システムＬＳＩ設計技術　104－9
(2002. 1. 23)

〔招待論文〕

SpecC 言語に基づくシステムレベル設計手法

藤田 昌宏[*1]　木下 常雄[*2]　石井 忠俊[*3]　酒井 良哲[*4]
Mike Olivarez[*5]　冨山 宏之[*6]　高田 広章[*7]　本田 晋也[*7]　竹井 良彦[*8]
*1 東京大学工学系研究科　〒113-8656 東京都文京区本郷 7-3-1
*2 SpecC Technology Open Consortium 事務局
〒160-0022 東京都新宿区新宿 2-4-3
*3 (株)ｲﾝﾀｰﾃﾞｻﾞｲﾝ･ﾃｸﾉﾛｼﾞｰ　〒105-0014 東京都港区芝 3-43-16
*4 (株)東芝　〒212-8582 川崎市幸区小向東芝町 1
*5 Motorola, Inc. 7700 West Parmer Lane, Austin, Texas, USA
*6 (財)九州システム情報技術研究所 〒814-0001 福岡市早良区百道浜 2-1-22
*7 豊橋技術科学大学 情報工学系　〒441-8580 豊橋市天伯町雲雀ヵ丘 1-1
*8 松下通信工業(株)　〒224-0054 横浜市都筑区佐江戸町 600
E-mail: fujita@ee.t.u-tokyo.ac.jp

あらまし 本稿では、C 言語ベースのシステム記述言語の１つである SpecC 言語について、その目標や機能についての解説とともに、現在提供されているレファレンスコンパイラの概要、SpecC を利用したレジスタ転送レベルの記述法、さらにソフトウェア、ハードウェア両方の記述例について紹介する。SpecC 言語は、現在バージョン２．０の制定が最終段階にあり、バージョン１．０からの改良点についても、その考え方について説明する。

キーワード システム記述言語、ハードウェア／ソフトウェア協調設計、ハードウェア記述言語、シミュレーション、合成、検証、システム LSI,システムオンチップ、レジスタ転送レベル（RTL）

System Level Design Methodology based on SpecC Language

Masahiro Fujita[*1]　Tsuneo Kinoshita[*2]　Tadatoshi Ishii[*3]　Yoshisato Sakai[*4]
Mike Olivarez[*5]　Hiroyuki Tomiyama[*6]　Hiroaki Takada[*7]　Shinya Honda[*7]　Yoshihiko Takei[*8]
*1 University of Tokyo 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656 Japan
*2 SpecC Technology Open Consortium 2-4-3 Shinjyuku, Shinjyuku-ku, Tokyo 160-0022 Japan
*3 InterDesign Technologies, Inc. 3-43-16 Shiba, Minato-ku, Tokyo 105-0014 Japan
*4 Toshiba Corp. 1 Komukai-Toshiba-cho, Saiwai-ku, Kawasaki 212-8582 Japan
*5 Motorola, Inc. 7700 West Parmer Lane, Austin, Texas 78729 USA
*6 ISIT/Kyushu 2-1-22 Momochihama, Sawara-ku, Fukuoka 814-0001 Japan
*7 Toyohashi University of Technology Tempaku-cho, Toyohashi, 441-8580 JAPAN
*8 Matsushita Communication Industrial Co., Ltd. 600 Saedo, Tuduki-ku, Yokohama 224-0054 Japan
E-mail: fujita@ee.t.u-tokyo.ac.jp

Abstract This paper describes SpecC language, a system level description language based on C programming language. After presenting brief summary of SpecC, the reference compiler for SpecC by which SpecC descriptions can be simulated is explained. Then a way to describe RTL in SpecC is proposed. Finally, a couple of description examples of SpecC for both hardware and software are shown. SpecC version 2.0 is now being discussed and almost finalized. The comparison between version 2.0 and the previous version 1.0 is also given.

Key words System Level Design, Hardware/Software Co-design, Hardware Description Language (HDL), Simulation, Synthesis, Verification, System LSI, System on Chip (SoC), Register Transfer Level (RTL)

1. はじめに

ここ数年、システムLSI設計をターゲットとした設計の上位レベルの記述とその支援手法に関する研究が活発に行われてきている。システムLSIでは1つのデジタルシステムが実現されるため、LSI自体のハードウェア設計だけでなく、その上で実行されるソフトウェアについても、設計記述のターゲットとしており、ハードウェア・ソフトウェア協調設計に対する支援手法の研究が進んでいる。その中でもC言語をベースとしたシステムレベル設計記述法に関しては、いくつかの有力な提案がある。本稿では、その1つであるSpecC言語[1]とその標準化案について現状を説明するとともに、SpecC設計支援ツールの1つであるリファレンスコンパイラの解説、現状のSpecC言語を利用したハードウェア・レジスタ転送レベル（RTL）の記述法、並びに、SpecCを利用したハードウェア、ソフトウェア設計例について紹介する。

SpecCでは、基本的にハードウェア・ソフトウェアが一体化している仕様記述レベルから始まり、アーキテクチャ設計や各モジュール間の通信設計がすべてSpecC言語で記述できるようになっている。最終的にシステムをハードウェア部分とソフトウェア部分に分割し、ハードウェア、ソフトウェア、それぞれの合成系につながるようになっている。このようにSpecC言語は、デジタルシステムの仕様記述から始め、設計を詳細化する過程を一貫して記述し、SpecC言語支援ツールを利用して効率的かつ高性能な設計を行うことを目指している。

