情報処理学会論文誌 Vol.53 No.12 2682–2701 (Dec. 2012)

μITRONベースのマルチプロセッサ向けRTOSのテスト

鴫原 一人[1,a)] 一場 利幸[1] 本田 晋也[1] 高田 広章[1]

受付日 2012年3月5日, 採録日 2012年9月10日

概要：組込みシステムは高い品質が求められるので，ソフトウェアの中核をなすRTOSに対するテストの実施は重要である．近年，組込みシステムにおいてもマルチプロセッサの利用が進んでいるが，マルチプロセッサ向けRTOSに対するテストプロセスやテスト手法，テストの規模は明らかになっていない．本研究では，μITRONベースのマルチプロセッサ向けRTOSであるTOPPERS/FMPカーネルに対するテスト設計，テストプロセス，テスト手法，およびテストスイートの開発と実施について述べる．我々は，22個のテストカテゴリを抽出し，その中から仕様とソースコードカバレッジの網羅を目的として，実施すべきテストを決定した．テスト手法として，テストケース数を抑止するためのテストケース設計ポリシ策定や，ツールによるテストプログラムの開発工数削減，各プロセッサの実行順序に依存するテストの実行効率化などを行った．テストスイートの開発と実施を通じて，マルチプロセッサ向けRTOSのテストの規模や，ツールによる効率化の効果を確認した．また，開発したテストスイートを用いたテストを実施することで，ソースコードカバレッジが100%となることを確認し，合計70件の不具合を検出した．

キーワード：組込みシステム，リアルタイムOS，μITRON，TOPPERS，オープンソース，テスト

The Tests for Multiprocessor RTOS Based on μITRON Specification

Kazuto Shigihara[1,a)] Toshiyuki Ichiba[1] Shinya Honda[1] Hiroaki Takada[1]

Received: March 5, 2012, Accepted: September 10, 2012

Abstract: RTOS comprises the core of a software therefore it should be well tested in order to supply high quality embedded systems. Multiprocessor is widely used in the contemporary embedded systems, but the test process, the test methods and the test scale for multiprocessor RTOS are not well documented. In this study, we described test design, test process, test methods and the development as well as operation of test suite for a multiprocessor RTOS based on μITRON specification, the TOPPERS/FMP kernel. We gathered 22 test categories and designed tests that cover the specification as well as source code coverage. The characteristics of the test methods were as follows: design policy for test cases to suppress the number of test cases, a tool aiming to reduce the cost of test programs development, and streamlined execution of the test depending on the execution timing. Through the development and operation of the test suite, the scale and test for multiprocessor RTOS as well as the efficiency of the tool were confirmed. We operated the test suites and verified that the source code coverage was 100% and found 70 defects in total.

Keywords: embedded systems, real-time OS, μITRON, TOPPERS, open-source, test

1. はじめに

組込みシステムにおいてもマルチプロセッサが利用されるようになりつつある．その理由として，発熱を抑えつつ性能を向上させるためには，高性能なシングルプロセッサを用いるよりも，低性能なプロセッサを複数用いた方が有利だということがあげられる．マルチプロセッサの利用が広がるにつれて，マルチプロセッサ向けリアルタイムOS（RTOS）の必要性も高まっている．こうした必要性の高まりを受け，我々は，マルチプロセッサ向けRTOSである，TOPPERS/FMPカーネル（以下，FMPカーネル）[1], [2]を開発し，オープンソースとして公開している．FMPカーネルの仕様や実装は，シングルプロセッサ向けRTOSであ

[1] 名古屋大学
Nagoya University, Nagoya, Aichi 464–8603, Japan
[a)] shigihara@nces.is.nagoya-u.ac.jp

るTOPPERS/ASPカーネル（以下，ASPカーネル）[3]をベースにマルチプロセッサに拡張している．ASPカーネル，FMPカーネルは，国内で広く用いられているμITRON仕様[4]をベースとしている．また，ASPカーネル，FMPカーネルは，産業界での利用を想定して開発しており，利用実績もある[5]．

近年，組込みシステムのソフトウェアを原因とする不具合が問題視されている中で，ソフトウェアの品質確保が重要な課題となっている．特に，RTOSはソフトウェアの中核をなすため，高い品質が求められる．そこで我々は，ASPカーネル，およびFMPカーネルの品質を高める取り組みとして，テストスイートの開発を実施している．テストの目的として，FMPカーネルの品質向上，および仕様と実装のトレーサビリティ確認を掲げた．前者はソフトウェアテスト本来の目的であり，このためには通常，品質評価に重きをおいたテストと，不具合を摘出することに重きをおいたテストの両方を行う[6], [7], [8]．他方，トレーサビリティ確認を掲げた理由は，RTOSはクリティカルなシステムに用いられることが多く，仕様として記述されていない動作をすることが好まれないという背景があるためである．

マルチプロセッサ向けRTOSは歴史が浅く，仕様とソースコードカバレッジの網羅を達成するためのテストプロセスや，テスト手法，および必要となるツールが不明であった．具体的には，マルチプロセッサではシングルプロセッサと比較して，テスト条件が複雑になるため実施すべきテストが膨大になり，ツールによるテストスイートの開発効率化が必要となると考えられる．また，各プロセッサの実行順序に依存してパスが定まる分岐が存在するため，これらのソースコードを網羅する方法が必要になる．これまで，汎用OSやシングルプロセッサ向けRTOSに対するテストの取り組みは存在するが，マルチプロセッサ向けRTOSに適用可能な取り組みや，ツールによるテストの効率化などは存在しない．

そこで，名古屋大学大学院情報科学研究科附属組込みシステム研究センターでは，コンソーシアム型共同研究という新しい研究スキームを提案し，複数の企業の協力を得て，マルチプロセッサ向けRTOSの検証に関するコンソーシアムを立ち上げた．本研究の目的は，マルチプロセッサ向けRTOSであるFMPカーネルに対するテスト設計の実施，テストプロセスの構築，テスト手法の考案，ツールの開発を行い，それらをもとにしたテストスイートを開発することである．

我々は，まずシングルプロセッサ向けRTOSであるASPカーネルに対するテストを実施したうえで，それを拡張する形でFMPカーネルに対するテストを実施した．対象とするASPカーネル，およびFMPカーネルに対してテスト分析を行い，抽出したテストカテゴリの中から，テストの目的である仕様とソースコードカバレッジを網羅するために必要なAPIテストと，実行順序依存分岐テストを実施した．テスト手法として，テストプログラム生成ツールの開発やテストケース設計ポリシの策定，命令セットシミュレータの拡張などを行い，テストスイート開発工数の削減やテスト実施の効率化を実現した．開発したテストスイートを用いてテストを実施した結果，ASPカーネルとFMPカーネルに対するソースコードカバレッジを100%とし，合計70件の不具合を検出した．本論文では，考案したテストプロセス，テスト手法と，開発したテストスイートの内容，およびテストの実施結果について報告し，それらを通じて得た知見を述べる．

本論文の構成は次のとおりである．2章で，テスト対象としたマルチプロセッサ向けRTOSであるFMPカーネルについて述べ，3章で関連研究について述べる．4章で我々が実施したテストの概要を，5章でテストプロセスを，6章でマルチプロセッサ向けRTOSに対するテスト手法を説明する．7章で実施したテスト実施結果について述べ，8章で検出した不具合の分析を行う．9章で，本研究で得られた知見と他のマルチプロセッサ向けRTOSへの適用可能性について考察し，10章でまとめる．

2. マルチプロセッサ向けRTOS：FMPカーネル

本章では，まずマルチプロセッサ向けOSの種類について述べ，本研究で対象としたマルチプロセッサ向けRTOSである，FMPカーネルについて述べる．

2.1 マルチプロセッサ向けOSの種類

マルチプロセッサ向けOSはタスクを実行するプロセッサを動的に変更（マイグレーション）可能かどうかによって，AMP型とSMP型に分類される．どちらのOSでも，タスクはプロセッサをまたいでOSの提供するAPIを呼び出すことが可能である．たとえば，他のプロセッサのタスクを起動したり，セマフォなどにより排他制御を実現したりできる．

2.1.1 AMP（Asymmetric Multi Processor）型

AMP型は，マイグレーションをサポートせず，タスクを特定のプロセッサに固定して実行する．タスクを管理するスケジューラや，レディキューをプロセッサごとに持つため，OSの並列実行性を高めやすいという利点がある．一方，タスクを実行するプロセッサが固定されるため，各プロセッサの負荷状況によっては，マルチプロセッサを効率良く使用することができないという問題がある．AMP型のRTOSの例として，RTEMS[9]，AMP T-Kernel[10]などがある．

2.1.2 SMP（Symmetric Multi Processor）型

SMP型は，マイグレーションをサポートしており，タ

スクを適切にプロセッサに割り付けることにより，システムのスループットを高めることが可能である．SMP 型は，単一のスケジューラとレディキューを，システム全体で共有する構成であるため，OS の並列実行性は低いという欠点がある．SMP 型の RTOS の例として，VxWorks [11]，SMP T-Kernel [10] などがある．

2.2 FMP カーネル

2.1 節で述べたように，AMP 型と SMP 型では，OS の並列実行性とスループットに一長一短があり，両立させることは困難である．

この問題に対して，汎用 OS の Linux では，スケジューラの構成を工夫することで，OS の並列実行性とスループットの両立を試みている．具体的には，AMP 型と同様にプロセッサごとにスケジューリングを行うことで，OS の並列実行性を高める．そのうえで，OS 内部のロードバランスモジュールが定期的に動作して負荷が平準化するようにタスクを動的にプロセッサに割り当てる．

FMP カーネルは，Linux のスケジューラ構成を適用し，組込みシステム向けに開発した RTOS である．組込みシステムはシステムごとに性質が異なり，有効なロードバランス方式も異なるため，OS 内部にロードバランスモジュールを実装すると，ユーザが容易にロードバランス方式を変更できないという問題がある．そのため，FMP カーネルでは OS 内にはロードバランスモジュールを実装せず，アプリケーションに対してタスクをマイグレーションする API を提供する．ユーザは，この API を用いてシステムに適したロードバランス方式を，アプリケーションレベルで実現する．

2.3 API 仕様

テストスイートが主な対象としている，FMP カーネルの API の仕様について説明する．

FMP カーネルは，シングルプロセッサ向け RTOS である ASP カーネルをベースにマルチプロセッサ向けの機能を追加している．ASP カーネルに対する拡張点は次のとおりである．

- プロセッサをまたいだ API 呼び出し
  他のプロセッサのタスクに対して ASP カーネル互換の API を発行する．
- マイグレーション機能と API
  タスクを実行するプロセッサを変更する．
- その他マルチプロセッサ向け機能と API
  スピンロック機能やプロセッサ ID 取得機能など．スピンロック機能により，OS の状態にスピンロックを取得した状態であるロック状態が追加されている．

なお，FMP カーネルの仕様の詳細については，「TOPPERS 新世代カーネル統合仕様書」（以下，仕様書）[12] を参照のこと．

2.4 コンフィギュレーション

マルチプロセッサシステムには，プロセッサの数，メモリ，タイマ，プロセッサ間排他制御方法などに関して，様々なハードウェアアーキテクチャが存在する．FMP カーネルは，様々なコンフィギュレーションを用意することにより，ハードウェアアーキテクチャの違いに対応している．コンフィギュレーションによって，サポートされる API の数や，コンパイルされるソースコードが異なる．

FMP カーネルがサポートしている主なコンフィギュレーション項目を以下に示す．

- タイマ方式：ローカルタイマ（LT）方式/グローバルタイマ（GT）方式
- ロック方式：プロセッサロック（PL）方式/ジャイアントロック（GL）方式
- スピンロック方式：ネイティブ（NS）方式/エミュレーション（ES）方式

