第1章

SystemCの概要と設計フロー

SystemCは，C++に基づいてハードウェアのモデリングに必要な構文を構築した設計記述言語です．SystemCの抽象度の高いモデリング能力によって，CPU，DSP，メモリ，バス，汎用デバイスなどをモデル化し，ハードウェア・システムを構築し，さらに，システムを制御するC/C++ソフトウェアとともに動作させ，検証することが可能です．このようなソフトウェアを含めたシステム全体の検証環境は，**ハードウェア・ソフトウェア協調設計**とも呼ばれ，協調設計により最適なシステム構成を検討できます．SystemCは，システム検証のための言語として注目されてきました．

しかし，システム検証をSystemCを用いて実現して効率化しても，ハードウェア化するデザインをHDLで記述するのであれば，再度，記述する手間がかかります．SystemCは，アルゴリズム記述からRTLを生成するビヘイビア合成ツールの入力言語としても使用できます．RTLより抽象度の高い記述レベルを用いて検証とビヘイビア合成を行い，ハードウェアを生成できるため，より大規模なハードウェアを短期間に得ることができます．

本章では，まず，SystemCとシステム設計フローの概要，SystemC言語の特徴と使用メリットについて紹介します．また，SystemCによるモデリング・レベルやビヘイビア合成の概要について紹介します．

1-1 SystemCの概要

SystemCは，HDLより抽象度の高いハードウェアやシステムを記述する言語として，2000年にバージョン1.0が，2002年にバージョン2.0がリリースされました．2006年にバージョン2.1がIEEE1666として標準化され，設計記述言語としての本格的な環境が整いました．SystemCは今後，バージョン3でリアルタイムOSモデルを，バージョン4でディジタルとアナログの混在したミックスト・シグナル・モデルをサポートする予定になっています．

SystemCの標準化は，**OSCI**（Open SystemC Initiative）という組織が行っています．OSCIでは，言語やモデリングの標準化だけでなく，SystemC普及のための啓蒙活動やSystemCリファレンス・シミュレータの開発と提供を行っています．

見本

SystemCを実行するためのSystemCリファレンス・シミュレータは，OSCIのホームページhttp://www.systemc.org/から無償でダウロードできます．本書で紹介するSystemC記述は，すべ

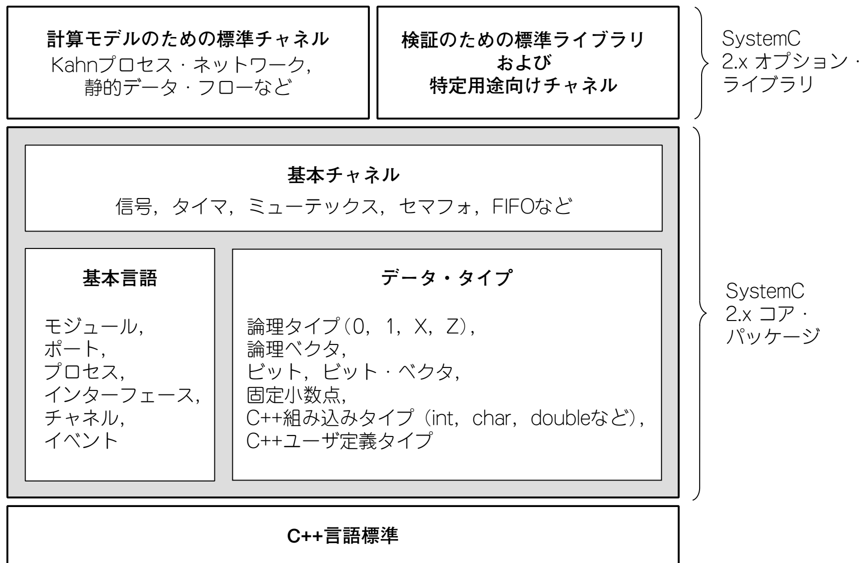

図 1-1　SystemC 言語の構造

てダウンロードしたパッケージを用い，シミュレーションを実行できます（インストール方法については，「付録 A　SystemC のインストール」を参照していただきたい）．

OSCI には，以下のようなワーキング・グループが設置され，SystemC を利用する上でのモデリングやパッケージの標準化を進めています（2006 年現在）．

- SystemC 言語ワーキング・グループ
- トランザクション・モデリング・ワーキング・グループ
- ビヘイビア合成の記述スタイル・ワーキング・グループ
- 検証ライブラリ・ワーキング・グループ

本書で紹介する記述例は，基本的にビヘイビア合成ワーキング・グループが作成したビヘイビア合成サブセット仕様に基づいています．

● SystemC の構造

SystemC の基本構文やデータ型は C++ に基づいて構築されています（**図 1-1**）．主に C++ のクラス構文を利用して，ハードウェア設計に必要なクロックや時間の概念，モジュール，ポート，プロセス，データ型，チャネルなどのハードウェア記述用のライブラリを構築しています．C++ 言語に基づいているため，C/C++ 言語で記述したアルゴリズムやソフトウェアを容易に取り込むことができます．

OSCI のホームページからダウンロード可能なデータには，ライブラリ，シミュレーション・カーネル，ヘッダ・ファイルなどが含まれています．SystemC で記述した設計データを，標準的な C++