以下、2章で、現在策定を進めているSpecCバージョン2.0の主な議論点を簡単に紹介する。3章では、SpecCのリファレンスコンパイラについて説明する。つづく4章では、SpecCバージョン1.0でハードウェア・レジスタ転送レベルの設計記述を行う1つの方法について説明する。5章では、SpecC言語を利用したハードウェアやソフトウェアの記述例の紹介を行う。

2. SpecC言語バージョン2.0策定状況

SpecC言語バージョン1.0は2001年2月にリリースされ、6月には、対応するリファレンスコンパイラ（後述）も提供されている。これらは、現在、SpecC Technology Open Consortium（STOC）のウェブサイト[2]からダウンロード可能となっている。STOCでは、現在、言語仕様WG（LSWG）とケーススタディWG（LSWG）の2つのワーキンググループが活動しており、LSWGでは、バージョン1.0を踏まえ、主に下記の点からバージョン2.0の検討を行っている。

- SpecC言語のセマンティックスの詳細の定義
- いくつかのシンタックスの追加・改良による記述しやすさの向上
- その他、ツールとの整合性など

言語のセマンティックスについては、SpecC言語リファレンスマニュアルやリファレンスコンパイラが提供されているが、下記のような点で不明確な要素があるため、その明確化を行っている。

- 並列動作を記述するPARやPIPE文の厳密な意味
- 資源に対する排他的アクセスなどで利用するイベント制御分である、NOTIFY, WAIT, WAITFORの厳密な意味

これらの点はいずれも並列プログラミング言語やハードウェア記述言語など、並列処理を記述する言語すべてについての本質的な部分であり、厳密かつ明確な意味を定義して初めて、システムレベルの記述をスムーズに行えると考えている。

