³ SystemC1.2 からコミュニケーション・リファインメントは可能である。

以下、各構成要素の説明を行う。

- プロセス：

モデルの機能はプロセス内に記述する。プロセスは同時並行的に実行可能であり、プロセス内部のコードはシーケンシャルに実行される。プロセスには、以下の 3 種類のプロセスタイプ⁴がある。

- `sc_method`：非同期ファンクションプロセス

センシティブリティリストにある信号にイベントが発生する度に、プロセスが起動され、プロセス内部のコードを最後まで実行する。組合せ回路を記述することが可能。

- `sc_thread`：非同期スレッドプロセス

センシティブリティリストにある信号にイベントが発生する度に、プロセスが起動され、コード中の `wait()` 文まで実行され中断する。次にプロセスが起動されるときには、この `wait()` 文以下が実行される。非同期の RTL 記述が可能。

- `sc_ckpt`：同期スレッドプロセス

`sc_thread` と同じだが、クロックエッジのみに同期して実行される。

- データタイプ：

C++の基本データタイプに加えて、ハードウェアモデルを記述するための以下のデータタイプを用意している。モジュールはポートを持ち、ポートは型を持つ。

- 整数タイプ

<code>sc_int<length></code>	: 64 ビット以下の符号付き整数 (2 の補数)
<code>sc_uint<length></code>	: 64 ビット以下の符号なし整数
<code>sc_bigint<length></code>	: 64 ビット以上の符号付き整数 (2 の補数)
<code>sc_bignint<length></code>	: 64 ビット以上の符号なし整数

- ビットタイプ

<code>sc_bit</code>	: 2 値の論理 (0/1) の 1 ビット
<code>sc_logic</code>	: 多値の論理 (0/1/Z/X) の 1 ビット

- ベクタタイプ

<code>sc_bv<length></code>	: <code>sc_bit</code> の配列タイプ
<code>sc_lv<length></code>	: <code>sc_logic</code> の配列タイプ

⁴ SystemC1.0 での名称。バージョンによって名称が異なることがあるので注意が必要である。

➤ 固定小数点タイプ

sc_fixed	: 定数値を引数にとる符号付き固定小数点
sc_ufixed	: 定数値を引数にとる符号なし固定小数点
sc_fix	: 変数を引数にとる符号付き固定小数点
sc_ufix	: 変数を引数にとる符号なし固定小数点

固定小数点タイプには、すべて量子化とオーバーフロー時の処理方式を引数で指定する。

● シグナル：

シグナルは、ハードウェアにおける配線を表現しており、プロセスはシグナルを通して通信を行う。

● 待機：

wait()	: イベントが発生するまでプロセスの実行を待機
wait_until()	: 引数の数式が真になるまでプロセスの実行を待機 ⁵

【2】 シミュレーション方法

SystemC で記述されたモデルは、SystemC のシミュレーションカーネルを用いてシミュレーションされる。このシミュレーションカーネルは、サイクルベース方式のシミュレータであり、以下のような手順（評価・更新パラダイム）で実行される。

入力にイベントがある全てのプロセスを実行する。

プロセスが出力シグナルに値を書き込むときは、その値はすぐには反映されない。

全てのプロセスが実行された後にシグナルの値が更新される。

プロセスの実行順序はシミュレーション結果に影響を与えない。

【3】 記述例

実際に SystemC1.0 の記述例を図 5.2-2 のような簡単な 2 入力加算器の例で説明する。

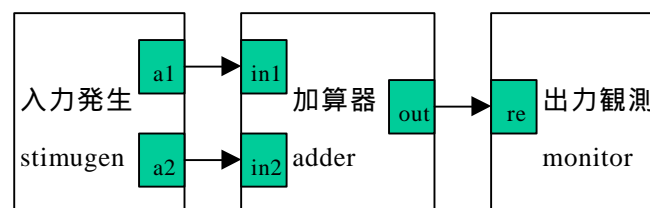


図 5.2-2 記述例モデル：加算器

⁵ SystemC1.0 の wait_until() の機能は、SystemC2.0 では、wait() の機能拡張によって統合されている。

また、図 5.2-3 には、SystemC 記述のファイル構成を示す。

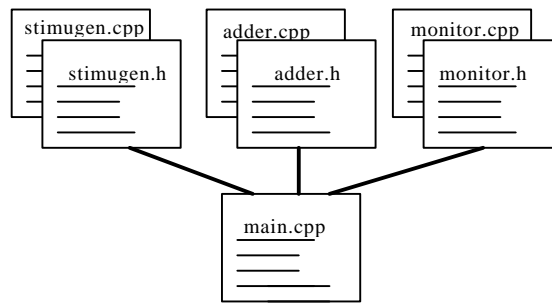


図 5.2-3 SystemC 記述のファイル構成

- インタフェース・ファイル (xxx.h)
各プロセスの入出力宣言、プロセスの種類などのプロセス宣言を記述する。
- インプリメンテーション・ファイル (xxx.cpp)
各プロセスの実際の機能 (処理・動作内容) を記述する。
- トップレベル記述・ファイル (main.cpp)
プロセス・モジュール間の接続、シミュレーション実行開始などのトップレベル記述を行う。

以下、実際の記述例を図5.2-4、図5.2-5、図5.2-6に示す。

```
// 加算器のインタフェース・ファイル adder.h

SC_MODULE( adder ) {
    // 入力ポート宣言
    sc_in_clk      CLK;
    sc_in<int>      in1;
    sc_in<int>      in2;

    // 出力ポート宣言
    sc_out<int>     out;

    // 機能実行メソッド宣言
    void entry();

    // コンストラクタ (モジュールのインスタンス化)
    SC_CTOR( adder ) {
        SC_CTHREAD( entry, CLK.pos() );
    }
};
```

図 5.2-4 インタフェース・ファイル (adder.h)

```
// インプリメンテーション・ファイル adder.cpp

#include "systemc.h"
#include "adder.h"

void adder :: entry()
{
    while( true ) {
        out = in1 + in2;
        wait();
        out = in1 + in2 + 2;
        wait();
    }
}
```

図 5.2-5 インプリメンテーション・ファイル (adder.cpp)

```
// トップレベル記述・ファイル main.cpp
#include "systemc.h"
#include "adder.h"
#include "stimugen.h"
#include "monitor.h"

int sc_main( int ac, char *av[] ) {
    // シグナル宣言
    sc_signal<int>    s1;
    sc_signal<int>    s2;
    sc_signal<int>    s3;

    // クロック生成
    sc_clock    clock( "CLOCK", 10, 0.5 );
    // モジュールのインスタンス化
    adder        Add( "MyAdd" );
    Add << clock << s1 << s2 << s3;
    stimugen    ST( "STIM" );
    ST( clock, s1, s2 );
    monitor    M( "MON" );
    M.clockint( clock );
    M.s3int( s3 );

    // シミュレーション実行 (200ステップ)
    sc_start( 200 );
}
```

図 5.2-6 トップレベル記述・ファイル (main.cpp)

5.2.3. SystemC2.0 の言語仕様

SystemC1.0 では、C/C++を用いたハードウェアのモデル化を主眼にしており、既存の HDL での RT レベルおよび動作レベルの記述が可能であるが、SystemC2.0 では、SystemC1.0 でのハードウェア記述に加えて、より抽象度の高いレベルからの設計を容易にするための機能拡張が行われている。

以下に SystemC1.0 から機能拡張された代表的なものについて列挙する。

- シグナル以外に抽象的なコミュニケーション手段（チャンネルやインタフェース）を用いてより抽象的なコミュニケーション記述が可能（定義済みチャンネルに加えてユーザ定義も可能）
- 段階的なコミュニケーションの詳細化（コミュニケーション・リファインメント）によって、効率的なトップダウン設計が可能。詳細は、5.2.4 項を参照
- 待機 wait()の機能拡張（引数としてイベントや時間を指定可能）
- プロセスの駆動条件（センシティビリティ）をシミュレーション中に動的に変更可能
- イベントをオブジェクト（sc_event）として扱うことが可能

また、互換性については SystemC1.0 の上位互換であるため、SystemC1.0 の記述は SystemC2.0 でもシミュレーション可能である。

5.2.4. SystemC による設計フロー

SystemC2.0 では、言語仕様と設計フロー（設計手法）を分離して考えており、特定の設計フローではなくさまざまな設計フローに適用できるように、自由度の高い言語仕様になっている。特定の設計フローに依存した言語仕様（クラスライブラリ等）は、Methodology-Specific ライブラリとして基本クラスライブラリと区別している。

Methodology-Specific ライブラリの一例として提案されているものに、マスター／スレーブ・コミュニケーション・ライブラリがある。これは、プロセッサや DSP、周辺回路、カスタム ASIC などの構成要素をマスター／スレーブ型のバスで接続するシステムを効率よくモデル化するために用意された専用クラスライブラリであり、抽象的なバス・コミュニケーション記述が可能になる。

本項では、マスター／スレーブ・コミュニケーション・ライブラリの仕様、このライブラリを用いた設計フロー、および具体的なコミュニケーション・リファインメントの例について説明する。

【1】 マスター／スレーブ・コミュニケーション・ライブラリの仕様

マスター／スレーブ・コミュニケーション・ライブラリは、バス構造を抽象的なレ

ベルで表現するためのアブストラクト・ポート型と、コミュニケーション・リファインメントによって実装レベルのバスへ詳細化するためのバス・プロトコル型がある。

- アブストラクト・ポート型：

アブストラクト・ポート型には、マスター・ポートとスレーブ・ポートが存在し、マスターとスレーブは対にして使用する。

- マスター・ポート：

`sc_master<>` , `sc_inmaster<>` , `sc_outmaster<>` , `sc_inoutmaster<>`

バス・マスター（CPU など）になるポートに対して指定する。

- スレーブ・ポート：

`sc_slave<>` , `sc_inslave<>` , `sc_outslave<>` , `sc_inoutslave<>`

バス・スレーブ（周辺回路など）になるポートに対して指定する。

これらマスター・ポートとスレーブ・ポートを用いたバス・コミュニケーションの一例を図 5.2-7 に示す。図中では、マスター・ポートには影を付け、スレーブ・ポートには影を付けないようにして区別している。

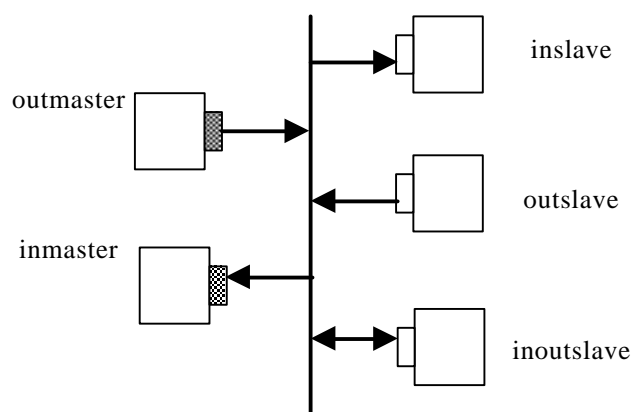


図 5.2-7 アブストラクト・ポート型を用いたバス・コミュニケーションの一例

- バス・プロトコル型：

バス・プロトコル型には、以下の 3 種類があり、図 5.2-8 に示す。

- 完全ハンドシェイク（Full Handshake）
- イネーブル・ハンドシェイク（Enable Handshake）
- ハンドシェイクなし（No Handshake）

