高信頼性システム向け
リアルタイムOSのテスト手法

鴫 原　一 人

高信頼性システム向けリアルタイム OS のテスト手法

要旨

IoT 社会に向かう中で，組込みシステムの重要性が一層高まっている．また，組込みシステムの高機能化・大規模化に伴い，マルチプロセッサやメモリ保護機能も必要となっている．一方で複雑化した組込みソフトウェアの不具合が問題視されている．組込みソフトウェア開発においては，プログラムの再利用性や，リアルタイム性を確保するために，組込み向けリアルタイム OS を採用して開発することが一般的である．つまりリアルタイム OS は，組込みシステムの品質を支える重要なソフトウェアであると言える．また，近年，自動車に搭載される電子制御ユニットにおいても高機能化・大規模化が進み，車載ソフトウェアの開発コストが急増している．1 つの電子制御ユニットで実現する機能の数も増えており，複数のアプリケーションを 1 つのプロセッサ上に混在させたいニーズが増え，アプリケーション間での故障や不具合の伝搬を防止するために，メモリ保護機能の必要性も高まっている．車載ソフトウェア分野においては，開発の効率化，開発コストの削減を目的として，AUTOSAR という標準化されたソフトウェアプラットフォームの導入が，欧州を中心に進んでいる．AUTOSAR においても，重要な役割を持つモジュールの 1 つとして，メモリ保護機能を提供するリアルタイム OS が規定されており，やはりリアルタイム OS が電子制御ユニットの品質を支える重要なソフトウェアとなっている．

このように，リアルタイム OS の品質確保は重要な課題であるが，リアルタイム OS に特化したテスト手法やツールなどで，一般的に知られているものは存在しない．また，マルチプロセッサやメモリ保護に対応したリアルタイム OS は，歴史が浅く，テスト手法が確立されていないという問題がある．実際に，リアルタイム OS に対してテスト分析を行った結果，リアルタイム OS 特有の性質から，スクラッチでテストプログラムを開発・保守することが困難であることや，マルチプロセッサ環境やメモリ保護機能におけるテストの実現に課題があることが明確となった．リアルタイム OS のテストを各企業で実施する場合，同様の課題が発生し，各企業で課題解決に取り組むと重複投資となってしまう恐れがある．また，オープンソースのリアルタイム OS を実製品に利用する場合，製品の品質を保証するために，利用者はアプリケーションのテストだけではなく，利用したリアルタイム OS のテストも実施する必要があり，オープンソースのメリットを損なっていると言える．

本研究では，これらの問題を解決するために，リアルタイム OS に対するテストの課題や規模を明確にし，リアルタイム OS をテストするための手法やツールを考案するため，以下の 4 つの研究を実施した．

1 つ目の研究として，μITRON ベースのリアルタイム OS を対象に，テストプログラムを直接コーディングするのではなく，テストプログラムより抽象度の高いテストシナリオから，テストプログラムを生成するツールを開発した．まず，テストシナリオを形式的に表現するために，TESRY(TEst Scenario for Rtos by Yaml) 記法と呼ぶ記法を考案した．次に，TESRY 記法で記述されたテストシナリオから，テストプログラムを生成するツールとして，TTG(Toppers Test Generator) を開発した．テストプログラムを生成することで，テストスイート開発を効率化し，テストスイートの拡張性や保守性が向上したことを確認した．また，TTG で生成したテストプログラムを実行することで，仕様と実装に相違がある不具合や，ターゲット依存部の実装不具合を検出した．

2 つ目の研究として，μITRON ベースのマルチプロセッサ向け RTOS である FMP カーネルに対して，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ーを策定し，プロセッサ間同期制御ライブラリ，拡張 ISS を開発し，マルチプロセッサ特有の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，かつテスト対象としたソースコードに対する 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．

3 つ目の研究として，AUTOSAR OS を対象に API テストを実施するため，TTG をベースとして，AUTOSAR OS 向けにテストプログラム生成ツール AKTG (Automotive Kernel Test Generator) を開発した．AUTOSAR OS 仕様の RTOS に対するテストでは，テストプログラムとは別に，RTOS のコンフィギュレーション情報に応じて，XML ファイルとコードやデータの配置を決定するためのヘッダファイルも用意する必要があるため，入力するテストシナリオのデータに応じて，AKTG でこれらのファイルも生成可能とし，テストに必要なファイルの開発コストを大幅に削減した．テスト実行時間が膨大になる問題に対しては，連続して実行することが困難なテストケースを強制的にマージして実行する手法を考案した．実際に，商用 AUTOSAR OS である JulinarOs に対して本テスト手法を適用し，テ

ストプログラム開発コストおよび，テスト実行時間の大幅な削減を実現した．

4 つ目の研究として，メモリ保護機能に対応した AUTOSAR OS である ATK2 に対して API テスト，およびアクセス保護テストを実施した．AUTOSAR OS 仕様は，μITRON 仕様と比べて，OS オブジェクトに設定可能なパラメータが多く，テストケース数が膨大となる問題に対して，テストケースの組み合わせツールを用いたテストケース生成を行った．アクセス保護テストでは，アクセス保護機能に対するテストパターンを導出し，テストのポータビリティ向上のため，プロセッサに依存する部分の明確化を行った．これにより，ATK2 の不具合を 19 件検出し，ATK2 の品質向上に貢献した．

以上の 4 つの研究成果により，提案したテスト手法，ツールが RTOS の品質向上に有効であることを示した．また，開発したツール，テストスイートの保守性，再利用性，拡張性の高さを示した．

目 次

第 1 章　序論 **1**
1.1　研究の背景 1
1.2　論文の概要 2
1.3　論文の構成 5

第 2 章　RTOS とそのテストにおける課題 **7**
2.1　本論文で対象とする RTOS 7
2.1.1　RTOS の主な機能 7
2.1.2　μITRON 9
2.1.3　AUTOSAR OS 13
2.2　RTOS に対するテスト 16
2.2.1　テストの全体像 16
2.2.2　RTOS に対するテストの課題 19
2.3　本研究のアプローチ 22
2.3.1　μITRON ベースの RTOS 向けテストプログラム生成ツール . 22
2.3.2　μITRON ベースのマルチプロセッサ向け RTOS のテスト . . 22
2.3.3　テストプログラム生成ツールの AUTOSAR OS への適用 . . 22
2.3.4　AUTOSAR OS 仕様ベースの RTOS に対するテスト 23

第 3 章　μITRON ベースの RTOS 向けテストプログラム生成ツール **25**
3.1　概要 25
3.2　API テスト 26
3.2.1　RTOS に対するテスト 26
3.2.2　API テストの概要 27
3.2.3　テストケース 27
3.2.4　テストシナリオ 29
3.2.5　カバレッジの確認 30
3.3　テストシナリオ記法とテストプログラム生成ツールの要件 30

3.3.1 要件 31
3.3.2 テストシナリオからのテストプログラム生成 31
3.3.3 統合仕様書に準拠した RTOS のサポート 32
3.3.4 任意数のテストシナリオの統合 32
3.3.5 テストシナリオの正当性チェック 33
3.3.6 リソース共有によるメモリ使用量の削減 33
3.4 TESRY 記法 33
3.4.1 構成 33
3.4.2 複雑なテストシナリオへの対応 36
3.5 TTG 37
3.5.1 要件に対する TTG の設計 38
3.5.2 TTG により生成するテストプログラム 41
3.5.3 TTG の実装 44
3.6 評価 47
3.6.1 TESRY 記法によるテストシナリオの記述性 47
3.6.2 開発の効率化 48
3.6.3 他のプロファイルへの拡張性 48
3.6.4 多様なターゲットシステムへの対応 48
3.6.5 スタック共有の効果 49
3.6.6 テストプログラムの正当性 50
3.6.7 検出した不具合 50
3.7 関連研究 50
3.8 まとめ 52

第 4 章 μITRON ベースのマルチプロセッサ向け RTOS のテスト 53
4.1 概要 53
4.2 テストの概要 55
4.2.1 外部仕様 (仕様書) を網羅するためのテスト 55
4.2.2 すべてのソースコードを実行するためのテスト 56
4.2.3 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 . 58
4.3 テストプロセス 59
4.3.1 テスト設計フェーズ 59
4.3.2 テストプログラム作成フェーズ 62

4.3.3 テスト実施フェーズ 64
4.4 マルチプロセッサ向けRTOSに対するテスト手法 64
4.4.1 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ー 64
4.4.2 TTGのマルチプロセッサ拡張 65
4.4.3 拡張ISS 70
4.5 テスト実施結果 73
4.5.1 APIテスト 74
4.5.2 実行順序依存分岐テスト 74
4.6 検出した不具合の分析 75
4.6.1 APIテストで検出した不具合 76
4.6.2 実行順序依存分岐テストで検出した不具合 78
4.7 得られた知見と他のRTOSへの適用 80
4.7.1 得られた知見 82
4.7.2 異なる仕様のRTOSへの適用 84
4.7.3 他のタイプのマルチプロセッサ向けRTOSへの適用 85
4.7.4 今後の課題 85
4.8 関連研究 86
4.9 まとめ 86

第5章 テストプログラム生成ツールのAUTOSAR OSへの適用 87
5.1 概要 87
5.2 AUTOSAR仕様 88
5.2.1 OS 88
5.2.2 メモリ保護機能 89
5.2.3 メモリマッピング 91
5.2.4 標準化の対象外ファイル 91
5.3 AUTOSAR OSに対するテストの課題 92
5.3.1 基本的なテストシナリオ 93
5.3.2 膨大なテストケース数 93
5.3.3 開発コスト 95
5.3.4 テスト実行時間 96
5.3.5 メモリ保護機能による制約事項 97

5.4 関連研究 98
5.5 テストプログラム生成ツール 99
5.5.1 TESRY 記法 99
5.5.2 AKTG 101
5.5.3 PictMaster による TESRY データの生成 101
5.5.4 強制マージ 102
5.5.5 テストライブラリ 103
5.6 評価 105
5.6.1 開発コストの削減 106
5.6.2 テスト実行時間の削減 106
5.6.3 有効性 106
5.7 まとめ 108

第 6 章 AUTOSAR OS 仕様ベースの RTOS に対するテスト 109
6.1 概要 109
6.2 ATK2 のメモリ保護機能 110
6.2.1 A-OS 仕様と ATK2 実装の差異 112
6.2.2 API 保護の実現方法 113
6.2.3 アクセス保護の仕様設計 114
6.2.4 アクセス保護の実現方法 115
6.3 TTSP の概要 118
6.3.1 API テスト 118
6.3.2 TESRY 記法 119
6.3.3 TTG 119
6.4 API テストの A-OS 仕様への適合とアクセス保護テスト 121
6.4.1 API テスト 121
6.4.2 アクセス保護テスト 128
6.5 評価 135
6.5.1 API テスト 136
6.5.2 アクセス保護テスト 139
6.5.3 A-OS 仕様への対応 140
6.6 関連研究 140
6.7 まとめ 141

第 7 章　結論 **143**
7.1　まとめ 143
7.2　今後の課題 144

謝辞 **145**

研究業績 **152**

図 目 次

1.1 本論文の各章の関係 5

2.1 goto 文の使用例 11
2.2 FMP カーネルの概要 12
2.3 AUTOSAR のアーキテクチャ 14

3.1 act_tsk のテストケースの例 28
3.2 act_tsk のテストシナリオの例 29
3.3 API テストにおけるツールの位置づけ 31
3.4 TESRY 記法の例 34
3.5 タスクの状態定義 34
3.6 時間指定の例 35
3.7 処理と後状態の複数定義の例 35
3.8 TTG の構成と処理フロー 45

4.1 共通部ロック関数の例 58
4.2 テストケースの例 60
4.3 テストシナリオの例 60
4.4 TESRY 記法の例 60
4.5 テストプロセスのフロー 61
4.6 ASP カーネル向けテストプログラムの例 66
4.7 FMP カーネル向けテストプログラムの例 66
4.8 同期制御を含むテストプログラムの処理フロー 68
4.9 ロック取得失敗パスが実行される例 72
4.10 ISS 拡張によるロック取得失敗パスの実行フロー 72
4.11 不具合発生のフロー 79
4.12 不具合が検出された処理の疑似コード 81

5.1 OS ソースコードの実装例 91

5.2 Os_MemMap.h の例 92
5.3 テストシナリオの例 92
5.4 アプローチの全体像 99
5.5 A-OS 仕様に対応した TESRY 記法の例 100
5.6 OS シャットダウンを伴う TESRY データの例 102
5.7 実行回数の実装イメージ 103
5.8 非信頼 OSAP から状態確認 API を実行する実装イメージ 104
5.9 TESRY データの行数と C ソースコード+arxml の行数の比較結果 . 107
5.10 強制マージ使用時と個別実行時のテスト実行時間の比較結果 107

6.1 ATK2 によるメモリ保護設定の例 111
6.2 アクセス保護の実現方法 117
6.3 テストケースの例 (図 3.1 の再掲) 120
6.4 テストシナリオの例 (図 3.2 の再掲) 120
6.5 TESRY 記法の例 (図 3.4 の再掲) 120
6.6 A-OS 仕様に対応した TESRY 記法の例 (図 5.5 の再掲) 123
6.7 PictMaster ファイルの例 127
6.8 PictMaster を用いた処理フロー (図 5.4 の再掲) 128
6.9 アクセス保護テストのテストシナリオと TESRY 記法の例 130
6.10 処理単位の切り替えパターン 131
6.11 メモリアクセスパターン (抜粋) 133
6.12 アクセス保護テスト設計のデータフロー 136

表 目 次

2.1 タスクのコンフィギュレーション項目の差異 15
2.2 テストカテゴリー一覧 18

3.1 メモリ使用量 49

4.1 同期パターンと同期メカニズム 69
4.2 テスト実施結果一覧 73
4.3 ターゲットシステム一覧 75
4.4 API テストにおける不具合の種類と件数 76

6.1 メモリ領域に対するアクセス保護属性 116
6.2 テスト実施結果一覧 137

第1章　序論

1.1　研究の背景

IoT 社会に向かう中で，組込みシステムの重要性が一層高まっている．一方で複雑化した組込みソフトウェアの不具合が問題視されている [1]．組込みソフトウェア開発においては，プログラムの再利用性や，リアルタイム性を確保するために，組込み向けリアルタイム OS(以下，RTOS) を採用して開発することが一般的である．つまり RTOS は，組込みシステムの品質を支える重要なソフトウェアであると言える．組込みシステムの規模が比較的小さかった時代は，RTOS に要求される機能もシンプルであったため，組込みシステムを開発する企業が，RTOS を自社開発するケースが多かった．しかしながら，近年は他社製やオープンソースの RTOS を利用する企業が増加している．これは，近年の組込みシステムの高機能化・大規模化に伴い，マルチプロセッサやメモリ保護に対応した RTOS が必要となり，自社内での開発が，困難になっているからである．

また，近年，自動車に搭載される電子制御ユニット (以下，ECU) においても高機能化・大規模化が進み，車載ソフトウェアの開発コストが急増している．1 つの ECU で実現する機能の数も増えており，複数のアプリケーションを 1 つのプロセッサ上に混在させたいニーズが増え，アプリケーション間での故障や不具合の伝搬を防止するために，メモリ保護機能の必要性も高まっている．車載ソフトウェア分野においては，開発の効率化，開発コストの削減を目的として，AUTOSAR[2] という標準化されたソフトウェアプラットフォーム (以下，SPF) の導入が，欧州を中心に進んでいる．AUTOSAR においても，重要な役割を持つモジュールの 1 つとして，メモリ保護機能を提供する RTOS が規定されており，やはり RTOS が ECU の品質を支える重要なソフトウェアとなっている．

このように，RTOS の品質確保は重要な課題であるが，RTOS に特化したテスト手法やツールなどで，一般的に知られているものは存在しない．また，マルチプロセッサやメモリ保護に対応した RTOS は，歴史が浅く，テスト手法が確立されていないという問題がある．実際に，RTOS に対してテスト分析を行った結果，RTOS

特有の性質から，スクラッチ[1]でテストプログラムを開発・保守することが困難であることや，マルチプロセッサ環境やメモリ保護機能におけるテストの実現に課題があることが明確となった．RTOS のテストを各企業で実施する場合，同様の課題が発生し，各企業で課題解決に取り組むと重複投資となってしまう恐れがある．また，オープンソースの RTOS を実製品に利用する場合，製品の品質を保証するために，利用者はアプリケーションのテストだけではなく，利用した RTOS のテストも実施する必要があり，オープンソースのメリットを損なっていると言える．これらの問題を解決するために，RTOS に対するテストの課題や規模を明確にし，RTOS をテストするためのツールや手法を考案する必要がある．

過去にも，RTOS のテスト手法に関する様々な研究が実施されているが，タスクの振る舞いや割込み処理に限定していたり，テスト項目や確認事項を提示することに留まっており，RTOS に対するテストの分析や，課題を明確にした事例はなく，本研究で対象とした RTOS 向けのツールや手法は提案されていない．特に，マルチプロセッサ向けの RTOS やメモリ保護機能に対応した RTOS に対するテストに言及した研究は存在しない．

1.2　論文の概要

本研究では，前節で延べた課題を解決するため，RTOS に対するテスト手法について，4 つの研究を実施した．

1 つ目の研究として，RTOS に対するテストケース，テストシナリオを分析し，テストプログラム開発における課題を解決した．RTOS では，OS オブジェクトの属性や状態の組み合わせが多数存在するため，1 つの API を呼び出すテストを行う場合でも，API を呼び出す際の状態・条件の組み合わせが膨大となり，テストプログラムの開発コストが大きくなる．また，タスクの起動順序や期待結果の確認方法の違いなど，同じテストシナリオであっても，テストプログラムの実装方法は多様に存在するため，異なる開発者がテストプログラムを開発すると，実装方法を統一することが困難であり，レビューや修正作業の負担が大きくなることが予想される．さらに，RTOS を実行する組込みシステム向けのプロセッサは，メーカーや品番によって ROM/RAM サイズが異なるため，1 つのテストプログラムで実行可能なテストケース数を，テストプログラム開発段階では決定することができない．そこで，μITRON ベースの RTOS を対象に，テストプログラムを直接コーディン

[1]本論文では，手作業でソースコードを記述することを意味する．

グするのではなく，テストプログラムより抽象度の高いテストシナリオから，テストプログラムを生成するツールを開発することにした．まず，テストシナリオを形式的に表現するために，TESRY(TEst Scenario for Rtos by Yaml) 記法と呼ぶ記法を考案した．次に，TESRY 記法で記述されたテストシナリオから，テストプログラムを生成するツールとして，TTG(Toppers Test Generator) を開発した．これにより，開発する対象をテストプログラムではなく，TESRY 記法で記述したデータ (以下，TESRY データ) とすることができ，テストプログラム開発工数を大幅に削減することができた．ツールで生成することで，テストプログラムの構造やテストの実現方法も一意とすることができ，テストプログラムのレビュー容易性を向上した．また，TTG へ入力する TESRY データの数を増減させることで，生成されるテストプログラムの ROM/RAM サイズを制御することが可能となり，テスト対象とするプロセッサに応じたテストプログラムの生成を可能とした．

2 つ目の研究では，μITRON ベースのマルチプロセッサ向け RTOS を対象とし，マルチプロセッサ環境におけるテストの課題を解決した．マルチプロセッサでは，プロセッサ毎に処理が独立して実行されるため，あるプロセッサが特定の状態で，他のプロセッサから API を呼び出すようなテストを行う場合，呼び出される側のプロセッサが確実に特定の状態になっているタイミングで API を呼び出す必要がある．つまり，プロセッサ間で API 呼び出しのタイミングを待ち合わせるような同期制御を行うテストプログラムを実装する必要がある．また，マルチプロセッサ向け RTOS のソースコード内には，各プロセッサの実行順序に依存してパスが定まる分岐が存在するため，テストプログラム側でタイミングを調整するだけでは，RTOS のソースコード内のすべてのパスを決定的に実行することは困難である．そこで，まず，対象とした RTOS において必要となるプロセッサ間の同期制御を明確にし，TTG に対して，プロセッサ間同期制御機能の拡張を行った．これにより，プロセッサ間の処理を同期するテストプログラムの生成を実現し，意図したタイミングでプロセッサを跨いで API を呼び出すテストを可能とした．そして，RTOS を実行可能な命令セットシミュレータを拡張することで，各プロセッサを意図したプログラムカウンタで停止・再開する制御を可能とした．これにより，RTOS のソースコードに含まれる，各プロセッサの実行順序に依存してパスが定まる分岐に対しても，決定的にパスを通すテストを実現した．