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

behavior a{
main(){    z=y;            /*st1*/
        x=z+20;    /*st2*/ }}

behavior b{
main(){  y=x+z+1; /*st3*/ }}
```

図1　PAR文の利用例

例えば、図1に示すようなPAR文では、プロセスaとプロセスbがPAR文により並列動作することになっているが、各プロセス内の実行文（st1, st2, st3）の間の時間順序関係としては、どういうものが許されるかが、バージョン1.0ではっきりしていない。そこで、バージョン2.0では、明確化するため、図2に示すような実行順序のいずれもが許されるということを定義する。つまり、「各実行文の開始時刻

と終了時刻間には、以下の関係が保障される」と定義する。

Tas <= T1s < T1e <= T2s < T2e <= Tae （a の中は順に実行）

Tbs<= T3s < T3e <= Tbe （b の中は順に実行）

Tas = Tbs, Tae = Tbe （a と b の並列動作からくる関係）

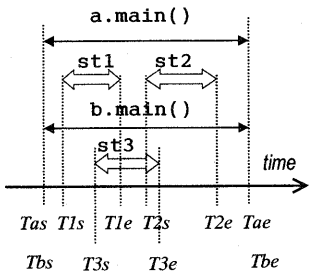

図2　図1の例題の実行の様子

これから、st1, st2, st3 の間の時間順序関係として、(st1 st2 st3), (s1 st3 st2), (st3 st1 st2)の 3 通りすべてが許される。これらに制限を加えたい場合には、チャネルやイベントの伝播を行って制御することになり、その詳細なセマンティックスについても、バージョン 2.0 の策定作業の一環として進めている[5]。

3. SpecC リファレンスコンパイラ(SCRC)

SpecC 言語バージョン 1.0 が 2001 年 2 月にリリースされたのに呼応して、SCRC の開発プロジェクトがスタートした。

SCRC は SpecC 言語バージョン 1.0 の言語リファレンスマニュアル(LRM)[3] の全ての規則を満たしている SpecC コンパイラの規範となる実装例として開発された。他の全ての SpecC コンパイラの規範となるため、誤解を引き起こしにくいコードを目指しており、実行の観点から最適化されたコードではない。

一般に言語の標準化にはフリーかつオープンなリファレンスコンパイラをリリースすることが、議論や策定、普及など色々な面で効果的であり、本 SCRC も BSD ライセンスにて配布されている。[4]

3.1 リファレンスコンパイラの開発体制と出荷

SCRC の開発は、4 つのグループ(組織)が支えている。

STOC は LRM を承認し、SCRC を推進する母体である。

STOC 言語 WG は、将来の SpecC LRM を開発すべく議論し、改良案を提案する。

カリフォルニア大学アーバイン校にある組込み計算機システムセンター(CECS)が、SCRC を開発し、保守する。また、SCRC のユーザに対するサポートも行っている。

開発費は、産業界から CECS への基金を寄せることにより賄われている。対価として、詳細な技術レポートが配布される他、優先的なサポートを受けることが出来る。是非とも趣旨を鑑みて、ご協力願いたい。

3.2 リファレンスコンパイラの構造

リファレンスコンパイラは SpecC コンパイラ、SpecC シミュレーションエンジン、テストスイートの三つから成り立っている。

SpecC コンパイラは SpecC 言語のテキスト記述を読み込む SpecC パーサ、内部表現 SIR、セマンティックチェッカー、C++のコード生成からなる。直接バイナリーを出力せず、C++のソースコードを出力する。Linux、Solaris でコンパイルが可能である。

SpecC シミュレーションエンジンは SpecC コンパイラが生成した C++ソースコードを C++コンパイラにてコンパイルする際に必要となる実行エンジンのライブラリである。

以上、二点はソースコードで公開され、出力と同様に Linux、Solaris でコンパイルが可能である。図 3 に機能構造図を示した。

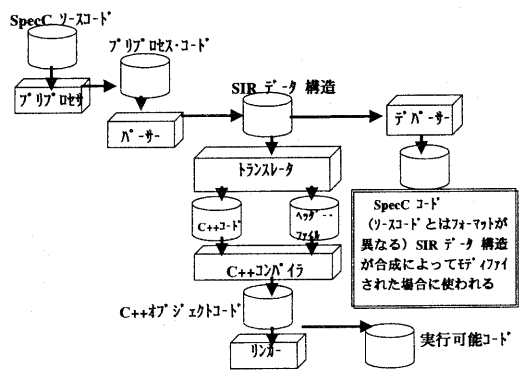

図3　SpecC リファレンスコンパイラの構造

テストスイートは SCRC を開発する際に使用した試験データである。LRM の基本機能が実現できていることの確認のため、SCRC の品質をユーザに理解してもらうため、また、ユーザが不具合を発見した際の試験のために、公開されており、今後も追加さ

れる。

3.3 リファレンスコンパイラの出荷

最初の SCRC v1.0 は 2001 年 6 月にリリースされた。それ以降、8 月にバグフィックスした v1.1 が出荷されている。

UCI の CECS にある開発チームは、SpecC 言語の生みの親であるガイスキー教授の下に組織され、SpecC 言語開発に長年携わっているメンバーで構成されている。今回の SCRC の開発では、以前からの研究用コードを流用するのではなく、全体のコードが書き直され、コードの可読性を高め、LRM の完全準拠、安定動作、等を実現している。

コンパイラは約 7 万行、シミュレータは約 4 千行のコード規模となっている。

3.4 フリーかつオープンソース

SCRC の開発ではフリーかつオープンソースなリファレンスコンパイラを供給することを最大の目標としているため、BSD ライセンスを採用した。

BSD ライセンスは Berkeley Software Distribution ライセンスの略である。昔 UNIX がリリースされていたのと同じライセンスで、GNU ライセンスとは異なる。