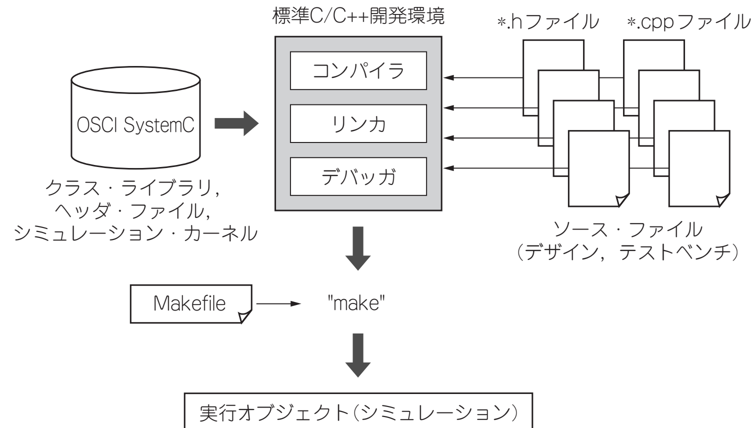

図1-2　SystemCの実行フロー

リスト1-1　Makefileの定義例

```
# ターゲット・アーキテクチャの指定(linux, sparcOS5など)
TARGET_ARCH = linux
# コンパイラとオプションの指定
CC      = g++
OPT     = -03
DEBUG     = -g
OTHER     = -Wall
EXTRA_CFLAGS = $(OPT) $(OTHER)
# EXTRA_CFLAGS = $(DEBUG) $(OTHER)

# 実行オブジェクト名の指定
MODULE = run

# コンパイルするターゲット・ファイルの指定
# 例 : SRCS = filter.cpp tb.cpp main.cpp
SRCS         =
OBJS         = $(SRCS:.cpp = .o)

#共通定義ファイルの取り込み
include  ../Makefile.defs
```

コンパイラを用いてコンパイルし，シミュレーションを実行します．SystemCを用いた設計環境は無償で構築できるため，導入しやすい環境であると言えます．

● シミュレーションの実行

g++/gccコンパイラを用いてSystemCの設計ファイルをコンパイルし，オブジェクトをリンクして実行オブジェクトを生成し，シミュレーションを実行します．**図1-2**に，設計ファイルからのコンパイルとシミュレーションの実行フローを示します．通常，SystemCの設計ファイルのファイル名には，モジュールやライブラリを含んだヘッダ・ファイルに.hの拡張子を，本体の記述に.cc，.cppなどの拡張子を付けます．

設計ファイルやコンパイル環境はファイル**Makefile**に定義し，makeコマンドを実行することで一連の作業を自動的に実行します．Makefileは，通常，コンパイルを実行する作業ディレクトリに置き，コンパイル・オプションやコンパイルする設計ファイル名を指定します．**リスト1-1**にMakefile

見本

リスト1-2　Makefile.defsの記述例

```
# SYSTEMC変数の設定(インストールしたパッケージの位置を指定)
#  例 : SYSTEMC = /usr/tools/systemc-2.1.v1
SYSTEMC =
# ライブラリの指定
INCDIR = -I. -I.. -I$(SYSTEMC)/include
LIBDIR = -L. -L.. -L$(SYSTEMC)/lib-$(TARGET_ARCH)
LIBS   =  -lsystemc -lm $(EXTRA_LIBS)
EXE    = $(MODULE).x

.SUFFIXES: .cc .cpp .o .x

# 実行方法の指定
$(EXE): $(OBJS) $(SYSTEMC)/lib$(TARGET_ARCH)/libsystemc.a
$(CC) $(CFLAGS) $(INCDIR) $(LIBDIR) -o $@ $(OBJS)$(LIBS) 2>&1 | c++filt
.cpp.o:
$(CC) $(CFLAGS) $(INCDIR) -c $<
.cc.o:
$(CC) $(CFLAGS) $(INCDIR) -c $<

clean::
rm -f $(OBJS) *~ $(EXE) core
ultraclean: clean
rm -f Makefile.deps
```

の定義例を示します．コンパイルするファイル名は，“SRSC=”の後に指定します．記述エラーがなく，すべてのファイルのコンパイルが無事に終了すると，実行オブジェクトが生成されます(このMakefileでは実行オブジェクトrun.xを生成する)．実行オブジェクトrun.xを起動することで，シミュレーションを実行します．

設計ファイルに依存しない共通的な環境は，**Makefile.defs**に指定します．Makefile.defsに，インストールしたSystemCパッケージの位置や，コンパイルに必要なライブラリなどを指定します[注1-1]．Makefile.defの定義例を**リスト1-2**に示します．

1-2　基本用語

SystemCで用いる基本的な用語について，**図1-3**を用いて説明します．多くの用語はHDLとほぼ同じです．

モジュール(**SC_MODULE**)はデザインの基本単位を示し，Verilog HDLのmoduleやVHDLのentityと同じです．モジュールは外部との接続を行う**ポート**を持ち，外部と通信します．モジュールはプロセスを持つことができ，プロセスは，クロックや信号の変化(**イベント**)によって起動されます．プロセスそのものは並行的に実行されますが，プロセスの内部のプログラムは手続き的(順序的)に処理されます．SystemCのプロセスには，通常のスレッド・プロセスとして**SC_THREAD**，クロック同期スレッド・プロセスとして**SC_CTHREAD**，メソッド・プロセスとして**SC_METHOD**があります．プロセスは，Verilog HDLのalways文，VHDLのprocess文とほぼ同じです．

注1-1：付録Aに従ってインストールした場合，インストール・ディレクトリは /usr/tools/systemc-2.1.v1 となるが，インストール・ディレクトリは任意である．