タイマ方式には，システム時刻をプロセッサごとに管理する LT 方式と，全プロセッサで 1 つのシステム時刻を管理する GT 方式がある．ロック方式には，OS 内部のプロセッサ間の排他制御ロックをプロセッサごとに持つ PL 方式と，全プロセッサで 1 つのロックを使用する，いわゆるジャイアントロックの GL 方式がある．スピンロック方式には，OS が提供するスピンロック機能の実現に，ハードウェア排他制御機能（Test&Set 命令など）を直接用いる NS 方式と，OS 内部のロックを用いて実現する ES 方式がある．

3. 関連研究

本章では，汎用 OS や RTOS に対するテストの事例について述べる．

Linux では，Linux Test Project [13] というコミュニティにより，テストスイートを開発・公開しているが，テスト分析や，テスト手法の評価は公開されていない．また，Linux Test Project では，約 3,000 件のテストケースに対するテストプログラムを開発しているが，カーネル実装部に対するソースコードカバレッジは 40%程度にとどまり [14]，テストケースやテストプログラムは，人手で開発している．

宇宙航空研究開発機構（JAXA）では，宇宙機に使用する RTOS を高信頼化するために，「リアルタイム OS 高信頼化ハンドブック」を作成している [15]．本ハンドブックは，シングルプロセッサ向け RTOS に対して実施すべきテスト項目や，確認事項を網羅的にまとめている．

車載向けプラットフォームである AUTOSAR [16] においては，仕様に準拠して実装されているかをテストするための，コンフォーマンステストの仕様を公開している．しかし，AUTOSAR の OS に対するコンフォーマンステストは，全仕様の約 30%の網羅にとどまっている．

以上のように，汎用 OS やシングプロセッサ向け RTOS

に対するテストの取り組みは存在するが，マルチプロセッサ向けRTOSに適用可能な取り組み，たとえばツールによるテストの効率化などは存在しない．また，マルチプロセッサ向けRTOSに対するテスト設計や規模感，シングルプロセッサ向けRTOSとの違いなどの事例の報告なども存在しない．

4. テストの概要

本章では，テストスイートにおける，テスト設計の全体像を示し，テストスイートの中核であるAPIテストと，ソースコードカバレッジを100%とするために実施した実行順序依存分岐テストについて説明する．

4.1 テスト設計の全体像

我々は，1章で掲げたテストの目的に沿い，テスト設計を行うための以下の方針を作成した．

(1) 外部仕様（仕様書）を可能な限り網羅する．
(2) 1度もテストしていないコードをなくす．
(3) 不具合が発生しやすいと推測される箇所を重点的にテストする．

(1)は，品質評価に重きをおいたテストを実施するために定めた．(2)は，トレーサビリティを確認するテストを実施するために定めた．(3)は，不具合を摘出することに重きをおいたテストを実施するために定めた．この方針に沿ってテスト設計を行った結果，22個のテストカテゴリを抽出し，それらを仕様ベース（方針(1)），設計・ソースコードベース（方針(2)），およびエラー推測（方針(3)）の3つに分類した．結果を**表1**に示す．

RTOSのテストとしては，表1に示した機能要件だけでなく，APIの最悪実行時間や，応答時間予測性などの非機能要件のテストも重要であるが，本研究では工数の兼ね合いから対象外とした．また，RTOSは汎用OSとは異なり，I/Oの隠蔽化はデバイスドライバで行うので，RTOSのテストとしてはI/O制御のテストは実施しない．

本研究を実施するコンソーシアム型共同研究は，2009，2010年度の2年間実施され，研究担当者は参加企業のエンジニアであり，2009年度が4名，2010年度が7名であった．この限られたリソースの中で，すべてのテストカテゴリを実施するのは困難であったので，我々は優先度を決めて順に実施していくことにした．その結果，ブラックボックスAPIテスト，ホワイトボックスAPIテスト，実行順序依存分岐テストの3つのテストを，2年間で完了した．優先度を決めるにあたり，ソフトウェアテスト本来の目的であることと，テスト実施方法が明確であることから，テストスイート開発の第1段階としては，方針(1)，(2)すなわち，仕様とソースコードカバレッジの網羅を目的とすることにした．

なお，前述のように，FMPカーネルは，ASPカーネルをベースとしているため，まず，ASPカーネルに対するテストを実施したうえで，それを拡張する形でFMPカーネルに対してテストを実施することにした．以降で両カーネルに共通する事項について説明する場合は，単にRTOSと呼ぶ．

表1 テストカテゴリ一覧
Table 1 List of the test categories.

(1) 仕様ベースのテスト	
A	ブラックボックスAPIテスト
B	処理単位テスト
C	SILテスト
D	クラス関連テスト
E	カーネル管理外割込みのテスト
F	カーネル起動/終了時の同期テスト
(2) 設計・ソースコードベースのテスト	
A	ホワイトボックスAPIテスト
G	実行順序依存分岐テスト
H	ターゲットシステム依存テスト
I	スタートアップモジュールテスト
J	コンフィギュレータテスト
K	機能拡張・チューニングテスト
(3) エラー推測テスト	
L	ロック区間テスト
M	割込み禁止区間テスト
N	タイマ割込みテスト
O	スピンロック中の割込みテスト
P	マイグレーションテスト
Q	割込み出入口処理テスト
R	CPU例外出入口処理テスト
S	アイドル処理テスト
T	デッドロック回避テスト
U	バリア同期テスト

4.2 方針(1)に対するテスト

方針(1)の主な目的は，RTOSの品質の向上である．RTOSのユーザは，仕様書に記述された仕様に従ってアプリケーションを開発する．つまり，仕様に従っていない実装が存在すると，RTOSで開発されたアプリケーションが誤作動する原因となりうる．この不具合は，仕様書に記述されたすべての仕様をテストすることで，検出することが可能である．

仕様ベースのテストの中で，我々はまずブラックボックスAPIテスト（以下，B-APIテスト）から着手することにした．B-APIテストは，RTOSが提供するAPIが，仕様書のとおりに正しく振る舞うことを確認する．B-APIテストから着手した理由は，APIの実装がRTOSのソースコードの大部分を占めていることから，RTOSが保証すべき最も重要な品質であると考えたからである．

多くのAPIは，発行により，ロック状態やディスパッチ禁止状態などのカーネルの状態や各オブジェクトの状態

（以下，システム状態）を変化させる．そこで，B-API テストでは，あるシステム状態で API を発行し，仕様どおりのシステム状態の変化が起きることを確認する．具体的には，テストプログラムで，API 発行前のシステム状態（前状態）を実現し，その状態でテスト対象となる API を発行（処理）し，API 発行後のシステム状態（後状態）を確認する．

4.3 方針 (2) に対するテスト

方針 (2) の主な目的は，方針 (1) に対するテストの評価，および RTOS の設計の正当性検証である．方針 (2) に対するテストとして，設計・ソースコードベースのテストの，ホワイトボックス API テスト（以下，W-API テスト）と実行順序依存分岐テストを実施することにした．これらのテストは，RTOS の API に関するソースコードカバレッジを，100%とすることを目的とする．なお，命令網羅[*1]によるソースコードカバレッジを 100%としても，仕様に記述のない実装が存在しないことを確認できないため [17]，条件網羅[*2]によるソースコードカバレッジ（以下，C1 カバレッジ）を 100%とすることを目的とする．

なお，RTOS のソースコードは，大きくターゲットシステムごとに異なる実装（以下，ターゲット依存部）と，ターゲットシステムに関係なく共通な実装（以下，ターゲット非依存部）に分けられるが，本テストにおけるカバレッジ測定の範囲は，ターゲット非依存部のみとした．これは，ターゲット依存部が主にアセンブリ言語で記述されており，カバレッジ測定が困難であることと，API の実装からターゲット依存部を呼び出しているため，API が仕様どおりに振る舞うことをテストすることで，ターゲット依存部が正しく実装されていることも同時にテストできると考えたからである．

4.3.1 W-API テスト

まず，B-API テストによる C1 カバレッジを確認する．C1 カバレッジが 100%とならない場合，以下の要因が考えられる．

(i) 仕様に現れない実装依存となる処理が存在する．
(ii) 各プロセッサの実行順序に依存してパスが定まる分岐が存在する．

要因 (i) は，RTOS のソースコードに，ヒープ操作やビットマップサーチなど，仕様に現れない部分があるためである．これらのコードは，B-API テストだけでは C1 カバレッジを 100%にはできない．そこで，W-API テストとして，意図的に要因 (i) の処理を実行するためのシステム状態を前状態として実現し，API の発行を行うテストを実施する．

なお，B-API テストと W-API テストは，テストの目的は異なるものの，テストケースやテストプログラムの共通点が多いため，以降で両者に共通する事項について説明する場合は，単に API テストと呼ぶ．

4.3.2 実行順序依存分岐テスト

FMP カーネルでは，API の実行が各プロセッサの実行順序に依存しないように，スピンロックによるロックでプロセッサ間の排他制御を行ってから処理を実行している．ただし，設計上，FMP カーネルには，ロックを取得していない状態で実行する必要があるコードを含む処理が，共通部ロック関数と，デッドロック回避処理にのみ存在する．これらは，各プロセッサの実行順序に依存してパスが定まる分岐（以下，実行順序依存分岐）を有しているので，前項で述べた要因 (ii) に該当する．

API テストは，API 発行によるシステム状態の変化を確認するテストであるので，実行順序依存分岐は，W-API テストでは網羅できない．そこで，実行順序依存分岐テストとして，後述の命令セットシミュレータ（以下，ISS）を拡張する手法（以下，拡張 ISS）によって，実行順序依存分岐を網羅するためのテストを実施することにした．

以下に，共通部ロック関数とデッドロック回避処理について説明する．

共通部ロック関数

API 内では，データの整合性を保つために排他制御を行う．ASP カーネルの場合は，割込み禁止により実現している．一方，FMP カーネルはマルチプロセッサで動作するため，割込み禁止に加えてハードウェア排他制御機能（Test&Set 命令など）によるスピンロックを用いたプロセッサ間の排他制御が必要となる．API 発行により，ロックの取得を試みた際に，別のプロセッサがすでにロックを取得していた場合は，ロックを取得できるまでループして待機する．共通部ロック関数の例を**図 1** に示す．この例では，すでに他のプロセッサ上で API が発行されていて，ロックを取得できない場合にのみ，3 行目の判定が false となり，7 行目以降のロック取得失敗パスを通る．この 3 行目の分岐が，実行順序依存分岐である．8，9 行目は，割込

```
 1: void acquire_lock() {
 2:   while(true) {
 3:     if ( try_lock() ) {  /* 実行順序依存分岐 */
 4:       /* ロック取得成功パス */
 5:       break;
 6:     }
 7:     /* ロック取得失敗パス */
 8:     enable_int();  /* 割込み許可 */
 9:     disable_int(); /* 割込み禁止 */
10:   }
11: }
```

図 1 共通部ロック関数の例
Fig. 1 An example of the common lock function.

*1 すべての命令を少なくとも 1 回は実行する．
*2 すべての条件式の判定による真偽を少なくとも 1 回は実行する．

```
低優先度の実行状態のタスクから，高優先度の
起床待ち状態のタスクに対して wup_tsk を発行すると，
対象タスクが実行状態になること．
```

図 2 テストケースの例
Fig. 2 An example of the test case.