著作権はカリフォルニア大学に残るが、改変部は改変者の著作権となり、改変部の公開/非公開の判断は改変者に委ねられる。また、改変を含んだ派生コードは、ほぼ無条件に商用に供することが出来る。主な制限はカリフォルニア大学の著作権者表記を残しておくことだけである。

このようなライセンスで開発されたコードは、真のリファレンスとしての役割を果たすことが出来る。

3.5 今後の予定

今後も、バグの修正や、新しい LRM の登場にあわせて開発が続行される。その活動は一重に産業界の基金協力によるものであり、産学連携による技術振興の成功例となれるよう、産業界のご協力を仰ぎたい。

4. SpecC 言語によるレジスタ・トランスファ・レベル(RTL)のハードウェア記述

4.1 SpecC 言語と RTL

SpecC 言語はシステムレベルの設計記述を主な対象としており、RTL 記述は必ずしも要求されていない。しかし、全工程で使用言語が一貫していれば、シミュレータを含む開発ツールが単純になり、設計者の負担も軽減される。

ここでは、SpecC 言語の文法を変更することなく RTL 記述を行う方法として、現在検討中の記述形式を説明する。ただし、紙面の都合もあり、概要にとどめ、具体的な提案は別の機会に行うことにする。

なお、STOC の言語仕様 WG では、文法を拡張して RTL 記述を行う方法を議論している。これは Accellera のワーキンググループによる RTL 意味論[6]の標準化に沿ったものとなっている。

両者のアプローチのすり合わせをこれから始めるところであるが、その作業を通して文法の改良と関連ツールの整備を進めてゆくことになるであろう。

4.2 提案する記述形式について

提案予定の RTL 記述形式について概要を述べる。

4.2.1 目的と特徴

この記述形式は、図4 に示す環境で利用することを想定し、関連ツールの内部処理に都合が良いように考えられている。その特徴は次のとおりである。

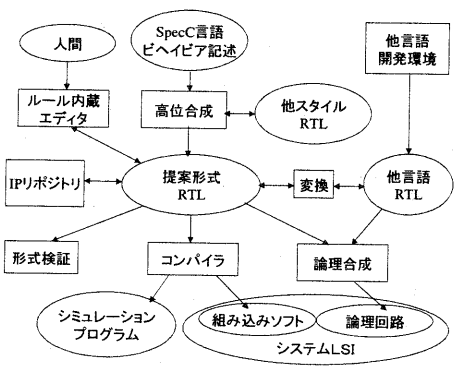

図4 想定する応用

- 言語仕様を変更しない範囲で RTL 記述が可能となるように記述ルールを定めている。
- サイクルベースの記述に限定している。

利点として以下の項目があげられる。

- シミュレーションが高速に実行できる。
- 既存のハードウェア記述言語へ変換しやすい。
- ドントケアやトライステートも記述できる。
- 数理モデルとの対応が明確であり、形式検証に応用できる。

一方、以下の項目については重点をおいていない。

- 他の記述スタイルと類似性がない。
- C 言語から論理回路への変換について配慮をし

ていない。

- イベント駆動形式の記述はできない。
- 既存のコンパイラでは不定値が扱えない。

記述自体はテキストエディタでも作成可能であるが、自動化が望ましい部分もある。

4.2.2 基礎的モデル

提案予定の記述形式で採用している数理モデルとは、単なる状態方程式である。k 番目の状態遷移後の状態を x_k、入力を u_k としたとき、出力 y_k は

$$y_k = g(x_k, u_k)$$

という式で決定される。また、さらに次の状態 x_{k+1} は、x_k、u_k、および状態遷移の原因となるトリガ信号の種類 s から

$$x_{k+1} = f_s(x_k, u_k)$$

という式で決定されるものとする。g が出力関数であり、f_s が状態遷移関数である。

提案の方法は、これらの関数をモジュールごとに記述する方法を与えるとともに、その記述結果をコンパイルすればシミュレーションプログラムとして動作するように作られている。

4.2.3 モジュール階層と分類

回路を構成するモジュールは、それぞれ SpecC 言語の `behavior` という単位で記述する。SpecC 言語にはインターフェースという概念があり、`behavior` で実装する関数を管理できる。これを利用してモジュールの種類を区別する。

たとえば、`LogicCircuit` というインターフェースは、回路モデルとテストベンチを区別するために用いる。また、ハードウェアマクロと論理合成の対象を区別するために `Synthesizable` というインターフェースを用いる。

4.2.4 組み合わせ回路

`LogicCircuit` インターフェースは `propagate(void)` という関数の記述を要求する。この関数は出力関数 g に対応するものであると同時に、シミュレーションではレジスタの値の変化を他のモジュールへ伝播させる役割を果たす。

4.2.5 クロック

クロック信号ごとに `ClockedBy(...)` という interface を実装する。さらに `updateWith(...)` という関数を実装することによって、状態遷移関数とクロック信号の対応関係を表す。

また、入れ子になっている内部モジュールが同じクロックを利用するならば、関数呼び出しの形を借りてクロック信号の分配を表現する。

4.2.6 非同期リセット

リセットについても、特別なインターフェースで入力ポートの存在を表明し、配線は関数呼び出しで表現する。

4.2.7 トライステート、ドントケア

ハイインピーダンスをあらわす値が SpecC 言語に存在しないので、バスを駆動する条件を C 言語のマクロで表現し、ツール依存の詳細を隠蔽する。また、比較演算におけるドントケアや、代入におけるドントケアの指定も、マクロで表現する。

4.3 記述例

以下の例は、非同期リセットつきの D フリップフロップを、提案の形式で記述したものである。