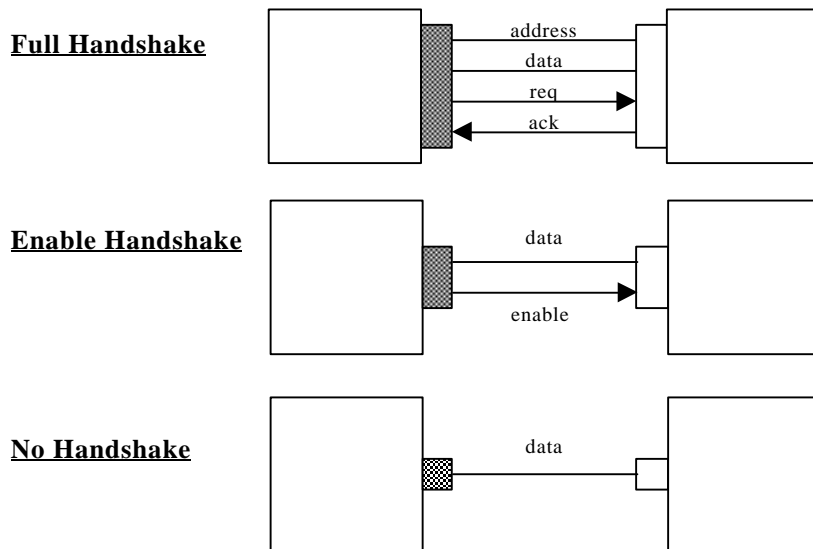


図 5.2-8 定義済みバス・プロトコル型

【2】 マスター/スレーブ・コミュニケーション・ライブラリを用いた設計フロー
 図 5.2-9 に、マスター/スレーブ・コミュニケーション・ライブラリを用いた設計フローを示す。

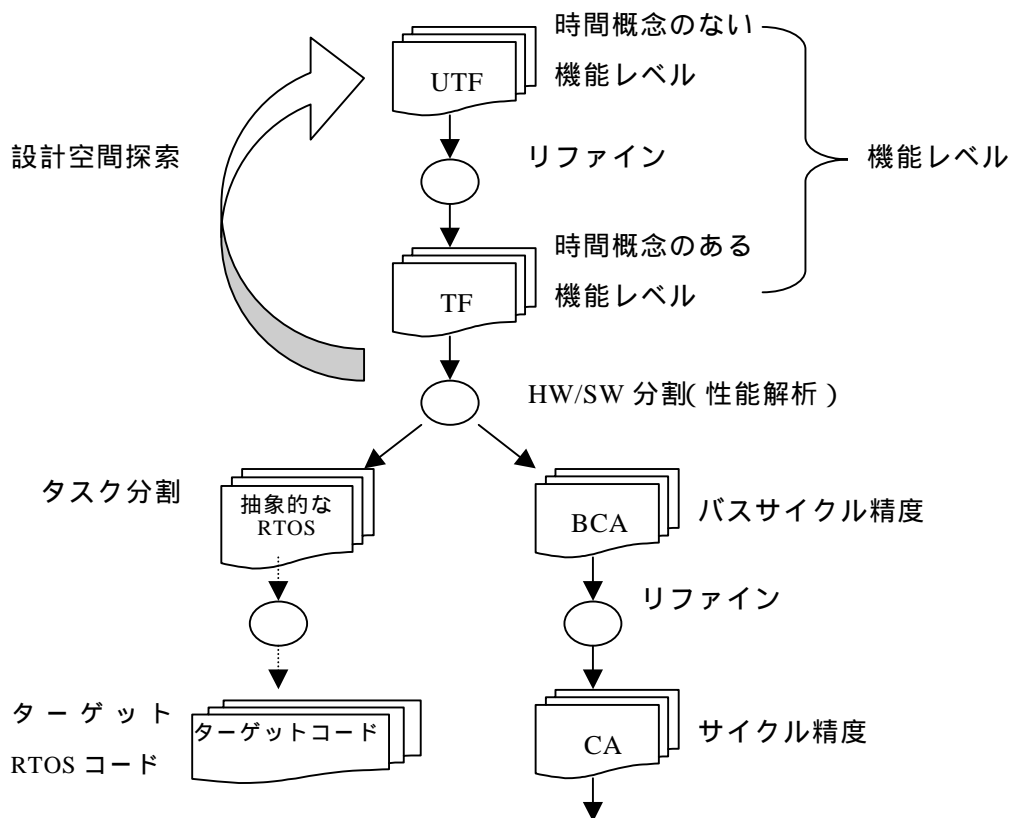


図 5.2-9 SystemC を用いた設計フローの一例

この設計フローでは、以下の設計抽象度を想定している。

- 機能レベル (Functional Level)

- 時間概念を含まない機能レベル (UTF: Untimed Functional)

まず、システムをクロックなどの時間概念をもたない手続き的な記述で機能を記述する。機能はシーケンシャルに実行され、ここでシステムの仕様、機能などの設計を行う。

- 時間概念を含む機能レベル (TF: Timed Functional)

次に、各プロセスに実行時間を与え、抽象度の高いレベルでの処理時間や性能の見積りなどを高速に行う。プロセスの相対的な実行順序は UTF と同じ

- バス・サイクル精度 (BCA: Bus Cycle Accurate)

上記、機能レベルで定義した抽象度の高いプロセス間のコミュニケーションは、クロック・サイクルの精度まで詳細化 (コミュニケーション・リファインメント) し、具体的なバス・プロトコル (Full Handshake、Enable Handshake、No Handshake) で実現する。ここでは、内部のプロセスは UT の記述でも可能。

- サイクル精度 (CA: Cycle Accurate)

プロセス内部も、クロック・サイクルで正確な動作になるように記述を修正することで、完全なクロック・サイクル精度のモデルとなる。これは、RT レベルと同等の記述レベル。

以上のように、時間概念のない抽象度の高い機能レベルから実装レベルであるサイクル精度レベルまで、設計レベルを段階的に詳細化していくことによって、設計の効率化・最適化を図ることができる⁶。

【3】 コミュニケーション・リファインメント

機能レベルからバス・サイクル/サイクル精度まで、プロセス間のコミュニケーションを詳細化するコミュニケーション・リファインメントの一例を図 5.2-10 に示す。

機能レベルの抽象度の高いレベルでは、各プロセスの入出力ポートにマスターかスレーブかの属性だけを指定する。例えば CPU はマスター、周辺回路はスレーブといった指定だけを行い、具体的なバス・プロトコルを意識しない抽象度の高いコミュニケーション記述が可能になる。次に、この抽象的なコミュニケーションを詳細化するために、具体的なコミュニケーション方法をバス・プロトコル (Full Handshake、Enable Handshake、No Handshake) から指定する⁷。機能レベルでは一つの信号線であったコ

⁶ SystemC2.0 では、ソフトウェアの設計フローについては、詳細に述べられていないが、HW/SW 分割後、抽象的な RTOS 上でタスク分割を行い、その後ターゲットの RTOS で実行可能なコードまで詳細化していく。

⁷ SystemC2.0 では、ユーザー定義のバス・プロトコルも指定可能である。

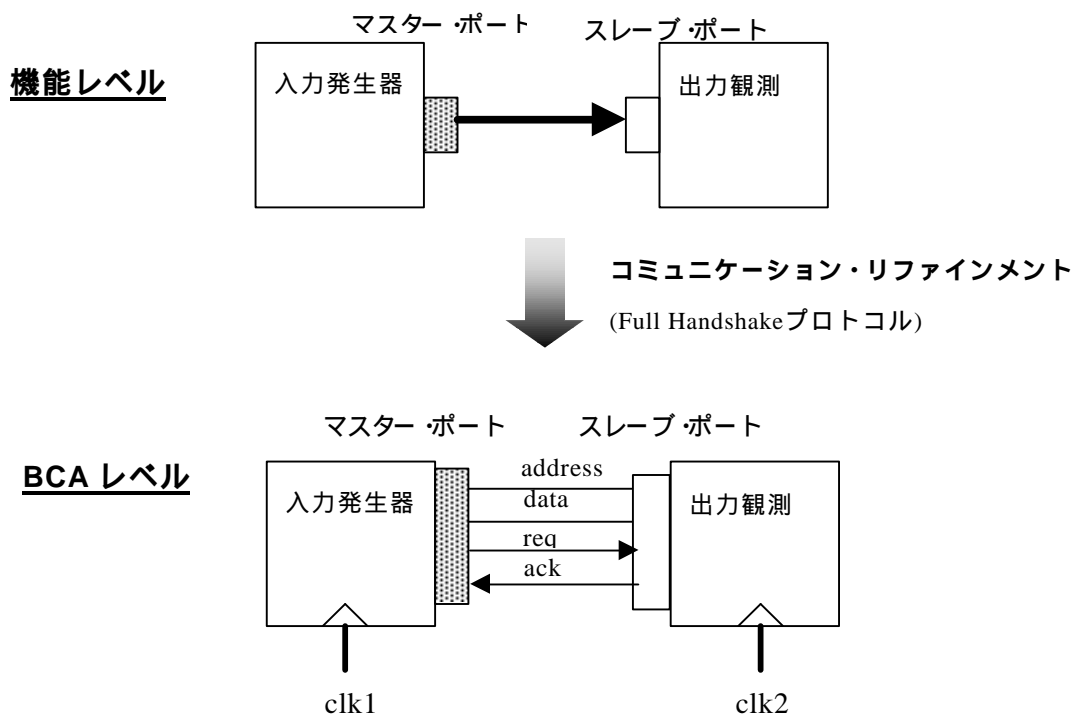


図 5.2-10 コミュニケーション・リファインメントの例

コミュニケーションが、BCA レベルでは指定したバス・プロトコルに従って、明示的に制御信号・データ信号線等を意識した設計を行う。例えば、図 5.2-8 の Full Handshake プロトコルの場合、address 信号、data 信号、req 信号、ack 信号でコミュニケーションが明示的に詳細化されている。バス・プロトコルを Enable Handshake に変更する場合は、機能レベルでバス・プロトコルの指定を変更すれば、容易に BCA レベルに反映される。以上のように、抽象度の高いレベルから設計を行うことによって、設計変更に対応できる設計が可能となる。

5.2.5. SystemC をサポートするツール

SystemC をサポートするツールは 2001 年 2 月 1 日現在、20 社 25 製品以上がリリースされており、ハードウェア/ソフトウェア協調設計ツール、ハードウェア/ソフトウェア協調検証ツール、ブロックエントリツール、シミュレータ、デバッガ、動作合成ツールなど、SystemC を用いた設計環境が整いつつある。以下に、代表的なツールを列挙しておく。上記以外のツールおよび最新情報については OSCI のホームページ^[20]または、各 EDA ベンダーのホームページを参照。

【1】 Synopsys 社 CoCentric family

3.3.1 項を参照。

【2】 CoWare 社 N2C

3.3.2 項を参照。

【3】 Frontier Design 社 A|RT Family

ANSI-C、SystemC 入力の動作合成ツール A|RT Designer および、固定小数点データ型ライブラリの A|RT library、RT レベルの SystemC 記述から HDL 記述への自動変換ツール A|RT builder から構成される。詳細は、Frontier Design 社のホームページ^[32]を参照。

5.2.6. 標準化団体 OSCI (Open SystemC Initiative)