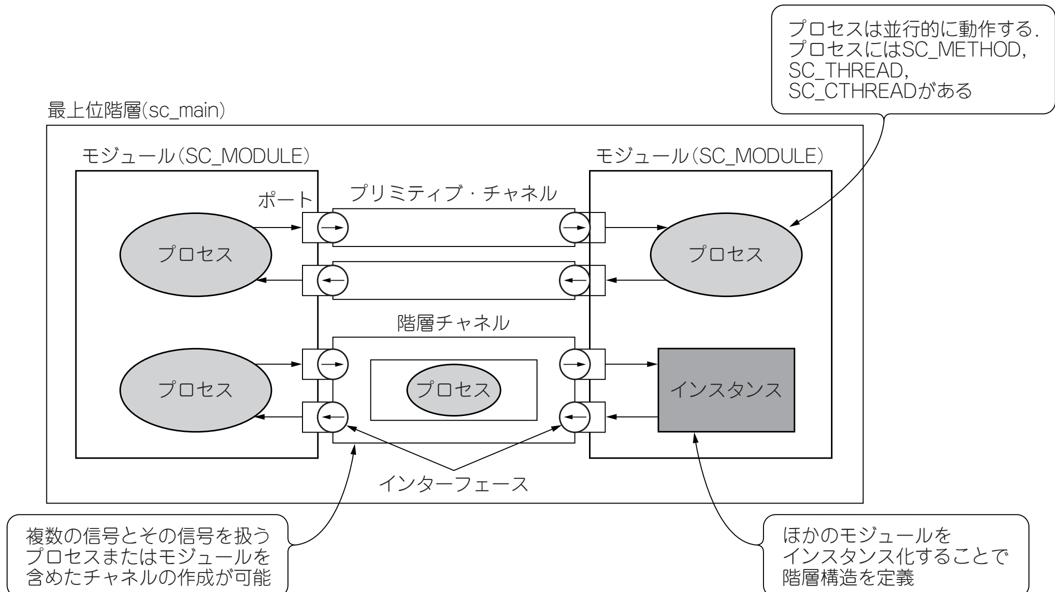

図1-3　SystemC モジュールの基本構造

モジュールは，ほかの階層から**インスタンス化**して実体化することができます．インスタンス化されたモジュールはコンポーネントとも呼ばれ，ほかのコンポーネントとチャネルを用いて接続します．

チャネルは，通信機能をモデル化したものです．信号を伝える基本的なチャネルはプリミティブ・チャネルと呼ばれ，プリミティブ・チャネルには`sc_signal`や`sc_fifo`などがあります．チャネルには，階層チャネルと呼ばれる高機能な通信プロセスの構築に利用できるチャネルも作成できます．階層チャネルは，`SC_MODULE`またはプロセスを持つチャネルで，モジュールとはほとんど区別がありません．階層チャネルを用いることで，複雑なバス・モデルを作成できます．チャネルは，通信する方法を実装したスレッド(関数)を持ちます．**インターフェース**は，アクセス可能なチャネル内のスレッドをまとめ，プロセスからポートを介してインターフェース[注1-2]内のスレッドにアクセスできます．

SystemCを用いるメリットの一つは，機能本体と，外部とのやり取りを行うインターフェースの設計を分離できることです．機能が同じでインターフェースが異なる場合，RTL設計では，インターフェースに合わせて多くの設計変更が必要になりますが，SystemCを用いた設計では，チャネル・モデルを置き換えるだけで済みます．

1-3 モデリング・レベル

SystemCを用いたモデリングでは，設計工程に応じて異なる抽象度による表現を行うことができ

見本

注1-2：SystemCの狭義のインターフェースという名称に対して，本書ではほかのデザインとの通信を行う機能を総称してインターフェースと呼んでいる．

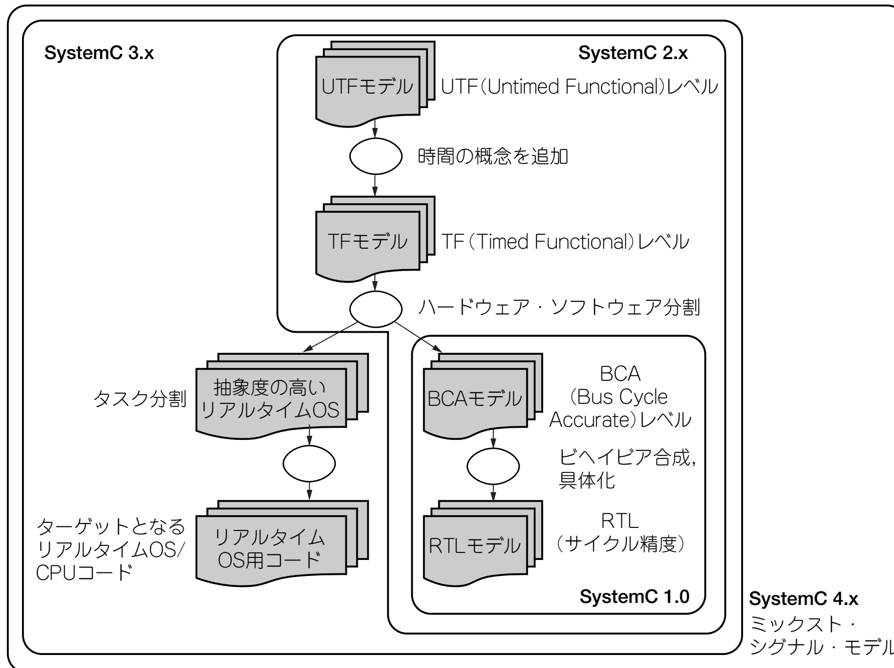

図1-4　SystemCのバージョンとモデリング・レベル

ます．設計フローの中でそれらのモデリングを適切に用いることで，設計の効率化が期待できます．ここでは，SystemCによるモデリングの抽象度を説明します．**図1-4**にSystemCのバージョンによる変遷と，そのモデル表現のレベルを示します．