```
#include "specc_rtl.sh"
DeclareClock(clock);
behavior dffa(in bool din, out bool dout)
  implements Synthesizable,Resettable,LogicCircuit,
             ClockedBy(clock)
{
  bool q;
  void reset() { q = 0; }
  void propagate() { dout = q; }
  void updateWith(clock) { q = din; }
  void main() {}
};
```

`q` が 1 ビットの状態変数であり、`updateWith(clock)` が状態遷移関数、`propagate(void)` が出力関数に相当する。

4.4 考察

本形式の RTL 記述でシミュレーションを行うと、モジュール間の制御の移動が関数呼び出しで行われ、スレッドの切り替えを伴わないため、モジュールごとにスレッドを立ち上げる方法よりも、シミュレーションが高速に実行できる。

また、クロック周期ごとの計算内容を記述する形

式であるため、たとえばディジタルフィルタの特性を確認するようなシミュレーションに向いている。

しかし、現状の処理系で扱える値が 0,1 のビットパターンに限られ、異常をあらわす値が存在しないため、たとえばバスで出力が衝突するなどの記述エラーを検出することができない。そのような問題を発見するためには、処理系のデータ型を拡張するか、他言語に変換してからシミュレーションする必要がある。

4.5 今後の予定

今後は、言語仕様 WG とともに、記述形式の改良に取り組む。Gated clock などの未定事項について検討をすすめ、完成度を向上する。

また、実験のために、提案形式の RTL 記述を Verilog-HDL に変換するツールを作成し、それを用いた設計事例を作成する予定である。具体的な記述形式を公開するときに、このツールも同時に提供することになるかもしれない。

5. 事例紹介

本章では SpecC にて設計した事例を紹介する。

5.1 モトローラの事例

5.1.1 背景と目的

現在半導体産業は，プロセッサと ASIC を用いた従来式のシステム設計から，システムオンチップ（SOC: System-on-Chip）によるシステム設計へと移行する転換期に差し掛かっている．SOC 設計とは単なる半導体回路の設計ではなく、ハードウェアとソフトウェアが統合されたシステム全体を設計することである。最適な SOC 設計を可能とするため、モトローラ社ではハードウェア設計フローとソフトウェア設計フローを統合し、新しいシステム設計フローを構築している。それにより、より短期間で、より良い品質の製品を開発することが可能となる。新設計プロセスは SpecC 言語とその方法論に基づいている。本稿では、如何にして SpecC 方法論が現在の設計を補うかを示し、更に、ハードウェア設計とソフトウェア設計を統合することにより高品質な SOC 解が得られた例を紹介する。

5.1.2 ハードウェアとソフトウェアの設計統合

最適なハードウェアとソフトウェアを設計するためには、両者の設計フローの統合は必須である。従来は、設計すべきハードウェアを C 言語でモデル化し、そのモデル上でテストケースを実行していた。しかし、時代の流れと共に OS やアプリケーションなどのソフトウェアがシステムの主要な部分となっており、トップダウン式のシステム設計フローを構築するためには、ソフトウェア工学における設計工程をハードウェア工学の設計工程へと併合する必要がある。設計フローは、要求の収集、仕様の開発、テスト計画とテストケースの作成、アプリケーションのソフトウェア部分のコーディング、ハードウェア部分の RTL コーディング、そして、統合テストなどの工程を含む。多くの場合、このフローはソフトウェア工学の工程から開始し、ソフトウェアまたはハードウェアで実現されるべきもののモデルを作成する。モデルの作成は、SpecC 方法論で定義される工程を通じて行われる。この工程は、実行可能な仕様の記述、アーキテクチャの探索、通信の探索、実装からなる。[1]

筆者が離散システム設計と呼んでいる実装を除いたこの分野は、システムのプロトタイプモデルを作成するものである。システムのハードウェア部分はサイクル精度のタイミングを有し、既存の合成ツールを用いて容易に変換される。ソフトウェア部分は容易にマシンコードへと変換され、顧客へ出荷可能な状態になる。これらは全てライブラリへ登録され、将来 IP として再利用が可能となる。この設計資産のサイクルにより、設計済みシステムを元に、より複雑な新しいシステムを容易に構築することができる。

5.1.3 実証実験と設計自動化の試み

モトローラで行われたプロジェクトは、製品設計サイクル方法論の基礎として有益であるだけでなく、設計サイクル時間を短縮するツールの基礎としても有益であることを実証した。実行可能な仕様は、最初の製品から誤りが無いことを保証するための鍵である。なぜなら、抽象度が高い方が、正しさを保証するためにコードを変更することがずっと容易だからである。このことは、現在の「記述－合成」、あるいは、過去の「キャプチャーシミュレーション」という方法論に代わって、「仕様記述－探索－リファイン」という方法論を可能にする。以上述べた能力により、次に述べる利点が得られる。(a) より高い抽象度で製品設計が行われ、それにより、問題の理解と分割が容易になる、(b) 設計サイクル期間が短縮される、(c) 高い抽象度において検証することに