OSCI は、システムレベルの協調設計や IP 再利用を可能にするための標準モデリングプラットフォームの策定および普及、推進活動を目的として、1999 年に米カリフォルニア州で設立された独立非営利の SystemC 標準化団体である。2001 年 3 月の時点でのメンバーには、米国を中心とした EDA ベンダーや半導体メーカー、IP ベンダー、システムハウスなどの企業・団体が参加している⁸。活動状況の最新情報は、OSCI のホームページ^[20]で公開されている。

⁸ 2001 年 2 月現在、13 社 (ARM, Cadence, CoWare, Ericsson, Fujitsu Microelectronics, Infineon Technologies AGs, Lucent Technologies, Motorola, NEC, Sony, STMicroelectronics, Synopsys and Texas Instruments) が中心となって活動中である。

5.3. SpecC 言語

5.3.1. SpecC 言語とは

SpecC 言語は 1997 年に UCI(カリフォルニア大学アーバイン校)の D.Gajski 教授等が開発したシステムレベル設計言語である^[38,39,40,41]。ANSI-C 言語をベースにしておりハードウェア / ソフトウェア混在システムの設計仕様を記述することができる。以下に、SpecC 言語の特徴を列挙する。

- SpecC 言語は、ANSI-C 言語をベースにて、並列性、割込み処理、状態遷移、通信といった動作仕様を記述するための拡張構文を用意して文法的な拡張を行っている。これにより、コンパクトで可読性の高い記述が可能であるが、この一方で、SpecC コードを処理するためには専用ツールが必要となる。
- SpecC 言語で記述される設計仕様は、システムレベルの仕様記述として位置づけられるために、記法上はハードウェアとソフトウェアの文法の差異がない。すなわち、実装の詳細をニュートラルな状態のままで、各々の機能をハードウェアあるいはソフトウェアとして実現した場合のトレードオフを容易に分析・検討することができる。
- システムレベル設計において必要なさまざまな抽象レベルのモデル、たとえば、システムの機能仕様記述、システムのアーキテクチャ記述、さらにはプロセッサやバスなどのシステム構築部品までを単一の言語で統一的に記述できる。また、各設計段階で常にシミュレーションが可能である。
- SpecC 言語を利用することによって、仕様から設計へとシステム開発を進める際に、連続的に開発の基幹データを共有することができる。また、ビヘイビア (behavior) とチャネル (channel) の二つのクラス構造を使って仕様記述を行う点でオブジェクト指向設計の手法を取り入れており、IP あるいは設計データベースの構築にも向いており、これにより設計再利用の実現が容易になる。
- モデルの機能と接続 (通信) が完全に分離でき、コンポーネントを階層的に分解して定義することができる。動作の階層化ではシステムを逐次実行と並列実行の部分動作ごとに分解して定義できる。構造の階層化によって、コンポーネント間の接続構造を表現できる。さらに仕様記述のモジュール化によって設計の再利用も容易に行えるよう考慮されている。

5.3.2. SpecC 言語の文法

【1】 SpecC プログラムの基本構造

SpecC 言語の持つ計算モデルは Program-State Machine(PSM)^[38,39]と呼ばれる。PSM とは、階層型並列ステートマシンと手続き型プログラミング言語を組み合わせたモデルである。

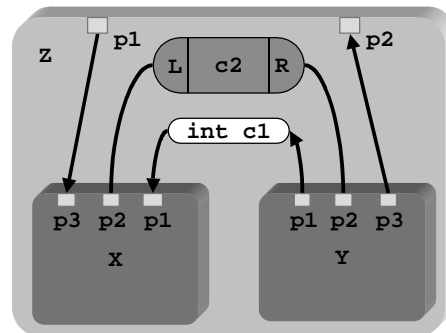


図 5.3-1 SpecC プログラムの基本構造

図 5.3-1 に SpecC プログラムの構造を簡単に示す。システムの全体は、階層的に積み上げた「ビヘイビア」と呼ぶブロック（図 5.3-1 の X,Y,Z）によって表現する。それぞれのビヘイビア間で行う通信は、各ビヘイビアに定義するポートとポート間の接続関係によって確立する。通信プロトコルは「チャンネル」と呼ぶブロック（図 5.3-1 の c1 , c2）の中で手続きを定義することによって表現する。

ビヘイビアあるいはチャンネルの記述は、オブジェクト指向的なプログラミングによって行える。すなわち、各ビヘイビアとチャンネルの仕様をクラスとして定義して、インスタンスの展開によってシステムの構造的な階層構造が決定される。

図 5.3-1 のモデルを SpecC 言語で記述したコードが図 5.3-2 である。SpecC プログラムは、ビヘイビア（behavior）、チャンネル（channel）、インタフェース（interface）の三つの要素からなる。この例ではビヘイビア Z が二つのサブ・ビヘイビア x と y を持っている。x はビヘイビア X のインスタンスであり、y はビヘイビア Y のインスタンスである。par 文の中で、x と y が並列に実行されると定義されている。x と y の間では整数 c1 とチャンネル c2 を経由して通信を行う。

```

interface L {
    void write(int x);
};

interface R {
    int read(void);
};

channel C implements L, R // チャネルCはLとRのインタフェースを持つ
{
    void write(int x) {
        ...
    }

    int read(void) {
        ...
    }
};

behavior X(in int p1, L p2, in int p3) // ビヘイビアXの定義
{
    void main(void) {
        p2.write(p1);
    }
};

behavior Y(out int p1, R p2, out int p3) // ビヘイビアYの定義
{
    void main(void) {
        p3 = p2.read();
    }
};

behavior Z(in int p1, out int p2) // ビヘイビアZの定義
{
    int    c1;    // 整数変数
    C      c2;    // チャネル変数
    X x(p1, c2, c1); // 子ビヘイビアxの宣言とポート接続
    Y y(c1, c2, p2); // 子ビヘイビアyの宣言とポート接続

    void main(void)
    {
        par{ x.main( ); y.main( ); } // xとyは並列
    }
};

```

図 5.3-2 SpecC プログラムの例

【2】 ビヘイビアの階層化

ビヘイビア階層には、逐次実行と並列実行の二つの実行形態がある。逐次実行は C 言語と同じ命令文あるいはステートマシンによって表現する。並列型の実行は、par あるいは pipe 構文を使って表現する。

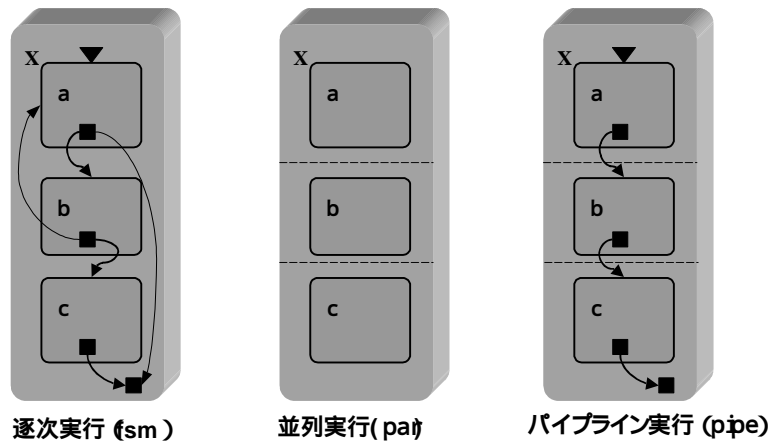


図 5.3-3 ビヘイビアの階層実行

図 5.3-3 の fsm はステートマシンを表す。par は並列動作を表し、ビヘイビア X の中では、サブ・ビヘイビア a、b、c が並列に実行される。サブ・ビヘイビア a、b、c がすべて終了すると X の実行が終了する。図 5.3-3 の pipe はパイプライン実行を表す。一回目は a だけを、二回目は a と b を、三回目の繰り返しでは a と b と c を実行し、以後はエンドレス・ループになる。

ステートマシンの実行は fsm 文を使って図 5.3-4 のように宣言する。C 言語の case 文の記法と似ており、条件文の評価と goto 文で状態遷移先のビヘイビアが決まる。break で fsm 文が終了する。例では、ビヘイビア a を実行した後で x が正の場合は fsm 文が終了する。0 または負の場合はビヘイビア b に遷移する。階層的なステートマシンを定義するときは、下位ビヘイビアにおいて fsm 文を使用する。

```
fsm{
  a:
    if(x>0) break;  // fsm文の終了
    if(x<=0) goto b;
  b:
    if(y>0) goto a;
    if(y==0) goto b;
    else goto c;
  c:
    break;  // fsm文の終了
}
```

図 5.3-4 fsm 構文の使用例

並列実行は par 文を使って図 5.3-5 のように宣言する。例では、ビヘイビア a、b、c を並列に実行し、すべてが終了すると par 文が終了する。

```
par{ a.main(); b.main(); c.main(); }
```

図 5.3-5 par 構文の使用例

パイプライン実行は pipe 文を使って図 5.3-6 のように宣言する。

```
pipe{ a.main(); b.main(); c.main(); }
```

図 5.3-6 pipe 構文の使用例

【3】 同期

並列に動作するブロック間での同期を取るための手段としてイベント変数がある。イベント変数は値を持たないが、wait 文や notify 文、あるいは例外処理構文で利用する。wait 文では、指定されたイベントの到着するまで停止する。notify 文はイベントの通知を意味する。

【4】 例外処理

図 5.3-7 に SpecC 言語が持つ二種類の例外処理を示す。例外処理には放棄 (trap) と割り込み (interrupt) がある。

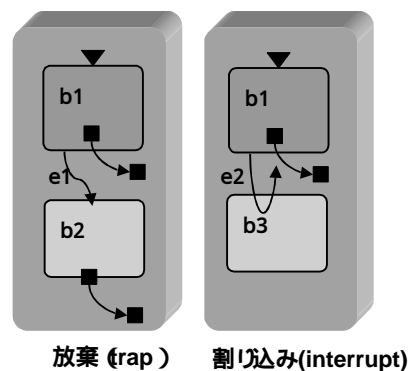


図 5.3-7 例外処理

図 5.3-8 の trap では、イベント e1 が発生すればビヘイビア b1 の実行を停止して割り込み処理 b2 を実行して try 文を終了する。このタイプの例外処理は主にシステムのリセットを記述するために用いる。一方、interrupt では中断処理を記述することができる。図 5.3-8 では、イベント e2 が発生すればビヘイビア b1 を中断して b3 を実行する。b3 の実行後は、残りの b1 の処理が行なわれる。

```

behavior B;
behavior B1;

behavior B_except(in event e1, in event e2)
{
    B      b1;
    B1     b2, b3;

    void main(void)
    {
        try { b1.main( ); }
        trap(e1){ b2.main( ); }           // 放棄
        interrupt(e2){ b3.main( ); }      // 割り込み
    }
};

```

図 5.3-8 例外処理の記述例

一つの try 文の中で、trap あるいは interrupt でモニターされているイベントの中の二つ以上が同時に発生した場合は、最初にリストされているイベントに優先権がある。

【5】 タイミング

タイミングを記述するための構文としては、waitfor 文と do-timing 文が用意されている。waitfor 文は、指定された時間だけ正確にビヘイビアの実行をサスペンドする仕様を記述する際に用いる。do-timing 文では時間範囲の形でタイミングに関する制約条件を記述する際に用いる。

【6】 拡張された変数型

SpecC 言語には ANSI-C 言語ではサポートされていないが、追加された変数型として以下の三つがある。

- ブーリアン・データ型 (bool)
- 任意長のビット・ベクタ型 (bit)
- 同期処理用のイベント型 (event)

