ソフトウェア工学 120－14
（1998．7．24）

# OSシミュレーションによる リアクティブソフトウェアのテスト工程の改善

小山 徳章　原 圭吾　植木 克彦
株式会社東芝 研究開発センター

**抄録**

マイクロコンピュータによって制御されるリアクティブシステムと呼ばれる機器は、制御対象となるハードウェアに特化した専用のソフトウェア（リアクティブプログラム）を必要とする。リアクティブプログラムの開発では、ソフトウェアの動作結果を確認するためにテスト用ハードウェアの完成を前提としている。近年、情報端末や多機能家電などのリアクティブシステムの多くは、短期サイクルで市場に投入されるため、その開発ライフサイクルはできる限り短く計画される。それゆえ、ハードウェアの完成を待たずして、ハードウェアとソフトウェアの並行開発を行うことがますます必要となってきている。本論文で我々は、ハードウェア/ソフトウェアの並行開発を可能とするためのソフトウェアのテスト/デバッグ工程の改善施策とテストシステムについて報告する。

# Testing Phase Improvement by Using OS Simulation Technique in Reactive Program Development

Noriaki KOYAMA　Keigo HARA　Katsuhiko UEKI
R&D Center, TOSHIBA Corporation

**Abstract**

*Most industrial machines which we call the Reactive System are currently controlled by a specific software program named the Reactive Program. The reactive system development generally has a very short life cycle. Such software, therefore, is programmed in parallel with hardware product assembly. Furthermore, indispensable hardware tools for the program testing is usually completed only in the final phase of the system development. This paper describes a new testing phase and an original verifying system specifically designed for the reactive program. We aim at verifying some of program modules in the earlier development stage and improving their quality.*

## 1　はじめに

マイクロコンピュータによって制御されるリアクティブシステムは、制御対象となるハードウェアに特化した専用のソフトウェア（リアクティブプログラム）を必要とする。リアクティブプログラムはPCやワークステーションのアプリケーションとは異なり、対象システムのハードウェアの制御が主目的である。このため、リアクティブプログラムの開発/テスト/デバッグには、対象ハードウェアが必須とされている。近年、情報端末や多機能家電などのリアクティブシステムの多くは、短期サイクルで市場に投入されるため、その開発ライフサイクルはできる限り短く計画される。したがって、ハードウェアの完成を待たずしてソフトウェア開発を進める製品ハードウェアとソフトウェアの並行開発と、ソフトウェア開発工程自体の工程短縮が要求される。

本論文で我々は、ハードウェア/ソフトウェアの並行開発を可能とするためのソフトウェアのテスト/デバッグ工程の改善施策と、開発の早期から並行開発を可能

とするテストシステムについて報告する。

## 2 リアクティブプログラム開発

### 2.1 リアクティブプログラムの特徴

リアクティブシステムは一般に、ハードウェアの挙動を通じて外部事象を変化させたり、外部事象に応じてハードウェアの挙動が変化するシステムである。マイクロコンピュータ (MPU) が発達した現在では、ハードウェアの挙動はほとんど全てが専用のソフトウァアによって制御される。

リアクティブシステムの開発では、一般に次のような特徴が挙げられる。

- ハードウェア、ソフトウェア共に、外部事象にたいする実時間応答処理 (リアクティブ性) を満足する機能仕様を作り込むことが求められる。
- 開発工程の初期～中期では、制御対象となる実機ハードウェアの開発と、それを制御する SW 開発が別々に進められ、開発の後期工程ではハード/ソフトを組み合せ、テスト/デバッグが実施される。
- 実機ハードウェアの構成部品など、物理的要因による製品仕様変更が発生しやすい。
- 製品仕様が複雑であり、テスト項目が多い。特に実機ハードウェアの物理現象に依存した総合的なテスト項目が多い。