より、設計が固まった部分をより迅速に、低いコストで実装することができる、(d) 既設計の IP を再利用することにより、検証時間を短縮できる。これらの利点は、カリフォルニア大学アーバイン校と共同で行った試験プロジェクト「GSM ボコーダの設計」を通じて詳細に調査、検討された。このプロジェクトに関しては図書[1]の中で詳しく記述されている。

また，方法論を拡張するための更なる研究が行われ，非可逆圧縮符号化を例題に用いて，アーキテクチャのトレードオフ決定を更に自動化する試みがなされた．この JPEG 符号化の事例は、如何にして迅速かつ容易にシステムの並列性を抽出し、設計サイクルの初期段階でターゲットアーキテクチャを決定し、高い抽象度で設計を改良するかを示している。これは、計算と通信を分離することによって可能となる。アルゴリズムをリファインするために必要な工程を定義することにより、設計プロセスをドキュメント化し、再実行することが可能となる。また、この作業の自動化ツールの基礎ともなる。このように設計探索を行うことにより、全ての設計要求を満たすまで動作記述はリファインされる。我々が行った JPEG アルゴリズムの事例の場合、元々203ms だった処理時間が、72ms まで改善された。選択可能な多様な設計解を有することで、最も効率の良い解を見つけることが可能となる。

5.1.4 まとめ

SpecC 方法論を使うことにより、過去のように単にシリコンを提供するのではなく、ハードウェアとソフトウェアの両方から構成されるシステムを提供することが可能となる。それにより、我々の顧客は、より最適なシステム設計が可能となり、また、より迅速に新しい製品を生み出すことが可能となる。SpecC 言語と方法論は、高い抽象度で設計することにより設計サイクルを加速し、また、迅速な設計空間の探索を実現する。

5.2 豊橋技科大の事例

5.2.1 背景と目的

組込みシステムの大規模化・複雑化に伴い，開発期間やコストの増大に加え，設計品質や信頼性の低下が大きな問題になっている．組込みシステムは，システム毎にハードウェア構成や周辺デバイスが異なるため，デバイスを直接制御するソフトウェアであるデバイスドライバは，システム毎に開発する必要がある．

デバイスドライバ開発に関する問題点として，ソフトウェアの規模が小さいにも関わらず，大きな開発工数がかかっていることが挙げられる．その主な原因として，デバイス設計者とデバイスドライバ設計者の間の意志疎通の問題と，検証やデバッグの難しさを挙げることができる．特に前者の問題は重要で，技術者間の意志疎通の悪さが，本質的でない開発工数の増加につながっている．例えば，デバイスのマニュアルの多くは，デバイスドライバ設計者が理解しやすいようには記述されておらず，仕様の誤解が設計ミスの原因となることも多い．また，デバイスドライバ設計を考慮せずにデバイスのインタフェース部が設計されているケースも多く，その結果デバイスドライバのオーバヘッドが大きくなる場合もある．

この問題を解決するために我々は，デバイスドライバとデバイスの一体設計手法について研究を行っている．具体的には，デバイスドライバとデバイスを SpecC を用いて一体に記述し，そこからデバイス，デバイスドライバ，およびその間を接続するインタフェースを生成するアプローチをとる（図4）．これにより，前述の意志疎通の問題が解決され，デバイスドライバ開発の効率化が期待される．また，コデザイン技術を適用することで，デバイスドライバ（ソフトウェア）とデバイス（ハードウェア）の切り分けを柔軟に変更できるという利点も期待できる．

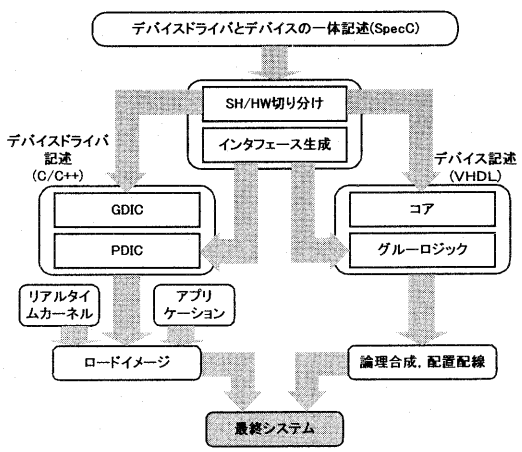

図4 デバイスドライバとデバイスの一体設計手法

ここで，組込みシステムのソフトウェアはリアルタイム OS を使って構築することを前提とし，生成するデバイスドライバは，リアルタイム OS の使用

を前提としたものとする．例えば，デバイスからのイベント待ちは，ポーリングではなく，リアルタイム OS におけるタスクの待ち状態と割込みからのタスクの起床機能を使って実現する必要がある．

なお本研究では，SpecC による一体記述からのハードウェア生成については他の研究成果を利用することとし，ソフトウェア生成およびインタフェース生成に焦点をあてて研究を進めている．

5.2.2 これまでの研究成果

本研究の第一段階として我々は，シリアル I/O システム（SIO システム）を例に，デバイスドライバとデバイスを SpecC を用いて一体に記述し，そのための記述言語として SpecC が十分な記述性を有しているかと，そこから効率的なハードウェアおよびソフトウェアの生成が可能かどうかの検討を行った[7]．

SpecC により記述した SIO システムは，シリアルライン上に TCP/IP パケットを送るためのプロトコルである PPP（Point-to-Point Protocol）の処理機能を含むものである．SIO システムを例としたのは，システム LSI の重要な適用分野である通信システムとして，簡単ではあるが典型的な構造を持っており，最初の評価対象として適当と考えたためである．

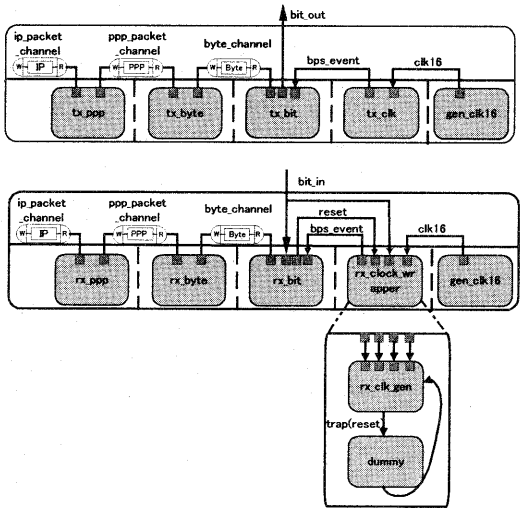

図 5　SpecC で記述した SIO システムの構成