● アンタイムド（UTF ： Untimed Functional）

時間的な概念を含まないレベルです．ソフトウェアでは，一般にクロックや時間の経過などの概念が含まれず，データの読み込みからアルゴリズム処理，出力までがゼロ時間で行われます．モデルでは，計算に必要なデータを関数への引き数として渡し，その計算結果がゼロ時間で戻ります．

● タイムド（TF ： Timed Functional）

時間的な概念を含んだレベルです．例えばクロックは，クロック周期という時間を持ち，クロックに同期してモデル記述を実行し，時間の経過に従って処理します．通常のハードウェアはクロックに基づいて処理するので，その記述レベルはタイムドです．SystemCのタイムドに基づいた動作の記述には，主にセンシティビティと`wait()`文を用います．クロックや時間を考慮したハードウェアのモデルは，一般にタイムドです．

● TLM（Transaction Level Modeling）

データの流れを定義したレベルです．TLMでは，クロックを含まないアンタイムドのモデルでも，

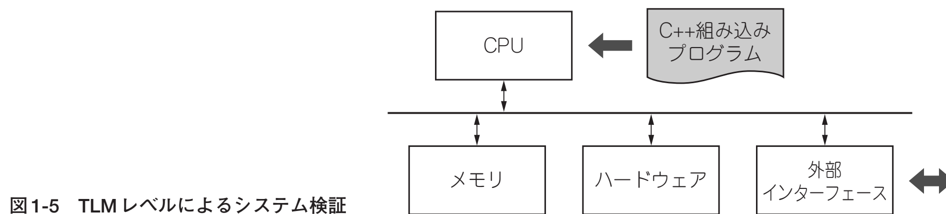

図1-5　TLMレベルによるシステム検証

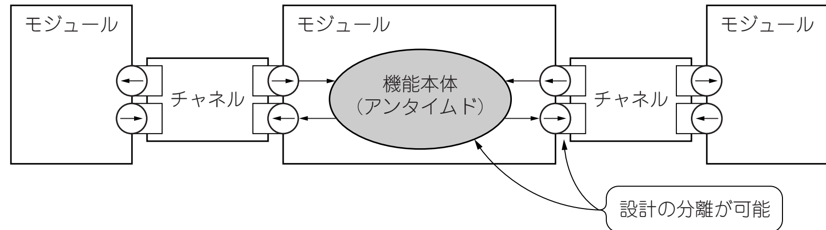

図1-6　BCAレベルの構造

クロックを含めたタイムドでも，どちらでもモデリング可能です．TLMでは，ハードウェアの詳細なバス構造やインターフェース機能はモデリングしません．多くの場合，`sc_fifo`などのチャネル・モデルを用います．`sc_fifo`は，FIFO（First-in First-out）にデータが入力されると，モジュールで計算が実行され，空の場合には`sc_fifo`にデータが入力されるまで待ちます．計算された結果は`sc_fifo`に書き込まれますが，出力側の`sc_fifo`がいっぱいの場合，空きがでるまで待ちます．このようにモジュール間に`sc_fifo`を配置し，データの流れに従って処理します．

　また，TLMでは，`sc_mutex`，`sc_master`，`sc_slave`などの詳細な制御信号を持たないバス・モデルを利用できます．さらに，階層チャネルを用いてバス・モデルを作成することも可能です．これらのモデルは，詳細な信号制御を持たないためにシミュレーションが高速です．そのため，CPU，メモリ，バスなどのモデルによって構成されたシステムLSIに対するシステム検証（**図1-5**）を実行できます．SystemCはC++に基づいているので，システムを動作させる組み込みプログラムとともにハードウェアを検証するハードウェア・ソフトウェア協調検証を可能とします．

● BCA（Bus Cycle Accurate）

　BCAの場合，入出力インターフェースについてはRTLと同じサイクル精度を持ち，記述本体はアンタイムドもしくは抽象度の高いタイムドで表現します（**図1-6**）．インターフェースはピン精度でRTLと同じ制御信号を持ち，外部と同期してデータの入出力を行います．そのため，同期の手順が正確であれば，RTLで構成されたシステムの一部のモジュールとして置き換えることができます．BCAでは，機能本体はタイムド記述を必要としないため，シミュレーションはRTLより一般に高速です．

　ビヘイビア合成可能なビヘイビア・モデルは基本的にBCAモデルです[注1-3]．BCAモデルは，入出力インターフェースにRTLと同じタイミング精度を必要とします．しかし，インターフェースの仕様を共通化し，ライブラリとして登録すれば，異なるインターフェースを持つBCAモデルを容易に

見本

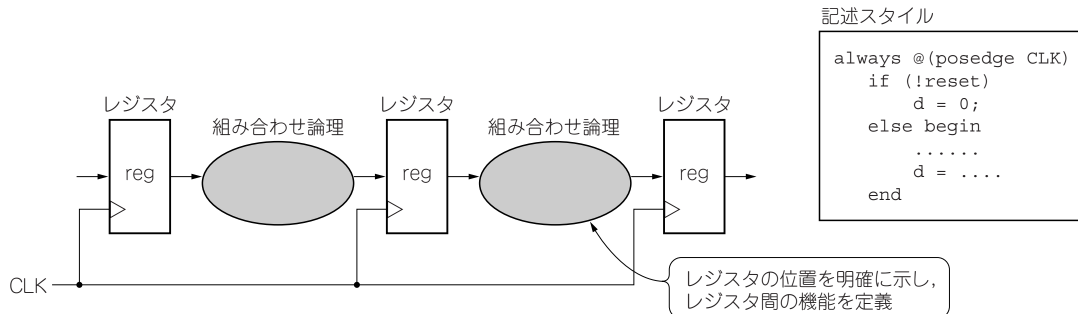

図 1-7　RTL の基本構造

作成できます．

● RTL（Register Transfer Level）

RTL は，ハードウェア内部の構造を明確に定義し，クロック動作に基づいたレジスタからレジスタへの機能を記述します（**図 1-7**）．論理合成可能な記述レベルは RTL です．RTL では，インターフェース仕様だけでなく，内部のデザイン構造全体をサイクル精度に基づいて記述します．