このようなリアクティブシステムの特徴に伴い、制御ソフトウェアであるリアクティブプログラムの開発においては、更に次のような特徴がある。

- リアルタイム性能、使用メモリ容量の制限など、実機ハードウェアの制約に、プログラム性能が左右される。
- 制御対象である実機ハードウェアの開発が終わらなければ、プログラムのテスト/デバッグを開始することができない。
- プログラムの動作環境と、開発/デバッグ/テスト環境が異なる。PC 等のアプリ開発では、動作環境と開発環境は同一であるため、コーディング→プロトタイピング→動作確認→デバッグ、というサイクルを開発環境を意識することなく進めることができる。一方リアクティブプログラム開発では、ソースコードをコンパイルしオブジェクトを作成した後は、実機環境を利用しなければオブジェクトを動作させ、デバッグ/テストを実施することができない。動作環境と開発環境が異なる開発環境は、クロス開発環境と呼ばれる。
- ICE (In Circuit Emulator) やテスト用ハードウェアなど、テスト/デバッグのためのハードウェア装置が欠かせない。リアクティブプログラムは、外部事象の変化によって動作が変化する、典型的なイベントドリブンシステムの 1 つである。したがって、テスト/デバッグ工程に置いても、外部事象を発生させるための製品ハードウェアまたは、テスト専用のハードウェアを必要とする。

### 2.2 リアクティブプログラム開発の問題点

前節で述べたような特徴ゆえ、リアクティブプログラムの開発は、PC 等のアプリ開発に対して次のような問題点が挙げられている。

**ハードウェアとソフトウェアの並行開発の同期:** ソフトウェア側のテストを実施する際には、少なくともテスト用のハードウェア、できれば完成された (不具合のない) 製品ハードウェアを必要とする。しかしながら、短期開発サイクルが求められる現在、ハードウェア開発とソフトウェア開発は、ほぼ同時進行 (並行開発) が要求される。

このため、ハードウェア側の工程進捗、品質によっては、ソフトウェア側のテスト/デバッグが工程通りに進められない、という問題点が生じる。

またこの結果、製品開発工程の後半に、大量の工数を投入して品質を確保する必要が生じる。一般に、一定期間に対する投入工数の推移が、開発工程全体で平坦なほど、開発のトータルコストを減じることができる。投入工数が開発後期工程に集中することで、人的リソースの移動などコスト増加要因が生まれ、トータルコストを上げる原因の 1 つとなる。

**実機ハードウェアの制約/クロス開発環境の機能制約:** テスト/デバッグに実機ハードウェアの制約 (またはそれに近い制約を持ったテスト用ハードウェア) によって、ソフトウェア側にデバッグのための特別な、言い換えれば冗長なテスト/デバッグ用機能を組み込めない場合が多い。具体的には、製品ハードウェアに搭載するROM/RAM容量のため、ソフトウェアモジュール内にデバッグ用機能モジュールを組み込めない場合がある。

また、実機ハードウェアの動作状況を監視するため、クロス開発環境はICEなど特殊なハードウェアを有するが、これらの装置を使い慣れるには多くの修練を必要とし、また工数も要する。尚且つ、これら装置は高価なため、開発コストを考慮した場合、必要台数を用意することも難しいのが現状である。

# 3 効率的な並行開発実現の施策

## 3.1 リアクティブシステムの特質

前述に挙げた2つの問題点は、いずれも次の3つのリアクティブシステムの特質が影響していると考えられる。

1. 並行開発
2. ハードウェアの必要性
3. リアクティブ性

ソフトウェア側のテスト/デバッグを実施するためには、完成されたハードウェアを必要とする。しかしながら工程の短縮化によるソフトウェア/ハードウェアの並行開発では、ソフトウェア側のテスト/デバッグ工程開始時期とハードウェアの完成時期が一致しない場合が多くなる。またリアクティブシステムでは、リアクティブ性 (実時間応答性) のテストを実施するために、実時間動作が保証されるハードウェアが必須となる。

## 3.2 既存の問題解決方法

リアクティブシステムの特質に沿った問題点解決方法として、以下のような技術が研究されている。

**ハードウェアシミュレーション:** 回路、制御機器などのハードウェアと、温度、湿度などの外部環境をコンピュータシミュレーションすることによって、実際の製品ハードウェアまたはテスト用ハードウェアがなくとも、ソフトウェア部分のテストを可能とする試みである。