SpecC により記述した SIO システムの構成を図 5 に示す．この図で，角の丸い四角形はビヘイビア，両端が半円の四角形はチャネルを表す．送信部・受信部ともに，扱うデータの単位毎に別々のビヘイビアとし，その間をチャネルで接続した構造となっている．SIO システムにおいては，扱うデータの単位は処理の実行頻度が同一の単位でもある．処理の実行頻度は，ハードウェアとソフトウェアの役割分担を決める際に重要な指標となるもので，実行頻度が異なる単位でビヘイビアに分割することは，両者の切り分けを柔軟に行うためには都合がよい．なお，記述した SIO システムは，シミュレータにより動作確認を行った．

次に，記述した SIO システムを，デバイスドライバの C 言語記述とデバイスの VHDL 記述に，手作業により変換した．具体的には，図 5 でバイトデータの送受信を行うチャネルである{byte¥_channel}より右側をデバイス（ハードウェア）で，左側をデバイスドライバ（ソフトウェア）で実現した．また，両者の境界となるチャネルは，デバイスとデバイスドライバの間のインタフェースへと変換した．インタフェースは，ハードウェア側はデバイスレジスタやバスインタフェース回路（バスインタフェース回路は IP を利用する），ソフトウェア側はデバイスドライバの最下位層になるデバイスアクセスのルーチンや割込みハンドラから構成される．これらの変換作業においては，自動生成の可能性を探るために，できる限り機械的な変換を行うように努めた．

以上を通じて，SpecC が，デバイスドライバとデバイスの一体設計のための記述言語として，（少なくとも SIO システムに関しては）必要な機能を備えていることがわかった．特に，ビヘイビア間の同期・通信をチャネルに集約して記述することは，システムの記述し易さや記述の読みやすさを向上させるのに有益であるばかりでなく，デバイスドライバとデバイスの間のインタフェース生成にも重要であることがわかった．ただし，効率的なインタフェースを生成するためには，SpecC 記述からの変換時にかなりの最適化処理が必要である．現時点の技術では，しばしば使うチャネルを，変換後の記述とともにライブラリ（テンプレートライブラリが望ましい）にしておく方法が現実的である．

5.2.3 進行中の研究と今後

上述の結果を踏まえて，我々は現在，デバイスドライバとデバイスの切り分けを変更できるかについて検討を行っている．具体的には，両者の切り分け箇所を，図 5 の byte_channel から ppp_packet_channel に変更し，前述と同様に，それより右側を VHDL 記述されたデバイスに，左側を C 言語記述されたデバイスドライバに，両者の境界と

なるチャネルをインタフェースに，それぞれ手作業で変換している．

ppp_packet_channel では，先頭番地を渡すことでPPP のパケットを送受信しているため，先頭番地を受け取ったデバイスは，DMA によりメインメモリからデータを取り出す必要がある点が，byte_channel で分割した場合との大きな違いである．また，デバイス側の機能が増えるために，ハードウェアへの変換を機械的に行うのが難しくなる．詳しい内容は，別の機会に報告したい．

今後の方向性として，SIO システム以外の事例についても同様の記述と変換方法の検討を行いたいと考えている．特に，今回使用したのと性質の異なるチャネルからインタフェースへの変換方法について検討するなど，検討範囲を広げることが必要と考えている．さらにその結果を踏まえて，デバイスドライバとデバイスの自動生成システムの実現へと研究を進めていく計画である．

5.2.4 SpecC の問題点と期待

SpecC は，C 言語をベースとして，ハードウェア記述を可能にするために，並行性，同期・通信，タイミング，FSM などの概念とそのための構文を追加した言語である．しかしながら，新たに追加された概念と構文に関して，ソフトウェアでどのように実現するかの検討が十分ではない．また，その間のインタフェースの実現方法についても，ほとんど考えられていない．以上で紹介した研究結果により，最低限の記述性や変換可能性については確認できたが，現状では SpecC のセマンティクスに曖昧な点があり，我々なりに解釈して進めている．

SpecC の並行性や同期・通信のセマンティクスに関しては，STOC において検討が行われているが，ハードウェア記述ないしはハードウェア生成の観点からの議論が中心で，上述の問題に応えるものとはなっていない．とは言え，ソフトウェア生成やインタフェース生成に関する検討が進んでいないために，ハードウェアの観点からしか議論できないというのが現状ではないかと思われる．

SpecC は，ハードウェア設計者とソフトウェア設計者の共通言語として有力なものであると期待しているが，このような現状では一気に理想的な言語仕様を作り上げるのは難しいと思われる．我々は，本稿で紹介した研究活動の成果を踏まえて，必要があれば，より理想的な言語仕様（例えば，SpecC の拡張仕様）を提案していきたいと考えている．

5.3 松下通信工業の事例

5.3.1 背景

柔軟な QoS（Quality of Service）制御を特徴とするパケット交換装置の開発にあたり、実際の装置開発と並行して、装置内データパスの仕様表現や複雑なパケット制御アルゴリズムの仕様検討に SpecC を適用した。本稿では特にパケット廃棄制御を司る WRED（Weighted Random Early Detection）機能の仕様検討への適用例を詳説する。本開発における SpecC の導入目的を以下に示す。