み応答性向上のために，ロック取得に失敗した場合に，割込みを許可する微小期間を入れる処理である．

デッドロック回避処理

FMP カーネルには，複数のロックを取得する際に，デッドロックを回避するための処理が存在する．API や OS の内部関数によっては，タスクロックとオブジェクトロックという 2 種類のロックを同時に取得する必要がある [18]．

多くの API や OS の内部関数では，オブジェクトロック→タスクロックの順序でロックを取得するが，一部は逆順に取得するため，デッドロックが発生する可能性がある．この一部の処理では，タスクロックを取得した後でなければ，取得すべきオブジェクトロックを特定できないので，タスクロックを取得して，取得すべきオブジェクトロックを特定した後，いったんタスクロックを解放して，オブジェクトロック→タスクロックの順序でロックを取得することで，デッドロックを回避する．このロックを解放して 2 種類のロックを取得する間に，他のプロセッサによって操作対象の状態が変更される可能性があるため，ロック取得後に操作対象の状態をチェックし，必要ならロックを再取得する必要がある．このロック取得後の操作対象の状態チェックが，実行順序依存分岐となる．

4.4 API テストにおけるテストケースとテストシナリオの定義

我々は，各種の標準 [19], [20] を参考にして，テストケースを次のように定義した．テストケースとは，特定のプログラムパスの実行や，指定された要求に適合していることの検証など，特定の目的のために作成された，入力値，実行事前条件，期待結果の対である．API テストにおけるテストケースの例を**図 2** に示す．このテストケースは，テストされるべき状況を動作ルールの形で抽象的に示している．この例は，起床待ち状態のタスクを起床する API である wup_tsk が，正しく動作することを確認するためのテストケースの 1 つである．

テストシナリオは，テストケースで指定された状況を再現するための具体的なタスクやシステム状態，および API 呼び出しの操作手順を示したものである．図 2 のテストケースの例を，テストシナリオで表現したものを**図 3** に示す．テストケースでは，TASK1 が wup_tsk を発行した後に，実行状態からどの状態へ遷移するかは，明記されていないが，テストシナリオでは，後状態で実行可能状態となることも表現される．

```
前状態
  優先度（低）の TASK1 が実行状態
  優先度（高）の TASK2 が起床待ち状態

処理
  TASK1 が wup_tsk(TASK2) を発行し，
  エラーコードとして E_OK が返る

後状態
  TASK1 が実行可能状態となる
  TASK2 が実行状態となる
```

図 3 テストシナリオの例
Fig. 3 An example of the test scenario.

```
pre_condition:
  TASK1:
    type   : TASK
    tskpri : LOW
    tskstat: running
  TASK2:
    type   : TASK
    tskpri : HIGH
    tskstat: waiting
    wobjid : SLEEP

do:
  id     : TASK1
  syscall: wup_tsk(TASK2)
  ercd   : E_OK

post_condition:
  TASK1:
    tskstat: ready
  TASK2:
    tskstat: running
```

図 4 TESRY 記法の例
Fig. 4 Example of the TESRY notation.

5. テストプロセス

本章では，API テスト，および実行順序依存分岐テストに対するテストプロセスについて述べる．テストプロセスのフローを**図 5** に示す．網掛けした部分が，FMP カーネルでのみ必要であることを示す．テストプロセスは，テスト設計，テストプログラム作成，テスト実施の 3 つのフェーズで構成される．

5.1 テスト設計フェーズ

テスト設計フェーズでは，テスト分析からテストケースの設計までを行う．

5.1.1 (1) テスト分析

4 章で述べたように，まず，テスト分析を行い，テストカテゴリの抽出と実施するテストの選定を行った．

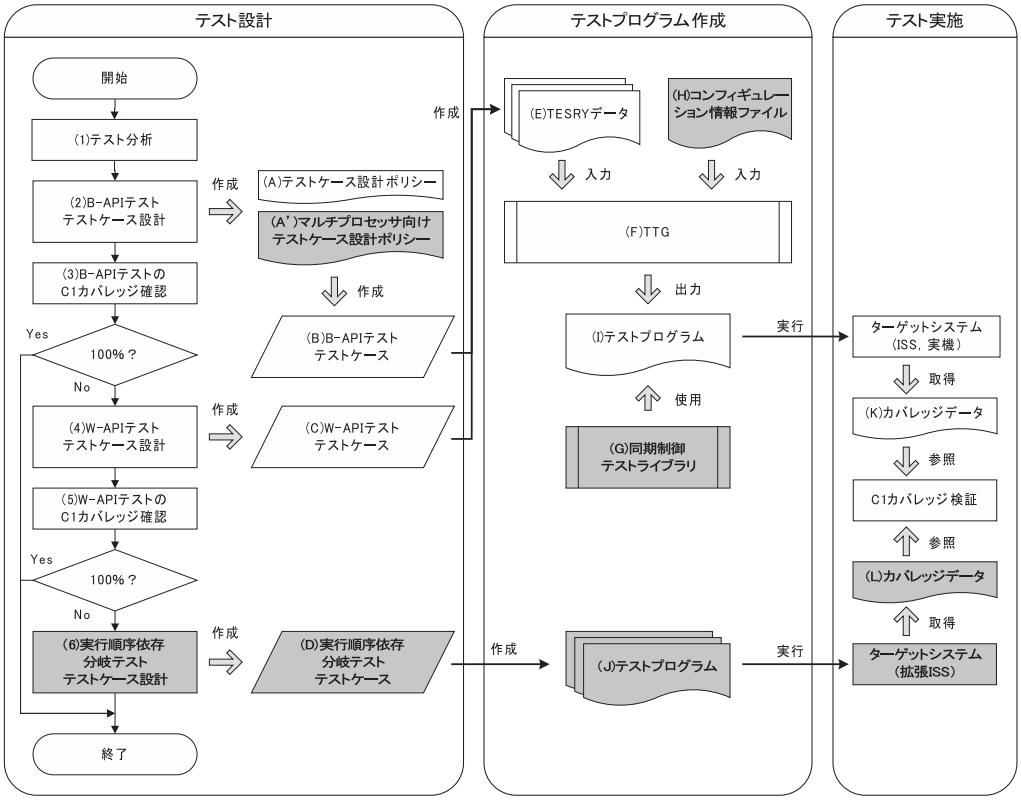


図 5 テストプロセスのフロー
Fig. 5 Flow of the test process.

5.1.2 (2) B-API テストのテストケース設計

B-API テストでは，仕様書に記載された全 API の振舞い，具体的には，タスクの状態やシステム状態によって変化する API の挙動を網羅的にテストする．テストケースは，仕様書に記載されている全 API の仕様のカバレッジ（以下，仕様カバレッジ）を 100%とするように抽出する．

仕様書は，抽象的な表現で記述されているため，開発者によって抽出するテストケースにばらつきが生じる可能性がある．たとえば，wup_tsk には，『対象タスクが起床待ち状態でなく，休止状態でもない場合には，対象タスクの起床要求キューイング数に 1 が加えられる』という仕様が存在する．ここで，“起床待ち状態でも休止状態でもない状態” は，ASP カーネル，FMP カーネルともに，20 種類以上存在する．このような抽象的な表現で記述された仕様に対してテストケースを設計する場合，いずれかの具体的な状態を指定する必要があるが，ルールやポリシがないと，設計者によって指定する状態が異なってしまう．

そこで，テストの実施範囲や，同値分割の粒度などのポリシを定め，API ごとに隔たりなく，テストケースを抽出することにした．我々はまず ASP カーネルに対するテストケース設計ポリシを定め [21]（図 5 (A)），それを FMP カーネルに拡張する形で設計した [22]（図 5 (A')）．詳細については，6.1 節で述べる．

定めたポリシに従って，B-API テストのテストケースを作成する（図 5 (B)）．

5.1.3 (3) B-API テストの C1 カバレッジ確認

API ごとに B-API テストのテストケースが実行された場合に，RTOS 内のどのパスを通るかを，人手で調査する．具体的には，API ごとに RTOS 内の条件式を抽出し，テストケースごとの判定結果を条件網羅表にまとめる．この結果，すべての条件式が，true と false にそれぞれ 1 回以上判定されている API は，C1 カバレッジが 100%と判断し，テストケースの設計は完了となる．true，false のどちらかしか判定されていない，あるいは 1 度も判定されていない条件式が存在する API は，次項で述べるホワイトボックスの観点でのテストケース設計を行う．

5.1.4 (4) W-API テストのテストケース設計

4.3 節で述べたように，RTOS 内には，仕様に現れない部分があるため，B-API テストのすべてのテストケースを実施したとしても，C1 カバレッジは 100%にならない．これに対し，ホワイトボックステストとして，網羅していないパスを網羅するためのテストケースを追加する（図 5 (C)）．具体的には，(3) B-API テストの C1 カバレッジ確認でまとめた条件網羅表を参考に，true と false をそれぞれ 1 回以上判定されていない条件式を網羅するためのテストケースを設計する．

5.1.5 (5) W-API テストの C1 カバレッジ確認

(3) B-API テストの C1 カバレッジ確認で網羅できなかった分岐とパスに対して，W-API テストによる C1 カバレッジを確認する．4.3 節で述べたように，FMP カーネルに

は，W-API テストでは網羅できない実行順序依存分岐が存在するため，次項で述べる実行順序依存分岐テストのテストケース設計を行う．

5.1.6 (6) 実行順序依存分岐テストのテストケース設計

4.3.2 項で述べたように，FMP カーネルの共通部ロック関数，およびデッドロック回避処理に存在する，実行順序依存分岐を網羅するテストケースを設計する（図 5 (D)）．実行順序依存分岐テストのテストケースは，FMP カーネル内に存在する実行順序依存分岐を列挙したものであり，1 つの実行順序依存分岐に対して 1 テストケースとする．

5.2 テストプログラム作成フェーズ

テストプログラム作成フェーズでは，テスト設計フェーズによって作成したテストケースを実現するためのテストプログラムを開発する（図 5 (I)，(J)）．

API テストのテストプログラムは，後述する独自に考案した記法でテストシナリオを記述し（図 5 (E)），ツールを用いることで自動生成する（図 5 (F)）．さらに FMP カーネルに対しては，後述するコンフィギュレーション情報ファイル（図 5 (H)）と，プロセッサ同期制御ライブラリ（図 5 (G)）を使用する．

実行順序依存分岐テストのテストプログラムは，4.3.2 項で述べたように拡張 ISS を使用する必要があり，API テストとはテストプログラムの内容が大きく異なるので，前述のツールは使用せず，人手で開発する．詳細は 6.3 節で述べる．

5.2.1 TESRY データ

次項で説明するテストプログラム生成ツールのために，テストシナリオの記述方法として，TESRY（TEst Scenario for Rtos by Yaml）記法を定めた．TESRY 記法で記述したデータファイルを TESRY データと呼ぶ（図 5 (E)）．図 3 のテストシナリオを TESRY 記法で記述した例を**図 4** に示す．

5.2.2 TTG

TTG（Toppers Test Generator）は，TESRY データ（テストシナリオ）を入力として，テストプログラムを生成するツールである（図 5 (F)）．TTG を開発した動機は，テストプログラムを人手で開発すると，開発工数や保守性に問題が発生するためである．ASP カーネル，および FMP カーネルの全 API に対するテストケースは，数千件と予想されたため，これらに対応するテストプログラムを一定の品質を保ちつつ人手で実装することは困難である．具体的には，あるテストシナリオを実現するテストプログラムの実装方法は複数存在しうる．そのため，複数の開発者でテストプログラムを実装した場合，実装方法を統一することが困難となり，レビューや修正作業の負担が大きくなる．

一方，テストケースからテストシナリオを作成する作業は，変換法則さえ決めてしまえば，開発者間でそれほどばらつくことはなく，統一が容易である．そこで，人手で行う作業は TESRY 記法によるテストシナリオの作成までとし，そこから先は TTG を用いてテストプログラムを生成することにした．