既に自動車や航空機などのハードウェア製造においては、幾つかの物理現象についての解析/モデリング技術が実用化されている。しかしながら、一般的な物理現象のモデリングには大きなコストが必要であること、シミュレーション装置等設備面での導入/運用コストが嵩むことから、一般産業機器、情報家電など比較的安価なリアクティブシステムの開発では費用対効果が小さい。

**回路シミュレーション:** 外部環境ではなく、主に論理回路をシミュレーションすることで、シミュレーションモデリングコストを少なくし、ソフトウェアのテスト/デバッグに利用する試みである。VHDLなどの記述言語を用いた回路設計が進んでおり、論理回路レベルでのシミュレーションも実用化されている。しかしながら、一般にリアクティブプログラムは、初期化やイベント処理のために数万回の単純無限ループを使用していたり、外部環境の応答時間を考慮したリアクティブ処理を行っている場合が多く、論理回路のみのシミュレーションではリアクティブ性のテストの正確性を欠く。

**ハード/ソフトのコデザイン:** 回路 (ハードウェア) 設計と同時にソフトウェア設計を実施することで、プログラムはプログラム部品 (コンポーネントウェア) 合成によって生成する試みである。ハードウェアの設計変更 (後戻り工程) に応じてソフトウェア設計も変更することで、プログラムの再開発よりは少ない変更工数で後戻りが可能である。

しかしながら現状では、ソフトウェア仕様が多種多様であり、必ずしも部品合成だけでは仕様を満たせないこと、生成されたプログラムの要求性能 (メモリ容量、動作速度、リアクティブ性) が仕様を満足しないことなどから、コデザイン手法は実験段階といえる。

## 3.3 実施可能な施策

前述のリアクティブシステムに関する3つの特質のうちハードウェアの必要性、リアクティブ性について

は、ハードウェア実装レベルの正確性を保証するシミュレーション技術が必須である。しかしながらこのシミュレーション技術は実験段階であり、即現場適用するには至っていない。そこで我々は、並行開発の実現についてのみ考慮することとし、即現場適用可能な次の施策を検討した。

すなわち、ハードウェアが完成していなくともシステム開発の早期から、リアクティブプログラムのテスト/デバッグ工程の一部を実施可能にするために、

1. リアクティブプログラムの工程を分割し、
2. シミュレーションによってハードウェアを必要としないテスト/デバッグ環境

を用意することとした。以下の節で、これら施策について述べる。

# 4 工程改善の施策

## 4.1 サブ工程の分割

リアクティブプログラムは通常、次の3つの動作手順をもつ。

1. イベント発生
2. 状態遷移
3. イベント処理

リアクティブプログラムのほとんどがイベントドリブンシステムである。プログラムは実行開始後、イベントを待ち受ける通常状態となる。ハードウェアを操作するなどのイベントが発生すると、イベントに応じて内部状態を変更し、相当するイベント処理プログラムを呼び出す。通常状態からイベント処理状態に変化することが状態遷移である。続いて、イベント処理プログラムによって必要な処理がなされ、再び通常状態に状態遷移する。プログラムの実装によっては、状態遷移とイベント処理が複数回繰り返される場合もある。

プログラムのリアクティブ性は、イベント発生から、イベント処理プログラムが終了するまでの時間応答性を意味する。

ソフトウェアのテスト/デバッグにハードウェアが必要となるのは、イベントである外部事象を発生、検出する部分、イベント処理の結果、ハードウェア等外部の状況を変化させる部分、応答性能の計測である。上記動作手順のなかで、状態遷移部分と大部分のイベント処理部分については、ハードウェアとの依存性が低い。

そこでリアクティブプログラムのテスト/デバッグ工程を、次の3つのサブ工程に分割して実施することにし、ハードウェアへの依存性を低くすることにした。

### 4.1.1 論理テスト工程

入出力に応じてモジュールの内部状態が仕様通りに変化することを確認、検証するテスト工程である。例えばモジュールがC言語で記述されている場合は、イベントの内容を表す変数値の変化などによって条件分岐を引き起こす、というインプリメントがある。この場合、各々の変数値と分岐後の場所とのペアが、仕様通りであるかどうかを確認することになる。