5.3.3. SpecC 言語による設計方法論

SpecC 言語による設計手法を図 5.3-9 に示す。図に示されるように設計プロセスは大きく「機能設計」、「アーキテクチャ設計」、「実装設計」からなる。

「機能設計」では、ステートマシンとアルゴリズム記述によってシステムの機能的な振る舞いを記述する。この機能仕様には実装の詳細は含まれていないが、通信と計算のそれぞれの機能が一通り記述されるので、シミュレータによる実行が可能となる。このステップは、製品設計の初期段階におけるラピッド・プロトタイプを開発する工

程に相当する。つまり、製品企画と基本機能を検討している段階で、機器の動作、製品の見た目、使い勝手などをユーザと同じ視点で検証するためのモックアップを開発することで、仕様の誤認や解釈の違いによる下流での設計変更・修正の削減が図れるなどの効果が期待できる。

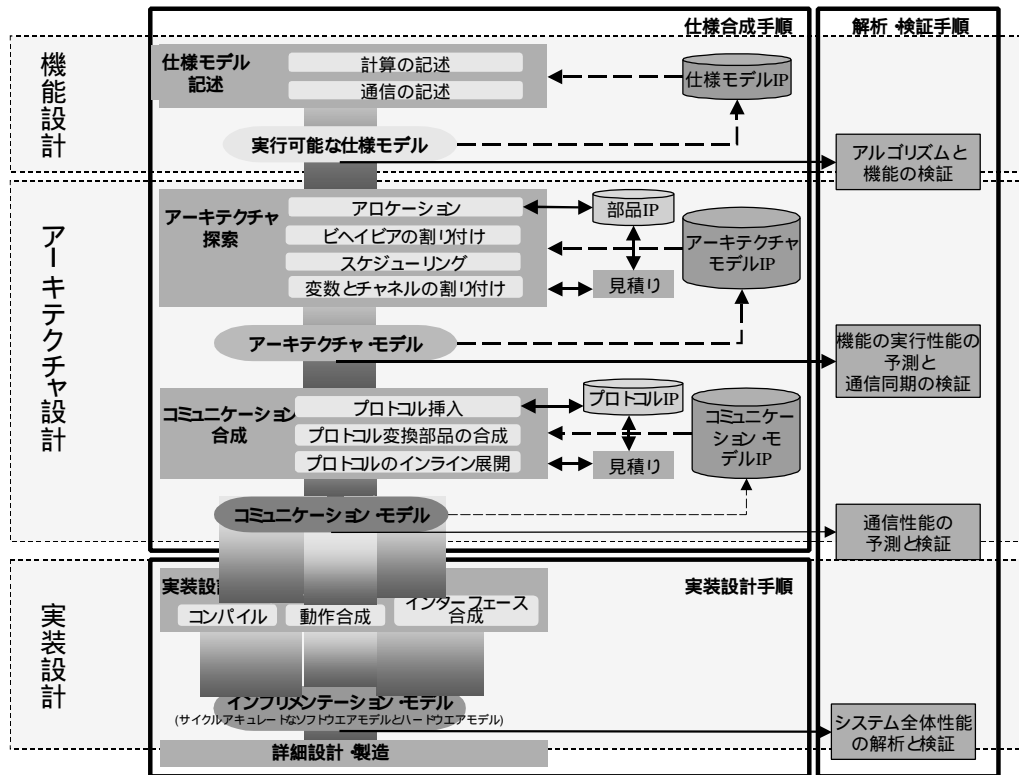


図 5.3-9 SpecC 言語による設計フロー（文献[38]の p.70 より）

「アーキテクチャ設計」は大きく「アーキテクチャ探索」と「コミュニケーション合成」に分かれる。「アーキテクチャ探索」では、まず、プロセッサ、メモリ、バス、カスタム・ハードウェアといったシステムを構成するコンポーネントの割り付け（アロケーション）を行う。次に、各ビヘイビアを各ハードウェアあるいはソフトウェア・コンポーネントに分割する。さらに、並列に動作する機能間での割り当ての変更と実行順序の依存関係の解析による並び替え（スケジューリング）を行う。これとともに、変数およびチャネルで表現されたビヘイビア間の通信をバス・コンポーネントに割り付ける。

「コミュニケーション合成」では、まず、バス・プロトコルの選択を行って通信部のモデル記述の具体化を行う。これは標準的なプロトコル IP を蓄積したデータベースから検索する形になる。次に、インタフェース回路とプロトコル変換部の合成を行う。最後に、プロトコル仕様の部分をチャネルからビヘイビアにインライン展開を行う。

「アーキテクチャ設計」を終えて「実装設計」に引き渡される設計仕様（コミュニケーション・モデル）は、バックエンドの下流設計ツールにそのまま渡せる形式になる。つまり、インタフェース・モデルの設計仕様に含まれるソフトウェア部分の SpecC 言語コードは、C 言語のソースコードとして取り出せる。一方のハードウェア部分はビヘイビアレベルの HDL 記述として取り出すことができる。下流設計ツールとは、既存のコンパイラ、動作合成ツール、インタフェース合成ツール等が相当する。

SpecC 言語による設計手法の特徴は次の二点にある。

- 機能設計から実装設計に至る各設計作業が、すべて、SpecC 言語によって記述された前段階の設計モデルに対して、なんらかの置き換えあるいは洗練を行う形で定義されている。
- それぞれの設計作業を一貫的にかつ連続的に実行でき、抽象的な機能仕様の段階から実装設計までがシームレスにつながる。

また、各設計ステップでは、シミュレーションによる動作検証を行うことができる。たとえば、機能仕様の段階では、記述されている機能が設計意図に合致するかをシミュレーションで検証できる。機能分割とスケジューリングが終わった後では、異なるプロセッサ部品に振り分けられたビヘイビア間が仕様どおりに同期するかをシミュレーションで検証できる。コミュニケーション合成の後では、設計モデルを使ったシミュレーションで、演算部と通信部を含めたシステムの性能まで検証することができる。

さらに、各設計ステップでの設計資産を IP データベースとして蓄積・再利用ができるがこれらの IP 部品も SpecC 言語を使って開発する。

5.3.4. SpecC 言語をサポートするツール

図 5.3-10 は SpecC 言語の標準化団体である STOC (SpecC Technology Open Consortium) が公開した SpecC 技術に関する設計ツールの関連図である。上流設計言語として UML や SDL、ソフトウェアの下流設計言語として Java や C++、ハードウェアの下流設計言語として既存の HDL や C/C++ 拡張型の HDL などを想定し、システムレベル設計の中核として SpecC 言語を位置づけている。STOC では、上流・下流設計言語との変換手法とツールの整備を推進している。特に下流設計へのパスの観点では、SpecC 言語による設計と実装設計あるいはハードウェア・ソフトウェア協調シミュレーションとの有機的な結合を目指している点で注目される。

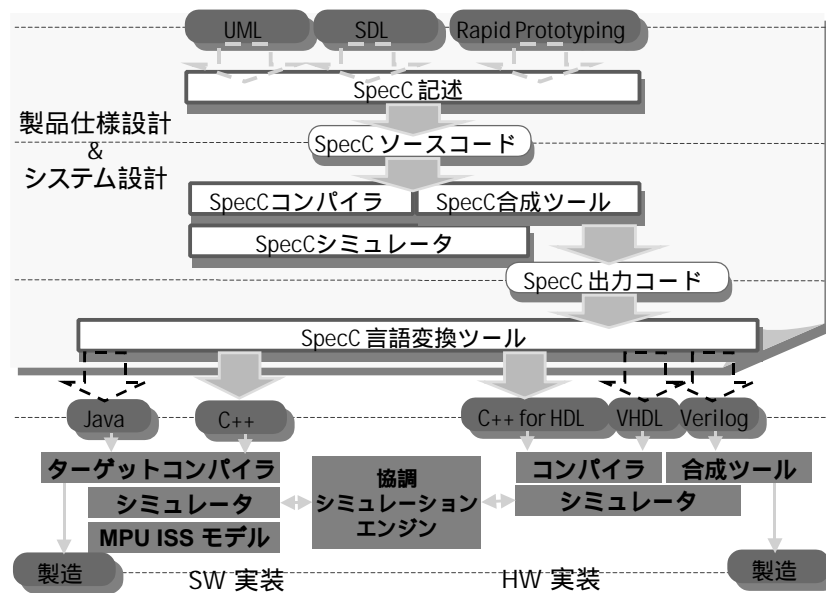


図 5.3-10 SpecC 設計ツールチェーン（STOC 公開資料^[18]による）

【1】 SpecC コンパイラ

UCI（カリフォルニア大学アーバイン校）から公開されている SpecC コードの実行環境。SpecC コードを C++コードに変換するコンバータと、C++コードを実行するためのランタイムエンジンからなる。Solaris と Linux 用が UCI の SpecC プロジェクトのホームページ^[17]で公開されている。

なお、2001 年 6 月には、後述の SpecC Technology Open Consortium（STOC）が公認するリファレンス環境が、ライセンスフリーのソースコードとして公開される予定である。

【2】 VisualSpec

ソリトン社および CATS 社から販売されている、SpecC 言語を利用した仕様記述の作成、および機能仕様をアーキテクチャレベルの設計仕様に合成するプロセスを支援する仕様オーサリングツール^[42, 43]である。

VisualSpec は、SpecC 言語の持つ各構文に合わせて、プログラム状態間の階層関係や有限状態機械、通信、例外処理の構造と動作をビジュアルに編集することができ、仕様の記述・詳細化を支援する。また、SpecC コード上でのソフトウェア/ハードウェア分割などの合成コマンドが用意されている。

これ以外に、Windows、WindowsCE、iTRON などのターゲット OS 上で稼動する SpecC 言語のランタイムエンジンおよびデバッガが用意されている。また、SpecC コードが

ら VHDL コードへの変換ツール、SpecC コードの IP ライブラリの開発ツール、ドキュメント作成ツールなどがオプションとして用意されている。

5.3.5. 標準化団体 STOC (SpecC Technology Open Consortium)

STOC は、SpecC 言語をもとにした業界標準のシステム仕様記述言語と仕様データ・フォーマット、および設計手法の開発と普及を目的として、1999 年 11 月 10 日に設立された。

STOC には日米の組込みソフトウェア開発ツールベンダー、EDA ベンダー、自動車、電機、通信、情報・電子機器などの応用システム製造業、および大学など、約 50 の団体が参加している (2001 年 1 月現在)。活動状況は、STOC のホームページ^[18]で公開されているとともに、米国での DAC (Design Automation Conference)、ESC (Embedded Systems Conference)、国内の主要展示会でコンソーシアムのブースが設置されるとともに、セミナー等も開催されている。

2001 年 1 月には言語仕様 1.0 が公表されたが、この言語仕様は UCI が開発したオリジナルの言語仕様とほとんど差異はなく、コンソーシアムとしての公式バージョンを定めた形になっている。また、2001 年 6 月にオープン・ソースコードでライセンス・フリーなりファレンス・コンパイラの公開が予定されている。

STOC では二つのワーキンググループ活動が行なわれている。

- 事例 WG は、SpecC 言語による事例の仕様記述とその過程で得られるノウハウの収集を行い、SpecC 言語を用いる設計のガイドラインや方法論などの整備を目的とした WG である。WG メンバーが作成する SpecC 言語の応用アプリケーションやソースコードをベースにして、評価やノウハウの交換が行われている。