以下に，TTG によるテストプログラム生成の概要について説明する．TTG は，まず TESRY データに定義された前状態を実現するために，RTOS の仕様に基づいて，各オブジェクトを対象とする状態へ遷移させるソースコードを生成する．前状態を実現した後で，システム状態を確認するための API（以下，システム状態確認 API）を用いてすべてのオブジェクトが，TESRY データに定義された前状態と一致していることを確認するソースコードを生成する．次に，TESRY データに定義された処理を，そのまま実行するソースコードを生成する．API の戻り値が指定されている場合，戻り値のチェックを行うソースコードも生成する．最後に，システム状態確認 API を用いて，すべてのオブジェクトが，後状態で定義された状態となっていることを確認するソースコードを生成する．

また，TTG は，複数の TESRY データを入力することで，それらをまとめたテストプログラムを生成する．TTG へ入力する TESRY データの数を調整することで，ターゲットシステムが搭載するメモリサイズに収まるテストプログラムを生成することが可能である．

TTG，および TESRY 記法の詳細については文献 [23] を参照のこと．

5.3 テスト実施フェーズ

テスト実施フェーズでは，作成したテストプログラムを実行してカバレッジデータを取得する（図 5 (K)，(L)）．取得したカバレッジデータが，テスト設計で想定したパスを通過して，C1 カバレッジが 100%となることを確認する．

6. マルチプロセッサ向け RTOS に対するテスト手法

本章では，5 章で述べたテストプロセスによって設計したテストを，実施するために考案した，マルチプロセッサ向け RTOS に対するテスト手法について述べる．

6.1 マルチプロセッサ向け RTOS に対するテストケース設計ポリシ

ASP カーネルでサポートされる API は，すべて FMP カーネルでもサポートされるため，ASP カーネル向けに作成したテストケースは，FMP カーネルにおいて，いずれかのプロセッサで実行することで流用が可能である．さらにその API に対して，プロセッサをまたいだパターンのテストケースの追加が必要になる．また，FMP カーネルで追加された API やシステム状態（ロック状態）で API を発行するテストケースも必要である．したがって，FMP カー

ネルに対するテストケースは以下の 4 つに分けられる．
(a) ASP カーネル互換機能のテストケース
(b) プロセッサをまたいだパターンのテストケース
(c) FMP カーネルで追加された API に対するテストケース
(d) ロック状態で API を発行するテストケース

(a) は，ASP カーネルに対して作成したテストケースを，そのまま使用する．(b) は，(a) のテストケースに対して，マルチプロセッサでしか起こりえない条件に対するポリシを追加する形で作成した．ポリシの一部を以下に列挙する．

- API 発行対象オブジェクトの他プロセッサへの割付け
- ディスパッチ保留状態の他のプロセッサに対する API 発行
- 実行状態タスクが存在しないプロセッサに対する API 発行
- FMP カーネルで追加された状態のタスクに対する API 発行

(c) は，主にマイグレーションを実現するための API である．(c) に対しても，ASP カーネルと同等のテストケース設計ポリシを適用し，同時に (b)，(d) で行った拡張も実施することで作成した．(d) は，ロック状態での API 発行によって，エラーが発生する仕様に対するテストケースである．

6.2 TTG のマルチプロセッサ拡張

TTG は，まず ASP カーネル向けに開発し，その後，FMP カーネル向けに拡張した．FMP カーネル向けに拡張した点について説明する．

6.2.1 ASP カーネル用 TESRY データの再利用

前述の (a) のテストケースに対するテストプログラムは，ASP カーネル向けのテストプログラムを FMP カーネルで利用することができれば，開発工数を大幅に減らすことができる．しかしながら，ASP カーネルと FMP カーネルでは，API の互換性はあるものの，タスクなどの生成記法や，システム状態確認 API による確認項目などがプログラムレベルで異なるため，テストプログラムの互換性はない．

図 6 に ASP カーネル向けテストプログラムの例を，**図 7** に FMP カーネル向けテストプログラムの例を示す．上側のプログラムは，タスクの定義を行うシステム設定ファイルの抜粋で，下側のプログラムが，図 4 の前状態（pre_condition）において，TASK1 から TASK2 の状態を確認する処理の抜粋と，処理（do）で API を発行する処理である．

FMP カーネルでは，システム設定ファイルにおいて，各タスクを割り付けるプロセッサを指定するために，CLASS という識別子で囲む必要がある．TCL_1 はプロセッサ 1 に割り付けるグループを示す．また，TASK2 の状態確認を行うプログラムにおいて，図 7 の 8 行目のように，FMP カーネルでは TASK2 が割り付けられているプロセッサ

```
CRE_TSK(TASK1, 1, task1, 7, ...);
CRE_TSK(TASK2, 1, task2, 8, ...);
```

```
 1: void task1(intptr_t exinf){
 2:   …
 3:  ercd = ttsp_ref_tsk(TASK2, &rt);
 4:  assert(ercd == E_OK);
 5:  assert(rt.tskstat == TTS_WAI);
 6:  assert(rt.tskpri == 8);
 7:  assert(rt.exinf == 1);
 8:
 9:  ttsp_check_point(1);
10:
11:  ercd = wup_tsk(TASK2)
12:  assert(ercd == E_OK);
13:   …
14: }
```

図 6 ASP カーネル向けテストプログラムの例
Fig. 6 An example of the test program for ASP kernel.

```
CLASS(TCL_1){
  CRE_TSK(TASK1, 1, task1, 7, ...);
  CRE_TSK(TASK2, 1, task2, 8, ...);
}
```

```
 1: void task1(intptr_t exinf){
 2:   …
 3:  ercd = ttsp_ref_tsk(TASK2, &rt);
 4:  assert(ercd == E_OK);
 5:  assert(rt.tskstat == TTS_WAI);
 6:  assert(rt.tskpri == 8);
 7:  assert(rt.exinf == 1);
 8:  assert(rt.prcid == 1);
 9:  ttsp_mp_check_point(1, 1);
10:
11:  ercd = wup_tsk(TASK2)
12:  assert(ercd == E_OK);
13:   …
14: }
```

図 7 FMP カーネル向けテストプログラムの例
Fig. 7 An example of the test program for FMP kernel.

ID の確認が追加される．さらに，9 行目は ASP カーネル，FMP カーネルともに，実行順序をチェックするための関数を呼び出すが，関数名と引数が異なる．FMP カーネルの場合は，実行順序チェック用のシーケンシャルな数字（第 1 引数）に加えて，処理が実行されたプロセッサ ID（第 2 引数）の確認も必要となるからである．ASP カーネルと FMP カーネルは，API の互換性はあるので，11，12 行目のテスト対象の API 発行と戻り値の確認は，共通のコードとなる．

このテストプログラムの互換性の問題に対して，本研究では，テストケースを TESRY データで作成していること

を利用し，TTG により，ASP カーネル用の TESRY データから FMP カーネル用のテストプログラムを生成することにした．具体的には，TTG 実行時のオプションで，対象とする RTOS を指定可能とし，指定された RTOS に対応するテストプログラムを生成する．しかし，図 4 のように，ASP カーネル用の TESRY データには，タスクなどのオブジェクトを割り付けるプロセッサが指定されていない．そこで，FMP カーネル用に生成する場合は，未指定のオブジェクトを割り付けるプロセッサを別のオプションとして指定することで，テストプログラムを生成する．これにより，図 4 の TESRY データを TTG に入力した場合，指定されたオプションによって，図 6 のテストプログラムと，図 7 のテストプログラムのどちらも生成可能とした．

結果として，(a) のテストケースは，ASP カーネルに対して作成した TESRY データを再利用することで，FMP カーネルに対する TESRY データ開発工数を削減することが可能である．

6.2.2 プロセッサ間同期制御

マルチプロセッサ向け RTOS に対するテストプログラムでは，プロセッサ間の同期制御が必要となる．図 4 のテストケースにおいて，TASK1 と TASK2 が異なるプロセッサに割り付けられている場合を例として考える．TASK1 がプロセッサ 1 に，TASK2 がプロセッサ 2 に割り付けられている状態で，TASK1 がプロセッサをまたいで，TASK2 へ wup_tsk を発行する．プロセッサ 1 には TASK1 しか存在しないため，図 4 の後状態とは異なり，TASK1 は wup_tsk 発行後も実行状態のままとなる．処理フローを**図 8** に示す．

プロセッサ 1 とプロセッサ 2 は独立して処理が実行されるので，プロセッサ 2 において，TASK2 がいったん起動して起床待ち状態へ遷移する前に，TASK1 が wup_tsk を発行してしまう可能性がある．この場合，TASK1 は，TASK2 が起床待ち状態となったことを確認してから wup_tsk を発行する必要がある．これは，同期制御 1 によって，TASK1 から，TASK2 が起床待ち状態となることをループして待機することで解決する（StateSync）．また，API 発行によって意図したシステム状態へ変化したことの確認を，TASK2 が実行状態へと遷移する前に実行すると問題が発生するので，TASK2 が実行状態となってからシステム状態を確認する必要がある．そのため，同期制御 2 によって，TASK2 が実行状態になったのを待ち合わせる（ConditionSync）．

このように，正しくテストプログラムを実行するには，適切な同期制御を行う必要がある．我々は，作成した API テストの全テストケースにおいて必要となる同期の状況を同期パターンとして 10 種類にまとめ，同期するための方法を同期メカニズムとして 4 種類にまとめた．そして，テストシナリオの内容に応じて，適切な同期処理をテストプログラムへ組み込む機能を，TTG に実装した [24]．

同期パターンと同期メカニズムの一覧を，**表 2** に示す．

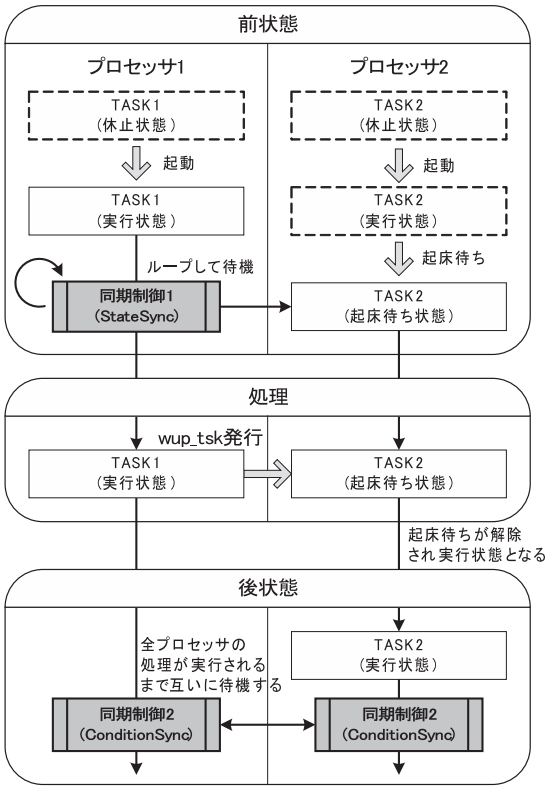


図 8 同期制御を含むテストプログラムの処理フロー
Fig. 8 Flow of a test program including synchronization.

表 2 同期パターンと同期メカニズム
Table 2 The synchronization pattern and the synchronization mechanism.