RTL は，最終的なハードウェアの構造を明確に定義できるメリットがあります．その一方で，設計するハードウェアが大規模になると，記述量が増えるため，記述が困難になり，シミュレーションや検証に長い時間がかかるというデメリットがあります．そのため，デザインが大規模化している現状では，より抽象度の高い設計レベルへの移行が望まれます．

1-4　システム設計フロー

SystemC を用いた設計フローの概要を示します（**図 1-8**）．ここで紹介する設計フローは一般的なフローであり，実際の設計プロジェクトの設計フローとは若干の違いがあるかもしれません．

まず，最初に実現する**アルゴリズム**（Algorithm）を ANSI-C や C/C++ 言語を用いて検証します．アルゴリズムには，フィルタやデータ圧縮/伸張などの画像処理，暗号化/復号化，通信，ハードウェア・ドライバなどが考えられます．通常，この段階では，実現するシステムの構成は考慮しません．

次に，アルゴリズムをどのようなシステム構成，すなわち**アーキテクチャ**（Architecture）で実現するかを検討します．例えば，既存の CPU やメモリなどの LSI を用いてボードを組み立て，組み込みプログラムを開発することでシステムを実現する場合があります．また，システムの性能を高めるために，新たに LSI や FPGA などのデバイスを開発する場合もあります．

新しいデバイスの開発では，従来，システムの組み込みプログラムを開発できるのはデバイスが完成した後になることがほとんどでした．デバイス開発が遅れれば，その遅れが組み込みプログラムの開発スケジュールにしわ寄せされ，短い期間でプログラムを開発しなければなりませんでした．

見本

注 1-3：本書では，BCA モデルという表現よりビヘイビア・モデルという表現を用いる．本書で用いるビヘイビア・モデルは，ビヘイビア合成可能な BCA モデルを意味する．

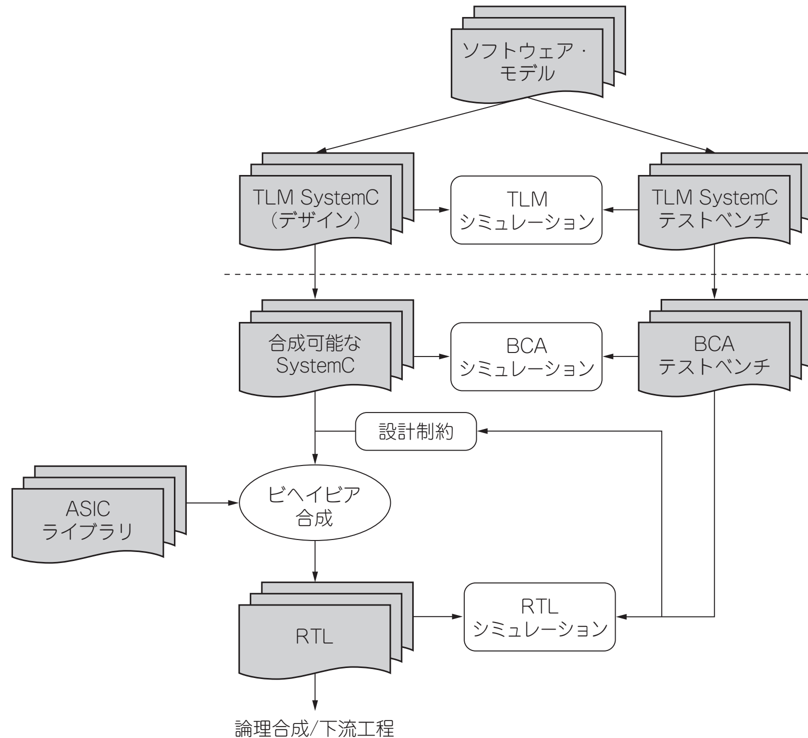

図1-8　SystemCを用いた設計フロー

SystemCを用いると，CPUやメモリ，コントローラ，オンチップ・バスなどのハードウェア・モデルを作成し，仮想的なハードウェア・システムを構築することができます．これにより，早い段階でシステム検討やプログラム開発を行えます．

このようなハードウェアとソフトウェアを協調して検証することを**協調検証**（Co-verification）と言います．従来のRTL記述でも協調検証は可能ですが，シミュレーションに時間がかかるため，大規模なシステムの検証速度は遅く，実用的な時間で実行できないことが多々ありました．しかし，SystemCのTLMレベルを用いると，実用的な時間で検証を行えます．それによって，実デバイスまたはエミュレーション環境が完成してから行っていた組み込みプログラムの開発を，先行させることができます．

アーキテクチャ検討の結果，どの機能をハードウェア化するかが決まったら，ハードウェア化するアルゴリズムに対してSystemCを用いてハードウェア階層とインターフェースを定義します．設計の初期には，詳細なインターフェースは確定していない場合があります．TLMレベルを用いることで，詳細なインターフェース仕様を定義せずに，システムを高速に検証できます．