言語仕様 WG は、SpecC 言語の言語仕様の策定を目的とした WG である。現行の SpecC 言語仕様の評価とその拡張を検討し、より強力で、効率良く仕様記述が可能な言語仕様を策定する。UCI などを含むワールドワイドなメンバー構成で、関連ツール (コンパイラなど) の標準化も視野に入れた活動が行われている。

5.4. UML

5.4.1. UML とは

UML は、Unified Modeling Language(統一モデリング言語)の省略語であり、言語という名前が付いているが、いわゆる一般のプログラミング言語とは異なり、ビジュアルな図表記を用いて、対象をモデル化するための言語である。この言語は、大規模で複雑なソフトウェア開発で生じる問題を解決するために、オブジェクト指向技術を集大成して生まれた。UML で注目すべき点として、この言語は、ソフトウェアシステム開発だけでなく、ハードウェアを含むシステム開発分野全般にも適用できる柔軟性を持っているということである。

【1】 開発背景

ハードウェア開発が年を追って規模が大きく複雑になってきたように、ソフトウェアシステム開発も、コンピュータネットワークや情報技術の発達により同様な道をたどってきている。ソフトウェアはハードウェアと異なり、メモリ上や磁気デバイスに格納されているため、ハードウェアのような実体として認識しにくいこと、完成してからプログラムの書き換えにより修正可能という特徴をもっている。そのため、特に大規模なシステムを構築するときには、要求された仕様を間違いなく実現するだけでなく、将来のための改良や保守を容易に実施できることを考慮して設計する必要がある。

ソフトウェア開発研究者達は、このようなソフトウェア設計上の課題に対し、再利用性を高めることや構造化設計手法などを発案し、システム開発の効率化に取り組んできた。1990 年以降では、さらに効果的な解決手段として、オブジェクト指向を取り入れた設計方法論が論じられてきた。特に、オブジェクト指向設計方法論のなかでは、Grady Booch 氏が開発した Booch 法、Ivar Jacobson 氏が開発した OOSE 法(Object Oriented Software Engineering)や James Rumbaugh 氏の開発した OMT 法(Object Modeling Technique)等が成果をあげた。ところが、これら 3 つの開発方法論は、システムを表現する方法をそれぞれ独自に定義していたため、仮に同じモデルを設計しても個々の方法論ごとに互換性のない設計図(モデル)ができあがる事態となってしまった。

大規模なシステム開発には、通常多数のエンジニアが関わる。このとき、個々のエンジニアが異なる設計方法論を採用すると、設計方法論ごとの表記法で記述されたサブシステムが完成することになる。この場合、完成したサブシステムモデルは、同じ設計手法を取り入れたエンジニア以外には表記法の違いにより理解しにくい。さらに異なる設計方法論を採用するエンジニア同士は、モデルに対してコミュニケーションも取りずらく、システム全体を見るとときに誤解を招きやすいなどの問題が生じていた。

【2】 開発歴史

このような状況のなか、1994年にBooch氏のいたRational社^[44]にRumbaugh氏が加わったことによって、Booch法とOMT法の統合作業が始まった。その後、そこにJacobson氏も加わり1996年にBooch法、OOSE法、OMT法を統合したモデリング言語としてUMLが考案された。Rational社は、UMLを標準化するためにUMLパートナーコンソーシアム設立を呼びかけ、そこでのブラッシュアップ後、1997年にオブジェクト指向モデリング言語標準化団体OMG(Object Modeling Group)^[45]に対してUMLをモデリング言語として提出した。その結果、UMLは標準的なモデリング言語として広まり、以後、図5.4-1に示すようにOMGの下でいくつかの改定をされ現在に至っている。

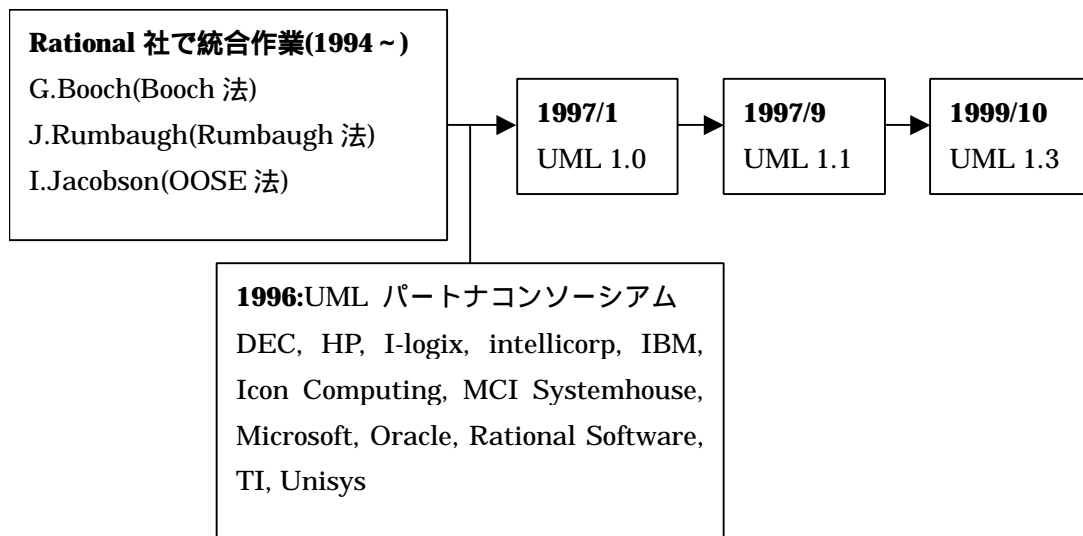


図 5.4-1 UML 開発の歴史

UMLは、文頭に記したがソフトウェアアプリケーションのみならず、ハードウェアを含むその他様々なシステムのモデリングが可能ないように設計されている。UMLに関する最新情報はRational社^[44]やOMG^[45]から得られる。

【3】 開発目的

UMLは、次に示すことを目的として開発されている。

- システム開発者の誰でも理解しやすい視覚的に優れたモデリング言語(記述法と構文)を提供することで、システム設計者が意味があり交換可能なモデルを容易に開発することができる。
- モデルを理解するための正式に定められた基礎(または基礎技術)を提供する。
- (モデルの特性に合わせて)記述表現を拡張するためのメカニズムを持つ。
- 特定のプログラミング言語や開発プロセスに依存しない。

「モデリング言語」は、システムの論理的側面や物理的側面を表現するための言語

である。開発者は、図形を使ってシステムを表現することにより、様々な視点からシステムを観察できるようになる。従って、モデリング言語の定義は、システム開発効率や情報交換において有効であると考えられた。

UML は正しい構文が定義されているので、モデル開発者だけでなく、開発したモデルを参照する他の開発者やツールが、記述したとおり正しく理解 / 認識することができる。この結果、UML モデルのプログラミング言語への変換（フォワードエンジニアリング）や、またはプログラミング言語から UML に逆変換（リバーズエンジニアリング）することが容易にできる。

UML は、モデルを表現するときに使用する記述要素や定義を柔軟に拡張できるように設計されているため、モデリング対象物が現在定義されている表現でカバーできない場合には、表記法を追加できる。この機能があるため、将来においてもモデリング能力の柔軟性が保証されている。それに加え、IBM によって開発されたオブジェクト制約言語 OCL(Object Constraint Language)^[46]も UML に含めて提供されているので、統一された記述法に従って制約を記述することもできる。OCL についての詳細な情報は、OCL Specification^[46]から得られる。

ただし、UML は開発プロセスに依存しないように開発されたので、どのようなモデルを作れば良いのか、またそれをいつ作れば良いのかについて、関知しない。作成すべきモデルの種類や作成時期等は、システム開発者や開発グループに委ねられる。

UML の発表以降、モデル開発において情報を交換するための手段として多くのソフトウェア開発者やツールベンダーからサポートされ、モデル開発ツールなど開発環境も整ってきた。特にオブジェクト指向ソフトウェア開発環境下では、UML は標準的な地位となり、1999 年ごろからは、組込みソフトウェア開発分野においても、注目されつつある。

5.4.2. UML の概要

以下の内容は、UML 1.1 Summary^[47] UML 1.1 Notation Guide^[48]及び UML 1.1 Semantics^[49]から抜粋した。また、本節と次節の内容は参照した企業ホームページ [44,45,46,51]や、参考文献 [52,53,54]に詳しい内容が記載されている。

【1】 ダイアグラム

UML は、複数の視点から捉えた様々なモデル（ダイアグラム）を用いて、開発システムを表現する。UML のダイアグラムには次のようなものがある。

- ユースケース図
- クラス図

- 振る舞い図
 - ステートチャート図
 - アクティビティ図
 - 相互作用図（シーケンス図、コラボレーション図）
- 実装図
 - コンポーネント図
 - 配置図

なお、システムのモデル化は、上記に記した全てのダイアグラムを用いて作成しなければならないわけでは無い。モデルに必要なダイアグラムのみ作成するだけで良い。UML は特定の開発プロセスに依存しないが、統一された 3 つのオブジェクト設計手法が持っていた次の設計手法とは相性が良い。

- システムが提供すべき機能単位にモデル設計する...ユースケースドリブン
- システムの構成や論理の組合せを中心に考える...アーキテクチャセントリック
- 繰り返ししながら徐々に開発する...イテレーション

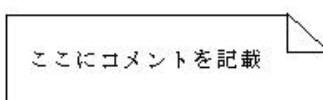
一般的に、UML を使ってシステムモデルを作成する場合には、要求仕様からシステムが提供すべきもの（ユースケース）を抽出し、形式的な仕様書（シナリオ）を作成した後、各種ダイアグラムを作成する方法が取られている場合が多い。

【2】 基本要素

UML の各ダイアグラムは、いくつかの基本要素を組み合わせて記述される。基本要素には、「基本部品」と基本部品同士を結びつける「関係」がある。

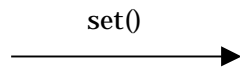
各ダイアグラムによっては使われる基本要素が異なるが（用いる基本要素によりダイアグラム名称が異なるとも言える）、どのダイアグラムにも用いられる一般的な基本要素のうち、いくつかを以下に説明する。基本要素など表記法については、ラショナル社や OMG から入手できる UML 表記法ガイド[47,48,49,50]に記述されている。UML 表記法ガイドは Web[45]で公開されている。

- ノート
 - モデルに関する説明や注釈をつけるための要素。端を折り曲げた四角形で表され、四角形の内部にコメントを記述する。



- 相互作用

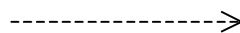
複数のオブジェクトの間で交換され、特定の目的を遂行するメッセージ群やアクションのシーケンスなどの要素。クラス図、コラボレーション図、シーケンス図、ステートチャート図などで使用される。実線矢印で表される。設計が進むと矢印の上に操作を示す関数名が記されることもある。



基本部品同士を結びつける「関係」を表す図形は主にクラス図、パッケージ図、コンポーネント図及び配置図など構造を示すダイアグラム内に使われる。

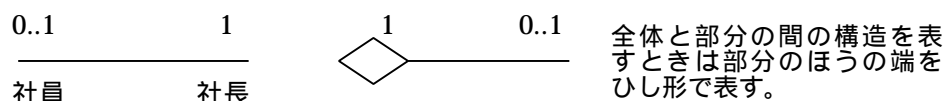
- 依存関係

2つの「もの」の依存状態を示す関係で、一方の「もの」への変更がもう一方の「もの」に対して影響を与えるときに、この関係を結ぶ。点線矢印で表される。



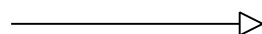
- 関連関係

クラスやオブジェクト間の構造や意味的なつながりを示す関係である。クラス間の場合は「関連関係」となり、オブジェクト間の場合では「リンク関係」になる。実線で記述され、中央に関連名や関連方向が、両端にはロールと呼ばれるその関連の役割を表す要素がついている場合もある。