あるいは、全ての分岐先を通過するかどうかの検証(いわゆるC0カバレッジ)などもこの工程で実施可能である。

対象システムのリアクティブプログラムがリアルタイムOS (RTOS) を採用している場合、モジュールであるタスクの状態変化、セマフォ、メールボックスなどリソースの状態変化も確認する。

### 4.1.2 状態遷移テスト工程

モジュール間I/Fが仕様通りであることを確認、検証するテスト/デバッグ工程である。例えばモジュールがC言語の関数で実装されている場合は、関数I/Fの確認、引数送り側/受け側の状態(内部状態)と引き数値の妥当性の検証などを実施する。多数のモジュール間のデータのやり取りを繰り返しテストするリグレッションテストなども、この工程で実施可能である。

対象システムがRTOSを利用している場合は、タスク間の遷移(コンテキストスイッチ)や、システムコールデータ(メモリプール内容)を確認する。

### 4.1.3 タイミングテスト工程

内部状態の変更、状態遷移、モジュール間I/Fの動作が仕様通りの処理時間(タイミング)で実行できることを確認、検証するテスト工程である。モジュール処理

開始位置から、モジュール通しの相互作用を経て、処理が終了するまでの時間を、ステップ数、サイクル数などから計測する。また、システムがRTOSを利用している場合は、RTOS処理時間も含めた実測値の計測などから、テストを実施する。

## 4.2 スパイラルモデル

多くの開発現場では、最もポピュラーなウォーターフォールモデルに基づく開発工程が組まれている(図1)。

ウォーターフォールモデルは、

- 工程を「設計」「コーディング」「単体デバッグ」「単体テスト」「結合デバッグ」「結合テスト」「総合デバッグ」「総合テスト」に分割する。
- 各工程を完全に終了してから次の工程に進む。

という特徴を持つ。

ウォーターフォールモデルに従えば、個々のプログラムモジュールは単体テスト時点で、(モジュール単体の)リアクティブ性が検証され、結合テスト時点でモジュール間I/Fの整合性までも検証されることになる。

実際のシステムでは、外部事象の変化、イベントの発生、複数のモジュールの処理、外界への出力までの応答速度がリアクティブ性として検証される必要がある。このため、単体テスト工程にてテストするモジュール単体のリアクティブ性はほとんど意味を持たず、総合デバッグ時に再度測定/検証することになる場合もあり、単体デバッグ工程への大幅な後戻り工程が発生する場合もある。

我々は、テスト/デバッグ工程にスパイラルモデルを適応する。スパイラルモデルでは、開発の早期からモジュールの基本動作をプロトタイピングし、作り込んだ個所から動作確認、デバッグ、テストを実施していく。

スパイラルモデルは、ウォーターフォールモデルとは異なり、全てのモジュールが同時に同じ工程を完了することを必要としない。すなわち、モジュールの仕様ボリューム、不具合件数、進捗に応じて、柔軟に工程を進めることができるという特徴を持つ。一方で、工程管理の側面では、スパイラルモデルでは進捗度を把握しづらい、どの時点でテストを切り上げて次のコーディングステップに戻すか(スパイラルさせるか)の判断がしづらい、といった欠点を持つ。