3 つ目の研究では，メモリプロテクションユニット (以下，MPU) を用いたメモリ保護機能を有する AUTOSAR OS 仕様をベースとした RTOS に対する API テストの課題を解決した．AUTOSAR OS 仕様の RTOS に対するテストでは，テスト

プログラムとは別に，RTOS のコンフィギュレーション情報に応じて，XML ファイルとコードやデータの配置を決定するためのヘッダファイルも用意する必要がある．また，メモリ保護機能に対応した RTOS の場合，ユーザモードで動作するタスクには，メモリ領域やレジスタへのアクセス制限があるので，テストプログラム中に，OS 内部の状態を確認するコードや，レジスタを操作するコードが存在した場合，そのままでは実行できない問題がある．さらに，AUTOSAR OS 仕様では，OS の仕様上，1 つのテストプログラムで実現できないテストケースが存在するため，多数のテストプログラムを，1 件ずつ個別にビルド・ロード・実行する必要があり，テスト実行時間が膨大になってしまう問題もある．これらの課題に対し，AUTOSAR OS 仕様に対応した TESRY 記法を考案し，TTG をベースとして，AUTOSAR OS 仕様に対応したテストプログラム生成ツール AKTG(Automotive Kernel Test Generator) を開発した．テストプログラムとは別に必要となる XML ファイルとヘッダファイルは，入力する TESRY データに応じて，AKTG で生成可能とし，テストに必要なファイルの開発コストを大幅に削減した．ユーザモードで動作するタスクに対する制約に対しては，意図的に CPU 例外を発生させることで特権モードへ切り替え，意図した処理を実行するためのテストライブラリを開発することで解決した．テスト実行時間が膨大になる問題に対しては，連続して実行することが困難なテストケースの実行を完了した後に，ソフトウェアリセットを発生させ，次のテストケースを実行する手法を考案した．これにより，OS の仕様上，1 つのテストプログラムで実現できない複数のテストケースを，1 つのテストプログラムで実行することを可能とし，テスト実行時間を大幅に削減した．

4 つ目の研究として，メモリ保護機能に対応した AUTOSAR OS である TOPPERS/ATK2 に対して API テスト，およびアクセス保護テストを実施した．AUTOSAR OS 仕様は，μITRON 仕様と比べて，OS オブジェクトに設定できるパラメータが多いことに加えて，メモリ保護機能を考慮したテストケースも必要であるので，API テストのテストケースはさらに膨大な数となる．また，メモリ保護の対象となるメモリ領域の種別や，メモリ保護設定切り替え方法など，メモリ領域へアクセスする条件は膨大であり，アクセス保護テストを効率的に実施するテスト手法が求められる．まず，API テストのテストケース数が膨大となる問題に対しては，テストケースの組み合わせツールを拡張することで，TESRY データを手作業で作成するのではなく，ツールで生成することで解決した．これにより，組み合わせツールで生成した TESRY データを，AKTG へ入力することで，膨大な数のテストケースを実行するテストプログラムを生成することを可能とした．ア

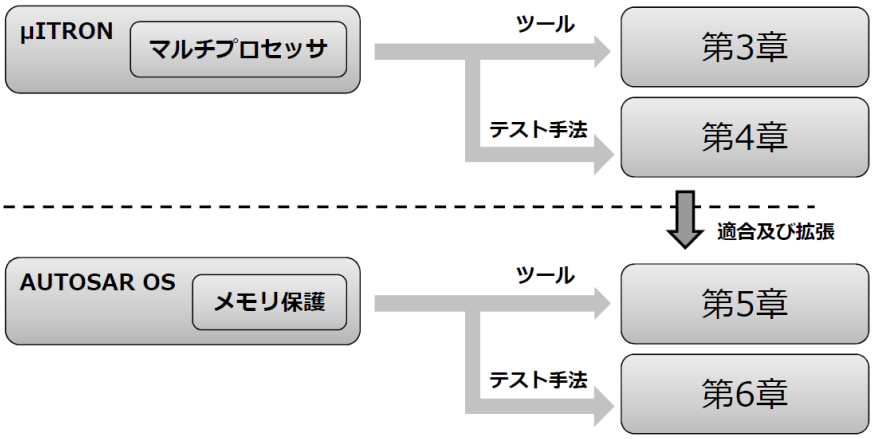


図 1.1: 本論文の各章の関係

クセス保護テストに対しては，アクセス保護機能に対するテストパターンを導出し，拡張した組み合わせツールと AKTG を用いることで，すべてのメモリ領域のテストパターンを効率的にテストする手法を考案した．

以上の 4 つの研究成果により，RTOS に対するテストの課題や規模が明確になり，提案したツールやテスト手法により，マルチプロセッサ環境やメモリ保護機能に対するテストが実現可能であることを示す．

1.3　論文の構成

本論文の構成は，次の通りである．

まず第 2 章で，本研究で対象とする RTOS の仕様，および実装について述べ，RTOS に対するテストの全体像，課題について説明する．第 3 章と第 4 章では，μITRON 仕様ベースの RTOS を対象とした研究について述べる．第 3 章では，本研究で提案する RTOS に対するテストプログラム生成ツールについて述べ，第 4 章では，マルチプロセッサ向け RTOS を対象としたテスト手法について述べる．次に，第 5 章と第 6 章では，AUTOSAR OS 仕様ベースの RTOS を対象とした研究について述べる．第 5 章では，テストプログラム生成ツールの AUTOSAR OS 仕様への適合について述べ，第 6 章では，AUTOSAR OS のメモリ保護機能に対するテスト手法について述べる．最後に，第 7 章で結論を述べる．

第 3 章から第 6 章までの関係について，図 1.1 に示す．第 3 章と第 4 章，および第 5 章と第 6 章は，ともに同じ仕様の RTOS を対象としたツールと，そのツール

を用いたテスト手法について述べているため，一部重複した記述があることに留意されたい．

第2章　RTOSとそのテストにおける課題

2.1　本論文で対象とするRTOS

RTOSは，リアルタイムシステムを構築する際に用いるOSである．リアルタイムシステムとは，処理の正しさが，出力される結果の正しさに加えて，結果を出す時間にも依存するようなシステムであり，主に，機械・機器を制御する組込みシステムが該当する．RTOSの仕様・実装は世界に数多く存在するが，本論文では，μ ITRON[3] およびAUTOSAR OS(以下，A-OS)[4] の2つの仕様をベースとしたRTOSを対象とする．

2.1.1　RTOSの主な機能

本節では，RTOSが提供する主な機能について説明する．

タスク管理

タスクとは，並行実行するプログラムの単位である．1つのタスクで実装されたプログラムは逐次的に実行され，異なるタスクで実装されたプログラムは並行して実行される．RTOSは，複数のタスクを擬似並列に実行するための機能 (マルチタスク機能) を有しており，シングルプロセッサ上であっても，複数のタスクが同時に実行しているかのように見せることが可能となる．これにより，独立した処理の流れを独立したタスクに実装することが可能となり，プログラムの保守性，再利用性の向上や，外部で開発されたソフトウェア部品の導入を容易にすることが可能となる．

どのタイミングでどのタスクを実行するかを決定することをスケジューリングと呼び，スケジューリングの方法をスケジューリングアルゴリズムと呼ぶ．多くのRTOSでは，スケジューリングアルゴリズムに，プリエンプティブな優先度ベー

ススケジューリングが採用されている．優先度ベーススケジューリングでは，最も優先度の高いタスクが実行され，優先度の高いタスクが実行できなくなるまで，優先度の低いタスクは実行されない．プリエンプティブなスケジューリングでは，優先度の高いタスクが実行可能になると，優先度の低いタスクが実行途中であっても，タスクの切り替えが発生する．

μITRON 仕様では，プリエンプティブな優先度ベーススケジューリングが採用されている．A-OS 仕様は，優先度ベーススケジューリングであるが，プリエンプティブかプリエンプティブでない (ノンプリエンプティブ) かは，タスク毎に設定が可能である．ノンプリエンプティブなスケジューリングでは，優先度の高いタスクが実行可能になっても，優先度の低いタスクが実行を継続し，タスクの切り替えが発生しない．

割込み管理

一般的なプロセッサでは，処理の実行中であっても，主にハードウェアなどの外的要因により処理が中断され，例外処理が実行されることがある．この機構は，文献によって割込みや CPU 例外と呼ばれるが，明確な定義が存在しない．統合仕様書では，以下のように定義されているため，本論文においても，統合仕様書の割込みと CPU 例外の定義を使用する．

- プロセッサが実行中の処理とは独立に発生するイベントによって起動される例外処理のことを割込みと呼ぶ．
- 割込みに対し，プロセッサが実行中の処理に依存して起動される例外処理を CPU 例外と呼ぶ．

本節では，RTOS における割込み管理について説明する．割込みが発生する仕組みや，割込みを制御するためのプログラムは，プロセッサによって多種多様であるため，RTOS で割込みを管理するための処理を抽象化することが一般的である．これにより，ユーザは，割込み発生時の処理だけを実装すればよい．

μITRON 仕様，A-OS 仕様のどちらも，割込み制御を抽象化する方法については実装依存としているが，プロセッサに依存せず各割込みに応じてユーザが実装可能なプログラムの単位は，割込みサービスルーチン (ISR) である．割込み処理は，タスクより高い優先度で実行されるため，タスク実行中に割込みが発生した場合，タスクの処理を中断して，割込みに対応する ISR が実行される．

時間管理

RTOS では，周期的にタスクを起動したり，指定した時間を経過した後にタスクを起動することを可能とするため，何らかの方法で時間を管理する．

μITRON 仕様では，RTOS 内にシステム時刻を持つことで，時間に依存した処理を行う．具体的には，システム時刻を使用して，指定した周期で繰り返し処理を実行する周期ハンドラや，指定した時刻に処理を実行するアラームハンドラなどを提供する．周期ハンドラやアラームハンドラから，タスクを起動することで，時間に依存したタスクの起動を制御することができる．また，実行中のタスクを指定した時間だけ待ち状態にする機能や，同期・通信機能におけるタイムアウトを行う機能においても，システム時刻を使用する．

一方，A-OS 仕様には，システム時刻という機能は存在しない．その代わり，何らかの事象をカウントし，接続された他の OS オブジェクトに対して処理タイミングの通知を行うためのカウンタという機能が用意されている．カウンタはティックという単位で事象をカウントするが，カウントのタイミングは実時間と関係する必要はなく，ティックを進める方法はユーザが制御することができる．カウンタには，アラームとスケジュールテーブルという OS オブジェクトを接続することができ，接続されたカウンタにおける指定したタイミングで，タスクの起動などの指定した処理を実行する．例えば，タスク起動を行うアラームに接続したカウンタを用いて，ハードウェアのタイマなどでティックを進めることで，周期的にタスクを起動することができる．

2.1.2　μITRON

μITRON は，国内外で広く使用される RTOS 仕様である．TOPPERS プロジェクト [5] では，μITRON 仕様をベースとして，TOPPERS 新世代カーネル統合仕様書 [6](以下，統合仕様書) を策定，公開し，この統合仕様書に準拠した RTOS の実装を一般公開している．統合仕様書は，信頼性と安全性とソフトウェアポータビリティを向上させるために，μITRON 仕様に拡張と改良が加えられている．

なお，TOPPERS プロジェクトでは，OS の中心になるモジュールという意味で，汎用の OS と区別するため，RTOS をカーネルと呼んでいる．本論文においては，RTOS とカーネルは同義である．

TOPPERS/ASP カーネル

TOPPERS/ASP カーネル (以下，ASP カーネル)[7] は，TOPPERS 新世代カーネルの基盤 (出発点) となる RTOS であり，統合仕様書に準拠している．ASP は，Advanced Standard Profile の略であり，統合仕様書準拠のスタンダードプロファイルである．プロファイルとは，同じ仕様に準拠した，使用目的の異なる RTOS の種別を意味する．具体的には，TOPPERS 新世代カーネルには，シングルプロセッサ対応 RTOS の機能セット，マルチプロセッサ対応 RTOS の機能セット，メモリ保護対応 RTOS の機能セットなどのプロファイルが存在する．

ASP カーネルに実装される API は，カーネル動作中に呼び出し可能なもの (以下，API) が 103 個，予めシステムコンフィギュレーションファイルに記述して，OS オブジェクト生成に用いるもの (以下，静的 API) が 17 個の，計 120 個である．OS オブジェクトとは，タスクやセマフォといったカーネルが管理するソフトウェア資源を指す．

ASP カーネルでは，API の実装を含むカーネルとアプリケーションを 1 つの実行モジュールとしてリンクする [1]．そのため，アプリケーションからカーネルの API を呼び出す際は，関数呼び出しにより呼び出すことが可能である．これを実現するためには，アプリケーションとカーネルはどちらも特権モードで実行する必要があるが，ASP カーネルはメモリ保護機能を持たないカーネルであるので，特権モードで実行して差し支えない．

ASP カーネルでは，主にエラーチェック処理において goto 文を使用する．タスクを起動する API である act_tsk の簡略化したソースコードを図 2.1 に示す．4〜7 行目の if 文で引数などのチェックを行い，異常があれば，エラーコードを戻り値に設定し，goto 文で関数の末尾へジャンプする．実際のコードでは，4〜7 行目の if 文全体がマクロ化され，複数のエラーチェックを，各 API の先頭で実行するように実装されている．一般に goto 文の使用は，コーディングルールで禁止とされることが多いが，各 API で必要なエラーチェック処理をパターン化し，ソースコードの簡潔さを保つことで可読性を向上させるとともに，記述ミスを減らすことを目的として，TOPPERS プロジェクトでは goto 文を使用している．

なお，1 リンクモデルや goto 文の使用といった RTOS の特徴は，本論文で対象とする他の RTOS も同様である．

[1] 1 リンクモデルと呼ばれる

```
 1: ER
 2: act_tsk(ID tskid)
 3: {
 4:     if (<エラーチェック>) {
 5:         ercd = <対象のエラーコード>;
 6:         goto error_exit;
 7:     }
 8:
 9:     t_lock_cpu(); /* 割込み禁止開始 */
10:
11:     /* act_tsk のメイン処理 */
12:
13:     t_unlock_cpu(); /* 割込み禁止終了 */
14:
15:   error_exit:
16:     return(ercd);
17: }
```

図 2.1: goto 文の使用例

TOPPERS/FMP カーネル

TOPPERS/FMP カーネル (以下，FMP カーネル)[8] は，統合仕様書に準拠したマルチプロセッサ向け RTOS である．本節では，まずマルチプロセッサ向け OS の種類について述べ，FMP カーネルについて述べる．

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跨いで OS の提供する API を呼び出すことが可能である．例えば，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できる．

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毎に持つため，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よく使用することができない問題がある．AMP 型の RTOS の例として，RTEMS[9]，AMP T-Kernel[10] などがある．

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

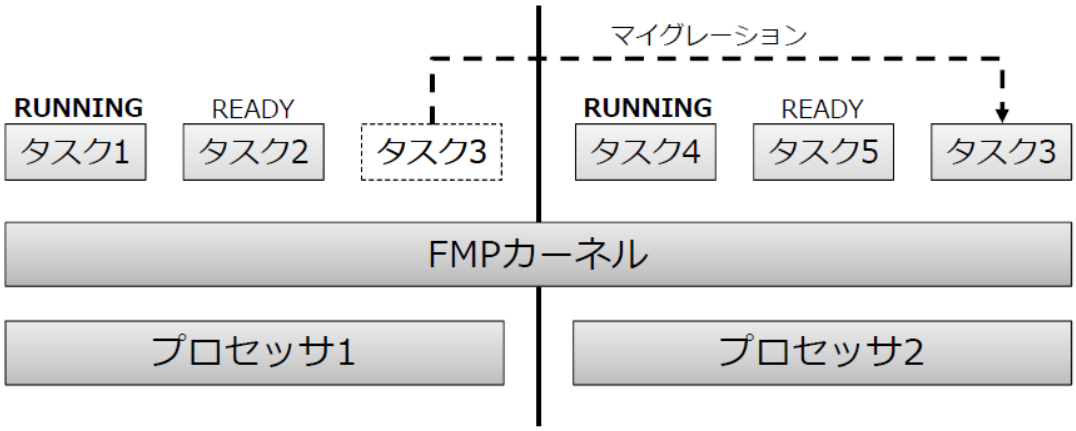


図 2.2: FMP カーネルの概要

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[12] などがある．

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，両立することは困難であるため，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毎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．その上で，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毎に性質が異なり，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．そこで，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．FMP カーネルの概要を図 2.2 に示す．

FMP は，Flexible Multiprocessor Profile の略であり，ASP カーネルを拡張して開発されている．ASP カーネルに対する拡張点は次の通りである．

- プロセッサを跨いだ 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．

また，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毎に管理する 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毎に持つ 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．

FMP カーネルに実装される API は，カーネル動作中に呼び出し可能な API が 120 個，予めシステムコンフィギュレーションファイルに記述して，OS オブジェクト生成に用いる静的 API が 18 個の，計 138 個である．

2.1.3　AUTOSAR OS

近年，車載ソフトウェア分野において，開発の効率化，開発コストの削減を目的として，AUTOSAR[2] という標準化された SPF の導入が，欧州を中心に進んでい

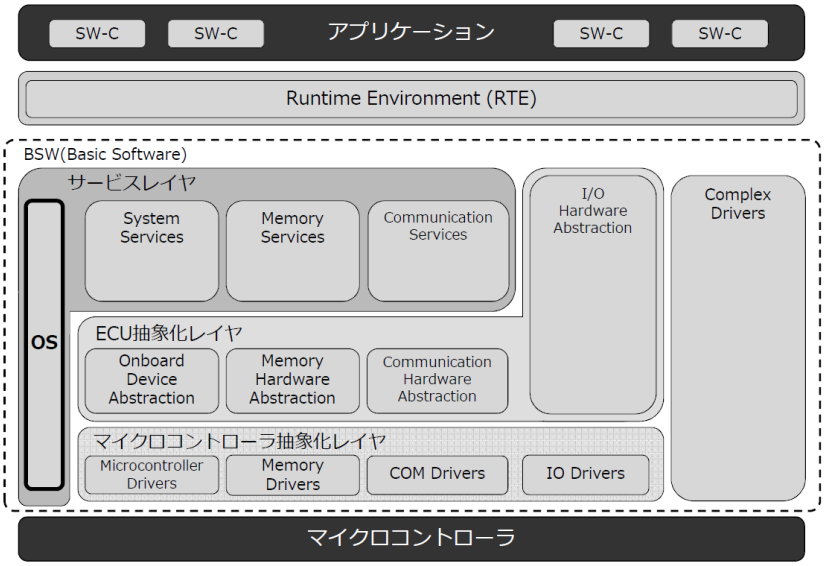


図 2.3: AUTOSAR のアーキテクチャ

る．AUTOSAR では，仕様のみ標準化・公開されており，実装は各ソフトウェアベンダが行う．AUTOSAR 仕様では，サービスレイヤ，ECU 抽象化レイヤ，マイクロコントローラ抽象化レイヤという 3 つの階層構造により，アプリケーションとハードウェアを分離し，ハードウェアに依存しないアプリケーションの開発を実現する．AUTOSAR のアーキテクチャ図 2.3 に示す．

これらのモジュールの中に，RTOS である A-OS がある．A-OS は，OSEK/VDX OS 仕様 [13] の上位互換として，OSEK/VDX OS 仕様との差分仕様のみ規定されている．A-OS は，優先度ベースのタスクスケジューリングを行う一般的な RTOS であり，タスクスケジューリング，割込み制御，リソース (上限優先度プロトコルによる排他制御)，カウンタ，アラームなどの OS オブジェクトが規定されている．さらに，メモリ保護やタイミング保護など，近年ニーズが高まっている技術に対応する機能も規定されている．A-OS には，以下の 4 つのスケーラビリティクラス (SC) が規定されている．