- 汎化関係

一般的な要素と、それを特化した要素の関係を表すものである。例えば、図形と三角形や四角形は、図形が汎用的なものであるのに対し、三角形や四角形は特殊なものである関係となっている。白抜ききの三角形がついた実線で表す。



【3】 拡張メカニズム

UMLはシステムを表すための記述言語であるが、標準的に提供された表現を用いて全ての領域に渡り全てのニュアンスを表現できるとは限らない場合も存在する。これを解決するために、決められた方法（ステレオタイプ、タグ付き値、制約）に従って開発者がUMLを拡張できる仕組みを備えている。

- ステレオタイプ

UML の基本要素を拡張して、開発者固有の要素を作るメカニズムである。例えば、ソフトウェア処理において例外発生を受け取るオブジェクトを定義するとき、それを明示するために<<例外>>と記述することができる。このように定義することにより、よりモデルの持つ意味を明確にしたり、あるいはツールがそれをサポートすることにより、UML からプログラミング言語に自動変換する手段（フォワードエンジニアリング）の実現が容易になる。

- タグ付き値

モデルの要素が持つプロパティを拡張し、開発者独自のプロパティを付加できる。プロパティ文字列は、{プロパティ=値}で定義でき、例えばモデル要素であるクラスなどでは、開発者名や改版履歴をプロパティとして持つように定義したりする。

- 制約

制約は、モデル要素やモデル要素間の関係に対し、制限や条件を定義するものである。制約を定義することによって、あるクラス（またはオブジェクト）の関係を示したり、ある操作方法の条件を示したりすることができる。{ }で囲まれた中に制約を記述する。UML では制約記述言語 OCL (Object Constraint Language) が規定されているので、これを用いて制約を記述することができる。

例

遅延は許す(絶対ある)が、遅延時間を 100ns に制限するという OCL 記述

```
{self.delay -> notEmpty implies self.delay.time <= 100 ns}
```

5.4.3. UML ダイアグラム

ここからは、各ダイアグラムについて例を示しながら説明する。あわせて、各ダイアグラム内で用いられる基本要素についても、説明する。

【1】 ユースケース図 (Use Case diagram)

システムがどのように機能すべきかの振る舞いをユースケースと呼び、そのシステムを用いる外部環境をアクターと呼ぶ。ユースケースとアクターの関係を表した図がユースケース図である。システムを分析し、そのシステムが持つユースケースの開始 / 終了条件や前提条件及び振る舞い後の状態を記述したイベントフロー(図 5.4-1)や、ユースケースの実例を示したシナリオ(図 5.4-2)を作成したのち、図 5.4-3 に示すユースケース図を作成する。イベントフローやシナリオは、テスト仕様書の源流として利用できる。

タイマー設計ボタンを押すと、時間設定が行える。

1. 時間（分）設定する場合

設定ボタンを押した直後は、分を設定するモードになる。このモードのときには、設定時間表示画面のうち、分を示す部分が点滅し分設定モードであることを示す。分は1から99まで入力することができる。数値を入力後、時間設定ボタンを押す。

2. 時間（秒）を設定する場合

...

図 5.4-2 キッチンタイマー時間設定シナリオ

1. 事前条件

キッチンタイマーの電源が入っていること。
時間設定モードになっていること。

2. メインフロー

このユースケースは、時間設定ボタンを押されたら開始する。
時間設定者が時間設定ボタンを押すと、分を示す表示部分が点滅する。
数字ボタンを押し、設定したい時間を入力する。

...

図 5.4-3 キッチンタイマー時間設定イベントフロー

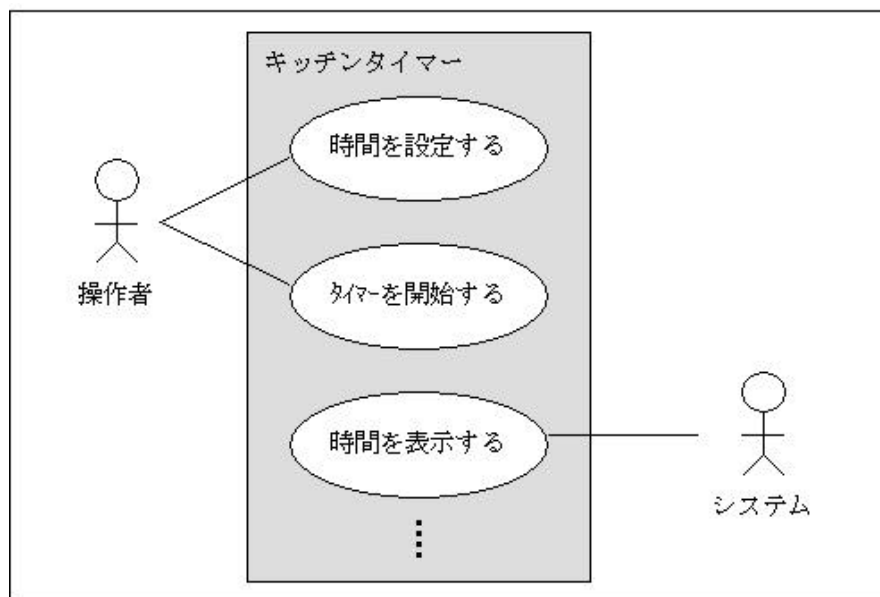


図 5.4-4 キッチンタイマー全体のユースケース図

[基本要素：ユースケース]

システムが実行する一連の動作を記述したもので、システムが提供するサービスを表す。ユースケース図は、モデル中の振る舞いのものを構造化するために使われる。

UMLでは、ユースケースに実線の楕円が割り当てられ、通常そこにはユースケース名しか記載せず、その詳細は、シナリオやイベントフローに記述する。

[基本要素：アクター]

ユースケースを起動するもの。UMLでは人の形で表す。アクターは、必ずしも人でなくてもよい。別のシステムが開発対象システムに関係するときには、その外部システムがアクターとなり得る場合もある。



【2】 アクティビティー図 (Activity diagram)

処理の流れや業務フローを示した図で、システム内の一連の作業や動作をワークフローとして順序だてて記述したものである。作業状態の遷移を記述した一種の状態遷移図である。フローチャートの持つ意味と似ている。図 5.4-5 に示すように、アクティビティー図は、1つのユースケース内で実行する一連の作業を明確にし、ビジネスフローとして関連する一連の作業を明確にするために利用する。

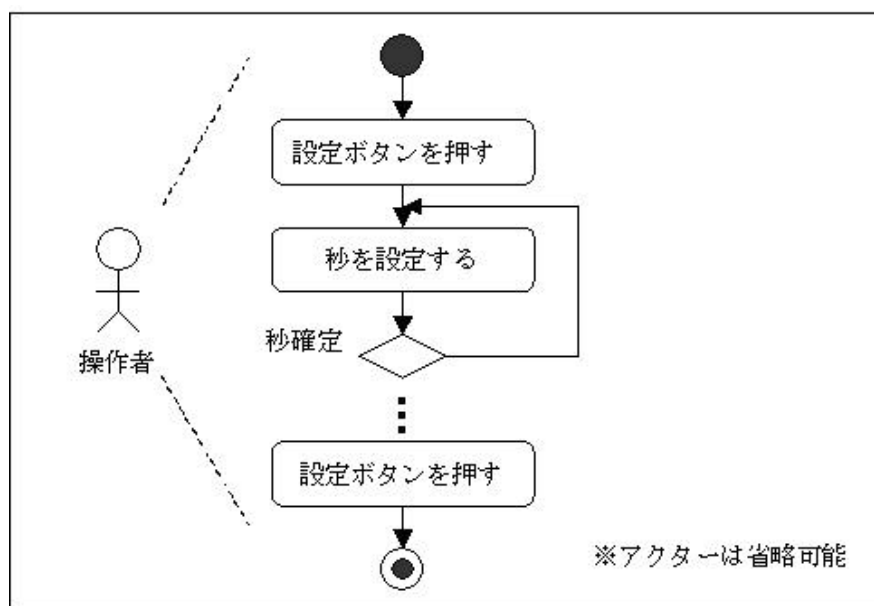


図 5.4-5 キッチンタイマー時間設定のアクティビティー図

[基本要素：開始点 / 終了点]

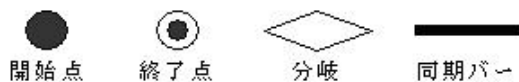
システムの開始や終了を表す。

[基本要素：分岐]

ある条件によって処理の流れが分岐する場合に用いる。フローチャート図と異なり条件は分岐要素の外側に記す。

[基本要素：同期バー]

処理の流れが並行に分岐したり、並行に流れていた処理がまとまるときに記す。



【3】 相互作用図 (Interaction diagrams)

- シーケンス図:Sequence diagram、コラボレーション図:Collaboration diagram

相互作用図には、シーケンス図とコラボレーション図がある。シーケンス図(図 5.4-6)は、時間を強調した図である。縦軸が時間の流れとなっているため、オブジェクト間のメッセージやデータの流れを時間順(処理順)に追やすい。コラボレーション図は、オブジェクト間のメッセージの流れや相互関係を示すダイヤグラムである。こちらは、システムが持つオブジェクトの構造を強調した図であり、オブジェクト間のメッセージやデータと構造の関係を把握しやすい。

シーケンス図とコラボレーション図は、同じ相互作用図であるため、どちらの図形からも相互変換することが可能である。⁹

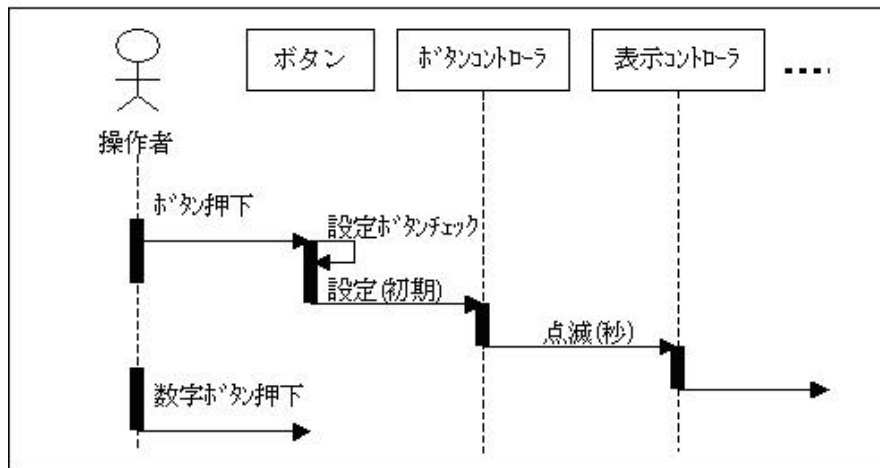


図 5.4-6 キッチンタイマー時間設定のシーケンス図

⁹ 各々の図を自動変換することができるツールもある。

【4】 ステートチャート図 (Statechart diagram)