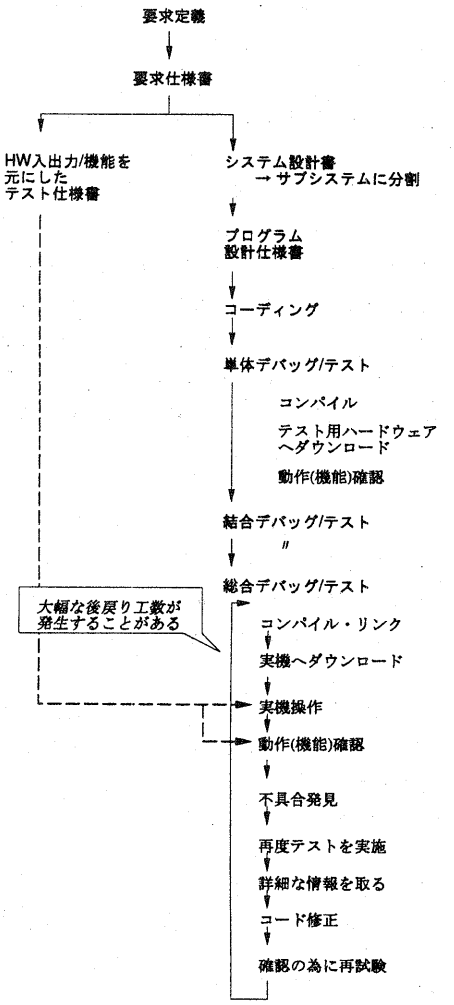


図 1: 一般的なウォーターフォールモデル

そこで、スパイラルモデルに、上述の3つのサブ工程を複合させることで、工程管理面の欠点を補うことにする。開発手順を図2に示す。

**ステップ1:** 対象となるモジュールの基本動作をスタブとして作成する。スタブ完成後、モジュールの論理動作をコーディングし、実装完了した機能から論理デバッグ、論理テストを実施する。

**ステップ2:** モジュールが関連する他のモジュールのスタブと組み合せ、状態遷移のコーディング、状態遷移デバッグ/テストと繰り返していく。

論理テストまでの工程が、ウォーターフォールモデルでの単体テスト工程に相当し、状態遷移テストまでの工程が結合テスト工程に相当する。ただし、これら工程はモジュールごとに実施すればよく、全てのモジュールで同一の工程にある必要はない。このことを実現するために、全てのモジュールについて、最初にスタブを作成することを義務づける。

**ステップ3:** 最終的に、デバイスドライバなどハードウェア制御モジュール、テスト用ハードウェア(または実機ハードウェア)と接続し、プログラム全体の、あるいはモジュール個々のタイミングテストとデバッグを行う。

このように3つの各サブ工程ごとに進捗管理を実施することで、工程管理面の欠点を補うと共に、スパイラルモデルによって各モジュールが独立してコーディング～テストすることが可能となる。

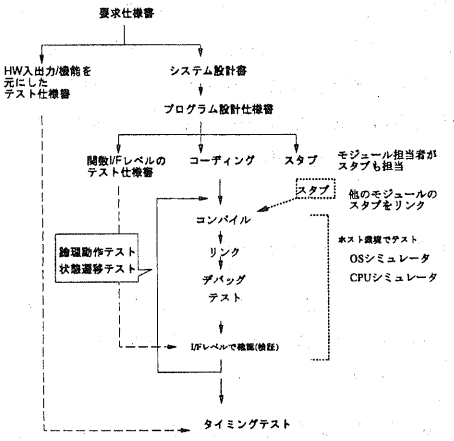


図2: スパイラルモデルと3つのサブ工程

# 5 シミュレーション環境の施策

## 5.1 OSシミュレーション

開発工程の分割とスパイラルモデルの融合によるソフトウェア開発では、個々のモジュールごとにプログラムを動作させ、結果の確認を実施できなければならない。なおかつ、この動作と結果の確認作業は、ハードウェアの開発工程の進捗如何に関わらず実現できる必要がある。

前述の通り、テスト/デバッグ工程の中でも「状態遷移テスト工程」と「論理テスト工程」については、リアクティブ性を必要としない。したがってこれら2つのテスト工程については、必ずしもテスト用ハードウェアが完成している必要はない。

一般にリアクティブシステムは、例えば、MPUやメモリマップなどがPCやワークステーションとは異なるハードウェア環境で構成されている。このため、リアクティブプログラムを、直接開発環境上で動作させるためには、次の3つの手段を採用することになる。