同期パターン	同期メカニズム
ActiveSync，LockSync，TimeSync，ExecSequenceSync，DoStartSync，LastCheckedSync	CheckPointSyncFunc
StateSync，DoFinishSync	StateSyncFunc
ConditionSync	BarrierSyncFunc
LoopTerminated	FinishSyncFunc

図 8 で使用した同期パターンが，StateSync と ConditionSync である．以下に，StateSync と ConditionSync を実現するための，同期メカニズムである，StateSyncFunc，および BarrierSyncFunc について説明する．

StateSyncFunc は，特定のタスクが特定の状態になるまで待つ同期方法である．対象のタスク ID と対象の状態を引数に渡す関数で実現し，待つ側のタスクから実行する．実行された関数は，指定したタスクが指定した状態へ遷移するまでリターンしないことで同期を行う．ただし，RTOS の不具合などで，指定したタスクが指定した状態へ遷移しない場合，無限ループとなってしまうため，一定時間でタイムアウトし，エラーを出力する．

BarrierSyncFunc は，2 つ以上の指定したプロセッサ間での実行タイミングを合わせる同期方法である．BarrierSyncFunc を使用するたびにカウントアップするシーケン

シャルな番号（シーケンス番号）と，実行タイミングを合わせるプロセッサの数を引数に渡す関数で実現し，実行タイミングを合わせるすべてのプロセッサから実行する．テスト開始時に，シーケンス番号を管理する変数（管理変数）を，プロセッサの数だけ確保し，0 で初期化しておき，関数の実行によって，実行されたプロセッサに対応する管理変数をインクリメントする．すべてのプロセッサの管理変数のうち，引数に指定された実行タイミングを合わせるプロセッサの数だけ，指定されたシーケンス番号と一致したことを確認できたらリターンする．BarrierSyncFunc においても，StateSyncFunc と同様に一定時間でタイムアウトし，エラーを出力する．

6.2.3 コンフィギュレーション対応

2.4 節で述べたように，FMP カーネルでは，コンフィギュレーションによって，サポートされる API の数や，コンパイルされるソースコードが異なることから，テストの観点では，すべてのコンフィギュレーションの組合せに対応したテストの実施が必要である．我々は，すべての組合せをカバーするように API テストのテストケースを作成したが，対象とするターゲットシステムで実施可能なテストケースのみを，ユーザが取捨選択するのは容易ではない．そこで，TTG 実行時に対象とするターゲットシステムのコンフィギュレーション情報も入力し，同時に入力された TESRY データの中から，実施可能なテストケースのみを取捨選択し，テストプログラムを生成する機能を設けた．

6.3 拡張 ISS

本節では，拡張 ISS の概要と，拡張 ISS によって実行順序依存分岐を網羅する方法について説明する．拡張 ISS の詳細については，文献 [25] を参照のこと．

6.3.1 実行順序依存分岐テストのテスト手法

共通部ロック関数による実行順序依存分岐を含む API の例として，セマフォを取得する API である wai_sem() を用いて，実行順序依存分岐テストのテスト手法について説明する．

wai_sem() では，複数のプロセッサから同時に発行された場合に，データの整合性を保つため，図 1 に示した共通部ロック関数の acquire_lock() を使用して，ロックを取得したプロセッサのみが，セマフォの取得処理を行う．B-API テストとして，wai_sem() をテストする場合，1 つのプロセッサからのみ wai_sem() を発行するので，1 回目の実行でロックを取得でき，ロック取得成功パスを通る．したがって，acquire_lock() の実行順序依存分岐における，ロック取得成功パスは，API テストによって網羅することができる．一方，ロック取得失敗パスは，API テストでは網羅できない．

次に，ロック取得失敗パスが実行されるプロセッサ間の実行順序の例を，**図 9** に示す．プロセッサ 1 の TASK1 と，プロセッサ 2 の TASK2 が，同一のセマフォに対して wai_sem() をほぼ同時に発行した場合の例である．TASK2 から発行した wai_sem() によって，TASK2 がロックを取得した後で，TASK2 がロックを取得中に，TASK1 の wai_sem() によるロック取得が行われると，ロックの取得に失敗し，acquire_lock() 内のロック取得失敗パスが実行される．このロック取得失敗パスを通すテストが，実行順序依存分岐テストである．

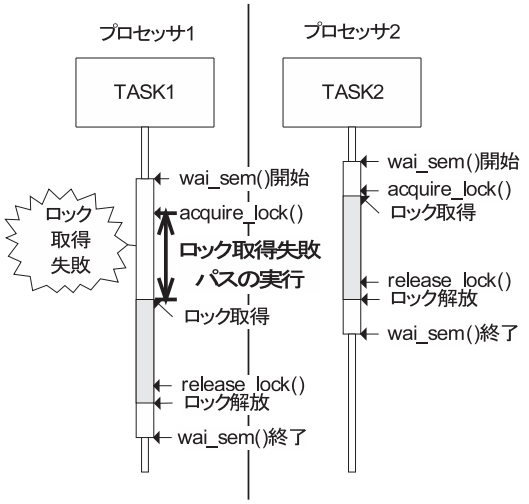


図 9 ロック取得失敗パスが実行される例
Fig. 9 An example of executing an acquiring failure path.

acquire_lock() のロック取得失敗パスを実行するテストを実施する場合，図 9 のように，異なるプロセッサから wai_sem() を同時に発行するテストを，繰り返し実行する方法が考えられるが，ロックを取得してから解放するまでの時間は非常に短いので，確実に通すことは困難である．我々はまず，FMP カーネルのすべての実行順序依存分岐に対して，テスト対象のパスを通る可能性が高いテストプログラムを，それぞれ 100 万回繰り返し実行したが，1 回も通らないパスが存在した．

テスト対象のパスを確実に通すための方法として，FMP カーネルのソースコード中に，実行順序を制御するためのコードを追加する方法が考えられるが，テスト対象のソースコードを修正する必要があるので，品質保証の観点から避けるべきである．したがって，FMP カーネルのソースコードを変更せずに，実行順序依存分岐を決定的に網羅するための手法が必要となる．

6.3.2 拡張 ISS による実行順序依存分岐の網羅

我々は，前項で述べた問題に対して，実行制御機構を実装した ISS を開発し，テストプログラムと ISS が協調して動作することで，API 内の実行順序依存分岐を決定的に網羅する機構を用いたテスト手法を考案した．具体的には，各プロセッサで動作させるテストプログラムに対して，指定したプログラムカウンタでの停止・再開を制御するインタフェースなど，意図したパスを確実に通すための機構を

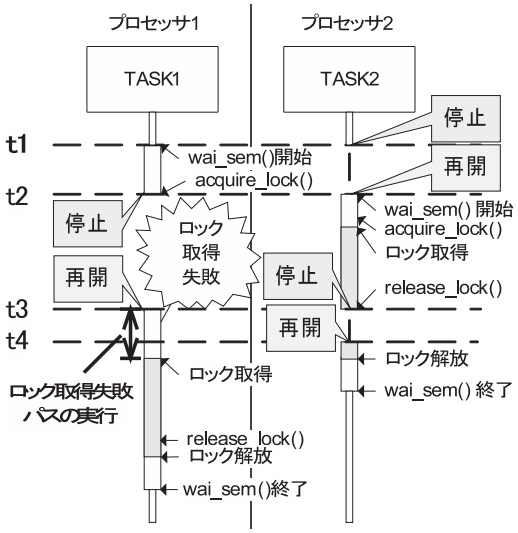


図 10 ISS 拡張によるロック取得失敗パスの実行フロー
Fig. 10 Execution flow an acquiring failure path by extended ISS.

用意した．

図 9 に示した，wai_sem() で実行される acquire_lock() 内のロック取得失敗パスの実行を例に，拡張 ISS を説明する．拡張 ISS を用いて，wai_sem() 内のロック取得失敗パスを実行する場合のフローを，**図 10** に示す．

まず，TASK1 が wai_sem() を呼び出す前（t1）に，プロセッサ 2 を停止させる．そして，ロック取得を試行する直前まで，TASK1 の処理を進めるため，acquire_lock() を呼び出すとき（t2）に，プロセッサ 2 を再開し，プロセッサ 1 を停止させる．プロセッサ 2 では，TASK2 が wai_sem() を呼び出してロックを取得する．TASK2 が release_lock() でロックを解放する前（t3）に，プロセッサ 2 を停止させ，プロセッサ 1 を再開させると，TASK1 はロックを取得できないためロック取得失敗パスを実行する．ここで，プロセッサ 2 がロックを取得したまま停止していると，プロセッサ 1 はロック取得の試行を繰り返すため，プログラムの実行が終了しない．そのため，TASK1 がロック取得失敗パス（図 1 の 7–9 行目）を実行してから，再度ロック取得を試行する（図 1 の 3 行目）までの間でプロセッサ 2 を再開させてロックを解放する必要がある．その間のアドレスを指定するため，関数の外からも参照可能なラベルを図 1 の 9 行目と 10 行目の間に入れる．TASK1 はそのラベルを実行したとき，プロセッサ 2 を実行させればよい．このようなラベルの挿入は，テスト対象のソースコードを変更することになるが，オブジェクトコードに影響は与えないため，許容した．

以上のように，各プロセッサに対して指定したプログラムカウンタでの停止・再開を制御する機構を用いることで，wai_sem() における実行順序依存分岐を網羅することができた．本例以外の FMP カーネルに存在するすべての実行順序依存分岐に対しても，考案した拡張 ISS を用いることで，決定的に網羅できることを確認した．なお，我々は，FMP カーネルの開発や動作確認に使用する ISS として，ARM プロセッサの ISS である SkyEye [26] を，マルチプロセッサの動作をシミュレートできるように拡張して使用している [27]．本研究においても，テスト対象の ISS として SkyEye を使用しており，拡張 ISS も，このマルチプロセッサ対応した SkyEye をベースに開発した．

7. テスト実施結果

本章では，5 章で述べたテストプロセスを，6 章で述べたテスト手法を用いて，ASP カーネル Release 1.4.0，および FMP カーネル Release 1.1.0 に対して実施した結果について述べる．テスト実施結果の一覧を**表 3** に示す．FMP カーネルのポリシ数やテストケース数は，FMP カーネルのみに必要なポリシやテストケースの数であり，() 内の値が ASP カーネルから流用したものとの合計である．ROM サイズ，RAM サイズは，SkyEye をターゲットとして，GCC（Sourcery G++ Lite 2011.03-42）[28] によりコンパイルした結果である．

7.1 API テスト

ASP カーネルは，TTG で生成したテストプログラムによって，C1 カバレッジが 100%となることを確認した．FMP カーネルでも同様に実施した結果，コンフィギュレーションによって値が異なるが，最も高いもので，C1 カバレッジは 96.7%であった．残りのパスは，4.3 節で想定したとおり，すべて共通部ロック関数，およびデッドロック回避処理にのみ存在する実行順序依存分岐によって実行されないパスであった．

なお，C1 カバレッジは，SkyEye を使用したテストで取得した．SkyEye は，ARM 純正コンパイラなどでサポートされるセミホスティングインタフェースと呼ばれる機能をサポートしている．この機能を用いると，標準関数を経由して，SkyEye が動作しているホスト（PC）のファイルの読み書きなどが可能となる．我々が使用した GCC もセミホスティングインタフェースに対応しており，GCC の実行カバレッジ取得機能である GCOV を使用すると，カバレッジ結果をホスト側に出力できる．我々はこの GCOV により，実行カバレッジを取得した．具体的な取得方法としては，カバレッジを取得するように，GCC にオプションを付加してコンパイルを行い，テストプログラムを SkyEye で実行した後，標準関数 exit() を呼び出すと，カバレッジ結果がホスト側にファイルとして出力される [29]．後はこのカバレッジ結果を GCOV により確認する．

実際にテストを実施したターゲットシステムと，それぞれのコンフィギュレーション，および検出した不具合の件数を**表 4** に示す．FMP カーネルに対する全テストケースをまとめて 1 つのテストプログラムとして生成して実行す

表 3 テスト実施結果一覧
Table 3 Results of tests.