- SC1：OSEK の上位互換の基本セット
- SC2：SC1 ＋タイミング保護
- SC3：SC1 ＋メモリ保護
- SC4：SC1 ＋タイミング保護＋メモリ保護

表 2.1: タスクのコンフィギュレーション項目の差異

	μITRON	A-OS
最大起動要求回数	固定 (値は実装依存)	タスク毎に指定可能
スケジューリングポリシー	フルプリエンプティブ固定	タスク毎にプリエンプティブ, ノンプリエンプティブを選択可能
タスク種別	区別なし	タスク毎に基本タスク, 拡張タスク * を選択可能

*:基本タスクと拡張タスクでタスクの状態遷移が異なる

A-OS のタイミング保護機能には，機能安全を達成するための課題が多いことから，本研究では，SC1 およびメモリ保護に対応した SC3 を対象としている．

なお，μITRON 仕様と比較して，A-OS 仕様は，OS オブジェクトに対するコンフィギュレーション項目が多い．例えば，タスクのコンフィギュレーション項目では，表 2.1 に示すような差異があるため，タスクの種類や振る舞いが μITRON 仕様よりも多くなる．

TOPPERS/ATK2

TOPPERS プロジェクトでは，A-OS 仕様をベースとした RTOS である TOPPERS/ATK2 (以下，ATK2)[14] を公開している．A-OS 仕様では，MPU を用いてメモリ保護機能を実現するように規定されているが，MPU を使った具体的なコンフィギュレーションの方法は規定されていない．ATK2 では，MPU を用いたメモリ保護機能を有する μITRON 仕様ベースの RTOS である，TOPPERS/HRP2 カーネル (以下，HRP2 カーネル)[15] の設計 [16] を参考に，未規定の仕様を定め，メモリ保護機能が実装されている．

なお，ATK2 も ASP カーネル同様，1 リンクモデルであるが，メモリ保護機能を持っているため，ユーザモードからの API 呼び出し時は，trap 命令などによりユーザモードから例外を発生させ，特権モードへ切り替えた上で API を実行する．

JulinarOs

JulinarOs は，APTJ 株式会社 [17] が開発・販売する A-OS 仕様をベースとした商用の RTOS である．JulinarOs の基本設計は ATK2 を踏襲しているが，AUTOSAR

プラットフォーム全体としての共通仕様への対応や，実製品開発向けの品質保証対応として，ATK2 とは以下の点で異なっている．

- メモリマッピング仕様 [18] への対応
- メモリ保護のコンフィギュレーション仕様変更
- ユーザビリティ/テスタビリティを考慮した ATK2 独自拡張の廃止
- AUTOSAR 標準化の対象外コードの分離
- リンカスクリプトに関するコンフィギュレーション項目の追加

分離した AUTOSAR 標準化の対象外コードとは，具体的にはスタートアップルーチンと例外ベクタテーブルである．リンカスクリプトも標準化の対象外であるが，リンカスクリプトの内容は，メモリ保護機能を実現する上で重要な役割であるので，ATK2 同様，OS の範囲としている．

2.2　RTOS に対するテスト

2.2.1　テストの全体像

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し，高い品質が求められるため，RTOS に対するテストが重要である．本研究では，RTOS に対するテストの目的を，RTOS の品質向上，および仕様と実装のトレーサビリティ[2] 確認とした．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 [19, 20, 21]．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許容されないという背景があるためである．

このテストの目的を鑑み，テスト設計を行うための方針を以下とした．

- 方針 (1)：外部仕様 (仕様書) を可能な限り網羅する

[2] 各開発工程の成果物の全項目が，上位文書の要求と紐づけて管理され，追跡できることを指す．

- 方針 (2)：一度もテストしていないコードをなくす
- 方針 (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，2.1 節で述べた RTOS を対象に，テスト設計を行った結果，23 個のテストカテゴリーを抽出し，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 2.2 に示す．

RTOS のテストとしては，表 2.2 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，本研究では機能要件に対するテストのみを対象とした．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 制御のテストは不要である．

本研究では，表 2.2 を踏まえ，優先度を決めて，テストを順に実施していくことにした．優先度を決めるに当たり，ソフトウェアテスト本来の目的であることと，テストの内容が明確であることから，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．本研究では，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，アクセス保護テストを実施した．ブラックボックス API テストとホワイトボックス API テストを総称して，API テストと呼ぶ．

なお，RTOS をテスト対象とした本論文におけるブラックボックステスト，ホワイトボックステストは，以下のテスト法を意味する．

ブラックボックステスト

ソフトウェアの内部構造やソースコードを意識せずに，対象ソフトウェアの仕様書に基づいて，入力に対して正しい出力が得られることを確認する．本論文では，仕様書に記載された通りに API が振る舞うことや，仕様書に記載された属性の OS オブジェクトが使用できることを，タスクや割込みハンドラといったアプリケーションソフトウェアから確認を行うテストとなる．

ホワイトボックステスト

ソフトウェアの内部構造やソースコードに着目し，対象ソフトウェアが意図した設計通りに実装されていること，仕様上実行されないパスが存在しないことを確認する．本論文では，ブラックボックステストを実施するだけでは，すべての

表 2.2: テストカテゴリー一覧

(1) 仕様ベースのテスト
ブラックボックス API テスト *
処理単位テスト
SIL テスト
クラス関連テスト
OS 管理外割込みのテスト
OS 起動/終了時の同期テスト
アクセス保護テスト *
(2) 設計・ソースコードベースのテスト
ホワイトボックス API テスト *
実行順序依存分岐テスト *
ターゲットシステム依存テスト
スタートアップモジュールテスト
ジェネレータテスト
機能拡張・チューニングテスト
(3) エラー推測テスト
ロック区間テスト
割込み禁止区間テスト
タイマ割込みテスト
スピンロック中の割込みテスト
マイグレーションテスト
割込み出入口処理テスト
CPU 例外出入口処理テスト
アイドル処理テスト
デッドロック回避テスト
バリア同期テスト

*:本研究で実施したテスト

パスを実行することが困難であるタスクのスケジューリング管理や，タスク間の同期を行う OS オブジェクトのデータ操作といったソースコードが対象となる．対象のソースコードに対して，意図したパスを通すための入力条件を満たすテストケースを作成する．

2.2.2 RTOS に対するテストの課題

テストを実施するにあたり，テスト手法を検討した結果，以下の課題が存在することが判明した．本節では，各課題について説明する．

- 課題 (1)：RTOS の特殊性による課題
- 課題 (2)：マルチプロセッサ環境における課題
- 課題 (3)：A-OS 仕様に対する課題
- 課題 (4)：メモリ保護機能に対する課題

課題 (1)：RTOS の特殊性による課題

複雑なプログラム構造

RTOS は，タスクディスパッチや割込み制御のために，少なからずアセンブリ言語による実装が必要となる．また，処理速度の向上やクリティカルセクションの時間短縮のため，goto 文やポインタ操作が多数使用されることにより，一般的なソフトウェアと比較して，プログラム構造が複雑である．構造が複雑であるが故に，特定の条件の組み合わせでのみ発生する不具合も多く，直交表などによるテストケースの削減が難しい．

RTOS に対するテストプログラムにおいても，割込みの発生や API 呼び出しによって，プログラム上の記述とは異なる順序で処理が実行されるので，テストが正しく実行されたことを確認する方法にも工夫が必要である．

リグレッションテストの重要性

前述の通り，RTOS は構造が複雑であることから，ソースコードを 1 行修正したとしても，副作用が無いことをレビューのみで確認することは困難である．したがって，理想的には網羅的なテストを容易に実施できることが望ましい．また，RTOS はプロセッサに依存したコードが必ず存在するため，新しいプロセッサへの

ポーティングの度に，テスト実施が求められる．つまり，RTOS の不具合や修正が無くても，ポーティング時には，新しいプロセッサにてテストを実施する必要がある．

以上から，RTOS に対しては，リグレッションテスト[3]を容易に実施可能であることが重要である．

テストプログラム開発コストの増大

RTOS では，OS オブジェクトの属性や状態の組み合わせが多数存在するため，例えば 1 つの API を呼び出すテストを行う場合であっても，API を呼び出す際の状態・条件の組み合わせが膨大となる．すべての組み合わせを実現するテストケースに対して，テストプログラムをスクラッチで開発すると，開発コストが大きくなってしまう．また，タスクの起動順序や期待結果の確認方法の違いなどにより，1 つのテストシナリオを実現するテストプログラムは何通りも考えられるので，複数の開発者で膨大な数のテストプログラムを開発した場合，テストシナリオの実現方法を統一することが困難であり，レビューや修正作業の負担が大きくなることが予想される．さらに，RTOS ではテストプログラムだけでなく，テストに登場する OS オブジェクトを定義するコンフィギュレーションファイルの用意も必要である．

実行環境上の制約

RTOS を実行する組込みシステム向けのプロセッサは，一般的に ROM/RAM サイズが小さいため，すべてのテストケースを 1 つのテストプログラムで実行できるとは限らない．したがって，スクラッチで 1 つのテストプログラムとして開発してしまうと，実施時に手作業で分割する必要が出てしまう．

逆に，各テストケースを 1 件ずつ独立した実装として開発した場合，1 件ずつ個別にビルド・ロード・実行する必要があるため，テスト実施時間が膨大になってしまう恐れがある．

課題 (2)：マルチプロセッサ環境における課題

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などは一般に知られていない．

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

[3] ソースコード修正時に，ソフトウェアの動作に影響が無いことを回帰的にテストすること．

同期制御が必要となる．例えば，プロセッサを跨いで API を呼び出すテストを行う場合，呼び出される側が意図した状態になっているタイミングで呼び出す必要があり，プロセッサ間でタイミングを待ち合わせる必要がある．

また，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，実施するべきテストケースが増加する．さらに，マルチプロセッサ向け RTOS のソースコード内には，各プロセッサの実行順序に依存してパスが定まる分岐が存在するため，RTOS のソースコードを修正せずにテストする場合，ソースコードカバレッジを網羅するのは困難である．

課題 (3)：A-OS 仕様に対する課題

A-OS 仕様の RTOS に対するテストでは，テストプログラムとは別に，RTOS のコンフィギュレーション情報に応じて，XML ファイルとコードやデータの配置を決定するためのヘッダファイルも用意する必要がある．また，メモリ保護機能に対応した RTOS の場合，ユーザモードで動作するタスクには，メモリ領域やレジスタへのアクセス制限があり，テストプログラム中に，OS 内部の状態を確認するコードや，レジスタを操作するコードが存在した場合，そのままでは実行できない問題がある．したがって，RTOS のソースコードを修正することなく，テストを実行可能とするテストプログラムの開発が必要である．

A-OS 仕様では，1 つのテストプログラムで実現できないテストケースが存在するため，多数のテストプログラムを，1 件ずつ個別にビルド・ロード・実行する必要があり，テスト実行時間が膨大になってしまう問題もある．

課題 (4)：メモリ保護機能に対する課題

A-OS 仕様は μITRON 仕様と比べて，設定できるパラメータが多いことに加えて，メモリ保護機能を考慮したテストケースも必要である．これにより，テストケースは，さらに膨大な数となる．

MPU は，プロセッサによって設定可能な属性や領域数，アライメント制約などが異なるため，メモリ保護機能に対応した RTOS のポーティングは複雑である．したがって，ポーティング後に，メモリ保護機能が正しく振る舞うことを網羅的かつ効率的にテストできることが望ましい．

しかし，MPU を用いたメモリ保護機能に対するテスト手法は，一般に知られていない．メモリ保護の対象となるメモリ領域の種別や，メモリ保護設定切り替え

方法など，メモリ領域へアクセスする条件は膨大であり，効率的なテスト手法が求められる．また，異なるプロセッサに対するテストを容易に実施可能とするために，テストのポータビリティ[4]も確保する必要がある．

2.3　本研究のアプローチ

RTOS をテストするには，2.2.2 節で述べた課題を解決するテスト手法やツールが必要である．本研究では，これらの課題を解決するためのテスト手法やツールを提案する．

2.3.1　μITRON ベースの RTOS 向けテストプログラム生成ツール

課題 (1) を解決するため，まず，ASP カーネルに対する API テストを実施するためのテストシナリオの構成およびテストプログラムの構造を考案した．その結果，テストプログラムを手作業で開発するのではなく，テストシナリオを形式的に記述したデータからツールにより生成するべきという結論に至り，テストプログラム生成ツール (TTG) を開発した．TTG は，他の課題を解決するテスト手法においても，重要な役割を持つツールである．

2.3.2　μITRON ベースのマルチプロセッサ向け RTOS のテスト

課題 (2) を解決するため，FMP カーネルに対して，ブラックボックス API テスト，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．API テストでは，マルチプロセッサ向け RTOS に対応するため，TTG に対して，プロセッサ間同期制御機構を追加した．実行順序依存分岐テストでは，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) を考案し，各プロセッサの実行順序に依存してパスも含めて，ソースコードカバレッジを 100%とした．

2.3.3　テストプログラム生成ツールの AUTOSAR OS への適用

課題 (3) を解決するため，まず，TTG をベースとして，A-OS 仕様に対応したテストプログラム生成ツールである AKTG を開発した．併せて，メモリ保護機能に

[4]本論文では，RTOS やテストプログラムを異なるプロセッサへポーティングする際の移植性を意味する．

よって発生する制約を解決するためのテストライブラリも開発した．テスト実行時間が長くなってしまう問題に対しては，強制的に 1 つのテストプログラムで実現する仕組みを考案し，AKTG でその仕組みを利用したテストプログラムを生成可能とした．JulinarOs を対象として API テストを行い，これらのテスト手法の有効性を確認した．

2.3.4　AUTOSAR OS 仕様ベースの RTOS に対するテスト

課題 (4) を解決するため，ATK2 に対して，API テスト，アクセス保護テストを実施した．API テストでは，膨大なテストケースに対し，AKTG への入力とする形式的に記述するデータも，組み合わせ生成ツールを用いて生成する手法を考案した．また，AKTG および組み合わせ生成ツールを応用し，ポータビリティの高い，アクセス保護テストの実現手法を考案した．

第3章　μITRONベースのRTOS向けテストプログラム生成ツール

3.1　概要

IoT 社会に向かう中で，組込みシステムの重要性は一層高まっている．一方で複雑化した組込みソフトウェアの不具合が問題視されている [1]．組込みソフトウェア開発においては，プログラムの再利用性や，リアルタイム性を確保するために，組込み向け RTOS を採用して開発することが一般的である．つまり RTOS は，組込みシステムの品質を支える重要なソフトウェアであると言える．

組込みシステムの規模が比較的小さかった時代は，RTOS に要求される機能もシンプルであったため，組込みシステムを開発する企業が，RTOS を自社開発するケースが多かった．しかしながら，近年は他社製やオープンソースの RTOS を利用する企業が増加している．これは，近年の組込みシステムの高機能化・大規模化に伴い，マルチプロセッサやメモリ保護に対応した RTOS が必要となり，自社内での開発が，困難になっているからである．

TOPPERS プロジェクト [5] では，TOPPERS 新世代カーネルとして複数の RTOS を開発している．μITRON 仕様 [3] をベースとしたシングルプロセッサ対応の RTOS である ASP カーネルをはじめとして，マルチプロセッサに対応した FMP カーネルや，メモリ保護機能をサポートした HRP2 カーネルなど，様々な RTOS を開発し，オープンソースで公開している．これらの RTOS は，実製品にも利用されているが，製品への組込み時に利用者側での改変や拡張が行われることが一般的であるので，製品の品質を保証するために，利用者はアプリケーションのテストだけではなく，利用した RTOS のテストも実施する必要がある．

しかし，RTOS に対するテストは，一定のコストを要するため，オープンソースのメリットを損なっていると言える．さらに，マルチプロセッサやメモリ保護に

対応した RTOS は，歴史が浅く，テスト手法が確立されていないという問題がある．また，RTOS をオープンソースで公開している団体自身で，RTOS に対するテスト手法やテストスイートの評価を公開している事例はない．

この問題に対して，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，複数の企業と団体の参加を得て，RTOS に対するテスト手法の確立と，テストスイートの開発を実施している．RTOS の品質を維持・向上していくために，テストスイートの拡張性や保守性についても研究している．開発したテストスイートは，オープンソースで公開している [22].

テストスイート開発にあたり，RTOS に対するテスト分析を実施した結果，まず API に着目したテスト (以下，API テスト) の開発から着手することにした．また，API テストは，RTOS 特有の性質から，スクラッチでテストプログラムを開発・保守することが困難であることや，利用者の利便性を損なうことが明確となった．

そこで本研究では，テストプログラムを直接コーディングするのではなく，テストプログラムより抽象度の高いテストシナリオから，テストプログラムを生成することで，この問題を解決した．まず，テストシナリオを形式的に表現するために，TESRY(TEst Scenario for Rtos by Yaml) 記法と呼ぶ記法を考案した．次に，TESRY 記法で記述されたテストシナリオから，テストプログラムを生成するツールとして，TTG(Toppers Test Generator) を開発した．テストプログラムを生成することで，テストスイート開発を効率化し，テストスイートの拡張性や保守性が向上したことを確認した．なお，TTG は「OJL による最先端技術適応能力を持つ IT 人材育成拠点の形成」[23] の OJL(On the Job Learning) のテーマとして開発した．

本章の構成は以下の通りである．3.2 節で API テストの内容を述べ，3.3 節では TTG と TESRY 記法に対する要件を述べる．3.4 節では TESRY 記法について，3.5 節では TTG について述べる．3.6 節で TTG および TESRY 記法の評価を行い，3.7 節で関連研究について述べ，3.8 節でまとめる．

3.2 APIテスト

3.2.1 RTOS に対するテスト

ソフトウェアテストは，対象とするソフトウェアを分析した上で，適切な手法やツールを用いて実施されるべきである [24]．しかし，オープンソースで公開されている多くの RTOS では，テストスイートは付属しているものの，テスト分析や，

テスト手法の評価まで実施している事例はない．そこで，本研究は，RTOSの品質を保証するために必要なテストの目的と手法を，整理することから着手した．

その結果，まずRTOSが提供するAPIが，統合仕様書の通りに正しく振る舞うこと，および仕様に記述のない実装が存在しないことをテストすることにした．なお，これをAPIテストと呼称することにした．APIテストから着手した理由は，APIが仕様通りに動作することを前提に，ユーザアプリケーションが開発されることや，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が保証するべき最も重要な品質であると考えたからである．

3.2.2 APIテストの概要

APIテストとは，ASPカーネルが提供するすべてのAPIが，統合仕様書に定められている通りに正しく振る舞うこと，および仕様に記述のない実装が存在しないことを確認するものである．多くのAPIは，発行により，CPUロック状態やディスパッチ禁止状態などのOSの状態や，各OS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，APIテストでは，あるシステム状態でAPIを発行し，仕様通りのシステム状態の変化が起きることを確認する．

APIテストの開発の流れは次の通りである．統合仕様書からテスト対象とするAPIの仕様と振る舞いを理解し，テストケースを抽出する．次に，このテストケースからテストシナリオを作成し，テストシナリオを実現するためのテストプログラムを開発する．最後に，テストプログラム実行により，対象とするAPIの正当性を確認する．

なお，静的APIは，プログラムをビルドする際に実行されるものであるので，APIテストの対象外として，別途静的API用のテストを設けることにした．

3.2.3 テストケース

本研究では，各種の標準 [25][26] 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．

テストケースは，統合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．統合仕様書は，抽象的な表

低優先度の実行状態のタスクから，高優先度の 休止状態のタスクに対して act_tsk を発行すると， 対象タスクが実行状態になること．

図 3.1: act_tsk のテストケースの例

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．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ーを定め，API 毎に隔たりなく，テストケースを抽出することにした．また，RTOS では，複数タスクの優先度や状態などの組み合わせが膨大となるが，本研究の参加企業における RTOS 開発の過去の経験を踏まえて，組み合わせ爆発を防ぐためのポリシーも合わせて定めた．テストケースの例を図 3.1 に示す．この例は，休止状態のタスクを起動する API である act_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．