1. 対象となるプログラムモジュールが、C言語のような高級言語のみで記述され、ハードウェアの制御を行わない場合
   プログラムを開発環境の言語処理系で実行することが可能である。クロス開発環境のデバッガ、プロファイラなどの豊富なツールが利用できる。ただし、モニターや簡易OSなど、モジュールの制御部を必要とする場合は、これら制御部も組み合せて実行できる必要がある。
2. 対象となるプログラムモジュールがアセンブラで記述されている場合、またはハードウェアの制御を行っている場合
   リアクティブシステムで利用されているMPUや周辺回路の論理動作を、クロス開発環境上でシミュレーションし、この上で対象モジュールを動作させることで実現可能である。ただし、MPU等のシミュレータは開発コストが高く、また動作スピードも実物の1/100以下になる場合が多い。

3. リアクティブプログラムが RTOS を利用しており、対象となるプログラムモジュールが高級言語と RTOS のシステムコールを利用している場合 RTOS の論理動作をクロス開発環境上でシミュレーションし、この上で対象モジュールを動作させることで実現できる。MPU 等のシミュレータに比べて開発コストと実行スピードの低下を抑えることができ、かつクロス開発環境上の豊富なツールを直接利用することができる。

現在多くのリアクティブシステムが RTOS を採用しており、また RTOS の多くは C 言語用システムコールインタフェースを備えている。RTOS 利用下では、個々のプログラムモジュールは「タスク」という形で構成され、個々のモジュール間 I/F もシステムコールという統一された API で扱うことができる。

そこで我々は、RTOS を利用しているリアクティブプログラムを対象とし、システムで採用されている RTOS の論理動作をクロス開発環境上でシミュレーションする「OS シミュレータ」を用意することにした。

## 5.2 システム概略

用意した OS シミュレータは、当社製 RTOS の論理動作をクロス開発環境である PC 上で動作するものである。図 3 に示すように、OS シミュレータは対象プログラムモジュール (タスク群) と結合し、PC 上の C コンパイラでコンパイルされ、C デバッガから動作を補足することができる。

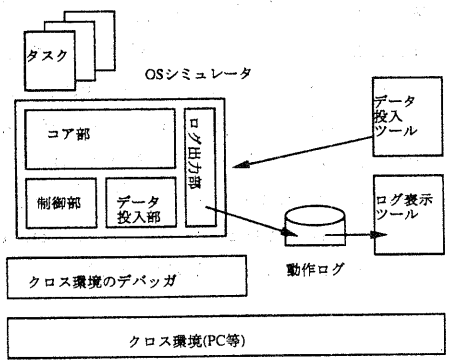


図 3: OS シミュレータ概略

OS シミュレータは、タスクの管理と制御、システムコールの処理を行うコア部と、OS 動作状態を記録するログ出力部、OS 自体の動作を外部から制御するコントロール部、対象タスク、内部メモリにデータを与えるデータ投入部を備えている。

また、OS シミュレータを用いて効率よくテスト/デバッグを実施するために、シミュレータの外部ツールとして、出力ログを解析表示するログ解析ツール、テストデータを投入するデータ投入ツールを用意した。

## 5.3 テスト方法

OS シミュレータを用いたテスト/デバッグ方法について説明する。まず対象となるモジュール (タスク) の状態遷移動作のテスト/デバッグを行う。対象タスクへのデータ投入のために、データ投入ツールを通じてデータを与える。例えば対象タスクがメールボックスの参照など、システムコールを利用している場合、対象タスクにシステムコールを発行し、メールボックスに変化を起こさせるような投入データを与える。動作確認は、PC 上のデバッガ、および OS 動作ログを用いる。

次に論理動作テストのためにスタブを用いる。スタブは、各タスク毎にスパイラルモデルによるコーディング工程の最初の段階で作成していることを義務づけている。論理動作テストでは、対象タスクと、対象タスクが利用する (または利用される) 他のタスクのスタブとを組み合せることで実施する。

対象タスク、スタブタスクへのデータ投入は、状態遷移動作時と同じである。実行タスクの推移、タスク間通信は、ログ解析ツールを利用する。尚、デバイスドライバに代表される製品ハードウェアを制御するモジュールは、テスト用スタブとして置き換えることで対応する。