TLMレベルで検証した後，詳細なインターフェースを定義するBCAモデル（ビヘイビア・モデル）を作成します．BCAモデルは入出力のインターフェースをピン精度で定義するレベルです．インターフェース仕様は，要求するタイミング仕様に基づいて手作業で記述する場合がありますが，TLMからの自動生成機能を利用したり，既存のチャネル・ライブラリを用いて実現する場合もあります．

見本

SystemC を使用すれば，機能本体とインターフェース部の分離設計が容易です．

詳細なインターフェースを用いた BCA モデルの検証が終了したら，ビヘイビア合成ツールに設計制約条件やスケジューリングのオプションを設定し，ビヘイビア合成を実行します．ビヘイビア合成可能な構文や記述スタイルは，OSCI のビヘイビア合成ワーキング・グループが定義しています．ビヘイビア合成の対象となるモジュールが大きい場合には，いくつかのモジュールやプロセスに分割します．

ビヘイビア合成によって生成された RTL が設計目標を満足していれば，そのまま論理合成に進みます．もし，設計目標を満足していない場合は，制約条件を変えたり，目標とするハードウェア構造を意識した記述に書き変えたりします．目標を満足したら，既存の設計フローに従って，論理合成，レイアウト設計へと進みます．

1-5 SystemC の特徴と使用メリット

ここで，SystemC を用いたメリットをまとめてみます．

● 標準言語である

SystemC は，OSCI という標準化団体で，言語の標準化やモデリング・スタイルなどの標準化を行っています．ツールやユーザごとに異なる言語を用いると，汎用的な設計資産として再利用することが難しくなります．SystemC は IEEE1666 として標準化された言語であるため，Verilog HDL や VHDL と同じように，多くのツール・ベンダから SystemC を用いた設計ツールや設計資産 **IP** (Intellectual Property) などの設計環境が提供されていくことが期待できます．

また，これまで，ソフトウェア設計者は C/C++，ハードウェア設計者は HDL と，異なる言語を用いていました．しかし，SystemC では，両者に対して共通の言語を用いた設計環境を提供することができます．

● 抽象度の高い検証が可能

SystemC の記述能力は高く，クロックの概念のないアンタイムド (Untimed) レベルや，抽象度の高い TLM レベルを用いたモデリングを可能とします．詳細なバス仕様やインターフェース仕様を定義しないでシステムを検証できるため，抽象度の高いレベルの検証も可能です．RTL 設計では，インターフェースおよび内部構造にサイクル精度の詳細な構造が要求されます．検証や実現化に TLM レベルや BCA レベルを用いることで，シミュレーションの高速化や設計期間の短縮が期待できます．

● 統一化したテスト環境を提供

SystemC はクロックの概念を持ちます．システム検証と RTL 検証に対して，異なるテストベンチを適用すると，開発に余計な手間や時間がかかります．さらに，どちらかの検証結果に問題があった場合，デザインの問題なのかテストベンチの問題なのかを判断することが困難です．SystemC によるクロック精度のテストベンチを使用することで，ビヘイビア合成と RTL 検証に対して同じテストベンチを用いた検証を行うことができます．

ANSI-Cを用いた場合，クロックやタイミングの概念は，ツール・ベンダが独自に追加したものになるので，汎用性に問題があります．場合によっては，ANSI-CとHDLのそれぞれの環境に異なるテストベンチが必要となることもあります．

● ビヘイビア合成に適する

ビヘイビア合成（または動作合成，高位合成とも呼ぶ）は，ごく最近，開発された新しい技術ではありません．論理合成ツールが実用化され始めたころから研究され，開発されてきました．しかし，実際の設計現場で本格的に使われてはいませんでした．当時のツールは，Verilog HDLやVHDLのビヘイビア記述をその入力としていました．こうしたHDLのビヘイビア記述の記述能力は高いとは言えず，さらに，アルゴリズム開発で用いたC/C++をHDLに書き直す手間がかかるため，敬遠されていました．また，ANSI-Cからのビヘイビア合成ツールも登場しましたが，ハードウェア記述のための構文が標準化されていなかったため，汎用性に問題がありました．

SystemCはC++言語のクラスを用いて定義されているため，ビヘイビア記述にそのままC++記述を用いることができます．さらに，標準化されたハードウェア記述構文を持つため，汎用性に優れています．

● 設計資産の再利用が容易

HDLでも設計資産の再利用は行えます．しかし，デザイン構造を変更することは簡単ではありません．例えば，クロック周波数を上げたい場合，演算器のタイミングが間に合わなくなるかもしれません．そのため，例えば演算器などの演算リソースをパイプライン化したり，それに合わせてステート・マシン（状態マシン）を変更したりするなど，結局，デザイン全体に対する変更が必要になってしまいます．RTLの固定化された構造では，異なるクロック周波数や異なるテクノロジへの変更は容易ではありません[注1-4]．

また，RTLでは，機能が同じでも入出力インターフェースが異なる場合，入出力タイミングを理解して注意深くインターフェース部を再設計しなければなりません．そのため，複雑なインターフェース仕様の変更を伴う設計資産の再利用では，多くの時間が必要です．

SystemCからのビヘイビア合成では，C++言語の持つオブジェクト指向の特徴をそのまま利用できます．オブジェクト指向に基づいた設計では，基本的な機能をテンプレート・クラスとして構造体に定義し，そのクラスをインスタンス化します．これにより，詳細な仕様を理解しなくても設計資産の再利用が容易になります．