ステートチャート図(図 5.4-7)は、オブジェクトの状態、遷移、イベントを記述するダイアグラムである。オブジェクトの状態はイベントによって遷移する。ステートチャートは(アクティビティ図同様に)角の丸い四角形で表し、状態名だけでなく必要に応じて属性値やイベントとアクション(状態に入るときや出るときのアクション)も記述できる。アクティビティ図の項目で説明した基本要素の同期バーを用いれば、並列状態のステートチャートも記述することができる。

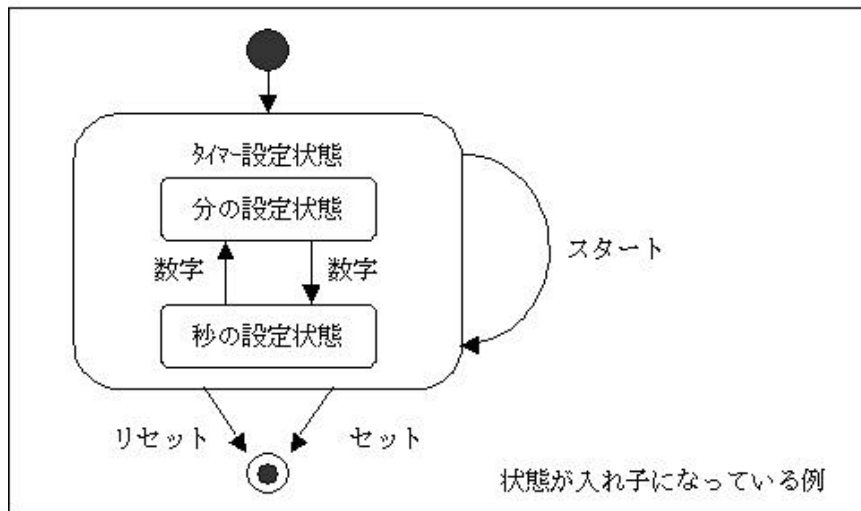
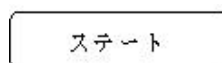


図 5.4-7 キッチンタイマー時間設定のステートチャート図

[基本要素：ステートマシン]

オブジェクトまたは相互作用が、生存期間を通じイベントへの応答として通過する状態を表したもの。角の丸い四角形で表され、通常は名前と状態が記述される。アクティビティ図でも用いられる。



【5】 クラス図 (Class diagram)

システムの静的な構造を記述するダイアグラムであり、オブジェクト指向のシステムをモデリングするときの中心的なものである。ステートチャートやシーケンス図などは、このクラス図で定義したクラスの動的な側面を補う位置づけとなっている。オブジェクト指向言語 (C++や Java) を実装言語として選択すれば、クラス図で書いたものをコードに落としこみやすいため、設計と実装間のギャップが小さい。

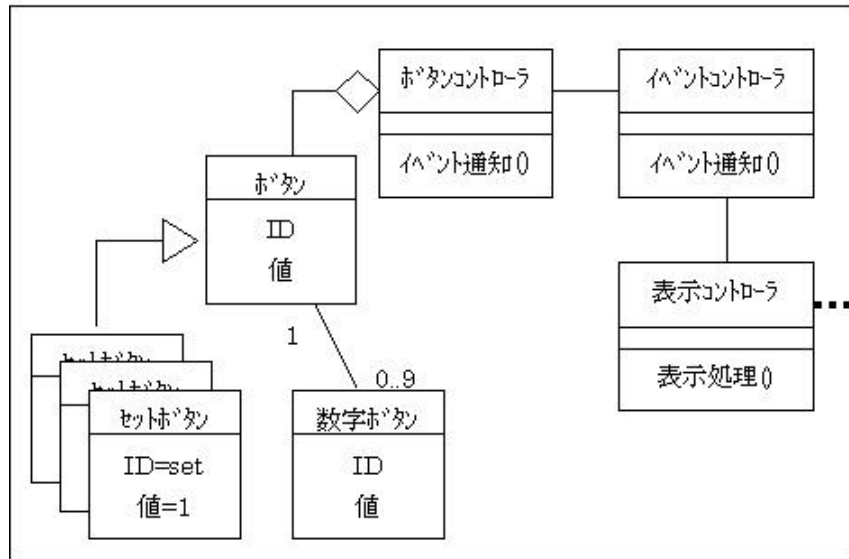


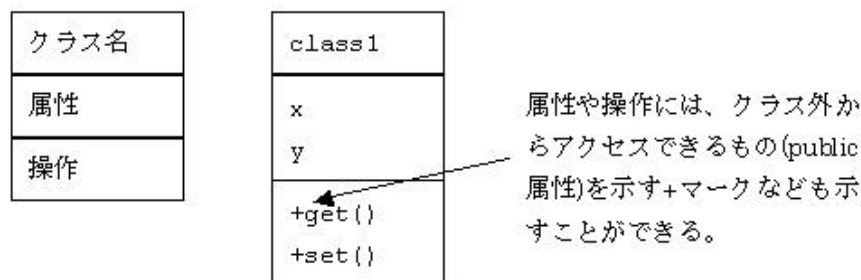
図 5.4-8 キッチンタイマーのクラス図例

[基本要素：クラス]

類似した属性、操作、関係、構造を持つオブジェクトを抽象化したもの。通常3つの区画を持つ四角形が割り当てられ、上部区画には名前、中区画には属性、下部区画には操作を記述する。なお、モデルの開発状況によっては属性と操作が省略されている場合もある。

[基本要素：インタフェース]

クラスやコンポーネントの外から見える振る舞いを記述する。インタフェースはクラスの操作仕様の集合を定義するものであるが、操作の実装を定義するものではない。また、定義されたインタフェースがクラス（またはコンポーネント）の一部しか表現していない場合もある。インタフェースは名前がついた円で表し、通常、クラス（またはコンポーネント）に付随する。



5.4.4. UML サポートツール

UML をサポートするツールは、UML の各ダイアグラムを入力 / 作成 / 編集することができ、多くのツールは作成した UML ダイアグラムから C/C++/Java 等のターゲット言語の雛型（スケルトン）を生成することができる。また、ツールによっては、シーケンス図や状態チャート図により記述された動作からスケルトンだけでなくターゲット CPU に合わせたコードを生成する機能も持っている。さらに、UML を用いてビジュアルな検証を行えるツールもある。以下に代表的なツールを紹介する。

【1】 Rational 社 Rational Rose/Rational Rose RealTime

Rational 社^[44]が開発 / 販売するツール。作成した UML からプログラムコードのスケルトンを出力したり、作成したプログラムから UML への逆変換機能（リバースエンジニアリング）を持つ。Rational Rose RealTime は、Rational Rose の持つ機能に加え、設計したモデルからスケルトンではなく完全な各種リアルタイム OS に対応した C または C++ のプログラムを生成する。さらにアニメーション化されたデバッグ環境も持つ。

【2】 Telelogic 社 Tau UML Suite

Telelogic 社^[55]が開発 / 販売する UML をサポートする分析 / モデリングツール。同社で開発販売される SDL Suite と組み合わせることによって、通信プロトコルを表現するための標準化言語 SDL と UML から C++ や Java などのプログラムコードを生成する。

【3】 i-Logix 社 Rhapsody

i-Logix 社^[56]が開発したツール。Rational Rose RealTime 同様に作成した UML からスケルトンではないプログラムコード（C/C++, Java）を生成する機能を持つ。またコード生成時にアニメーションデータを作成し、UML を用いて検証を行える。

【4】 Project Technology 社 BridgePoint/DesignPoint

Project Technology 社^[57]が開発する UML をサポートする分析 / モデリングツール。BridgePoint は、分析モデル¹⁰の作成、モデルの実行と検証、モデルからのコンパイル可能なソースコードを生成する。ソフトウェアアーキテクチャと呼ぶ"ソースコード生成ルール"に従って生成するため均一の品質のソースコードが生成される。DesignPoint は、BridgePoint で作成された分析モデルから、VxWorks 等を採用したターゲットマシン上で実行可能なソースコードを自動生成する為のモデルコンパイラ。

¹⁰ プログラミング言語に依存しないモデルを分析モデル、プログラミング言語に依存したモデルを設計モデルと定義している。設計モデルからソースコードを生成する。

【5】 Microsoft, VISIO 2000 Professional 社 VISIO 2000 Enterprise

Microsoft 社^[58]が開発 / 販売する様々なグラフィックスドキュメントを作成するツール。このなかで UML もサポートする。VisualC++環境と協調して、C++のスケルトン生成や VisualC++から UML へのリバースエンジニアリングを行える。

5.4.5. 標準化活動状況

UML は Open Modeling Group によって標準化作業が進められている。2001 年 1 月 1 日時点での正式に提供されている UML のバージョンは、UML1.3 である。OMG の下に UML RTF (Revision Task Force) があり、そこでは次期バージョンである UML1.4 やさらにその次のバージョンである 2.0 について議論されている。

UML1.4 は、次のような改良を目指している。

- 拡張メカニズムの改良
ベンダーやユーザの特定の設計手法や技術に合わせたモデリングのサポート
- ユーザからのフィードバックによる単純で明確な協調モデリング手法
強力な動作のモデル化技法を学び適用するため
- UML2.0 は、次のような改良を目指している。
- OMG で規定されるその他の技術を取り入れ、中核的な言語となる。
- 拡張メカニズムのさらなる改良
- コンポーネントベース設計への対応
- モデル管理とモデル交換の新しいメカニズム

各々の詳細については、UML RTF (Revision Task Force)^[59]、UML1.x WG^[60]、UML2.0 WG^[61]を参照のこと。

5.5. 提案フローへの適用評価

SLD研究会では、これまでにシステム設計者のニーズと最先端CAD技術分野のシーズの両面からシステムレベル設計に必要な設計工程や要素技術を洗い出して、あるべきシステム設計フローを提案し、その妥当性を検討してきた。その意味において、第4章で提案した設計フローは、将来的にシステムレベル設計言語を用いて実現すべき、さまざまな設計工程や要素技術を含んでいる。

そこで本節では、SLD研究会で提案したシステムレベル設計フローを1つの評価基準としてとらえ、各システムレベル設計言語を提案フローに適用してそれぞれの言語の能力を評価し比較する。表5.5-1にSystemC、SpecC、UMLの提案フローへの適用評価の結果を表の形で示した。表中で、○はその項目が言語の特徴となっているもの、×は適用可能、△は機能不足、×は適用不可を示している。

表 5.5-1 SLD 研究会提案フローへの適用評価結果

項目 \ 言語	SystemC (v1.0)	SpecC(v1.0)	UML (v1.3)
要求仕様定義 システム仕様 設計制約	× × 記述不可	× × 記述不可	ビジュアル表現可能 制約記述言語有
機能決定 機能定義 機能検証	○可読性おちる 既存インフラあり	明確な方法論	SW化が前提 ステートチャートのみ可
アーキテクチャ決定 プロファイリング 機能ブロック分割 分割整合性検証 アーキテクチャ生成 アーキテクチャマッピング 見積り	構造のみ記述可 トランザクションモデル書き難い 方法論不足	ツール未整備 HW/SW記述の差異なし ○ 構造のみ記述可 トランザクションモデル書き難い 方法論不足	×適用を想定せず × × × × ×
実装設計へのインターフェース HW/SW協調設計	RTLまでサポート	RTL以降は既存フロー	×適用を想定せず
テストベンチ作成	○	○	テスト仕様まで
IP設計			×