さらに，抽出したすべてのテストケースを実行することによって，RTOS のソースコードカバレッジが 100%となることを確認する．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，3.2.5 節で述べる不具合を検出できないため [27]，条件網羅 [2] によるソースコードカバレッジを確認する．

しかしながら，ソースコードには，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，仕様カバレッジを 100%にするテストケースを実施したとしても，ソースコードカバレッジが 100%にならない．これらは，ホワイトボックステストとして，該当するコードを網羅するためのテストケースを追加する．本研究で対象とした RTOS のソースコードでは，API 呼び出し時点でのシステム状態を特定すれば，実行タイミングに依存する条件分岐を除くすべての条件分岐で決定的に意図したパスを通すことが可能であるため，条件網羅を行うために必要なテストケースを導出することが可能である．実行タイミングに依存する分岐に対して網羅を行うテスト手法ついては，4.4.3 節で述べる．なお，ターゲットシステム毎に異なる実装 (以下，ターゲット依存部) に関しては，API テストの範囲外とし，網羅対象外とした．

[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．

前状態
　優先度 (低) の TASK1 が実行状態
　優先度 (高) の TASK2 が休止状態

処理
　TASK1 が act_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 3.2: act_tsk のテストシナリオの例

3.2.4 テストシナリオ

テストシナリオとは，テストケースの内容を「前状態」,「処理」,「後状態」に分けて，タスクなどの OS オブジェクトの振る舞いを具体化したものである．

3.2.2 節で述べたように，API テストでは，API 発行によるシステム状態の変化を確認する．具体的には，テストプログラムにて，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．テストケースは，統合仕様書の記述を引用した形で記述しているため，そのままでは API 発行前，発行後のシステム状態が明確となっていない場合がある．例えば，図 3.1 のテストケースの例では，TASK1 が act_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．そこで，テストケースを元に，確認するシステム状態を，手作業により具体化する．

このようにテストプログラムの実行内容を「前状態」,「処理」,「後状態」の 3 つの項目に分けて表現したものを，API テストのテストシナリオとする．なお，統合仕様書からテストシナリオを直接作成することも可能であるが，テストケースがテスト対象とする仕様から導かれる振る舞いを表すのに対し，テストシナリオはテストする方法を表しているため，テストシナリオでは抽象度が低く，仕様カバレッジを確認することが困難である．そこで，テストシナリオ作成の前に，統合仕様書の記述を引用したテストケースを作成し，仕様カバレッジはテストケースによって確認を行うことにした．

図 3.1 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 3.2 に示す．優先度の低い TASK1 と高い TASK2 が存在して，前状態は，TASK1 が実行され

ており (実行状態)，TASK2 が休止状態である．処理は，TASK1 が TASK2 に対して act_tsk を発行し，戻り値としてエラーコード E_OK を受け取ることである．後状態は，処理の結果として，TASK2 のほうが優先度が高いため，ディスパッチが発生し，TASK2 に処理が切り替わり (実行状態)，TASK1 は実行待ち (実行可能状態) となることを定義している．

3.2.5 カバレッジの確認

3.2.3 節で述べたように，API テストでは，仕様カバレッジを 100%にするように抽出したテストケースと，ホワイトボックステストとして追加したテストケースによって，ソースコードカバレッジが 100%となることを確認する．これにより，以下の 2 つの不具合を検出することができる．

1. 仕様に記載されている処理が実装されていない
2. 仕様に記載されていない処理が実装されている

(1) の不具合は，3.2.3 節で述べたテストケース抽出作業で検出する．(2) の不具合は，テストプログラムを実行する際，すべてのテストケースに対して，カバレッジを取得し，条件網羅によるソースコードカバレッジが 100%となることを確認することで，検出する．

3.3 テストシナリオ記法とテストプログラム生成ツールの要件

3.2.3 節で述べた方法でテストケースを抽出した結果，ASP カーネルの全 API に対するテストケースは，1,668 件となった．このすべてのテストケースに対して，テストプログラムをスクラッチで開発すると，開発コストが大きいこと，後述のテストスイート利用方法に制限が付くこと，ASP カーネル以外のプロファイルへの拡張が難しいことなどの問題がある．そこで，形式化したテストシナリオの記法と，その記法で記述された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を開発した．本節では，テストシナリオ記法とテストプログラム生成ツールの要件について述べる．

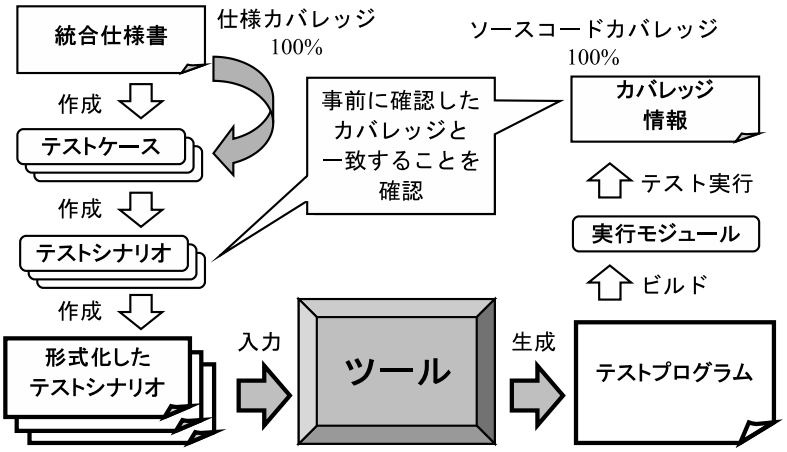


図 3.3: API テストにおけるツールの位置づけ

3.3.1　要件

テストシナリオ記法とテストプログラム生成ツールの要件を，以下のように定めた．また，API テストにおけるツールの位置づけを図 3.3 に示す．

1. テストシナリオからのテストプログラム生成
2. 統合仕様書に準拠した RTOS のサポート
3. 任意数のテストシナリオの統合
4. テストシナリオの正当性チェック
5. リソース共有によるメモリ使用量の削減

次節より，各要件について説明する．

3.3.2　テストシナリオからのテストプログラム生成

要件 (1) は，テストスイートの開発コスト削減，および保守性の向上が目的である．タスクの起動順序やシステム状態の確認方法の違いなどにより，1 つのテストシナリオを実現するテストプログラムは何通りも考えられるので，複数の開発者で 1,668 件のテストプログラムを開発した場合，テストシナリオの実現方法を統一することが困難であり，レビューや修正作業の負担が大きくなることが予想される．

一方，テストシナリオは単にシステム状態や実行内容だけを記述するため，開発者間の統一が容易である．そこで，テストシナリオを形式的に記述する記法を定め，その記法で記述したテストシナリオから，ツールを用いてテストプログラムを生成することにした．開発対象がテストシナリオとなることで，開発コストの削減と保守性の向上が期待できる．

また，考案する記法は，抽出したすべてのテストシナリオを記述可能であることにした．例外的に記述できないテストシナリオが存在すると，そのテストシナリオはスクラッチでテストプログラムを開発する必要があり，前述の問題を解決できないためである．

3.3.3 統合仕様書に準拠した RTOS のサポート

要件 (2) は，開発するツールの適用範囲を定義するものである．3.3.2 節で述べた記法や，生成するテストプログラムは，統合仕様書に準拠した RTOS に対応するものとする．2.1.2 節で述べたように，TOPPERS 新世代カーネルには様々なプロファイルが存在する．ツールは，これらすべてのプロファイルに対して適用可能とすることを目的とする．

また，ASP カーネルは，TOPPERS 新世代カーネルの基盤となるプロファイルであるため，ASP カーネルに対するテストシナリオは，他のプロファイルに対しても流用可能である．しかし，プロファイルによってコンフィギュレーションの記法がそれぞれ異なるため，テストプログラムのレベルでは流用できない．そこで，ツールは，ASP カーネルのテストシナリオから，異なるプロファイルに対応するテストプログラムも，生成可能とする．

3.3.4 任意数のテストシナリオの統合

要件 (3) は，テストスイートを利用したテストの実施容易性の向上が目的である．一般に組込みシステムにおいては，実行モジュールをターゲットシステムにロードして実行する．そのため，テストシナリオ毎に個別の実行モジュールとすると，テストシナリオの数だけロードと実行を繰り返す必要があり，テスト実施に多大な工数を要する．ロード可能なサイズの範囲で，複数のテストシナリオを 1 つの実行モジュールにすることができると，テスト実施の工数を大幅に削減できる．

3.3.5 テストシナリオの正当性チェック

要件 (4) は，テストスイートの開発効率の向上が目的である．作成したテストシナリオ自体に誤記や矛盾などの問題がある場合，テストプログラムをビルド，もしくは実行するまで気付けない可能性がある．しかし，3.3.2 節で述べたように，テストシナリオを自然言語ではなく，形式化して表現することによって，誤記や矛盾を検出する機能を，ツールに実装できる．

3.3.6 リソース共有によるメモリ使用量の削減

要件 (5) は，テストプログラム実行時における使用メモリ (RAM) の削減が目的である．3.3.4 節で述べたように，複数のテストシナリオを統合することにより，テストシナリオ間でリソースを共有できる可能性がある．ターゲットシステムによっては，RAM が非常に小さいため，テストプログラムの RAM 使用量が大きいと，実行中に問題が発生する可能性がある．そこで，ツールは，リソースを共有して RAM 使用量を圧縮したテストプログラムを生成する．

3.4 TESRY 記法

本節では，要件 (1)(2) を実現するために考案した，テストシナリオを形式的に記述する TESRY 記法について述べる．なお，TESRY 記法によってテストシナリオを記述したデータファイルを，TESRY データと呼ぶ．

3.4.1 構成

TESRY 記法は，テストシナリオを YAML 形式 [28] で記述したものである．YAML を採用した理由は，テストシナリオをデータ構造で階層的に表現できること，改行とインデントによって構造を表すので可読性が高いこと，などである．

TESRY 記法では，システム状態を，前状態として “pre_condition” 内に，後状態として “post_condition” 内に記述し，発行する API とその引数と，期待する戻り値を “do” 内に記述する．

図 3.2 のテストシナリオを，TESRY 記法で記述したものを，図 3.4 に示す．

```
pre_condition:
  TASK1:
    type   : TASK
    tskpri : LOW
    tskstat: running
  TASK2:
    type   : TASK
    tskpri : HIGH
    tskstat: dormant

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

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

図 3.4: TESRY 記法の例

```
<TASK_ID>:
type   : TASK /* OS オブジェクト種別 */
tskstat: [ running | ready | waiting | /* 実行状態 | 実行可能状態 | 待ち状態 | */
           dormant | suspended |       /* 休止状態 | 強制待ち状態 | */
           waiting-suspended ]         /* 二重待ち状態 */
wobjid : [ 対象待ち OS オブジェクト ID |  /* 対象待ち OS オブジェクト ID | */
           SLEEP | DELAY ]             /* slp_tsk() による待ち | dly_tsk() による待ち */
actcnt : [ 0 | 1 ] /* 起動要求キューイング数 */
wupcnt : [ 0 | 1 ] /* 起床要求キューイング数 */
itskpri: [ HIGH | MID | LOW | 整数 ] /* 起動時優先度 */
tskpri : [ HIGH | MID | LOW | 整数 ] /* 現在優先度 */
exinf  : [ 整数 ] /* 拡張情報 */
lefttmo: [ 整数 ] /* タイムアウト待ち時間 */
porder : [ 整数 ] /* 同一優先度タスク内での優先順位 */
bootcnt: [ 整数 ] /* 起動回数 */
```

図 3.5: タスクの状態定義

```
pre_condition:
  TASK1:
    type   : TASK
    tskpri : MID
    tskstat: running
  TASK2:
    type   : TASK
    tskpri : MID
    tskstat: ready

do:
  id     : TASK1
  syscall: dly_tsk(3)
  ercd   : E_OK

post_condition:
  0:
    TASK1:
      tskstat: waiting
      wobjid : DELAY
      lefttmo: 3
    TASK2:
      tskstat: running
  3:
    TASK1:
      lefttmo: 0
  4:
    TASK1:
      tskstat: ready
```

図 3.6: 時間指定の例

```
pre_condition:
  TASK1:
    type   : TASK
    tskpri : HIGH
    tskstat: running
  TASK2:
    type   : TASK
    tskpri : MID
    tskstat: ready

do_0:
  id     : TASK1
  syscall: slp_tsk()
  ercd   : E_RLWAI

post_condition_0:
  TASK1:
    tskstat: waiting
    wobjid : SLEEP
  TASK2:
    tskstat: running