## 5.4 並行開発の流れと期待効果

図 4 に本工程と OS シミュレータを用いたリアクティブプログラムの並行開発の流れを示す。

図中の 3 つのタスク (タスク A、タスク B、タスク C) を例にとる。例えば開発担当者工数上、タスク C はタスク A のテスト終了後にコーディングを開始、タスク B はタスク A、タスク C よりもボリュームが大きいとする。本報告で述べた開発工程と OS シミュレータに

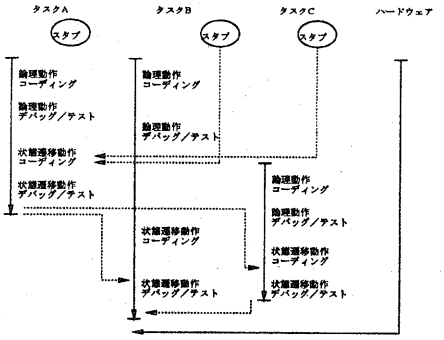


図 4: 並行開発例

よって、次のような開発の流れが可能となる。

タスクAの論理テスト/デバッグ、状態遷移テスト/デバッグは、タスクB、タスクCのコーディング作業中に実施可能である。またタスクCの状態遷移テスト/デバッグは、テストが終了したタスクAを利用することも可能となる。また、タスクBの状態遷移テスト終了時までにテスト用ハードウェアは完成すればよく、ハードウェア開発工程とソフトウェア開発工程の並行開発期間を多くとることができる。

以上の施策によって、ソフトウェア側の全テスト/デバッグ工程の40%〜60%を並行開発期間として割り当て、ハードウェア投入時期までに前倒しでテスト/デバッグを開始することが可能になると期待している。

また定性的な効果として、テスト/デバッグ工程の分割によって、不具合の種類を絞り込むこみ不具合発生個所が特定しやすくなること、ハードウェアとの結合時までに論理動作、状態遷移動作の不具合が大幅に取り除かれること、不具合による後戻り工数が削減できることなどが期待できる。

## 6 まとめ

本報告で我々は、リアクティブシステムの並行開発を可能とするための施策として、ソフトウェアのテスト/デバッグ工程の改善と、開発の早期から並行開発を可能とするテストシステムについて述べた。テスト/デバッグ工程については、リアクティブプログラムの特徴に合わせて3つのサブ工程に分割し、これらサブ工程をスパイラルモデルを適用することにした。また、論理動作、状態遷移動作のテスト/デバッグ工程でOSシミュレーション技術を利用し、クロス開発環境下でテスト/デバッグの実現を可能とするテストシステムを適用することにした。

本開発工程改善策は現在当社リアクティブシステム開発部門にて適用しており、工程短縮、品質向上といった効果を挙げている。

本工程改善策では、スタブ開発コスト、テストデータ作成コストなど、新たな工数発生要因が考えられる。今後、テストデータの作成支援、シミュレータを用いたテスト工程の自動化などの工数削減策を検討していく予定である。

### 参考文献


1. 河井、西山: 組み込みシステム用ソフトウェア開発環境、情報処理、Vol.37, No.9, pp.872-879 (Sep.1996).
2. 中本、高田、田丸: 組込みシステム技術の現状と動向、情報処理、Vol.38, No.10, pp.871-878 (Jul.1997).
3. 宿口: 組込みシステムのデバッグ手法、情報処理、Vol.38, No.10, pp.886-891 (Jul.1997).
4. 栗山、大野: シミュレーションによるリアルタイムシステム開発環境、計測と制御、Vol.31, No.7, pp.811-815 (1992).
5. 小林 他: 実行可能な仕様記述におけるプラント制御システムの環境のモデル化、情報処理学会論文誌、Vol.35, No.7, pp.1402-1409 (Jul 1994).
6. 特集「ハードウェア/ソフトウァア・コデザイン」、情報処理、Vol.36, No.7 (Jul 1995).
7. 岡島、五十嵐、小山、安田: 組込みソフト開発支援のためのシステムシミュレーション環境、51回情報処理学会論文集、Vol.5, pp.259-260 (1995).
8. Harel, D. et al.: STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE Trans. on Software Engineering, Vol.14, No.4, pp.403-414 (1990).