① 複雑なシステム仕様の早期明確化と曖昧性排除
② 実行可能モデルの活用による仕様検討とアルゴリズムの最適化
③ 高位合成手法確立を睨んだ設計資産の蓄積

5.3.2 適用例

WRED 機能とは、輻輳状態にある装置内バッファに到着したパケットに対し、予めパケット毎に定められた廃棄優先度と現在のトラヒック傾向に応じて廃棄確率を決定するアルゴリズムであり、輻輳状態からの早期回復を目的として実装される。一般に提案されている WRED 技術に対し「将来展開を勘案したスケーラビリティの確保」「限られたメモリリソースの利用効率向上」を狙った独自仕様を検討するとともに、装置独自の実装制約や速度性能にあわせたアルゴリズムの最適化を行う必要があった。しかし装置性能を正確に見積もるためには、複数のバッファにパケットが同時に入力する（ハードウェア的な）状況を正確にモデル化することが重要であったが、C 言語を用いた従来手法では非常に困難であった。そこで、容易に「並列性」を記述することができる SpecC を用いて、仕様モデルの作成とシミュレーション環境の構築を行った。

WRED 機能における従来の仕様表現と SpecC による仕様表現の対比を図 6 に示す。SpecC では局所的な処理を表現する「Behavior」とデータパスを表現する「Channel」でシステム構成を表現し、それぞれの内部に詳細動作を記述する。仕様表現レベルでは、各種制御の具体的な実装イメージを記述する必要がなく、コード量を必要最小限に留めることができる。

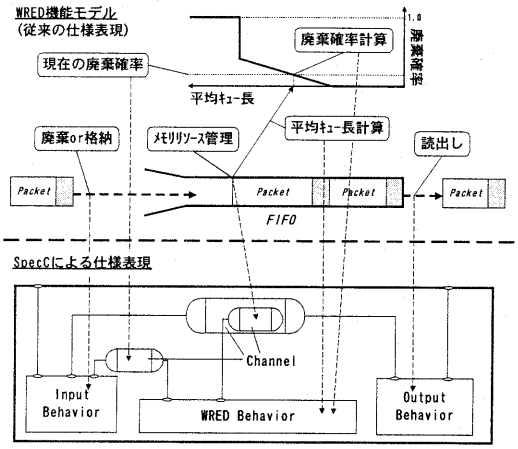

図6 usual model and SpecC model

またSpecCは実行可能言語であり、VisualBasic*1で作成したモックアップをGUIとしてシミュレーション環境の構築が可能である。WRED処理のシミュレーション用に作成したGUI環境を図7に示す。複数のWREDアルゴリズム候補に対して、パケット入力条件を変化させながら、各種中間制御パラメータ（カウンタ値等）の変化や評価指標の確認を行った。

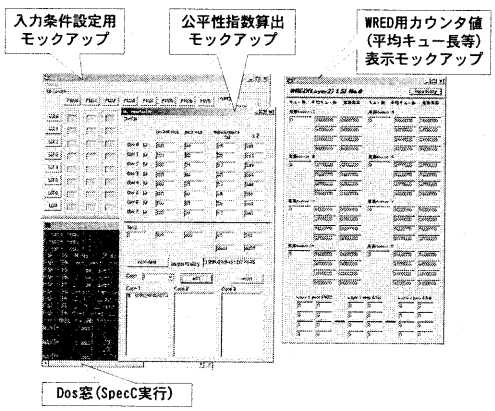

図7 GUI environment

5.3.3 成果

シミュレーション結果を図8に示す。パケットを順次入力し、その時々の各種中間制御パラメータの変化をモックアップで確認することで、アルゴリズムの妥当性を確認することができた。更に評価用に定義した公平性指数（1に近いほど望ましい）を算出することで、設定3が最も公平性に優れることが確認でき、装置仕様として採用することができた。

本取組みを通じて作成したSpecCコードは、それ自体が上記①の目的を達するドキュメント能力を有する。ここから更に各機能記述の抽象度を落とし

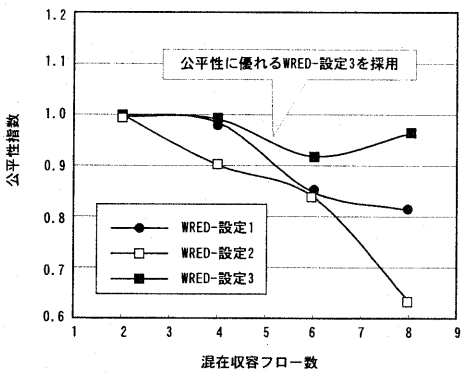

図8 fairness index and flow

て実装イメージを持たせるとともに、VisualSpec*2のパーティショニング機能を用いてハードウェア／ソフトウェアの機能分割を行い、実際にハードウェア設計仕様書としても活用することが出来た。

5.3.4 今後の取り組み

作成した仕様モデルからのHDL高位合成を試行し、関係他部門や業界標準団体との連帯のもとで高位合成環境の早期確立を目指すとともに、設計資産再利用の自動化により開発工数の大幅削減を図る。

*1 Microsoft 社商標

*2 InterDesign･Technology 社商標

文　献

[1] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao, “SpecC: Specification Language and Methodology,” Kluwer Academic Publishers, 2000. 木下、富山訳：SpecC仕様記述言語と方法論、CQ出版、2000年.

[2] http://www.specc.gr.jp/eng/wg_lang

[3] R. Doemer, A. Gerstlauer, and D. Gajski, “SpecC Language Reference Manual Version 1.0,” http://www.specc.gr.jp/eng/tech/SpecC_LRM.pdf

[4] http://www.ics.uci.edu/~specc/reference/

[5] M. Fujita, H. Nakamura “The Standard SpecC Language,” Proc. of ACM International Symposium on System Synthesis, pp.81-86, Montreal, Canada, Oct. 2001.

[6] The Accellera C/C++ Working Group of the Architectural Language Committee, “RTL Semantics Draft Specification Version 0.8,” http://www.specc.gr.jp/eng/wg_lang

[7] 本田晋也, 高田広章: 「シリアルデバイスの記述によるSpecCの記述性評価」, DAシンポジウム論文集, pp.55-60 (2001年7月).