do_1:
  id     : TASK2
  syscall: rel_wai(TASK1)
  ercd   : E_OK

post_condition_1:
  TASK1:
    tskstat: running
  TASK2:
    tskstat: ready
```

図 3.7: 処理と後状態の複数定義の例

前状態，後状態の定義

pre_condition，post_condition に対して，テスト対象とする ASP カーネルにおける，あらゆるシステム状態を記述可能とした．例として，タスクの状態定義の仕様を図 3.5 に示す．“<TASK_ID>” はタスクに付与する固有の ID である．

前状態では，テストに必要なシステム状態のみを定義する．前状態から変化しない状態は，後状態で省略可能とする．

処理の定義

do として，テスト対象とする API を，発行する処理単位 (id)，引数を含む実行コード (syscall)，戻り値 (ercd) に分けて記述する．図 3.4 の例では，処理単位 TASK1 が，実行コード act _tsk(TASK2) を発行して，戻り値 E_OK が返ることを示している．なお，API を発行するタスクや割込みハンドラなどの OS オブジェクトを処理単位と呼ぶ．

3.4.2 複雑なテストシナリオへの対応

テストシナリオの中には，前状態・処理・後状態だけでは表現できない，複雑なケースが存在した．これらの複雑なテストシナリオに対応するために，前状態・処理・後状態をベースに拡張した記法について述べる．

経過時間指定

3.2.2 節で述べたように，API は発行直後にシステム状態を変化させるものが多いが，発行直後だけでなく指定した一定時間後にシステム状態を変化させる機能を持つものもある．例えば，dly_tsk という API は，発行直後に発行したタスクを待ち状態とし，引数で指定した時間経過後に，実行可能状態とする (待ち状態を解除する)．このような API が正しく振る舞うことをテストする場合，テストシナリオで時間経過を明示的に指定する必要がある．

TESRY 記法では，前状態を実現した時刻を基準として後状態の中で指定した時刻でのシステム状態をチェックする記述を定めた．図 3.6 に dly_tsk の TESRY データにおける時間指定の例を示す．post_condition とシステム状態定義の間に，経過時間を指定する階層を定義することで，経過時間を指定する．時間指定の単位は，

RTOS が管理する時間の単位であるミリ秒とする．“0”，“3”，“4” は経過した時間を示す．“0” は発行直後であり，API を発行したタスクが，待ち状態 (waiting) へと変化し，タイマーとなる値 (lefttmo) が設定される．“3” は 3 ミリ秒後の状態であり，lefttmo が時間とともに減少し，0 になったことを示す．“4” で 4 ミリ秒となり，引数で指定した時間が経過したため，タスクは実行可能状態 (ready) となる．

処理と後状態の複数定義

API によっては，発行直後にリターンせずに，呼び出し元のタスクを実行状態以外の状態へ遷移させるものがある．その状態への遷移後に，特定の条件を満たすことで API がリターンし，特別な戻り値を返す．例えば，slp_tsk という API は，発行したタスクが起床待ち状態となるが，その後，別のタスクから rel_wai という待ち状態を強制解除する API を発行されると，起床待ち状態が解除され，戻り値として E_RLWAI が返るという仕様がある．この仕様をテストする場合，do の実行コード (syscall) を slp_tsk，戻り値 (ercd) を E_RLWAI と記述しても，post_condition で起床待ち状態となっているため，戻り値に E_RLWAI が返ったことを確認できない．

そこで，TESRY 記法では，複数の API を指定した順序で発行するための記述方式を定めた．図 3.7 に slp_tsk の TESRY データにおける処理と後状態の複数定義の例を示す．1 つ目の処理 (do_0) として TASK1 が slp_tsk を発行し，1 つ目の後状態 (post_condition_0) として TASK1 が起床待ち，TASK2 が実行状態となる．続いて，2 つ目の処理 (do_1) として実行状態となった TASK2 が TASK1 に対して rel_wai を発行する．2 つ目の後状態 (post_condition_1) として TASK1 の方が優先度が高いため，再び TASK1 が実行状態となり，ここで，slp_tsk に対する戻り値 E_RLWAI が返ることを確認する．

なお，図 3.7 の例では，処理と後状態の組み合わせは 2 組であるが，任意数の処理と後状態の定義が可能である．

3.5 TTG

TTG は，TESRY データを入力データとして，テストプログラムを生成するツールである．本節では，要件に対する TTG の設計，TTG により生成するテストプログラム，および TTG の実装について述べる．

3.5.1 要件に対する TTG の設計

本節では，3.3 節で述べた要件を満たすために行った TTG の設計について述べる．

異なるプロファイルへの対応

3.3.3 節で述べたように，ASP カーネル用に開発したテストシナリオは，FMP カーネルや HRP2 カーネルに対しても適用できるが，テストプログラムは異なる．しかし，テストプログラムは TESRY データから生成するため，異なるプロファイルに対応したテストプログラムを生成する機能を用意することで，要件 (2) を実現可能である．

具体的には，TTG 実行時のオプションで，対象とするプロファイルを指定可能とし，指定されたプロファイルに対するコンフィギュレーションを記述したテストプログラムを生成する．例えば，マルチプロセッサ対応 RTOS である FMP カーネルの場合，OS オブジェクトを割り付けるプロセッサ ID の指定が追加される．

また，3.5.2 節で述べるチェックポイントには，シーケンシャルな数値に加え，チェックポイント通過時のプロセッサ ID を付与して，正しいプロセッサ上で実行されることも確認する．これにより，ASP カーネル用に開発した TESRY データを，FMP カーネルに対しても使用可能となる．

テストシナリオ統合処理

要件 (3) を満たすために，任意数のテストシナリオをまとめて，1 つの実行モジュールを生成する必要がある．これを実現する方法としては，以下の 2 つが考えられる．

1. 全テストシナリオを 1 つのテストプログラムとして生成し，コンパイル時に ifdef などを用いてテスト対象のテストシナリオのみコンパイル対象とする方法
2. テスト対象とするテストシナリオのみを含むテストプログラムを生成する方法

方法 (1) の場合，常にすべてのテストシナリオが含まれることにより，テストプログラムが長くなり，デバッグ時などの可読性が低下する．また，ifdef などを用いて対象とするテストのみをコンパイルするための処理が必要となり，テストプ

ログラム生成処理が複雑になる．一方，方法 (2) の場合，テスト対象とするテストシナリオのみを含むテストプログラムを生成するので，これらの問題は発生しない．この場合，ユーザがテスト対象とするテストシナリオの TESRY データだけを TTG の入力とする必要があるが，TTG はコマンドライン上から実行可能であるので，TESRY データを選択して入力するのは容易である．

また，方法 (1) は，3.5.2 節で述べる実行順序確認のためのチェックポイントの実現が困難である．各チェックポイントには，シーケンシャルな数値を付与する必要があるため，任意のテストシナリオのみを実行するには，リナンバリングの必要があるからである．数値を変数によって動的に指定する方法もあるが，RTOS に不具合があり，意図しないチェックポイントを通過してエラーとなった場合，エラーが発生したチェックポイントが，テストプログラム上のどのチェックポイントであったかを特定するのが困難である．方法 (2) では，任意数のテストシナリオ毎に，1 つずつテストプログラムを生成し，それぞれのテストプログラムにおいてシーケンシャルな数値を付与するため，この問題は発生しない．

さらに，あるテストシナリオに不具合があった場合，方法 (1) では，すべてのテストシナリオに対するテストプログラムを生成し直す必要があるが，方法 (2) では，対象のテストシナリオを含むテストプログラムを生成し直すだけでよい．

以上より TTG では，方法 (2) を採用した．これにより，テストスイート利用者は，機能毎などターゲットシステムのメモリ制約に合わせた分割単位で，TESRY データを入力してテストプログラムを生成することで，テスト実施工数を最小限とすることが可能である．当然ながら，メモリ制約がない場合は，全 TESRY データを入力として 1 つのテストプログラムを生成し，全テストシナリオを 1 回で実行可能である．

テストシナリオ正当性チェック処理

要件 (4) より，TTG は入力された TESRY データの正当性のチェックを行う．はじめに 3.4 節で述べた TESRY 記法の構成が正しいことを確認し，各状態定義→各 OS オブジェクト→各コンディション (pre_condition，post_condition) のように YAML 形式の階層にしたがって，段階的に以下の 6 つのカテゴリに分け，チェック処理を実装した．チェック項目は合計で 275 件であった．

1. テストシナリオの構成 (26 件)

2. OS オブジェクトの状態定義 (145 件)

3. OS オブジェクト内の依存関係 (37 件)

4. コンディション内の依存関係 (44 件)

5. テストシナリオ内の依存関係 (16 件)

6. テストシナリオ間の依存関係 (7 件)

(1) は，pre_condition，do，post_condition が存在するかなどの，基本構造のチェックである．(2) は，タスクの状態に不正な文字列が指定されていないかなどの，各パラメータ毎のチェックである．(3) は，タスクが待ち状態にも関わらず，対象待ち OS オブジェクト ID が指定されていないなどの，OS オブジェクト内のパラメータ同士の依存関係のチェックである．(4) は，タスクの対象待ち OS オブジェクト ID に指定された ID の OS オブジェクトが存在しないなどの，コンディション内の依存関係のチェックである．(5) は，pre_condition で定義されていない OS オブジェクトが post_condition で登場していないかなどの，1 つのテストシナリオ内における依存関係のチェックである．(6) は，複数の TESRY データを 1 つのテストプログラムに統合できない要因となる OS オブジェクトが存在しないかなどの，入力したすべて TESRY データ間の依存関係のチェックである．

タスクのスタック共有

要件 (5) のリソース共有には，OS オブジェクトを共有する方法と，スタック領域を共有する方法が考えられる．

OS オブジェクトを共有する場合，テストシナリオに最も多く登場する OS オブジェクトであるタスクを共有する方法が望ましいが，コンフィギュレーション時に静的 API で設定するタスクの優先度や拡張情報などの値が，テストシナリオ毎に異なるため，共有できるタスクには制限がある．したがって，効率的にタスクを共有させるためには，テストシナリオ間で設定するタスクの状態定義を調整する必要がある．また，タスクを 1 つ管理するために必要な RAM 使用量は，32byte と小さいため，共有しても効果が低いと考えられる．タスク以外の OS オブジェクトの RAM 使用量は，タスクよりも小さいため，さらに共有の効果は低い．

一方，タスクのスタック領域は，テストシナリオに記述するタスクの状態定義とは無関係である．さらに，複数のテストシナリオが同時に実行されることがないため，最もタスクが多く登場するテストシナリオにおけるタスクの数だけ，ス

タック領域を確保できれば，他のテストシナリオでも再利用が可能である．また，一般にスタック領域は，比較的大きいサイズのメモリが必要であるため[3]，スタック領域を共有したほうが，OS オブジェクトを共有するより，RAM 使用量削減の効果が高い．

以上より，TTG では，複数のテストシナリオに対して，スタック領域を共有可能とした．なお，タスクを含めた OS オブジェクト自体の共有は，今後の課題とする．

3.5.2 TTG により生成するテストプログラム

テストプログラムの基本構造

テストプログラムでは，テストシナリオで指定されたタスクとは別に，前状態の実現や各システム状態のチェック用のタスク (管理タスク) を用意する．管理タスクは，まず，各 OS オブジェクトが 3.2.4 節で述べたテストシナリオの前状態で定義された状態となるように，API の発行などを行い，前状態を実現する．統合仕様書に基づいて，システム状態，各 OS オブジェクトの状態，API の仕様をモデル化して TTG に実装することにより，入力された前状態の正当性チェックや，前状態を実現するために必要なソースコードの生成が可能となる．テストシナリオの前状態において，実行状態と指定された処理単位から，実現したシステム状態が，前状態に記述された内容と一致していることの確認を行う．

前状態を実現した後に，テストシナリオの処理 (do) として記述された実行コードを，指定された処理単位から実行する．戻り値が指定されていれば，指定された戻り値と一致しているかの確認を行う．

処理の後で，後状態で実行状態である処理単位から，システム状態が後状態に記述された内容と一致しているかを確認する．後状態で実行状態の処理単位が存在しない場合は，管理タスクからシステム状態の確認を行う．なお，管理タスクが行う処理はテストシナリオ毎に異なるため，管理タスクのソースコードも，テストシナリオに合わせて TTG が生成する．

[3]ASP カーネルのテストシナリオを対象に，TTG で生成したテストプログラムでは，最低でも 512byte 必要であった．

システム状態の確認方法

テストプログラムは，前状態や後状態がテストシナリオで指定されたシステム状態と異なる状態となった場合に，エラーを出力する．システム状態の確認方法として，RTOS が提供する各 OS オブジェクトの状態を参照する機能を持つ API(以下，状態参照 API) を用いる方法がある．例えば，タスクの状態参照 API は，タスクの状態や優先度などの各パラメータに対応したメンバ変数を持つ構造体に，指定したタスクの現在の情報を格納して返す．しかし，状態参照 API には，以下の問題がある．

1. タスク以外の処理単位から発行できない
2. TESRY 記法で定めたすべてのパラメータを参照できない
3. 状態参照 API 自体の API テストを実施できない

(1) は，状態参照 API の仕様により，タスク以外の処理単位からシステム状態の確認ができないという問題である．テストシナリオによっては，割込みハンドラなどのタスク以外の処理単位が実行状態となっている状態も有り得るため，状態参照 API ではシステム状態の確認ができないテストシナリオが発生してしまう．タスク以外の処理単位からシステム状態を確認できるように，状態参照 API のソースコードを変更してしまうと，テスト対象のソフトウェアとは異なってしまうことから，変更するべきではない．そこで，状態参照 API とは別に，すべての処理単位からの発行を可能とした API (以下，代替 API) を開発して利用することにした．

(2) は，状態参照 API で確認できるシステム状態に制限があり，TESRY 記法で定めたすべてのパラメータを参照できないという問題である．例えば，タスクの初期優先度などのコンフィギュレーション時に静的に定義するパラメータは，状態参照 API で確認できない．また，同一優先度の実行可能状態のタスクが複数存在する場合に，どの順序で実行されるかの優先順位も，状態参照 API で確認できない．API テストでは，API の発行によって正しい優先順位になったことを確認することが必要であるが，これを確認するには，実際にタスクを実行状態に遷移させるしかないため，テストプログラムが複雑となる．そこで，代替 API では，状態参照 API で確認できるパラメータを含め，TESRY 記法で定めたすべてのパラメータを参照できるようにした．

(3) は，状態参照 API をシステム状態の確認に使用してしまうと，状態参照 API 自体の API テストを，TTG で生成したテストプログラムで実施できないという問

題である．状態参照 API も API テストの対象であるが，同じ API を用いて参照したシステム状態同士を比較してもテストとして評価できない．そこで，テストプログラムでは，代替 API を用いてシステム状態を管理する RTOS 内部の変数を直接取得し，状態参照 API による取得結果と一致していることで確認を行うことにした．

チェックポイントによる実行順序チェック

RTOS では，処理単位の切り替えが発生することにより，プログラムの実行される順序が複雑になる．例えば，タスクが 2 つ存在する場合，それぞれのタスクの処理を 2 つの関数で実装するが，片方のタスクの実行中にもう一方のタスクにディスパッチすると，実行されている関数が切り替わる．タスクの数が多くなり，ディスパッチが頻繁に発生すると，テストプログラム中の実行箇所も頻繁に前後することになる．そこで，想定した順序で各処理単位の処理が実行されることを確認するために，テストプログラム中にチェックポイントを入れることにした．チェックポイントには，実行されるべき順番に，シーケンシャルな数値を付与しておき，想定外の順序でチェックポイントを通過した場合，エラーを出力する．

TTG には統合仕様書に基づいた API 仕様を実装しているため，前状態を実現する過程，および後状態で次のテストシナリオを実行するためのシステム状態を実現する過程において，各 API を呼び出すことで実行状態となる処理単位が切り替わることを識別することが可能である．各過程において，実行状態の処理単位が切り替わったタイミングで，実行状態の処理単位の実行コードにチェックポイントを付与していくことで，想定する順序で実行されることの確認を行う．また，前状態で実行状態となっている処理単位の実行コードには，テスト対象の処理を実行する直前に，後状態で実行状態となっている処理単位の実行コードには，テスト対象の処理を実行した直後に，それぞれチェックポイントを付与することで，テスト対象の API 呼び出しによって意図した処理単位の切り替えが発生したことの確認を行う．このように，チェックポイントを付与することで，想定した通り順序でテストプログラムが実行されることを確認する．

テストプログラムの正当性

生成するテストプログラムの正当性については，2 つの側面から検証可能と考えた．1 つ目は，テストケース毎に生成したテストプログラムを実行して取得する

ソースコードカバレッジが，想定したパスを通っていることを確認することである．これは，3.2.3 節で述べたソースコードカバレッジを 100%とするテストケース抽出作業と，3.2.5 節で述べた実行結果のソースコードカバレッジを確認する作業により，確認する．

2 つ目は，代替 API によるシステム状態のチェックである．開発した代替 API では前状態，後状態における，すべてのシステム状態を参照可能としており，想定した状態と異なるものが 1 つでも存在すれば，アサートエラーとするコードを生成する．

以上より，正常にテストプログラムが実行され，想定したソースコードカバレッジとなるには，以下の 3 つの条件が満たされていると考えることができる．

- RTOS が仕様通りに実装されている
- TESRY データが仕様通りに記述されている
- TTG が正しいテストプログラムを生成している

3.5.3 TTG の実装

TTG は，フロントエンドとバックエンドに分けて実装した．フロントエンドでは，入力された TESRY データに対して，パラメータの補完処理やチェック処理などの前処理を行う．バックエンドでは，フロントエンドによって生成された全テストシナリオが含まれるデータを入力として，テストプログラムの生成処理を行う．本構成により，バックエンドでは，不正なテストシナリオに対する処理などを考慮する必要がなくなり，テストプログラム生成処理を簡潔にすることが可能である．TTG の構成と処理フローを図 3.8 に示す．図中の各処理内の番号は，次節以降の説明で使用する番号を意味する．

フロントエンド

フロントエンドでは，入力された TESRY データに対して前処理を行い，前処理の結果として得たテストシナリオのデータをバックエンドへの入力とする．前処理は，以下の流れで行う．

1. パラメータ補完

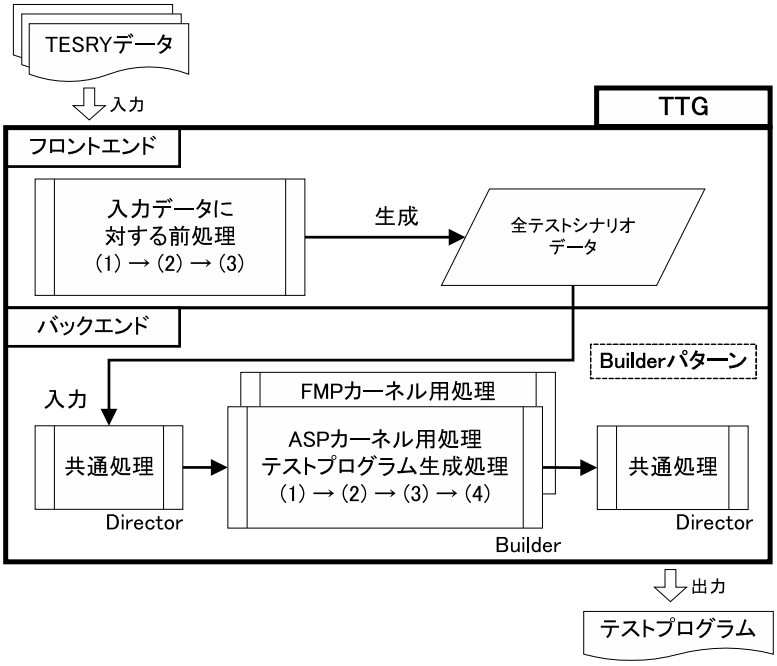


図 3.8: TTG の構成と処理フロー

2. 正当性チェック

3. OS オブジェクト ID 変更

(1) は，TESRY データにおいて，省略されているパラメータを補完する処理である．3.4.1 節で述べたように，TESRY データでは，前状態から変化のないシステム状態を定義するパラメータは，後状態で省略可能である．また，TESRY データには，テストシナリオに必要なパラメータ以外は定義されていない．バックエンドのテストプログラム生成処理ではこれらのパラメータが必要であるため，前者に関しては前状態のシステム状態を，後者に関してはデフォルト値を設定する補完処理をフロントエンドで行う．

(2) は，3.5.1 節で述べた TESRY データの正当性のチェックである．複数の TESRY データが入力された場合，入力された全 TESRY データに対して，まずチェック処理を実行してからテストプログラム生成処理を行うようにした．これにより，1 つでもテストシナリオに問題があれば，エラー終了するため，無駄なテストプログラム生成処理を省くことができる．

(3) は，各テストシナリオ毎に登場する OS オブジェクト ID を，全テストシナリオ間で重複しないように変更する処理である．3.5.1 節で述べたように，任意数

のテストシナリオを統合する必要があるが，異なるテストシナリオに同一 OS オブジェクト ID が存在すると，1 つのテストプログラムとした際に，OS オブジェクト ID の衝突が起きてしまう可能性がある．そこでフロントエンドにおいて，テスト ID を OS オブジェクト ID に付加することで，すべての OS オブジェクト ID を一意とする．入力したすべての TESRY データのテスト ID が一意でなければフロントエンドでエラーにする．

バックエンド

バックエンドでは，フロントエンドによって生成されたデータを入力として，テストプログラムを生成する．3.5.1 節で述べたように，複数のプロファイルに対応することを考慮して，バックエンドでは Builder パターンを適用した．プロファイル毎に Builder を用意することで，新しいプロファイルへの対応や，プロファイル毎の仕様変更への対応が容易となる．

Builder パターンの Director にあたる共通処理では，テストシナリオに登場する OS オブジェクト情報の生成や，テストプログラムの出力処理など，プロファイルに関係しない処理を行う．Builder パターンの Builder にあたるテストプログラム生成処理は，以下の流れで行う．

1. 前状態への遷移する処理の作成，およびシステム状態確認
2. 処理の実行
3. 後状態のシステム状態確認
4. テストシナリオ終了処理の生成

(1) では，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 OS 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，代替 API を用いてすべての OS 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．

(2) では，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．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．

(3) では，代替 API を用いてすべての OS 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．なお，図 3.6 のように経過時間指定されている場合は，時刻を調整するために用意したライブラリを使用して，定義された時刻とするためのソースコードを，(3) の処理の前に生成する．また，図 3.7 のように処理と後状態が複数定義されている場合は，(2)(3) の処理を繰り返す．

(4) では，次のテストシナリオを実行するために，実行状態や待ち状態のタスクをすべて休止状態へ遷移させるソースコードを生成する．

3.6　評価

本節では，TTG の効果を，ASP カーネルのテストスイート開発の結果に基づいて，評価する．対象とする ASP カーネルのバージョンは 1.6.0 とし，コンパイラは GCC4.5.1[29] を使用した．TTG が生成したテストプログラムは，ARM の命令セットシミュレータである SkyEye[30] を用いて実行した．カバレッジ確認には，GCC に付属している GCOV[31] を使用した．

評価項目として，TESRY 記法の記述性，開発の効率化，他のプロファイルへの拡張性，多様なターゲットシステムへの対応，スタック共有の効果，テストプログラムの正当性，検出した不具合の，7 点について行った．

3.6.1　TESRY 記法によるテストシナリオの記述性

考案した TESRY 記法によって，ASP カーネルに対する 1,668 件すべてのテストシナリオを，表現可能であることを確認した．また，TTG によって，すべてのテストシナリオに対するテストプログラムを，生成可能であることを確認した．

1,668 件のうち，経過時間指定が必要であったテストシナリオが 38 件，複数の処理/後状態が必要であったテストシナリオが 167 件であった．つまり，約 12%が 3.4.2 節で述べた複雑なテストシナリオであり，これらのテストシナリオのテストプログラムをスクラッチで開発した場合，要件 (1) を満たせなかったと考えられる．

3.6.2 開発の効率化

TTG を用いて，テストシナリオ毎に生成したテストプログラムの総行数は，352,268 行となった．スクラッチでテストプログラムを開発して，同等のテストを実現する場合，約 35 万行規模のテストプログラムを開発・保守する必要があるということである．

一方，全テストケースの TESRY データの総行数は 59,959 行であったため，単純にテストプログラムの開発規模を 6 分の 1 程度まで低減した．また，開発対象が TESRY データとなることや，TESRY データの可読性の高さからも，開発コストを削減できたと考えられる．

3.6.3 他のプロファイルへの拡張性

TTG は Ruby で開発し，総行数はコメントと空行を入れて 19,328 行であった．Ruby のバージョンは 1.8.7 を使用した．本研究では，まず ASP カーネルを対象として TTG を開発し，その後，FMP カーネルを対象に，TTG の拡張を行った．FMP カーネルに対応した TTG の総行数は，22,032 行であり，約 10%のコード追加により対応が可能であった．これは，Builder パターンにより，プロファイルの違いに関係のない，コード生成処理やチェック処理を共有できていることが大きい．さらに，ASP カーネル用に開発した TESRY データは，FMP カーネルに対しても使用可能であることを確認した．以上により，TTG の他のプロファイルに対する拡張性を確認した．

3.6.4 多様なターゲットシステムへの対応

ASP カーネルのテストシナリオ 1,668 件に対して，1 つのテストプログラムにまとめてビルドした場合の実行モジュールのサイズ，1 件毎にビルドした場合の最大値，機能毎に分割してビルドした場合の最大値を，表 3.1 に示す．ROM に配置されるのは，text セクションと rodata セクションであるので，この合計値で比較を行う．

テスト対象とするターゲットシステムの ROM が，3,576kbyte 以上搭載されていれば，全テストシナリオを 1 回の実行で可能である．逆に，最低でも ROM が 63kbyte 搭載されていればテスト実施は可能である．つまり，この間の ROM サイズのターゲットシステムの場合は，表 3.1 のサイズを元に，実施回数がなるべく少

表 3.1: メモリ使用量

	全件	1 件毎の最大値	機能毎の最大値
text	3,260	59	668
rodata	316	4	67
text+rodata	3,576	63	735
data	1	1	1
bss(共有なし)	3,307	14	619
bss(共有あり)	231	14	66

(単位は kbyte)

なくできるように分割して実施することが可能である．なお，ターゲットシステムの ROM サイズを指定して，適切に実行モジュールを分割する方法は，今後の課題として検討している．

3.6.5　スタック共有の効果

スタック領域は，初期値無しデータとして，bss セクションに配置されるので，この値で比較を行う．3.6.4 節と同様に，3 つの分割単位における，スタック共有を行った場合と行わなかった場合の，bss のサイズについて比較した結果を表 3.1 に示す．タスク毎のスタックサイズは 1kbyte として測定した．

全件を対象としたテストプログラムの場合，bss のサイズは，スタック共有を行わなかった場合で 3,307kbyte，スタック共有を行った場合で 231kbyte となり，7%以下に削減できた．

ASP カーネルがサポートしている ARM プロセッサを用いたシステム (チップ) の中で，最も小さい RAM サイズは 64kbyte である．表 3.1 より，RAM サイズが 64kbyte のターゲットシステムであっても，スタック共有を用いることで，より多くのテストシナリオを一括して実行可能となることを確認した．

以上より，スタック共有の有効性を確認した．

3.6.6 テストプログラムの正当性

全テストケースをまとめて実行し，ソースコードカバレッジを取得して，対象とするソースコードの条件網羅が 100%となることを確認した．さらにテストケース毎にカバレッジを取得し，事前に想定していたパスを通ることを確認した．これにより，3.5.2 節で述べたように，TTG が生成するテストプログラムの正当性を確認した．

なお，ASP カーネルのソースコードにおいて，カバレッジ対象としたライン数は，1,885 行であった．

3.6.7 検出した不具合

TTG を利用した API テストによって，統合仕様書および ASP カーネルに対して 17 件の不具合を検出した．これらの不具合は，主に以下の 4 種類に分類できる．

1. 不要なソースコード，分岐が存在する
2. 統合仕様書に記載のない実装が存在する
3. 期待した後状態とならない
4. 意図しないチェックポイントを通過した

(1)(2) は，3.2.5 節で述べた不具合であり，意図した不具合を検出できたと言える．(3)(4) は，ASP カーネルの実装に含まれた不具合であり，特にターゲット依存部の不具合が多かった．これは，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 が仕様通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．

3.7 関連研究

近年の汎用 OS や RTOS では，多機能化や複雑化によって，API が仕様通りに動作することを確認するテストだけでも，テストケース数が膨大となり，スクラッチでテストプログラムを開発することが困難となっている．

オープンソースの汎用 OS である Linux では，Linux Test Project[32] 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，OS 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 [33]，テストケースやテストプログラムは，スクラッチで開発しているために，開発コストが大きいと考えられる．

オープンソースの RTOS である FreeRTOS[34] に対しては，テストケースおよびテストプログラムを生成するツールである STG[35] を利用して，テストを実施した研究がある [36]．STG では，対象とするソフトウェアの仕様とテストの目的を，IOSTS[37] という形式で記述し，これを入力として，IOSTS 形式のテストケースを生成する．次に，生成したテストケースを入力として，テストプログラムを生成する．生成可能なプログラミング言語は C++と Java である．なお，FreeRTOS に対する研究では，テストケースの生成のみで，テストプログラムの生成には言及していない．また，テストケースの生成に関しても，テストの目的を RTOS の割込み処理に関する仕様に限定しており，RTOS 全体はテスト対象としていない．

一般的なソフトウェアに対しても，テストのコスト削減の取り組みとして，テストケースやテストプログラムを生成するツールが研究されている [38][39]．これらの研究では，対象とするソフトウェアの仕様を形式化して記述し，仕様を満たすことを確認するためのテストケースと，それを実行するテストプログラムを，ツールによって生成する．

これらのツールを今回対象とした RTOS の API テストに適用する場合，統合仕様書を何らかの形式的な記法で記述する必要がある．統合仕様書をすべて形式的に記述すれば，テストケースを生成できる可能性はあるが，タスクの数や優先度，状態の組み合わせは無数に存在するため，3.2.3 節で述べたように組み合わせ爆発が発生する恐れがある．また，これらのツールで生成されるテストケースは，実装が仕様に適合することを確認するコンフォーマンステストを目的としているが，本研究で実施したい API テストは，ソースコードカバレッジを 100%とすることも目的としているため，コンフォーマンステストよりテスト範囲が広く，これらの手法では十分なテストケースが生成されないという問題がある．コンフォーマンステスト以外では，リアルタイム性に関するテストに対して，テストケース生成からテスト実行までの一連の作業を自動化した事例 [40] があるが，使用しているツール [41] はリアルタイム性の解析に特化しているため，API テストには適用できない．

テストプログラムの生成に関しては，対象とするソフトウェアが，通信プロトコルや状態遷移系のシステムなど，特定の分野に特化されており，API テストに

は適用できない．特に，入力データとして定義した，API 発行前の RTOS のシステム状態を実現する処理のような，RTOS の API に特化したテストプログラムの生成が可能なツールの事例はない．また，既存のツールでは出力するプログラミング言語が限定されており，本研究で対象とする RTOS のように，静的 API によるコンフィギュレーションなど，独自の記述が必要なソフトウェアへの適用は困難である．

以上のように，既存のツールは，API テストに適用できないことから，本研究では，テストプログラム開発のコスト削減を含め，3.3 節で述べた問題点を解決するために，独自にテストシナリオの記法と，テストプログラム生成ツールを開発した．

3.8 まとめ

本章では，μITRON ベースの RTOS に対するテストを分析し，テストプログラム生成ツールの必要性について述べた．さらにツールを開発し，実際にテストスイートの開発・実施を行い，ツールの評価を行った．

TTG で生成したテストプログラムを実行することで，3.2.5 節で想定した不具合や，ターゲット依存部の実装不具合を検出した．今後，ASP カーネルの仕様が改変される際や，新しいターゲットシステムへポーティングされる際に，API テストを繰り返し実施することが予想されるため，テストスイートの再利用性や拡張性の高さによって，テストスイート開発コストが削減されることが期待できる．

TTG は，統合仕様書に準拠した μITRON ベースの RTOS にのみ対応しているため，タスクの状態やスケジューリング方式，API の仕様などが異なる他の RTOS に対しては，利用できない．しかし，3.3 節で述べた要件は，他の RTOS に対するテストプログラム生成ツールを開発する際に，適応が可能であり，TESRY 記法についても，対象とする RTOS に合わせて改変することは容易である．AUTOSAR[2] 仕様に準拠した RTOS に対するテストプログラム生成ツールについては，第 5 章，第 6 章で述べる．

第4章　μITRONベースのマルチプロセッサ向けRTOSのテスト

4.1　概要

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挙げられる．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，TOPPERSプロジェクトでは，マルチプロセッサ向けRTOSである，FMPカーネル[8, 42]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であるASPカーネル[7]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，国内外で広く用いられているμITRON仕様[3]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[43].

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．そこで本研究では，ASPカーネル，およびFMPカーネルの品質を高める取組みとして，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[19, 20, 21]．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る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 (以下，NCES) 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，それらを元にしたテストスイートを開発することである．

本研究では，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上で，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ーの中から，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ーの策定，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コスト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．本章では，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．

本章の構成は以下の通りである．4.2 節で本研究で実施したテストの概要を述べ，4.3 節でテストプロセスを，4.4 節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．4.5 節で実施したテスト実施結果について述べ，4.6 節で検出した不具合の分析を行う．4.7 節で，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，4.8 節で関連研究について述べ，4.9 節でまとめる．

なお，本研究で対象としたマルチプロセッサ向け RTOS である FMP カーネルについては，2.1.2 節を参照のこと．

4.2　テストの概要

本節では，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．テストの全体像については，2.2.1節を参照のこと．

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，2.2.1節で説明したすべてのテストカテゴリーを実施するのは困難であったので，本研究では，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つのテストを実施し，2年間で完了した．

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上で，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．

4.2.1　外部仕様(仕様書)を網羅するためのテスト

外部仕様(仕様書)を網羅するためのテストは，2.2.1節で述べた方針(1)のテストであり，主な目的はRTOSの品質の向上である．RTOSのユーザは，統合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，統合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，本研究では，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，統合仕様書の通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が保証する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OSの状態や各OS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，仕様通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，テストプログラムにて，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．

4.2.2 すべてのソースコードを実行するためのテスト

すべてのソースコードを実行するためのテストは，2.2.1 節で述べた方針 (2) のテストであり，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 [27]，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毎に異なる実装 (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 が仕様通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．

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

[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．

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．

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な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，前節で述べた要因 (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．

<u>共通部ロック関数</u>

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，別のプロセッサが既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 4.1 に示す．この例では，既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，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