- 要求仕様定義： 要求仕様定義とは実行可能なプログラムではなくシステムの要求事項が書いてあるものである。従って、自然言語を形式的な書式で記述することができ、IBM が考案した制約記述言語 OCL (Object Constraint Language)⁴⁶⁾を採用し、設計制約を記述できる UML が有利である。SystemC や

SpecC は、このような定義方法を仮定していないため、UML とのリンクを考えるか言語仕様の拡張が必要になる。SystemC、SpecC においても、少なくとも設計制約記述の拡張は検討しても良いと思われる。

- 機能決定： SystemC、SpecC とともに可能であるが、それぞれは異なる優位性を持つ。SpecC は方法論が明確に規定されていることが有利であり、SystemC は C++用の既存ツールが使えるという点が有利である。UML に関しては、ツールのサポートにより状態チャートの検証が可能であるが、現状では UML を用いたハードウェア設計の方法論は未だ研究レベルであり、この設計工程が UML 単独での使用の限界である。

- アーキテクチャ決定： アーキテクチャの決定は、言語 / ツール / 方法論のすべてがそろえることが望ましい。この観点から各項目を評価すると以下のようになる。

「プロファイリング」： SystemC は既存の C/C++ 用のプロファイリングツールがそのまま使える。その反面、SpecC は、そのままのコードでは既存ツールを直接使うことができないので専用ツールの整備が必要になる。

「機能ブロック分割」： SpecC ではハードウェアとソフトウェアの記述手法の差異がないので、分割の変更に伴う記述の変更が SystemC より少ない。

「分割整合性検証」： 機能ブロック記述のシミュレーション実行が可能な点で、SystemC、SpecC に差異はない。

「アーキテクチャ生成」： SystemC、SpecC とともに、アーキテクチャ中のコンポーネント間の接続関係を記述することはできるが、アーキテクチャ候補を選択・評価するための基準や制約条件などの記法は用意されていない。

「アーキテクチャ・マッピングと見積り」： 方法論が不足しているとともに、利用目的に応じたさまざまな精度での見積り情報をトランザクション・モデルに記述する能力の点で不足がある。特に、粗い精度の見積りモデルを記述する観点で弱い。

総合的に見て SystemC と SpecC には一長一短はあるものの、設計空間生成の機能ブロック分割とその検証までは可能である。しかし、アーキテクチャ生成からマッピング / 見積り評価までは、どちらも満足できるレベルにはない。

- 実装設計へのインタフェース： SystemC がクロックを明確に記述する BCA レベルをサポートしている分、このレベルの設計への適用に向いている。SpecC は言語仕様の拡張か、下流へのリンクを行うインタフェースが必要である。
- テストベンチ生成： SystemC、SpecC とともに、テストベンチ作成については設

計者が記述可能である。

- IP 設計：SystemC、SpecC とともに、既存の HDL と比べて、オブジェクト指向の導入や抽象度の高い記述が可能である点で、モデルを IP として記述したり再利用がしやすい。しかし、アーキテクチャ IP 化や見積りのための IP 化が行いにくい。

5.6. システムレベル設計言語のまとめ

本章では、まずシステムレベル設計言語全体をアプローチ別に分類し、それぞれの動向と代表的な言語についての概説を行った。次に、注目すべき言語としてSystemC、SpecC、UMLを取り上げ、それらの内容や現状について詳細に説明し評価した。これら3つの言語の特徴をまとめると以下のようになる。

- SystemC
 - ◇ 設計方法論の明確化が必要
 - ◇ 実装面では強いがアーキテクチャ生成から見積りに関しては検討が必要
- SpecC
 - ◇ 専用開発環境の整備と充実が普及の鍵
 - ◇ システム記述の方法論が充実しているが、アーキテクチャ生成以降が弱い
- UML
 - ◇ アーキテクチャ設計以降は考慮されていないので、実装につながる設計言語へのリンクを考慮することが必要

これらの3つの言語を提案したシステム設計フローに適用すると考えた場合、UMLとSystemC、SpecCは元々目的が異なっている。すなわち、UMLはソフトウェアシステムのオブジェクト指向設計の考え方から提案された言語であり、「システムのモデリングと解析」が目的であるのに対して、SystemCとSpecCの目的は「実装を意識したシステム設計」である。したがって、3つの言語を目的別に使い分けるか、あるいは言語仕様を互いに拡張することによって、提案したシステム設計フローの全体に対してシステムレベル設計言語の適用が可能となる。

また、SystemCとSpecCは、現状ではアーキテクチャ決定に適用するために十分な設計環境が整っていない。アーキテクチャ生成からマッピング／見積り評価までは、言語仕様／ツール／方法論を整える必要がある。すなわち、SystemC、SpecCともに、アーキテクチャ・モデルやトランザクション・モデルを扱える言語仕様を確立するとともに、それらを用いてアーキテクチャ評価を行う方法論とツールの整備が必要である。

[参考文献]

- [1] OOVHDL, <http://www.vhdl.org/oovhdl>
- [2] VHDL+, <http://www.vhdl.org/sid/>
- [3] SUAVE, <http://www.cs.adelaide.edu.au/~petera/suave.html>
- [4] Objective VHDL, http://eis.informatik.uni-oldenburg.de/research/objective_vhdl.shtml
- [5] Superlog, <http://www.superlog.org/>
- [6] Proceedings on IEEE FDL'98
- [7] VHDL International (VI), <http://www.vhdl.org/>
- [8] EDA アニュアルレポート (1999) VHDL プロジェクト : VHDL システムレベル
拡張調査レポート
- [9] <http://www.o vi.org>
- [10] Accellera, <http://www.accellera.org>
- [11] A.Yamada, K.Nishida, R.Sakurai, A.Kay, T.Toshio, T.Kambe, "Hardware Synthesis
with the Bach System", ISCAS '98, 1998
- [12] T.Kambe, A.Yamada, K.Nishida, K.Okada, M.Ohnishi, A.Kay, P.Boca, V.Zammit,
and T.Nomura, "A C-based synthesis system, Bach, and its application," Proc. ASP-
DAC 2001, pp.151-155, 2001.
- [13] Raul Camposano and Wayne Wolf, "High Level VLSI Synthesis", pp.127-151,
Kluwer Academic Publishers, 1991
- [14] K.Wakabayashi, "C-based Synthesis Experiences with a Behavior Synthesizer
"Cyber"", Proc. of DATE'99, pp.390-393, 1999
- [15] 若林、池上、大山、"C 言語ベースのシステムレベル合成・検証手法と開発事例",
第 13 回軽井沢ワークショップ、2000 年
- [16] <http://www.cleveledesign.com>
- [17] <http://www.ics.uci.edu/~specc>
- [18] STOC, <http://www.SpecC.org>
- [19] <http://www.cynapps.com>
- [20] OSCI, <http://www.systemc.org>
- [21] D.Davis, "Using the Java Language and Environment for High-Level Circuit Design",
http://www.lavalogic.com/forged/java_wp.htm
- [22] http://www.lavalogic.com/product/wp_java.html

- [23] LavaLogic 社, <http://www.lavalogic.com/>
- [24] <http://www.sldl.org/>
- [25] JavaTime Project Overview CAD Lunch
<http://www-cad.eecs.berkeley.edu/~jimy/research/cadlunch.4.98/index.htm>
- [26] ASP-DAC'97 Tutorial 4
- [27] Jorgen Staunstrup and Wayne Wolfk, "Hardware/Software Co-Design: Principles and Practice", Kluwer Academic Publishers, 1997
- [28] <http://www.telelogic.com>
- [29] <http://www-sop.inria.fr/meije/esterel/>
- [30] <http://www-dir.hon.melco.co.jp/DIR/select1/searchj/basFD/62000000.htm>
- [31] <http://www.daimi.aau.dk/~petrinet/>
- [32] Frontier Design, Inc., <http://www.frontierd.com>
- [33] Synopsys, Inc., CoWare, Inc., Frontier Design, Inc. "SystemC Version 1.0 User's Guide", 2000 年 4 月
- [34] Synopsys, Inc., CoWare, Inc., Frontier Design, Inc., and others. "FUNCTIONAL SPECIFICATION FOR SYSTEMC 2.0 Version 2.0-M", 2001 年 1 月
- [35] Stan Liao, Steve Tjiang, Rajesh Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware", Proc. of Design Automation Conference 1997
- [36] Dundar Dumlugol, Abhijit Ghosh 他, 「無償配布ツールを使って SystemC を体験しよう!」, Design Wave Magazine 2000 年 6 月号特集, CQ 出版, 2000 年 6 月
- [37] OSCI, "第一回 Japan SystemC User Forum 配布資料", ASP-DAC2001, 2001 年 2 月 1 日
- [38] D.Gajski et al., "SpecC: Specification Language and Methodology", Kluwer Publishers, 2000
- [39] D.Gajski et al., 木下常雄他訳 「SpecC:仕様記述言語と方法論」 CQ 出版, 2000 年 12 月
- [40] 木下常雄、石井忠俊 他, 「組込みソフトと LSI を C 言語でコデザイン!」, Design Wave Magazine 2000 年 7 月号特集, CQ 出版, 2000 年 7 月
- [41] J.Zhu, R.Doemer, D.Gajski, "Syntax and semantics of the SpecC language", Proc. of SASIMI'97 (7th Workshop on Synthesis and System Integration of Mixed Technologies), 1997 年 12 月
- [42] <http://www.lsi.soliton.co.jp>

- [43] http://www.zipc.com/VisualSpec/prodvs_j.htm
- [44] Rational Software Corporation: <http://www.rational.com>,
<http://www.rational.co.jp/>
- [45] OMG: <http://www.omg.org>
- [46] Object Constraint Language, IBM,
<http://www-4.ibm.com/software/ad/standard/ocl.html>
- [47] UML 1.1 Summary, Rational Software Corporation, Sep. 1, 1997
- [48] UML 1.1 Notation Guide, Rational Software Corporation, Sep. 1, 1997
- [49] UML 1.1 Semantics, Rational Software Corporation, Sep. 1, 1997
- [50] UML サマリ Ver1.1, 日本ラショナル株式会社, 1997 年 9 月 1 日
- [51] 株式会社オーグス総研, <http://www.ogis-ri.co.jp>
- [52] グラディーブーチ著, オーグス総研オブジェクト技術ソリューション事業部訳,
UML ユーザガイド, ISBN4-89471-155-9 ピアソンエデュケーション, 1999 年 11
月 20 日
- [53] オーグス総研著, かんたん UML, ISBN4-88135-759-X 翔泳社, 1999 年 6 月 30 日
- [54] 吉田裕之他著, UML によるオブジェクト指向開発実践ガイド, ISBN4-7741-
0714-X 技術評論社, 1999 年 1 月 25 日
- [55] Telelogic, Tau : <http://www.telelogic.co.jp/UML.htm>
- [56] i-Logix : <http://www.ilogix.com>,
<http://www.ctc-g.co.jp/~product/CTC/ctg63/prd315.html>
- [57] Project Technology, <http://www.projtech.com>,
<http://www.toyo.co.jp/ss/bridgepoint/bp.html>
- [58] Microsoft Corp,
<http://www.microsoft.com/japan/Office/Visio/default.htm>
- [59] UML RTF (Revision Task Force), <http://www.celigent.com/omg/umlrtf/>
- [60] UML1.x WG, <http://www.celigent.com/omg/umlrtf/wgs/workgroups.htm>
- [61] UML2.0 WG, <http://www.celigent.com/omg/adptf/wgs/uml2wg.htm>