1-6　ビヘイビア合成とは

ビヘイビア合成では，ビヘイビア記述に対して，その機能を実現するために必要な演算データパス部とそれを制御するステート・マシン回路を自動的に生成します．例えば，次式のような演算式を考えます．

見本

注1-4：もちろん，ゲート・レベル設計に比べれば変更は容易である．

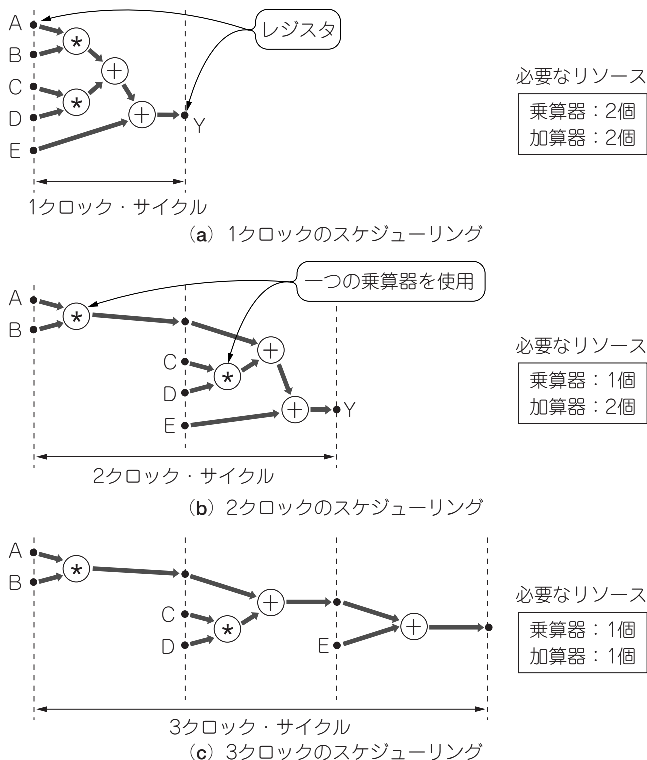

図1-9　演算式のスケジューリング例

```
Y = (A * B) + (C * D) + E;
```

図1-9に演算リソース数とスケジューリングの関係を示します．この式を実現する場合，1サイクルの実現では二つの乗算器と二つの加算器が必要です．これを2サイクルで実現する場合，一つの乗算器と二つの加算器で実現できます．さらに3サイクルでは，一つの乗算器と一つの加算器で実現できます．

ビヘイビア合成は，このようなクロック周期と演算リソースの持つタイミングに従ってスケジューリングを実行し，演算リソースとレジスタ，およびマルチプレクサから構成されるデータパス部，そのデータパス部を制御するステート・マシン部，配列をメモリにマッピングした場合にはメモリをインスタンス化したRTLデザインを生成します．RTL設計では，このようなデザイン構造は設計者の過去の経験や知識に基づいて決めていきました．しかし，ビヘイビア合成では，設計制約条件に従って，ツールが自動的にRTLデザインを作成します(**図1-10**)．

ビヘイビア合成の結果として，処理に必要な**レイテンシ**(クロック数)と面積のトレードオフが得られます(**図1-11**)．それぞれのポイントは異なる構造を持つRTLデザインで，論理合成を用いて，クリティカル・パス(Critical Path)と面積のトレードオフを検討しながら，ゲート・レベルに最適化します．品質の高いRTLデザインを得るためには，ビヘイビア合成時に適切な設計制約を与えなけ

見本

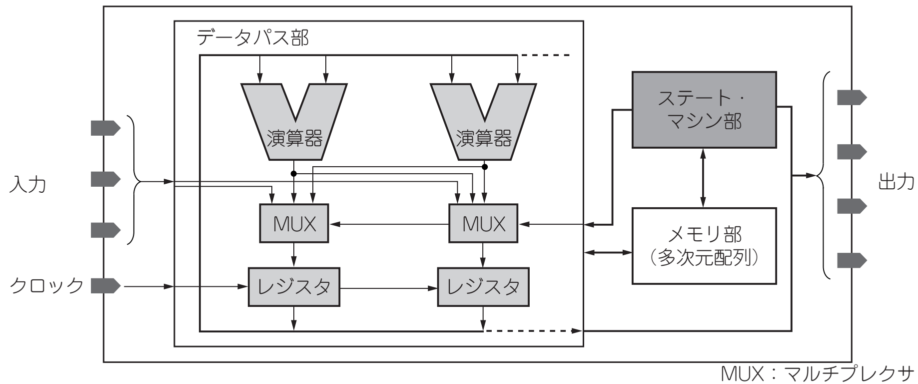

図1-10　ビヘイビア合成が生成するRTLデザインの構造

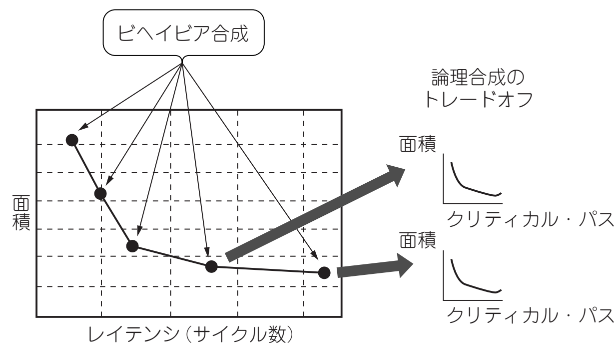

図1-11　ビヘイビア合成によるクロック数と面積のトレードオフ

ればなりません．

1-7　本章のまとめ

本章では，SystemCの概要や構造，SystemCの実行環境と実行方法，SystemCを用いた設計フローやSystemCを設計に用いた場合のメリットについて解説しました．また，SystemCによるTLM，BCA，RTLなどのモデリング・レベルについて紹介し，さらに，ビヘイビア合成の概要にも触れました．