```
 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 4.1: 共通部ロック関数の例

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 [44]．

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，取得する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，取得するべきオブジェクトロックを特定した後，一旦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．

4.2.3 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

3.2.3 節で述べた通り，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 4.2 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 4.2 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 4.3 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．

4.3　テストプロセス

本節では，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 4.5 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．

4.3.1　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．

(1) テスト分析

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

(2)B-API テストのテストケース設計

B-API テストでは，統合仕様書に記載された全 API の振る舞い，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，統合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．

統合仕様書は，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．例えば，wup_tsk には，『対象タ

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 4.2: テストケースの例

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 4.3: テストシナリオの例

```
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 4.4: TESRY 記法の例

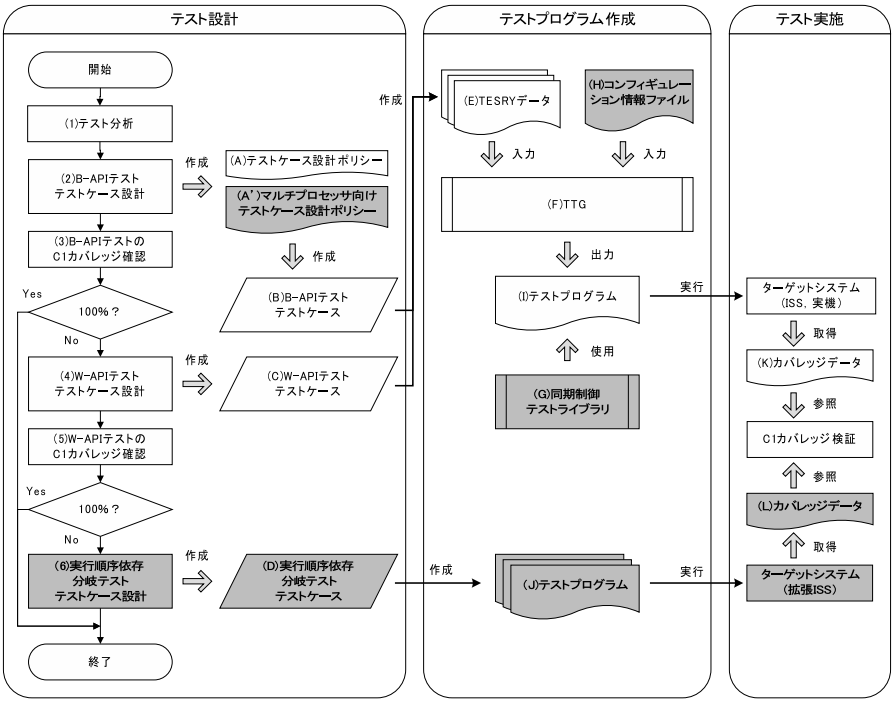


図 4.5: テストプロセスのフロー

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，ルールやポリシーがないと，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ーを定め，API 毎に隔たりなく，テストケースを抽出することにした．本研究では，まず ASP カーネルに対するテストケース設計ポリシーを定め (図 4.5(A))，それを FMP カーネルに拡張する形で設計した (図 4.5(A’))．詳細については，4.4.1 節で述べる．

定めたポリシーに従って，B-API テストのテストケースを作成する (図 4.5(B))．

(3)B-API テストの C1 カバレッジ確認

API 毎に 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 毎に RTOS 内の条件式を抽出し，テストケース毎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，あるいは一度も判定されていない条件式が存在する API は，次節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．

(4)W-API テストのテストケース設計

4.2.2 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 4.5(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)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．4.2.2 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，次節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．

(6) 実行順序依存分岐テストのテストケース設計

4.2.2 節で述べたように，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 4.5(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．

4.3.2 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 4.5(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 4.5(E))，ツールを用いて生成する (図 4.5(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 4.5(H)) と，プロセッサ同期制御ライブラリ (図 4.5(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，4.2.2 節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 4.4.3 節で述べる．

TESRY データ

次節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 4.5(E))．図 4.3 のテストシナリオを TESRY 記法で記述した例を図 4.4 に示す．

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 4.5(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，開発コスト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 OS 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 OS 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 OS 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，第 3 章を参照のこと．

4.3.3 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 4.5(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．

4.4 マルチプロセッサ向け RTOS に対するテスト手法

本節では，4.3 節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．

4.4.1 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ー

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，プロセッサを跨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) プロセッサを跨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き得ない条件に対するポリシーを追加する形で作成した．ポリシーの一部を以下に列挙する．

- API 発行対象 OS オブジェクトの他プロセッサへの割り付け
- 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ーを適用し，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．

4.4.2　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．

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，開発コスト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．

```
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 4.6: ASP カーネル向けテストプログラムの例

```
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 4.7: FMP カーネル向けテストプログラムの例

図 4.6 に ASP カーネル向けテストプログラムの例を，図 4.7 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 4.4 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 4.7 の 8 行目のように，FMP カーネルでは TASK2 が割り付いているプロセッサ 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 4.4 のように，ASP カーネル用の TESRY データには，タスクなどの OS 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 OS 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 4.4 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 4.6 のテストプログラムと，図 4.7 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 データ開発コストを削減することが可能である．

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 4.4 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 がプロセッサを跨いで，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 4.4 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 4.8 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 が一旦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

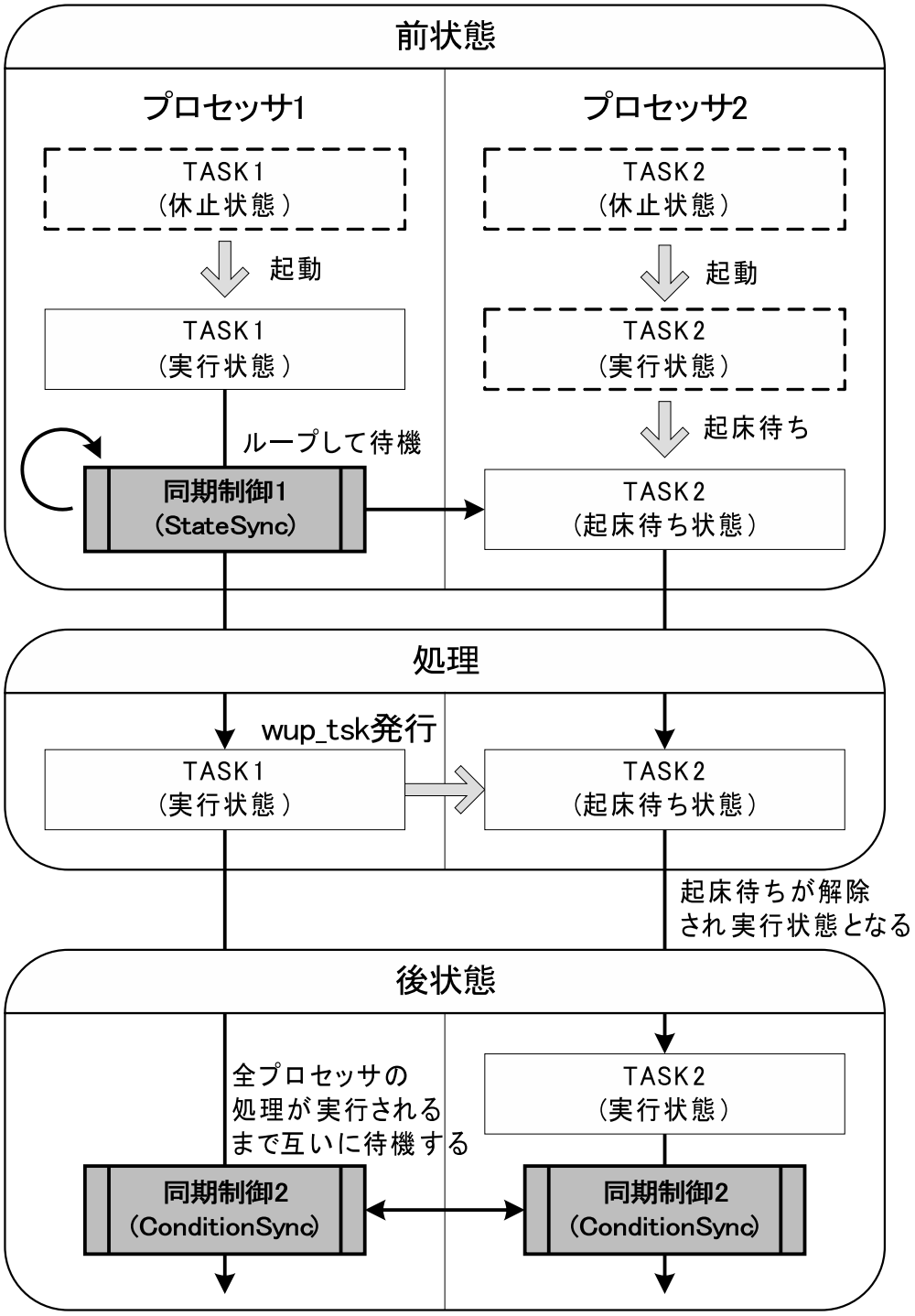


図 4.8: 同期制御を含むテストプログラムの処理フロー

表 4.1: 同期パターンと同期メカニズム

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

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．本研究では，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．

同期パターンと同期メカニズムの一覧を，表4.1に示す．図4.8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を使用する度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

の内，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．

なお，本研究で対象とした FMP カーネルにおいては，表 4.1 に示した同期パターンと同期メカニズムを用いた API テストにより，テスト対象とするソースコードのカバレッジを満たすことができたため，TTG に実装したプロセッサ間同期制御は表 4.1 の内容で十分であると言える．

コンフィギュレーション対応

2.1.2 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み合わせに対応したテストの実施が必要である．本研究では，すべての組み合わ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．

4.4.3 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 [45] を参照のこと．

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 4.1 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 4.9 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．

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 4.9 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．本研究では，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．

拡張 ISS による実行順序依存分岐の網羅

本研究では，前節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用意した．

図 4.9 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

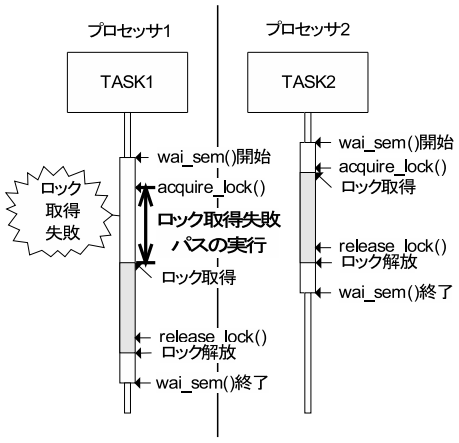


図 4.9: ロック取得失敗パスが実行される例

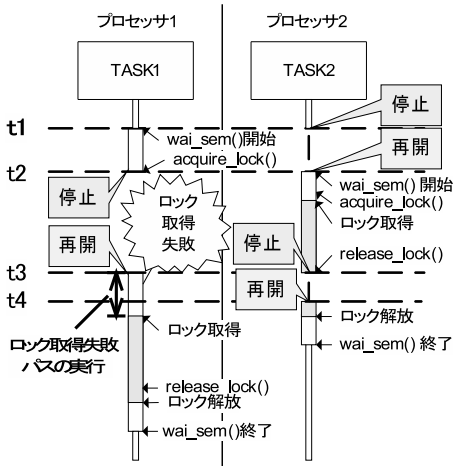


図 4.10: ISS 拡張によるロック取得失敗パスの実行フロー

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 4.10 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 4.1 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 4.1 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 4.1 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．本手法を用いることで，各プロセッサの実行タイミングを，ソース

表 4.2: テスト実施結果一覧

	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ー数	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)

コード上の任意の場所で制御できるようになったことから，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，本研究では，FMP カーネルのテスト環境に使用する ISS として，ARM プロセッサの ISS である SkyEye[30] 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た [46].

4.5　テスト実施結果

本節では，4.3 節で述べたテストプロセスを，4.4 節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 4.2 に示す．FMP カーネルのポリシー数やテストケース数は，FMP カーネルのみに必要なポリシー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)[29] によりコンパイルした結果である．

4.5.1 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，4.2.2 節で想定した通り，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ーフェイ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．本研究で使用した GCC もセミホスティングインターフェイスに対応しており，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．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 [47]．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 4.3 に示す．FMP カーネルに対する全テストケースをまとめて一つのテストプログラムとして生成して実行するには，ターゲットシステムに約 12MB の ROM と，約 1.3MB 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，4.3.2 節で述べたように，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．

4.5.2 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ヶ所，デッドロック回避処理が 8ヶ所の，合計 83ヶ所存在した．実行順序依存分岐テストでは，4.4.3 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ヶ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．

表 4.3: ターゲットシステム一覧

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	GL	NS	3
	NaviEngine	ARM11MPCore	4	LT	PL	NS	9 (※2)
	KZM-CA9-01	ARM11MPCore	4	LT	PL	NS	
	Core Tile for ARM11 MPCore Emulation Baseboard	ARM11MPCore	4	LT	PL	NS	
	NiosII Development Kit Stratix	NiosII	2	LT	PL	ES	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

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ヶ所と少ないため，これらの課題に対応するより，人手で開発したほうが，コストが小さいと判断した．

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ヶ所が網羅されることを確認した．これにより，4.5.1 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．

4.6　検出した不具合の分析

本節では，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．

表 4.4: API テストにおける不具合の種類と件数

	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) 統合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

4.6.1 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．不具合の種類毎に分類した件数の一覧を表 4.4 に示す．

(A)(B) は，4.1 節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言える．(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，4.2.2 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 が仕様通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．

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．

例えば，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．

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テストを一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．これに対し本研究では，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)毎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，明確な判断基準は筆者の知る限り存在しない．ただし，本研究では，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

可能性は極めて低いと考えられる．

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内，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．4.4.2 節で述べたように，すべてのコンフィギュレーションの組み合わ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 2.2 に挙げたターゲットシステム依存テスト (I) として，別途実施する必要がある．

4.6.2 実行順序依存分岐テストで検出した不具合

4.4.3 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 4.11 に，不具合が検出された処理の疑似コードを図 4.12 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

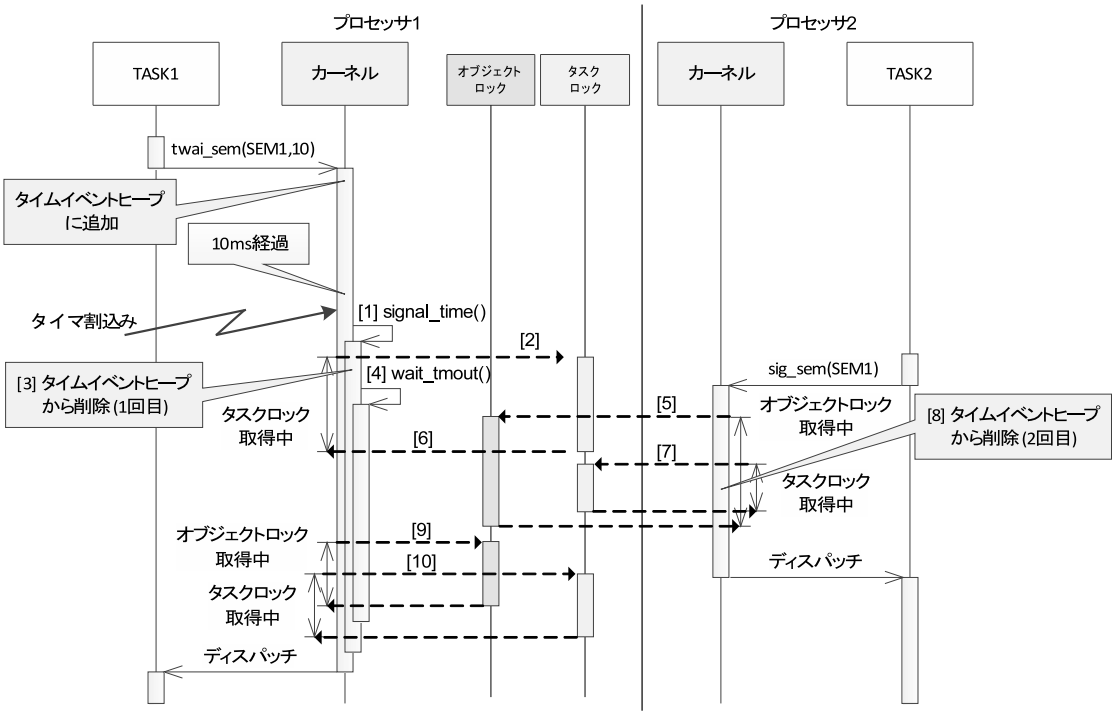


図 4.11: 不具合発生のフロー

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 ミリ秒毎に発生し，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 4.11[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 4.11[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 4.11[3])，タイムイベントを実行する (図 4.11[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 OS 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，OS 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，4.2.2 節で述べたように，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，一旦タスクロックを解放して (図 4.11[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 4.11[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 4.11[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 4.11[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() 内で一旦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 4.11[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 で既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 4.12 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，既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 4.11 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．

4.7 得られた知見と他の RTOS への適用

本節では，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．