	ASP カーネル	FMP カーネル
API 数	103	120
テストケース設計ポリシ数	14	12（計 26）
B-API テストによるテストケース数	1,629	2,535（計 4,164）
W-API テストによるテストケース数	46	11（計 57）
TESRY データ総行数	60,648	170,625
TTG で生成したテストプログラム行数	224,719	695,314（※ 1）
TTG で生成したテストプログラム ROM サイズ（kByte）	3,745	12,238（※ 1）
TTG で生成したテストプログラム RAM サイズ（kByte）	247	1,333（※ 1）
カバレッジ対象ソースコード行数	1,888	3,389（※ 1）
API テストによる C1 カバレッジ	100%	96.7%（※ 1）
実行順序依存分岐の数	0	83
実行順序依存分岐テストのテストプログラム総行数	-	9,356
実行順序依存分岐テストを含めた C1 カバレッジ	-	100%
ターゲット非依存部の不具合件数	9	43
ターゲット依存部の不具合件数	2	16

※ 1 最大値（コンフィギュレーションによって若干異なる）

表 4 ターゲットシステム一覧
Table 4 List of the target systems.

RTOS	ボード	プロセッサタイプ	プロセッサ数	コンフィギュレーション			不具合検出数
				タイマ方式	ロック方式	スピンロック方式	
ASP カーネル	SkyEye（ISS）	AT91 システム	-	-	-	-	0
	AP-SH2A-0A	SH2A（SH7211）	-	-	-	-	2
	MS7727CP01	SH3（SH7727）	-	-	-	-	0
FMP カーネル	SkyEye（ISS）	AT91 システム	4	※ 1			2
	AP-SH2AD-0A	SH2A-DUAL（SH7205）	2	GT 方式	GL 方式	NS 方式	3
	NaviEngine	ARM11MPCore	4	LT 方式	PL 方式	NS 方式	9 (※ 2)
	KZM-CA9-01	ARM11MPCore	4	LT 方式	PL 方式	NS 方式	
	Core Tile for ARM11 MPCore Emulation Baseboard	ARM11MPCore	4	LT 方式	PL 方式	NS 方式	
	NiosII Development Kit Stratix	NiosII	2	LT 方式	PL 方式	ES 方式	2

※ 1 シミュレータのため任意の方式を選択可能
※ 2 ARM11MPCore に対する不具合検出数

るには，ターゲットシステムに約 12 MB の ROM と，約 1.3 MB の RAM が必要であった．ただし，すべてのターゲットシステムで，このサイズのメモリを用意できないため，5.2.2 項で述べたように，ターゲットシステムのメモリサイズに応じて，TTG へ入力する TESRY データの数を調整し，テストプログラムを複数回に分けて実行した．

7.2 実行順序依存分岐テスト

FMP カーネルにおいて，API テストで網羅できない実行順序依存分岐は，共通部ロック関数が 75 カ所，デッドロック回避処理が 8 カ所の，合計 83 カ所存在した．実行順序依存分岐テストでは，6.3 節で述べたテスト手法を用いて，API 内の実行順序依存分岐 83 カ所を網羅するためのテストプログラムを人手で開発し，実行した．実行順序依存分岐テストのテストプログラムは 9,356 行であった．

実行順序依存分岐テストのテストプログラムも，技術的には TTG によって生成することも可能であったが，実行順序依存分岐テストは，API テストと実施する内容が大きく異なるので，TTG では対応しなかった．具体的には，実行順序依存分岐テストにおける前状態，処理，後状態を TESRY 記法で記述する場合，実行順序を制御するための詳細な情報（プログラムカウンタなど）を定義する必要があり，TESRY 記法自体の拡張が必要である．また，拡張した TESRY 記法に対する TTG の対応も必要である．しかし，実行順序依存分岐は 83 カ所と少ないため，これらの課題に対応するより，人手で開発したほうが，工数が少

ないと判断した．

実行順序依存分岐テストによって実行されたパスは，APIテスト同様，GCOV によって確認し，実行順序依存分岐83 カ所が網羅されることを確認した．これにより，7.1 節で述べた API テストの結果と合わせて，FMP カーネルに対する C1 カバレッジも 100%となった．

8. 検出した不具合の分析

本章では，検出した不具合について述べ，発生原因や内容の分析を行う．

8.1 API テストで検出した不具合

API テストで検出した不具合は合計 69 件であった．不具合の種類ごとに分類した件数の一覧を**表 5** に示す．

(A)，(B) は，1 章で述べたテストの目的に合致した不具合であり，意図した不具合を検出できたといえる．(C)，(D) は，ASP カーネル，FMP カーネルの実装に潜在していた不具合であり，特にターゲット依存部の不具合が多かった．これは，4.3 節で述べたように，API の実装からターゲット依存部を呼び出しているため，API が仕様どおりに振る舞うことをテストすることで，ターゲット依存部が正しく実装されていることも同時にテストできているためである．

(E) は，テスト実行結果には直接影響を与えなかったが，テスト設計や，C1 カバレッジ確認の過程で検出された不具合であり，ソースコード中のコメントやマクロ名の誤記などである．

8.1.1 マルチプロセッサ特有の不具合

API テストは，API 発行前後のシステム状態を確認するテストであるので，実行結果は決定的であり，何度実行しても同じ結果が得られると予想される．しかし，API テストを繰り返し実行することで，再現率が 100%ではない不具合が確認された．

たとえば，アプリケーションにスピンロックを提供するスピンロック機能における，ハードウェアのロック（HW ロック）と，OS 管理上のロック状態（OS ロックフラグ）の更新の順序に関する不具合である．スピンロックを解放する API において，誤って HW ロック解放→ OS ロックフラグオフという順序で実装してしまった場合，この間に他のプロセッサからスピンロックを取得する API を発行されると，他のプロセッサから HW ロック取得→ OS ロックフラグオンを行った後で，OS ロックフラグをオフにしてしまうので不整合が発生する．本現象は，スピンロックを解放する API のテスト実行時，稀に発生した．

表 5 API テストにおける不具合の種類と件数
Table 5 Types and amounts of bugs detected by API test.

	ASP カーネル	FMP カーネル
(A) 不要なソースコード，分岐が存在する	1	4
(B) 仕様書に記載のない実装が存在する	3	1
(C) 期待した後状態とならない	3	38
(D) 意図しない順序で処理が実行された	2	7
(E) その他	2	8

このような不具合は，各プロセッサのレースコンディションに依存する不具合であり，単に API テストを 1 度実行するだけでは検出できない可能性がある．また，該当するソースコードに対する C1 カバレッジは 100%となるので，実行順序依存分岐テストの対象として抽出することができない．これに対し我々は，各プロセッサの実行順序を意図的にばらつかせ，API テストを繰り返し実行することで検出した．

テストに使用したマルチプロセッサ対応の SkyEye は，1 つの PC 上で複数個起動し，それらがメモリ空間を共有することで，マルチプロセッサ環境をシミュレートする．SkyEye では，シミュレートするプロセッサにおいて 1 命令を実行することを，1 ステップとして処理する．プロセッサの数だけ起動したそれぞれの SkyEye は，事前に設定したステップ数（同期ステップ数）ごとに相互に同期処理を行い，特定のプロセッサだけ処理が進むことがないように制御する．同期ステップ数を小さい値とした場合，すべてのプロセッサ間の同期処理の頻度が高く，実用に耐えない処理速度となってしまうので，通常は同期ステップ数を 1,000 ステップで使用している．実機のマルチプロセッサにおいては，プロセッサ間の処理が 1,000 ステップずれることは考えにくいが，レースコンディションに依存する不具合検出のために，毎回異なる同期ステップ数に設定して，繰り返しテストを実施した．同期ステップ数は，1,000 から 10,000 ステップまでの間の値から，ランダムに選択した．10,000 ステップより大きくすることも可能であるが，同期間隔が極端に大きいと特定のプロセッサの処理だけ実行され続け，プロセッサ間同期制御でタイムアウトが発生する可能性があることや，デッドロックや無限ループの検出が遅れる可能性があるため，10,000 ステップを妥当な最大値と判断した．

レースコンディションに依存した不具合をすべて検出するには，どの程度繰り返し実行すればよいか，という点については，明確な判断基準は我々の知る限り存在しない．ただし，我々は，テストプログラムの生成からビルド，実行までをスクリプトで自動化し，時間が許す限り，ヒートラン的に繰り返し実施することで不具合が検出されなくなることで確認した．また，前述のように，実機ではプロセッサ間の処理のばらつきが発生しないと考えられるため，プロセッサ間の処理をばらつかせた SkyEye によって不具合が検出されなければ，実機で発生する可能性はきわめて低いと考えられる．

8.1.2 コンフィギュレーションに依存した不具合

FMP カーネルで検出された不具合のうち，9 件はコンフィギュレーションに依存した不具合であった．これは特定のコンフィギュレーションで実行した場合のみ発生する不具合である．6.2.3 項で述べたように，すべてのコンフィギュレーションの組合せに対してテストを実施することは重要であると考えられる．

また，コンフィギュレーションに依存した不具合の多くは，ロック方式が PL 方式の場合に発生した．PL 方式では，ロックの粒度を細かくして，プロセッサ数に対する性能のスケーラビリティを上げる方式である．ロックの粒度が細かくなるため，ソースコードが複雑になり，多くの不具合が発生したと考えられる．

なお，割込み制御や共有メモリ制御など，ターゲットシステムのハードウェアアーキテクチャに依存した処理に関する不具合は，API テストでは検出できないため，表 1 にあげたターゲットシステム依存テスト（H）として，別途実施する必要がある．

8.2 実行順序依存分岐テストで検出した不具合

6.3 節で述べた手法を用いてテストを実施したところ，システム時刻に関する制御を行う関数内で 1 件の不具合を発見した．以下に，この不具合について説明する．不具合発生のフローを**図 11** に，不具合が検出された処理の疑似コードを**図 12** にそれぞれ示す．

プロセッサ 1 とプロセッサ 2 にそれぞれ，TASK1, TASK2 が割り付けられている．TASK1 は，タイムアウト付きでセマフォ取得を行う API である twai_sem() を，セマフォ

```
 1: void signal_time(void) {
 2:     割込み禁止 ;
 3:     タスクロック取得 ;
 4:     現在時刻の更新 ;
 5:     while ( 現在時刻で発生するタイムイベントがある ) {
 6:         タイムイベントヒープからタイムイベントを削除 ;
 7:         タイムイベントを実行 ;
 8:     }
 9:     タスクロック解放 ;
10:     割込み許可 ;
11: }
12:
13: void wait_tmout(void) {
14:     if ( タスクがオブジェクト待ち状態 ) {
15:         タスクロック解放 ;
16:         オブジェクトロック取得 ;
17:         タスクロック取得 ;
18:         if ( タスクの待ち状態を解除された ) {
19:             オブジェクトロック解放 ;
20:             return;
21:         }
22:         オブジェクト待ちキューからタスクを削除 ;
23:         オブジェクトロックの解放 ;
24:     }
25:     タスクの待ち状態を解除 ;
26:     スケジューリング ;
27:     if ( ディスパッチが必要 ) {
28:         ディスパッチ ;
29:     }
30: }
```

図 12 不具合が検出された処理の疑似コード
Fig. 12 Pseudo-code for bug detection process.