SystemCは，C++に基づいてハードウェア概念を構築した設計言語です．SystemCを用いることでアルゴリズム設計やシステム設計に対する抽象度の高い設計や検証を行えます．さらに，SystemCはビヘイビア合成の入力となるハードウェア記述言語としても有効です．

見本

第2章

導入記述例

本章では，簡単なデータ変換関数を用いて，SystemC検証とビヘイビア合成による設計フロー，および記述スタイルを紹介します．細かい構文などの説明は次章以降で行います．ここでは，まず，SystemCによる設計手法の全体像をつかむようにしてください．

設計フローでは，まず，ハードウェア化する関数に対して，入出力に`sc_fifo`を用いたTLM（Transaction Level Modeling）モデルを作成し，テストベンチを用いて検証します．期待どおりの動作を確認したら，データの入出力をタイミング精度で記述したBCA（Bus Cycle Accurate）モデルに変更して動作を確認し，設計制約を与えてビヘイビア合成を実行します．RTLデザインが生成されたら，再度，RTLシミュレーションを実行します．性能目標を満足するRTLデザインが得られたら，従来の設計フローに進みます．

本章では，SystemC/C++を用いたオブジェクト指向設計の紹介として，クラス（`class`）文を用いた構造体を作成します．構造体クラス内にメンバ変数および変数を扱うメンバ関数を宣言することで，記述を効率化できます．ANSI-C言語に対するC++を使用するメリットの一つは，オブジェクト指向設計を行えることにあります．SystemCはC++に基づいているため，ハードウェア設計に対してオブジェクト指向を適用することができます．

2-1 アルゴリズム記述例

ソフトウェアによるアルゴリズム検証，もしくは，ハードウェアとソフトウェアの協調検証によって，ハードウェアとして実現するC/C++アルゴリズムを選択します．アルゴリズムには，画像処理や暗号化，ドライバ，通信などの多くのアルゴリズムがありますが，本章の導入例として，*RGB*データを*YUV*に変換する単純な関数を用いることにします．**リスト2-1**に変換関数を示します．*R*，*G*，*B*それぞれのデータを，輝度信号*Y*，色差信号*U*（またはC_r），*V*（またはC_b）に変換します．実際の設計では，より複雑な関数がハードウェア化の対象となりますが，基本的な記述スタイルや設計フローを説明するために，この簡単な関数を用いることにします．

浮動小数点演算はビヘイビア合成では扱えないか，扱えても必要なハードウェアの面積が大きくなるので，この変換関数では，データ型として整数を用います[注2-1]．

リスト 2-1　ハードウェア化する変換関数

```
void calc_rgb2yuv(int r, int g, int b,
  int &y, int &u int &v)
{
  // y (255,0)
  // u, v offset by 128 (127,-128)
  //
  // y = r*0.299 + g*0.587 + b*0.114;
  // u = r*-0.169 + g*-0.332 + b*0.5 + 128;
  // v = r*0.5 + g*-0.418 + b*-0.082 + 128;
  y = ( r*77  + g*151 + b*29  ) / 256;
  u = ( r*-43 - g*85  + b*128 ) / 256 + 128;
  v = ( r*128 - g*107 - b*21  ) / 256 + 128;
}
```

2-2 データ型と構造体の定義

C++ではデータ型としてintやshortなどを用いていますが，ハードウェア化する際には，ビット幅を明確にしたほうが，最終的なハードウェアが小さくなります．変換関数は，最大値が255で *Y*，*U*，*V* は正の整数に正規化しているので，8ビットの符号なしの整数を表すデータ型 sc_uint<8> を用いることにします．

RGBデータについては，*R*，*G*，*B* をそれぞれ個別のデータとして扱うより，一つのデータとして扱ったほうが便利なので，class構文を用いて構造体を定義します．C++の構造体では，データ・メンバとそれに関連した関数を同じ構造体の中にカプセル化することができます．このようなデータ構造をオブジェクトと呼びます．SystemCはC++上に構築されているので，ハードウェア設計に対してオブジェクト指向の設計スタイルを取り入れることができます．

ここでは *RGB* および *YUV* に対して，それぞれのデータをクラスに宣言し，データに関連する関数も同時に宣言します[注2-2]．**リスト 2-2** に示した記述例では，データ・メンバの読み書きを行う関数と，データ・メンバを個別に加算して結果を最大値にクリップする関数を宣言しています．ここで宣言した加算演算子をオーバロード演算子と呼び，C++コンパイラは演算子の引き数のデータ型を調べ，データ型が一致した演算子があれば，その演算子を自動的に呼び出します．ここでは加算演算だけを定義していますが，同じように乗算，減算，比較などの演算子のオーバロード記述も可能です（詳細は，「4.9 演算子のオーバロード」を参照）．SystemCでは，ポートに構造体を使用する場合，等号==，sc_trace，coutのオーバロード演算の定義が義務付けられているので，それらも宣言しています．

なお，ヘッダ・ファイルの複数回の読み込みは避けなければなりません．そのため，#ifndefを用いて環境変数がすでに定義されているかどうかを調べ，環境変数が定義されていれば，再度，読み込まないようにします．

見本

注2-1：固定小数点に対してはsc_fixedやsc_ufixedを用いることができ，通常，ビヘイビア合成可能である．

注2-2：*RGB* と *YUV* に共通なクラスを定義して，typedefによって異なる型名にすることもできるし，共通クラスを継承して，異なるクラスとして作成することもできる．