```
 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 OS 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:         OS 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 4.12: 不具合が検出された処理の疑似コード

4.7.1 得られた知見

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 件の内，4.4.1 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跨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，プロセッサを跨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 でプロセッサを跨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 当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 当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 つのプロセッサに跨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，プロセッサを跨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 の追加が無い，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．

プログラム生成による効率化

表 4.2 に示した通り，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した．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．本研究で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，4.4.2 節で述べたプロセッサ間同期制御ライブラリは，約 19,000ヶ所登場した．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伴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繋がった．

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．

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コスト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繋がる．さらに，4.6.1 節で述べたように，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．

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，4.2.2 節で述べたように，実行順序依存分岐が存在する．本研究において，4.4.3 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，2.2.1 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，一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．

4.7.2 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，2.1.2 節で述べた統合仕様書に準拠した RTOS に対応している．また，統合仕様書は 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，μ ITRON 仕様と異なる他の RTOS に対しては，利用できない．しかし，4.3 節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．

例えば，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，プロセッサを跨いで発行できることも規定されている．NCES では，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なっている [48]．タスクが取りうる状態や，OS 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．

4.7.3　他のタイプのマルチプロセッサ向け RTOS への適用

2.1.2 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．

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ー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，4.4.2 節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．

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ーは流用できない．特に API 発行前の状態を，全プロセッサに跨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ーも複雑になる．ただし，4.4.2 節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．

4.7.4　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．本研究では，TTG が生成したテストプログラムのレビューに加えて，テストケース毎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 4.5(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，統合仕様書の仕様と 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，統合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．統合仕様書，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．

4.8 関連研究

本節では，RTOS に対するテストの事例について述べる．

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 [49]．本ハンドブックは，シングルプロセッサ向け RTOS に対して実施するべきテスト項目や，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[2] 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%の網羅に留まっており，マルチプロセッサの仕様に対するテストは存在しない．

以上のように，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，例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．

4.9 まとめ

本章では，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ーを策定し，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．

第5章　テストプログラム生成ツールのAUTOSAR OSへの適用

5.1　概要

近年，車載ソフトウェア分野において，開発の効率化，開発コストの削減を目的として，AUTOSAR[2] という標準化されたソフトウェアプラットフォーム (以下，SPF) の導入が，欧州を中心に進んでいる．AUTOSAR は，仕様のみ標準化・公開され，実装は各ソフトウェアベンダが行う．AUTOSAR 仕様では，サービスレイヤ，ECU 抽象化レイヤ，マイクロコントローラ抽象化レイヤという 3 つの階層構造により，アプリケーションとハードウェアを分離し，ハードウェアに依存しないアプリケーションの開発を実現する．各レイヤには，通信やメモリ制御など，各機能に応じたモジュールが 80 以上[1] 定義されている．

それらのモジュールの中で重要な役割を持つモジュールの 1 つとして OS がある．AUTOSAR の OS(以下，A-OS) は，優先度ベーススケジューリングを行う RTOS(AUTOSAR では単に OS と記す) であり，すべてのモジュール，アプリケーションを動作させるベースとなるソフトウェアである．さらに，A-OS では，メモリ保護，タイミング保護といった，機能安全機構を実現するための重要な機能を提供する．

車載ソフトウェアは，高い信頼性が求められるので，すべてのソフトウェアのベースとなる A-OS に対するテストは重要である．しかし，RTOS は，コンテキストスイッチや割込み制御を伴うテストが必要になるので，一般にテスト実施が困難である．RTOS におけるメモリ保護機能では，Linux などの汎用 OS のメモリ保護機能と異なり，ユーザが信頼/非信頼といったメモリ保護設定をコンフィギュレーションで決定できるため，テストプログラムの構造が複雑となる．また，A-OS 仕様特有の理由により，膨大なテストケース数が必要になり，テストプログラムの開発コストが大きく，テスト実行時間も長くなる．

[1] Release 4.2.2 の場合．

これらの課題に対して，テストプログラムを生成するツールを開発し，テストプログラム開発のコスト削減およびテスト実行時間の短縮を実現した．本章では，A-OS に対して実施したテストについて紹介し，テストプログラム開発コスト削減およびテスト実行時間短縮のために開発したテストプログラム生成ツールについて紹介する．

本章の構成は以下の通りである．5.2 節で，テスト対象とする RTOS である A-OS 仕様について述べ，5.3 節では A-OS 仕様に対するテストの課題について述べる．5.4 節で関連研究について述べ，5.5 節で開発したテストプログラム生成ツールについて述べる．5.6 節でツールの評価を行い，5.7 節でまとめる．

5.2　AUTOSAR 仕様

5.2.1　OS

A-OS は，OSEK/VDX OS 仕様 [13] の上位互換として，OSEK/VDX OS 仕様との差分仕様のみ規定されている．A-OS は，優先度ベースのタスクスケジューリングを行う一般的な RTOS であり，タスクスケジューリング，割込み制御，リソース (上限優先度プロトコルによる排他制御)，カウンタ，アラームなどの OS オブジェクトが規定されている．さらに，メモリ保護やタイミング保護など，近年ニーズが高まっている技術に対応する機能も規定されている．A-OS には，4 つのスケーラビリティクラス (SC) が規定されている．

- SC1：OSEK の上位互換の基本セット
- SC2：SC1 ＋タイミング保護
- SC3：SC1 ＋メモリ保護
- SC4：SC1 ＋タイミング保護＋メモリ保護

A-OS のタイミング保護機能には，機能安全を達成するための課題が多いことから，本研究では，まず SC1 およびメモリ保護に対応した SC3 の開発を対象としている．本章において，単に A-OS と記述した場合，SC3 を示す．

OS オブジェクトの定義は静的に行い，コンフィギュレーション情報は AUTOSAR で規定された XML 仕様 (拡張子は arxml) で記述する．XML 仕様は，ECU コンフィギュレーション仕様 [50] で規定されている．コンフィギュレーション情報は GUI

ツールによって設定を行い，arxml ファイルを出力することが一般的である．arxml ファイルで記述されたコンフィギュレーション情報を，OS に付属する OS ジェネレータに入力することで，OS オブジェクトが静的に定義された C ソースコードが生成される．OS ジェネレータによって生成されたソースコードと，静的に実装した OS のソースコードをリンクし，OS が動作する．

5.2.2 メモリ保護機能

A-OS 仕様におけるメモリ保護機能では，タスクやリソースといった OS オブジェクトを，OS アプリケーション (以下，OSAP) というグループに所属させ，OSAP 毎にアクセスが許可された OS オブジェクトやメモリ領域にのみアクセスを可能とすることで，保護を実現する．なお，RTOS 上でプログラムを実行するタスクや割込みサービスルーチン (以下，ISR) といった OS オブジェクトを，本章では処理単位と呼ぶ．A-OS 仕様におけるメモリ保護機能は，以下に説明する API 保護とアクセス保護の 2 種類に分類される．

API 保護

API 保護は，OS から提供される API を，ある OSAP の処理単位から，異なる OSAP に所属する OS オブジェクトに発行した際に実行されるメモリ保護機能である．ある OSAP に所属する処理単位から，ある OS オブジェクトに対して API を発行すると，OS がその OS オブジェクトに対してアクセス権があるかをチェックし，アクセス権がない場合，API の処理を実行しない．また，引数にポインタを渡してデータを受け取る API の場合，API 呼び出し元の処理単位がポインタの指し示すアドレスへ，データを書き込むアクセス権があるかを OS がチェックし，アクセス権がない場合，API の処理を実行せず，不正なデータの書き込みを防止する．同一 OSAP に所属する OS オブジェクト同士はアクセス権が与えられるが，これとは別に，OS オブジェクト毎にアクセス可能な OSAP を，コンフィギュレーション時に指定することで，他の OSAP の OS オブジェクトに対してもアクセス権を与えることができる．アクセスが許可されていない OS オブジェクトやアドレスに対して API を発行した場合，違反を検出した結果として，OS がエラーフックという処理単位を実行する．なお，それぞれの OSAP に対して，OSAP 固有のエラーフック，OSAP 固有のスタートアップフック (OS 起動時に呼び出される)，OSAP 固有のシャットダウンフック (OS 終了時に呼び出される) を定義することができる．

アクセス保護

OSAP には，信頼 OSAP と非信頼 OSAP の 2 種類がある．アクセス保護は，API を使用することなく，非信頼 OSAP から，他の OSAP に所属する変数や関数などのメモリ領域へ直接アクセスした際に実行されるメモリ保護機能である．信頼 OSAP に所属した処理単位は一切のアクセス制限を受けず，すべてのメモリ領域にアクセスでき，非信頼 OSAP に所属した処理単位は，所属した OSAP に許可されたメモリ領域にのみアクセスできる．OS は，信頼 OSAP は特権モードで，非信頼 OSAP はユーザモードで動作するよう，プロセッサを制御する．非信頼 OSAP の処理単位から，許可されていないメモリ領域に対してアクセスを行うと，MPU(Memory Protection Unit) によってアクセス権があるかがチェックされ，アクセス権がない場合，アクセスが拒否され，CPU 例外が発生する．アクセス権のないメモリ領域へアクセスを行い，MPU による CPU 例外が発生することを，アクセス保護違反と呼ぶ．

A-OS 仕様には，MPU を用いて非信頼 OSAP の処理単位からの不正なアクセスを保護することは，要求仕様として規定されているが，具体的に MPU をどのように使用するかや，コンフィギュレーション方法などは一切規定されていない．これは，プロセッサによって MPU で設定できるメモリ領域数や設定方法が様々であり，標準化が困難であるためと考えられる．

アクセス保護違反が発生すると，違反を検出した結果として，OS がプロテクションフックという処理単位を実行する．OS のユーザは，プロテクションフックの中で，状況に応じた復帰処理やシャットダウンを行うことができる．プロテクションフックでは，プロテクションフックの戻り値によって，OS の振る舞いを指定することが可能である．また，プロテクションフックを呼び出さないように設定することも可能であり，この場合，アクセス保護違反が発生すると，OS をシャットダウンする．これらのアクセス保護違反が発生した際に行う処理のことを，保護違反時処理と呼ぶ．

信頼関数

A-OS には，信頼関数という，ユーザモードで動作する非信頼 OSAP から特権モードへ切り替えて，ユーザ定義の関数を実行するための機能が用意されている．信頼関数は，信頼 OSAP にのみ所属することができ，信頼関数から実行される処理は，信頼関数が所属する OSAP に対する API 保護，アクセス保護が適用される．

```
  :
#define OS_START_SEC_CODE
#include "Os_MemMap.h"
void Function(void) {
  :
}
#define OS_STOP_SEC_CODE
#include "Os_MemMap.h"
  :
```

図 5.1: OS ソースコードの実装例

5.2.3 メモリマッピング

AUTOSAR では，コンパイラやプロセッサを抽象化して実装するための取り決めが多く規定されている．例えば，各実装の text 領域，data 領域，bss 領域などを，どのセクションに配置するかを，コンパイラに依存しない形で実装するための仕組みが，メモリマッピング仕様として規定されている [18]．メモリマッピングを使用した OS のソースコードの実装例を図 5.1 に示す．OS のソースコードにおいて関数の実装を開始する前に，OS_START_SEC_CODE といったマクロ (以下，セクション指定マクロ) を定義し，Os_MemMap.h を include する．そして，関数を記述した後に，OS_STOP_SEC_CODE というセクション指定マクロを定義し，再度 Os_MemMap.h を include する．

Os_MemMap.h では，直前に定義されたマクロに応じたセクションに配置されるよう，pragma 命令などを使用した記述を行う．GHS コンパイラ [51] に対する Os_MemMap.h の例を図 5.2 に示す．Os_MemMap.h は，AUTOSAR を搭載する ECU 開発時に，使用するコンパイラに応じて，ユーザが用意するため，コンパイラに依存したソースコードとなる．

この仕組みにより，OS のソースコードはコンパイラに依存せずに開発した上で，ECU 開発時に，任意のセクションに各領域を配置することが可能となる．

5.2.4 標準化の対象外ファイル

RTOS において，以下にあげるファイルは重要な役割を持つが，AUTOSAR では標準化されていない．

- リンカスクリプト

```
  :
#if defined(OS_START_SEC_CODE)
#pragma ghs section text=".os_text"
#undef OS_START_SEC_CODE
#undef MEMMAP_ERROR

#elif defined(OS_STOP_SEC_CODE)
#pragma ghs section text=default
#undef OS_STOP_SEC_CODE
#undef MEMMAP_ERROR
  :
```

図 5.2: Os_MemMap.h の例

```
<前状態>
  TASK1(低優先度) が実行状態
  TASK2(高優先度) が休止状態
<処理>
  TASK1 が ActivateTask(TASK2) を呼び出す
  ActivateTask がエラーコード E_OK を返す
<後状態>
  TASK1 が実行可能状態へ遷移する
  TASK2 が実行状態へ遷移する (プリエンプト)
```

図 5.3: テストシナリオの例

- 例外ベクタテーブル
- スタートアップルーチン

これらは ECU 開発時に用意することになっているため，A-OS として提供するものではない．しかし，これらは OS 側で意図した内容となっていることが求められるので，OS の実装に準じた制約が課せられる．特にリンカスクリプトはメモリ保護機能を実現する上で重要なファイルである．

5.3 AUTOSAR OS に対するテストの課題

本節では，RTOS に対する基本的なテストシナリオを説明し，A-OS に対するテストケース数および，テスト実行時間が膨大になってしまう理由について説明する．

5.3.1 基本的なテストシナリオ

RTOS は，API 呼び出しや，割込み発生など，あるアクションをトリガとして，タスクの状態などの OS オブジェクトの状態や割込み禁止状態などの OS 内部状態 (以下，システム状態) が変化する．したがって，RTOS に対するテストシナリオは，あるシステム状態で，何らかのアクションを発生させ，仕様通りのシステム状態へ遷移することを確認することがベースとなる．例えば，休止状態のタスクを起動する ActivateTask という API をテストするテストシナリオの例を図 5.3 に示す．この例では，休止状態の TASK2 が，実行状態の TASK1 より優先度が高いため，ActivateTask 発行により TASK2 が起動し，プリエンプトが発生し，TASK2 が実行状態，TASK1 が実行可能状態となることをテストしている．API の呼び出しに限らず，このテストシナリオを用いて,「処理」を割込み発生や，メモリ保護違反発生などのアクションに置き換えることで，OS の様々な振る舞いをテストすることが可能である．テストケースによっては，複数の「処理」「後状態」が必要となる場合がある．

5.3.2 膨大なテストケース数

A-OS では，OS オブジェクトに設定可能なパラメータが多く，1 件の要求仕様を網羅的にテストするには，多くのテストケースを抽出する必要がある．図 5.3 は,「TASK1 が，TASK2 に対して ActivateTask を発行した場合，TASK2 が起動する」という要求仕様をテストするテストシナリオの 1 つの例である．このテストシナリオにおいて，考慮するべきパラメータ，およびパラメータの組み合わせは以下となる．

タスク優先度

A-OS では，タスクごとにタスク優先度を指定できる．図 5.3 の例では，TASK1 を低優先度，TASK2 を高優先度としているため，ActivateTask により TASK2 を起動後，TASK2 が実行状態となっている．もし，TASK1 と TASK2 の優先度が逆であれば，TASK2 は実行可能状態になるのみで，TASK1 は実行状態を継続する．このように，TASK1 と TASK2 の優先度の組み合わせにより，異なるテストケースが必要となる．同値分割の観点から，少なくとも以下の 3 パターンの実施は必要である．

- TASK1 < TASK2
- TASK1 = TASK2
- TASK1 > TASK2

スケジューリングポリシー

A-OS では，タスクごとにスケジューリングポリシーとして，フルプリエンプティブかノンプリエンプティブかを指定できる．図 5.3 のテストシナリオでは，図には記載はしていないが，TASK1 がフルプリエンプティブであることを前提としているので，TASK2 が起動することで，プリエンプトが発生している．もし，TASK1 がノンプリエンプティブであれば，TASK1 が実行状態であり続けることを確認する必要がある．

タスク状態

A-OS では，タスクは，実行状態，実行可能状態，待ち状態，休止状態の 4 つの状態を持つ．図 5.3 のテストシナリオでは，TASK2 を休止状態としているが，ActivateTask は実行可能状態のタスクにも発行可能である．また，実行状態である TASK1 自身に対して，ActivateTask を発行することも可能である．

タスク起動要求回数

A-OS では，タスクに対して多重に起動要求を行うことができる．よって，実行可能状態のタスクに対して ActivateTask を発行することで，多重に起動要求されたことの確認も必要である．