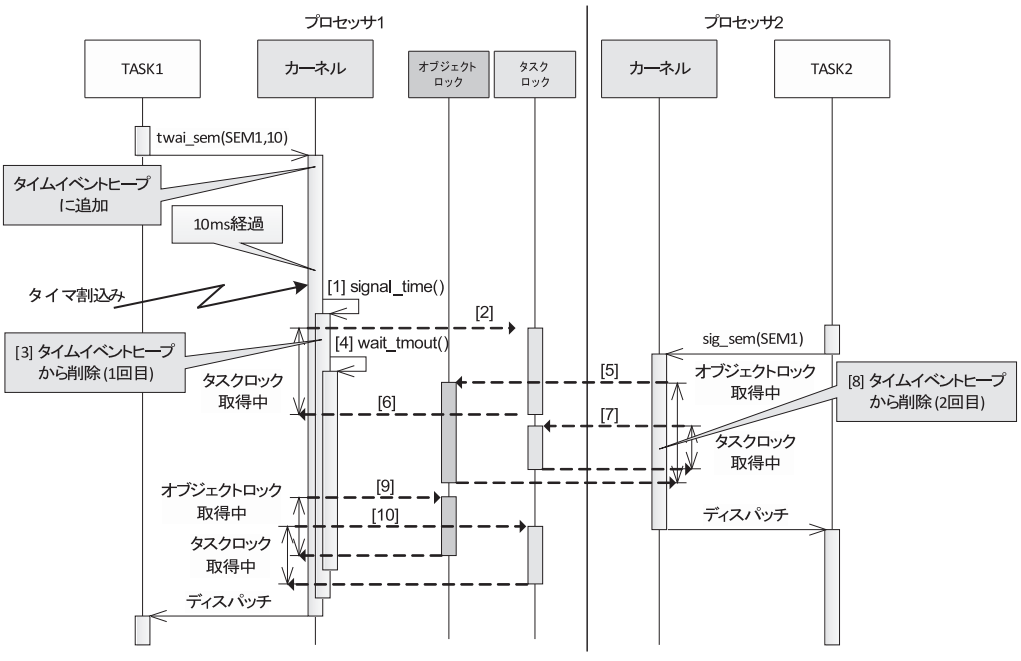


図 11 不具合発生のフロー
Fig. 11 Flow of a bug.

SEM1 に対して発行する．セマフォを取得できない場合，TASK1 は待ち状態へ遷移し，タイムアウトするタスクを管理するためのタイムイベントヒープに，カーネルによって追加される．タイムアウト時間に指定した 10 ミリ秒を経過してもセマフォを取得できない場合，タイムアウトが発生し，カーネルによってタイムイベントヒープから削除され，TASK1 の待ち状態が解除される．タイムアウトが発生する前に，セマフォを取得していた別のタスクから，セマフォを返却する API である sig_sem() を発行された場合は，TASK1 がセマフォを取得して，待ち状態が解除される．ここで，タイムアウトにより，カーネルによって TASK1 の待ち状態が解除される処理の実行中に，TASK2 が sig_sem() を SEM1 に対して発行した場合に，不具合が発生した．

プロセッサ 1 では，タイマ割込みが 1 ミリ秒ごとに発生し，プロセッサ 1 のシステム時刻を更新している．TASK1 が twai_sem() を発行してから，10 ミリ秒経過したときの処理は次のようになる．タイマ割込みが発生すると，プロセッサ 1 のシステム時刻を更新するための関数である signal_time() が呼ばれる（図 11 [1]）．signal_time() では，システム時刻更新中に他のプロセッサからタスクの状態を操作されないように，タスクロックを取得してからシステム時刻を更新する（図 11 [2]）．システム時刻を更新した後，タイムアウトが発生するタイムイベントがある場合に，タイムイベントヒープから該当するタイムイベントを削除して（図 11 [3]），タイムイベントを実行する（図 11 [4]）．TASK1 による twai_sem() 発行によって，このタイムイベントには，待ち状態を解除する関数である wait_tmout() が登録されている．

wait_tmout() では，待ち状態を解除する対象のタスクが，セマフォなどのオブジェクト待ちである場合は，オブジェクトロックを取得して，オブジェクト待ちキューからタスクを削除する．ここで，そのままオブジェクトロックを取得すると，ロックを取得する順番が，タスクロック→オブジェクトロックという順になるが，4.3.2 項で述べたように，FMP カーネルでは，デッドロック回避のためにオブジェクトロック→タスクロックの順でロックを取得する必要がある．そこで，いったんタスクロックを解放して（図 11 [6]），オブジェクトロック→タスクロックの順で取得し直す（図 11 [9]，[10]）．

一方，sig_sem() ではオブジェクトロック→タスクロックの順にロックを取得した後（図 11 [5]，[7]），セマフォ取得待ちのタスクが存在すれば，セマフォ待ちキューからタスクを削除し，さらにタイムイベントがあればタイムイベントヒープからタイムイベントを削除する（図 11 [8]）．プロセッサ 1 において，デッドロック回避のために，wait_tmout() 内でいったんタスクロックを解放したときに，プロセッサ 2 で動作する TASK2 が sig_sem（SEM1）を発行した場合，プロセッサ 2 がプロセッサ 1 のタスクロックを取得できる（図 11 [7]）．このタイミングで発行された sig_sem() の処理において，タイムイベントがあるかどうかの判定処理に不具合があり，結果として，TASK1 のタイムイベントヒープの削除はプロセッサ 1 ですでに実行されているが，プロセッサ 2 でも実行され，不整合が発生してしまった．

この不具合は，プロセッサ 1 が図 12 の 15 行目と 17 行目の間を実行中に，プロセッサ 2 が sig_sem() によってタスクロックを取得して，TASK1 の待ち状態を解除する処理を実行するときに発生する．そこで，プロセッサ 1 がタスクロックを解放する前（14 行目と 15 行目の間）に，すでにタイムイベントヒープから削除されたことを示すフラグを更新することで，sig_sem() によって不正にタイムイベントヒープの削除処理が実行されないよう，修正した．

wait_tmout() はタイムアウトが発生した時刻のみ呼び出されるので，頻繁に実行されるものではない．本不具合は，図 11 のフローが発生する可能性のあるテストプログラムを，100 万回繰り返し実行しても 1 回も発生しなかったため，拡張 ISS を用いなければ検出が困難な不具合であった．

9. 得られた知見と他の RTOS への適用

本章では，本研究を通じて得られた知見について述べ，開発したテストスイートを，他のマルチプロセッサ向け RTOS へ適用する場合について考察する．

9.1 得られた知見

9.1.1 マルチプロセッサ向け RTOS のテストの規模

API テストにおいては，ASP のテストケースに加えて，FMP カーネルではさらに同程度以上の規模の開発が必要となった．

FMP カーネルで追加したテストケース 2,535 件のうち，6.1 節で述べた (b) のテストケースは 1,783 件，(c) のテストケースは 588 件，(d) のテストケースは 164 件であった．つまり，FMP カーネルで追加された API に対するテストケースよりも，ASP カーネルから存在する API に対して追加したプロセッサをまたいだテストケースのほうが，3 倍近く多かった．これは，ASP カーネルの API（103 個）の中で，プロセッサをまたいだテストケースの追加が必要な API は 80 個であり，ほとんどの API でプロセッサをまたいだテストケースの追加が必要となったためである．

ASP カーネルでは，API あたりのテストケース数が約 16 件であるのに対し，FMP カーネルで追加された API（13 個）に対する (c) のテストケースでは，API あたりのテストケース数は約 45 件となった．これは，前述のマルチプロセッサ特有のテストケースに加えて，一部のマイグレーション API において，API 発行元のプロセッサ，マイグレーション元の他のプロセッサ，マイグレーション先の他のプロセッサの，3 つのプロセッサにまたがるパターンを

考慮したテストケースが必要となったためである．

なお，プロセッサをまたいだパターンである (b) のテストケースの件数が，ASP カーネルのテストケースの約 1.1 倍であることから，マイグレーションや新規 API の追加がない，FMP カーネル以外の RTOS であっても，マルチプロセッサに対する API テストのテストケースは，シングルプロセッサの 2 倍以上必要であると考えられる．

9.1.2 プログラム自動生成による効率化

表 3 に示したとおり，TTG で生成したテストプログラムに対して TESRY データの総行数は 4 分の 1 程度であり，開発規模を大幅に低減させた．仮に人手ですべてのテストプログラムを開発する場合，ASP カーネルのテストプログラムは，そのままでは FMP カーネルに再利用できないため，FMP カーネル用として新規に，4,221 件分のテストプログラムの開発が必要である．我々が考案したテスト手法によって，テストプログラムを開発しなくてもよいこと，ASP カーネル用に作成した TESRY データを FMP カーネルに再利用できることで，削減できる開発コストは非常に大きい．

また，FMP カーネルの全 TESRY データを入力として，テストプログラムを生成した結果，6.2.2 項で述べたプロセッサ間同期制御ライブラリは，約 19,000 カ所登場した [24]．これらの同期制御処理を，人手で開発した場合，適切な同期パターンに相当する同期制御処理を記述する必要があるのに加えて，テストシナリオの修正にともなう同期処理の修正が必要となる．したがって，テストシナリオの内容に応じて，適切な同期処理をテストプログラムへ組み込む機能は，開発コストの削減につながった．

さらに，FMP カーネルでは，コンフィギュレーションによって実施可否が異なるという問題に対して，TTG によって自動的に取捨選択できること，ターゲットシステムのメモリサイズに応じて，テストプログラムのサイズを調整できることは，テストスイートを使用する際の，テスト実施の効率化も実現している．

以上より，マルチプロセッサ向け RTOS に対するテストスイート開発，およびテストの実施では，ツールの活用による効率化が有用であることを確認した．

9.1.3 ISS の有用性

組込みシステム開発において，実機を用いてテストする場合，テストプログラムを実機へロードする必要があり，コードサイズが大きいとテストの実行に時間を要してしまう．一方，ISS であれば，PC 上で実行できるため，テストの実行は容易である．そのため，テストスイートの開発では，ISS を使用することで開発工数を削減することが可能である．

また，ISS は実行する PC 上のメモリを使用するので，事実上，メモリ制約もない．したがって，本研究で開発した規模のテストケース数であっても，一括して生成したテストプログラムを実行でき，テスト実施時間の短縮にもつながる．さらに，8.1.1 項で述べたように，マルチプロセッサ向け RTOS では，各プロセッサのレースコンディションに依存した不具合が存在する可能性があるため，意図的に各プロセッサ上で動作する ISS の実行順序をばらつかせて，かつヒートラン的にテストを実行する手法が有効である．

以上より，マルチプロセッサ向け RTOS に対するテストにおいて，ISS を活用することは有効である．

9.1.4 拡張 ISS による C1 カバレッジ網羅の重要性

マルチプロセッサ向け RTOS には，4.3 節で述べたように，実行順序依存分岐が存在する．我々は，6.3 節で述べた手法を用いて，これらのパスを網羅することによって，不具合を 1 件検出した．つまり，4.1 節で述べたように，一般的なソフトウェアに対するテストと同様に，マルチプロセッサ向け RTOS に対しても，1 度もテストしていないコードをなくすという方針が重要であることを確認した．

実行順序依存分岐の網羅は，実機では実現が困難であるが，ISS では比較的容易に実現することで可能であり，拡張性の面でも ISS は有効である．

9.2 異なる仕様の RTOS への適用

本研究で開発したテストスイートは，2.3 節で述べた「TOPPERS 新世代カーネル統合仕様書」に準拠した RTOS に対応している．また，本仕様書は μITRON ベースで規定されてるため，μITRON 仕様に準拠した RTOS に対しても対応可能であると考えられる．そのため，タスクの状態やスケジューリング方式，API の仕様などが，μITRON 仕様と異なる他の RTOS に対しては，利用できない．しかし，5 章で述べたテストプロセスは，他の仕様の RTOS に対しても適用可能であり，TESRY 記法や TTG を含めたテスト手法についても，対象とする RTOS の仕様に合わせて改変することは可能である．

たとえば，AUTOSAR 仕様の OS でも，FMP カーネルと同様に，タスクの起動，終了などの API が提供されており，マルチプロセッサ環境においては，プロセッサをまたいで発行できることも規定されている．我々は，AUTOSAR 仕様に対応した OS を開発しており，同時にテストスイートの開発も行っている [30], [31]．タスクがとりうる状態や，オブジェクトの種類などが異なるので，多少の修正は必要であるが，TESRY 記法と TTG を，AUTOSAR 仕様へ対応することは可能であった．また，AUTOSAR 仕様の OS においても，仕様カバレッジを 100%としても，C1 カバレッジは 100%とすることはできないことや，シングルプロセッサ向けに開発したテストケースを，マルチプロセッサ向けに流用できることも共通しており，大部分で本研究のテスト手法が適用できることを確認した．

9.3 他のタイプのマルチプロセッサ向け RTOS への適用

2.1 節で述べた，FMP カーネルと異なる AMP 型，および SMP 型のマルチプロセッサ向け RTOS への適用可能性について検討する．

9.3.1 AMP 型

FMP カーネルからマイグレーション機能を外すと AMP 型と同等になる．したがって，FMP カーネルで追加されたマイグレーションに関する項目を除けば，本研究で開発したテストケース設計ポリシやテストスイートをそのまま使用することが可能である．また，6.2.2 項で述べた同期制御ライブラリについては，AMP 型に対しても適用することが可能である．

9.3.2 SMP 型

SMP 型は，FMP カーネルとはスケジューリング方式が大きく異なるため，テストケース設計ポリシは流用できない．特に API 発行前の状態を，全プロセッサにまたがって考慮する必要があるので，FMP カーネルより複雑となることが予想される．したがってテストケース設計ポリシも複雑になる．ただし，6.2.2 項で述べた同期制御ライブラリについては，SMP 型であっても同様の同期が必要であるので，適用することが可能である．

9.4 今後の課題

TTG のようにツールによりテストプログラムを生成する場合，ツールの正当性を検証する作業が必要である．我々は，TTG が生成したテストプログラムのレビューに加えて，テストケースごとにテストプログラムを生成し，それぞれ実行した場合の C1 カバレッジが，事前に想定したパスを通っていることを確認することで（図 5 (3)，(5)），TTG が正しいテストプログラムを生成していることの確認とした．しかし，本作業は非常に工数を要するので，効率的にツールの正当性を検証する手法について，今後研究する予定である．

また，現状，仕様書の仕様と TESRY データ間は関連付けがなされていないため，仕様変更や不具合などの理由で，仕様書やソースコードに修正が発生した場合，その影響範囲や修正する必要のある TESRY データの抽出は，手作業で行う必要がある．仕様書，ソースコード，テストスイート間でのトレーサビリティを向上させる手法やツールについても，今後研究する予定である．

10. おわりに

本研究では，仕様ベース，設計・ソースコードベース，エラー推測の 3 つの観点による，μITRON ベースのマルチプロセッサ向け RTOS である FMP カーネルに対するテストの全体像を設計し，このうち仕様ベースのブラックボックス API テスト，設計・ソースコードベースのホワイトボックス API テスト，および実行順序依存分岐テストを実施した．また，マルチプロセッサ向け RTOS に対するテスト手法として，マルチプロセッサに特化したテストケース設計ポリシを策定し，プロセッサ間同期制御ライブラリ，拡張 ISS を開発し，コンフィギュレーションへの対応を行った．ASP カーネルと FMP カーネルに対する API テスト，FMP カーネルに対する実行順序依存分岐テストを完了し，かつ対象とした C1 カバレッジを 100%とし，合計 70 件の不具合を検出して RTOS の品質向上に貢献した．

今後の取り組みとして，抽出したテストカテゴリの残りのテストの実施，および非機能要件のテストがあげられる．非機能要件のテストとしては，最悪ロック取得待ち時間のプロセッサ数に対するスケーラビリティの研究 [32] などを実施しており，今後非機能要件のテストも可能なテストスイートの開発を検討している．

謝辞 本研究の推進に協力してくださった，コンソーシアム型研究の皆様に謹んで感謝の意を表する．

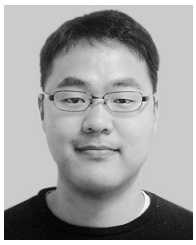
参考文献

[1] 石田利永子，本田晋也，高田広章，福井昭也，小川敏行，田原康宏：TOPPERS/FMP カーネル：リアルタイム性と高スループットを実現可能な組込みシステム向けマルチプロセッサ用 RTOS，コンピュータソフトウェア，Vol.29, No.4, pp.219–243 (2012).
[2] TOPPERS/FMP カーネル (online), available from ⟨http://www.toppers.jp/fmp-kernel.html⟩ (accessed 2012-05-28).
[3] TOPPERS/ASP カーネル (online), available from ⟨http://www.toppers.jp/asp-kernel.html⟩ (accessed 2012-05-28).
[4] 坂村 健（監修），高田広章（編）：μITRON4.0 仕様 Ver.4.02.00，トロン協会 (2004).
[5] TOPPERS プロジェクト/適用事例 (online), available from ⟨http://www.toppers.jp/applications.html⟩ (accessed 2012-05-28).
[6] Myers, G.J., Badgett, T., Thomas, T.M. and Sandler, C.: The art of software testing, John Wiley & Sons Inc. (2004).
[7] 玉井哲雄：ソフトウェア工学の基礎，岩波書店 (2004).
[8] 保田勝通：ソフトウェア品質保証の考え方と実際—オープン化時代に向けての体系的アプローチ，日科技連出版社 (1995).
[9] RTEMS (online), available from ⟨http://www.rtems.com/⟩ (accessed 2012-05-28).
[10] MP T-Kernel (online), available from ⟨http://www.t-engine.org/ja/what-is-t-kernel/mpt-kernel⟩ (accessed 2012-05-28).
[11] VxWorks (online), available from ⟨http://www.windriver.com/japan/products/vxworks/⟩ (accessed 2012-05-28).
[12] TOPPERS 新世代カーネル統合仕様書 (online), available from ⟨http://www.toppers.jp/documents.html⟩ (accessed 2012-05-28).
[13] Linux Test Project (online), available from ⟨http://ltp.sourceforge.net/⟩ (accessed 2012-05-28).
[14] Modak, S. and Singh, B.: Building a Robust Linux Kernel piggybacking The Linux Test Project, *Linux Symposium* (*OLS*) (2008).
[15] 佐藤伸子，石濱直樹，川崎朋実，片平真史：宇宙機搭載

用リアルタイム OS に適用した高信頼化技術のハンドブック化，組込みシステムシンポジウム 2011 論文集，pp.25-1–25-7 (2011).
[16] AUTOSAR (online), available from ⟨http://www.autosar.org/⟩ (accessed 2012-05-28).
[17] Glover, A.：コード品質を追求する：カバレッジ・レポートに騙されないために，IBM developerWorks (online), available from ⟨http://www.ibm.com/developerworks/jp/java/library/j-cq01316/⟩ (accessed 2012-05-28).
[18] 本田晋也，高田広章：ITRON 仕様 OS の機能分散マルチプロセッサ拡張，電子情報通信学会論文誌 D，Vol.J91-D, No.4, pp.934–944 (2008).
[19] IEEE: IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990 (1990).
[20] IEEE: IEEE Standard for Software Test Documentation, IEEE Std 829-1998 (1998).
[21] 鴫原一人，松浦光洋，金ハンソル，金スンヨプ，馬　鋭，廉　正烈，金　榮柱，木村貴寿，眞弓友宏，本田晋也，山本雅基，高田広章：組込みリアルタイム OS に対する API テストの実施，ソフトウェアテストシンポジウム 2010 予稿集，pp.46–53 (2010).
[22] 松浦光洋，金ハンソル，眞弓友宏，金スンヨプ，廉　正烈，金　榮柱，木村貴寿，鴫原一人，馬　鋭，森　孝夫，本田晋也，山本雅基，高田広章：マルチプロセッサ対応 RTOS のテスト開発，情報処理学会研究報告，Vol.2010-EMB-16, No.10, pp1–8 (2010).
[23] 鴫原一人，眞弓友宏，森　孝夫，本田晋也，高田広章：μITRON ベースの RTOS 向けテストプログラム生成ツール，電子情報通信学会論文誌 D，Vol.J95-D, No.4, pp.870–884 (2012).
[24] 浅見侑太，鴫原一人，本田晋也，山本晋一郎，高田広章：マルチプロセッサ対応 RTOS 向けテストプログラム生成ツールにおけるプロセッサ間同期の実現，情報処理学会研究報告，Vol.2011-EMB-20, No.11 (2011).
[25] 一場利幸，森　孝夫，高瀬英希，鴫原一人，本田晋也，高田広章：命令セットシミュレータの実行制御機構を用いたマルチプロセッサ RTOS のテスト効率化手法，電子情報通信学会論文誌 D，Vol.J95-D, No.3, pp.387–399 (2012).
[26] SkyEye (online), available from ⟨http://sourceforge.jp/projects/sfnet_skyeye/⟩ (accessed 2012-05-28).
[27] 安積卓也，古川貴士，相庭裕史，柴田誠也，本田晋也，冨山宏之，高田広章：オープンソース組込みシステム向けシミュレータのマルチプロセッサ拡張，コンピュータソフトウェア，Vol.27, No.4, pp.24–42 (2010).
[28] Sourcery G++ Lite (online), available from ⟨http://www.codesourcery.com/⟩ (accessed 2012-05-28).
[29] 本田晋也：TOPPERS カーネル用シミュレーション環境 TISE の使い方，CQ 出版社 Interface 2012 年 5 月号，pp.90–97 (2012).
[30] 「次世代車載システム向け RTOS の仕様検討および開発に関するコンソーシアム型共同研究を開始」プレス発表（オンライン），入手先 ⟨http://www.nces.is.nagoya-u.ac.jp/press/nces-release-1105.pdf⟩（参照 2012-05-28）.
[31] 風間佳之，平橋　航，鴫原一人，海上智昭，本田晋也，高田広章：AUTOSAR OS に対するテストケースおよびテストプログラムの自動生成，ソフトウェアテストシンポジウム 2012 予稿集，pp.17–24 (2012).
[32] 一場利幸，松原　豊，本田晋也，高田広章：中断可能な優先度継承キューイングスピンロックとそのハードウェア実装，情報処理学会論文誌コンピューティングシステム，Vol.4, No.3, pp.133–146 (2011).

鴫原 一人　（正会員）

2003 年武蔵工業大学（現，東京都市大学）工学部電子通信工学科卒業．同年首都圏松下テクニカルサービス（現，パナソニックテクニカルサービス）（株）入社．2007 年富士ソフト（株）入社．2010 年名古屋大学大学院情報科学研究科附属組込みシステム研究センター出向．リアルタイム OS の研究に従事．日本ソフトウェア科学会会員．



一場 利幸　（正会員）

2009 年名古屋大学大学院情報科学研究科博士前期課程修了．2012 年同博士後期課程満期退学．2012 年より同附属組込みシステム研究センター研究員．マルチコアシステムのリアルタイム性，リアルタイム OS の研究に従事．

本田 晋也　（正会員）

2002 年豊橋技術科学大学大学院情報工学専攻修士課程修了．2005 年同大学院電子・情報工学専攻博士課程修了．2005 年名古屋大学情報連携基盤センター名古屋大学組込みソフトウェア技術者人材養成プログラム産学官連携研究員．2006 年名古屋大学大学院情報科学研究科附属組込みシステム研究センター助教．現在，同准教授．リアルタイム OS，ソフトウェア・ハードウェアコデザインの研究に従事．博士（工学）．2002 年度情報処理学会論文賞受賞．IEEE，電子情報通信学会，日本ソフトウェア科学会各会員．

高田 広章 （正会員）

名古屋大学大学院情報科学研究科情報システム学専攻教授．1988 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同専攻助手，豊橋技術科学大学情報工学系助教授等を経て，2003 年より現職．2006 年より大学院情報科学研究科附属組込みシステム研究センター長を兼務．リアルタイム OS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．オープンソースの ITRON 仕様 OS 等を開発する TOPPERS プロジェクトを主宰．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．