タスク種別

A-OS では，タスクに対してタスク種別として基本タスクか拡張タスクかを指定できる．基本タスクは，待ち状態に移行しないが，多重起動要求を行うことができる．拡張タスクは，待ち状態に移行できるが，多重起動要求を行うことができない．ActivateTask の振る舞いには関係ないが，網羅性の観点では，ActivateTask を発行するタスク，発行されるタスクが，それぞれ基本タスクである場合と拡張タスクである場合をテストするべきである．

所属する OSAP

5.2.2 節で述べた通り，A-OS には，API 保護があるので，図 5.3 のテストシナリオで，TASK1 から TASK2 に対して，ActivateTask を発行できるということは，TASK1 から，TASK2 に対してアクセス権が付与されているということである．アクセス権を付与するには，TASK1 と TASK2 が同じ OSAP に所属するか，TASK2 に対して，TASK1 が所属する OSAP からのアクセス権が付与されているかの 2 パターン存在する．さらに，各タスクの所属する OSAP が，信頼か非信頼かの組み合わせもある．

呼び出し元の処理単位

A-OS では，ActivateTask を呼び出せる処理単位として，タスクと ISR がある．また，信頼関数経由での呼び出しも考慮するべきである．これらの処理単位を呼び出し元として，正しく振る舞うことの確認が必要である．

上記のようなパラメータの組み合わせを，PictMaster[52] というテスト組み合わせツールを用いて，生成を行ったところ，ActivateTask に対するテストケース数は，794 件となった．また，不正な条件で ActivateTask を呼び出した場合に，正しいエラーコードが返ることの確認を行うテストケースを含めると，823 件となった．メモリ保護機能のない SC1 では，ActivateTask に対して同様の組み合わせを行うと 170 件であったので，メモリ保護機能が増えることにより，いかにテストケース数が増えるかが分かる．A-OS 仕様の中で，本研究で対象とした A-OS の実装でサポートされた API は 38 個であり，すべての API に対して，同様にテストケースを作成したところ，テストケースは，23,434 件となった．なお，直交表などの観点でテストケース数を削減することも考えられるが，RTOS の実装は，一般的なソフトウェアと比較して，構造が複雑であるため，可能な限り，様々な組み合わせでテストを実施することが望ましい．

5.3.3 開発コスト

本節では，A-OS のテストに必要なテストプログラムなどの開発コストについて述べる．

数万件のテストプログラムをスクラッチで開発する場合，テストシナリオの実現方法を統一することが困難であり，レビューや修正作業の負担が大きくなることが予想される．また，数万件のテストケースを実現するテストプログラムを，スクラッチで開発できたとしても，仕様変更などでテストプログラムを修正する場合，影響範囲の分析や，修正箇所の特定が困難である．当然，保守性も著しく低い．また，A-OS では，テストプログラム開発に加え，コンフィギュレーションファイル (arxml) も，テストケースごとに作成する必要がある．AUTOSAR のメソドロジでは，一般的に arxml ファイルは GUI ツールを用いて生成するため，テストケースごとに作成するのは膨大な時間を要する．テストシナリオを修正する場合，変更内容に合わせて arxml ファイルの修正も必要であるので，この観点でも保守性が低い．

OS のソースコードからは，5.2.3 節で説明した Os_MemMap.h が include されるので，テストにおいても Os_MemMap.h が必要となる．A-OS 仕様では，割込みベクタテーブルを，ユーザが意図する任意のセクションに配置できるように実装することを規定している．同様に，OS の内部データや，OS・タスクのスタック領域も，ユーザ側で意図したセクションに配置できるように，それぞれ異なるセクション指定マクロが使用されるべきである．つまり，タスクや ISR の構成によって，必要なセクション指定マクロが変わるため，テストケースごとに異なる Os_MemMap.h が必要となる．

さらに，ポータビリティの観点でも課題がある．arxml ファイルには，スタック領域などのメモリ保護に関連する情報が記述されており，これらはプロセッサに依存するので，異なるプロセッサに対してテストする場合，プロセッサごとに arxml ファイルを用意する必要がある．また，Os_MemMap.h は，コンパイラに依存するため，異なるコンパイラに対してテストする場合，Os_MemMap.h をコンパイラごとに用意する必要がる．

5.3.4 テスト実行時間

A-OS の処理単位に，保護違反時に呼び出されるプロテクションフックがある．プロテクションフックからも，いくつかの API を呼び出すことが可能であるため，プロテクションフックから API を呼び出すテストケースが必要である．プロテクションフックを任意の処理単位から起動するには，CPU 例外を発生させる必要があるが，A-OS 仕様では，CPU 例外によってプロテクションフックが起動してしま

うと，元の処理単位に戻ることができず，OS シャットダウンとなる．また，A-OS 仕様では，OS シャットダウン時，無限ループすることが規定されている．

5.3.1 節で述べた 23,434 件のうち，プロテクションフックが登場するテストケースは 92 件であった．1 回のプロテクションフック起動により，すべてのテストケースを実施することで，1 つのテストプログラムとして実現する方法も考えられるが，テストシナリオによっては実現できない．例えば，実行中の OSAP を返す GetApplicationID という API をテストする場合，プロテクションフック起動前に動作していた OSAP を，信頼 OSAP と非信頼 OSAP の両者で実施する必要がある．つまり，信頼 OSAP 動作中にプロテクションフックが起動するテストケースと，非信頼 OSAP 動作中にプロテクションフックが起動するテストケースは，1 つのテストプログラムでは実現できない．また，メモリ保護違反が正しく検出されることをテストするような場合，様々な条件でメモリ保護違反を発生させ，プロテクションフックが起動することを確認するため，やはり 1 つのテストプログラムでは実現できない．

したがって，プロテクションフックが登場するテストケースは，テストケースごとに 1 つのテストプログラムで実現する必要がある．しかし，テストプログラムを分けて開発する場合，1 件ずつ個別にビルド・ロード・実行する必要があるため，テスト実行時間が膨大になってしまう恐れがある．

5.3.5 メモリ保護機能による制約事項

5.2.2 節で述べた通り，非信頼 OSAP には様々なアクセス制限がある．本節では，テストプログラム実現にあたり，弊害となる非信頼 OSAP のアクセス制限について説明する．

システム状態チェック

5.3.1 節で述べたテストシナリオの「後状態」では，実行状態の処理単位から，システム状態が期待通りに遷移したことを確かめる必要がある．システム状態の確認方法として，OS が提供する各 OS オブジェクトの状態を参照する機能を持つ API を用いる方法があるが，A-OS 仕様で規定されている API で得られる情報は限定的であるので，テスト用にシステム状態を確認するための API(以下，状態確認 API) を用意する必要がある．

しかし，非信頼 OSAP から状態確認 API を発行しても，OS 内部データに対するアクセス権がないため，メモリ保護違反が発生してしまうという問題がある．5.2.2 節で述べた信頼関数を使用して状態確認 API を呼び出す方法もあるが，信頼関数は割込み禁止状態で呼び出せないという制約があるため，テストシナリオによっては，使用ができない問題がある．

レジスタアクセス

A-OS に対するテストでは，以下のようなレジスタアクセスが必要となる．まず，テスト結果を確認するために，実行結果を何らかの方法で出力して確認する必要がある．本研究では，UART を用いて，ホスト PC へログメッセージを送信する方法を取っている．このためには UART レジスタへのアクセスが必要である．次に，意図したタイミングで意図した処理単位を起動するために，テストプログラムから割込みや CPU 例外を強制的に発生させる必要がある．これらの発生には，割込み制御レジスタなどを操作する必要がある．

このように，テストプログラムからは，いくつかのレジスタを操作する必要があるが，非信頼 OSAP から実行する場合，レジスタに対するアクセス権を付与する必要がある．しかし，MPU の領域数は有限であり，一般的に数が少ないため，これらのレジスタ領域を共有領域として設定してしまうと，MPU の領域が足りなくなる恐れがある．また，プロセッサによっては，書き込みアクセス権が MPU に設定されていても，ユーザモードでは変更できないレジスタが存在するため，単にアクセス権を付与するだけでは解決できない問題もある．

5.4 関連研究

5.3 節で述べた課題を解決するには，スクラッチでテストプログラムを開発するのではなく，ツールなどにより，テストプログラムを生成する必要があると考えた．

オープンソースの RTOS である FreeRTOS[34] に対しては，テストケースおよびテストプログラムを生成するツールである STG[35] を利用して，テストを実施した研究がある [36]．しかし，この研究では，テストケースの生成のみで，テストプログラムの生成には言及していない．また，テストケースの生成に関しても，テストの目的を RTOS の割込み処理に関する仕様に限定しており，RTOS 全体はテス

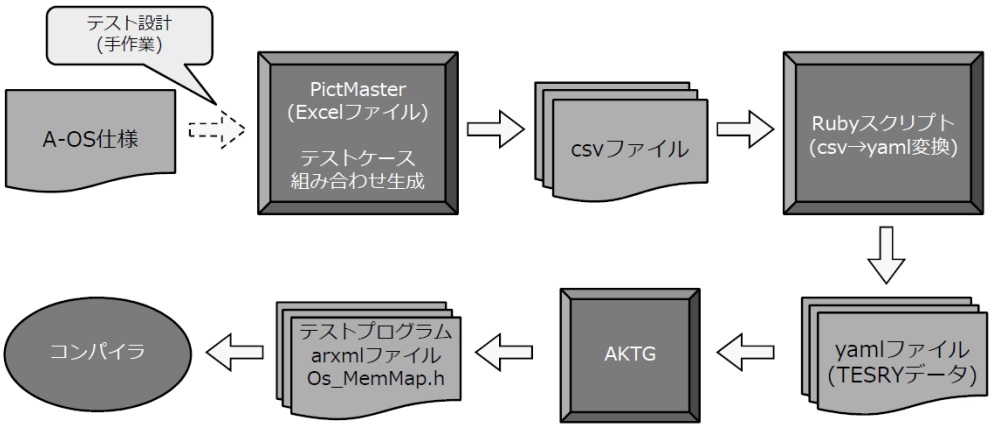


図 5.4: アプローチの全体像

ト対象としていない．また，STG で生成可能なプログラミング言語は C++と Java である．

テストプログラムの生成に関する研究も存在するが，対象とするソフトウェアが，通信プロトコルや状態遷移系のシステムなど，特定の分野に特化されており，RTOS のテストに適用できない [53][54]．特に，入力データとして定義した，API 発行前の RTOS のシステム状態を実現する処理のような，RTOS の API に特化したテストプログラムの生成が可能なツールの事例はない．また，既存のツールでは出力するプログラミング言語が限定されており，arxml ファイルや Os_MemMap.h など，A-OS 仕様に対応したファイル生成は困難である．

以上から，本研究では，5.3 節で述べた課題を解決するために，第 3 章で述べた TTG をベースとして，A-OS 仕様に対応したテストプログラム生成ツールを開発することにした．

5.5　テストプログラム生成ツール

本節では，5.3 節で述べた課題を解決するためのアプローチについて説明する．アプローチの全体像を図 5.4 に示す．

5.5.1　TESRY 記法

テストプログラムや arxml ファイル，Os_MemMap.h は，手動で開発せず，テストシナリオを抽象的に表現したデータから，テストプログラムを生成するツー

```
pre_condition:
  TASK1:
    type   : TASK
    curpri : LOW
    tstat  : RUNNING
    ttype  : AK_BASIC
    spolicy: AK_SCHFULL
    osap   : NON_TRUSTED_OSAP1
  TASK2:
    type   : TASK
    inipri : HIGH
    tstat  : SUSPENDED
    ttype  : AK_BASIC
    spolicy: AK_SCHFULL
    osap   : NON_TRUSTED_OSAP2
    accosap:
      - NON_TRUSTED_OSAP1

do:
  id     : TASK1
  syssrv : ActivateTask(TASK2)
  rettype: StatusType
  retval : E_OK

post_condition:
  TASK1:
    tstat: READY
  TASK2:
    tstat : RUNNING
    curpri: HIGH
```

図 5.5: A-OS 仕様に対応した TESRY 記法の例

ルを開発することで対応する．本研究では，第 3 章で μITRON ベースの RTOS 向けに考案した TESRY 記法を，A-OS 仕様向けに適合させた．A-OS 仕様に適合させた TESRY 記法の例として，図 5.3 を，TESRY 記法で記述した例を図 5.5 に示す．A-OS 仕様に適合させた TESRY 記法の詳細は，6.4.1 節を参照のこと．なお，TESRY 記法で記述されたファイルを，TESRY データと呼ぶ．

5.5.2　AKTG

本研究では，TESRY データを入力として，テストプログラム，arxml ファイル，Os_MemMap.h を生成するツールを開発した．ツールは，AKTG(Automotive Kernel Test Generator) と命名した．

AKTG は，複数の TESRY データを入力可能とし，入力されたテストシナリオをマージした 1 つのテストプログラムを生成する．これにより，ユーザが，AKTG に入力する TESRY データの数を調整することで，ターゲットプロセッサの ROM/RAM サイズに合わせて，最適なテストプログラムサイズを制御できるようになる．AKTG は，TTG と同様に，大きくフロントエンドとバックエンドで構成する．フロントエンドとバックエンドの構造については，3.5.3 節を参照のこと．

AKTG は，テストプログラムに加えて，テストシナリオに登場する OS オブジェクトを AUTOSAR 準拠の arxml で生成する．さらに，AKTG は，プロセッサやコンパイラに依存した定義を記述した設定ファイルも入力とし，ROM/RAM 領域のアドレスのようなプロセッサに依存する情報は，入力した設定ファイルにしたがって出力する．これにより，プロセッサやコンパイラを変更してテストを実施する場合，設定ファイルだけを修正すればよい．

5.5.3　PictMaster による TESRY データの生成

PictMaster は，組み合わせたテストケースの期待値も生成することができる．本研究では，PictMaster[2] での組み合わせ結果を，TESRY データへ変換可能な csv ファイルで出力するように，PictMaster を改造した．出力した csv ファイルを TESRY データへ変換し，AKTG への入力とした．PictMaster で生成した TESRY データは，23,434 個存在するが，テストを実行するプロセッサの ROM/RAM サイズに応じて，適切な数に分けて AKTG への入力とすることで，テストプログラムのサイ

[2]PictMaster は VBA で実装されている．

```
pre_condition:
  PROTECTION_HOOK1:
    type    : PROTECTION_HOOK1
    hookstat: AK_ACTIVATE

do:
  id    : PROTECTION_HOOK1
  syssrv: return(PRO_SHUTDOWN)
  finish: true

post_condition:
```

図 5.6: OS シャットダウンを伴う TESRY データの例

ズが可能である．なお，PictMaster による TESRY データの生成の詳細については，6.4.1 節を参照のこと．

5.5.4 強制マージ

プロテクションフックが起動するなど，OS シャットダウンを伴うテストケースを効率的に実行するために，AKTG にテストシナリオを強制マージする機能を設けた．強制マージが必要な場合，TESRY 記法において，OS シャットダウンが発生するトリガとなる do に対して finish というパラメータを true と指定する (図 5.6)．do に finish を指定した TESRY データを入力する場合は，AKTG 実行時に強制マージオプションを指定することで，各テストシナリオにおいて，finish が指定された do の処理の直前に，ソフトウェアリセットを発生させて，リセットベクタにジャンプするテストプログラムを生成することにした．しかし，単にリセットベクタに飛んだだけでは，毎回 1 つ目のテストシナリオが繰り返し実行されてしまう．何回目のリセットからの実行かは，以下の方法で区別する．まず，スタートアップルーチンで初期化しない RAM 領域に，実行回数とその反転値をそれぞれ保存する．反転値が一致しない場合を初回と見なし，実行回数と反転値を初期化し，以後，実行回数と反転値をセットで更新していくことで，何回目の実行かを判断する．実装のイメージを図 5.7 に示す．

強制マージによる実行時間の短縮は，AKTG のコード生成機能というよりは，テストプログラム構造の工夫である．しかし，強制マージ向けのテストプログラムを手動で開発する場合でも，テストケースの増減に対する保守性や，ターゲットプロセッサの ROM/RAM サイズに応じたテストプログラムサイズの制御が必要で

```
uint32 bootcnt;
uint32 bootcnt_inv;
  :
sint32 main(void) {
    if (bootcnt != ~(bootcnt_inv)) {
        bootcnt = 0;
    };
    bootcnt++;
    bootcnt_inv = ~(bootcnt);
    :
}
```

図 5.7: 実行回数の実装イメージ

あり，やはり，AKTG によるコード生成が必要である．

5.5.5　テストライブラリ

本節では，5.3.5 節で述べた非信頼 OSAP への制限を解決するためのテストライブラリについて述べる．AKTG で生成したテストプログラムからは本テストライブラリを使用する．

システム状態の確認

5.3.5 節で述べた通り，AKTG が生成するテストプログラムでは，状態確認 API が必要である．本研究では，各 OS オブジェクトの状態を参照して，TESRY データに記載されたシステム状態と一致するか確認する API を実装した．意図したシステム状態と異なる状態であった場合は，エラーを出力し OS シャットダウンを行う．

非信頼 OSAP に所属する処理単位から，OS の内部データにアクセスできない問題に対しては，以下の方法で対応した．5.2.4 節で述べた通り，例外ベクタテーブルは，テスト対象の OS ソースコードに含まれないため，テスト用に用意することが可能である．具体的には，trap 命令などによりユーザモードから例外を発生させ，特権モードに切り替え，意図した処理を実行する API を用意する．例えば，システム状態を確認する直前に，OS 内部データを読み出し許可とする MPU 設定を行い，システム状態確認後に，MPU 設定を元に戻すことで，非信頼からでも状態確認 API を発行可能となる．実装のイメージを図 5.8 に示す．AktgMakeTaskInfo で，TESRY データに記載されたタスクの状態を，vtsk という構造体に格納し，AktgVerifyTask

```
AktgStartCheckOsState();  /* start to share */

/* pre_condition */
AktgMakeTaskInfo(&vtsk, RUNNING, 12, ...);
ercd = AktgVerifyTask(TASK1, &vtsk);
CheckAssert(ercd == E_OK);
ercd = AktgVerifyTaskRelRes(TASK1, ...);
CheckAssert(ercd == E_OK);
ercd = AktgVerifyTaskGetRes(TASK1, ...);
CheckAssert(ercd == E_OK);

AktgStopCheckOsState();  /* stop to share */
```

図 5.8: 非信頼 OSAP から状態確認 API を実行する実装イメージ

で OS の内部状態と一致しているか確認する．この処理の前後に，OS 内部データを一時的に読み出し許可にする MPU 設定を行う関数である AktgStartCheckOsState，AktgStopCheckOsState を呼び出している．なお，コンフィギュレーションにより OS 内部データを一律読み出し可能と設定する方法もあるが，非信頼から不正に OS 内部データを参照してしまうような不具合を検出できなくなるため，本方法を採用した．

レジスタアクセス

テストシナリオには，割込み発生によって起動する ISR，CPU 例外によって起動するプロテクションフックなどの処理単位が登場する．前状態で ISR が実行中のテストシナリオを実現するためには，意図したタイミングで割込みや CPU 例外を発生させる必要がある．そこで，意図的に割込みや CPU 例外を発生させる API を用意し，ISR やプロテクションフックの起動が必要なタイミングで，これらの API を呼び出す．具体的には，以下の API が必要となる．

- ISR 起動 API
- プロテクションフック起動 API
- ソフトウェアリセット発生 API

5.3.5 節で述べたように，上記のような API を実現するには，非信頼 OSAP 実行中に以下のレジスタへのアクセスが必要となる．

- 割込み発生用レジスタ
- CPU 例外発生用レジスタ
- ソフトウェアリセット発生用レジスタ

これらのレジスタ操作を行う関数の実現方法は，ターゲットプロセッサに依存するため，テストライブラリの構成を，プロセッサ依存部/非依存部に分けて実装し，ポータビリティを向上した．テスト対象のプロセッサの MPU の領域数に余裕があれば，これらの領域をすべて共有しておくことでも問題はない．しかし，本研究におけるテストで使用したプロセッサでは，MPU の領域数に余裕がなく，割込み発生用レジスタは，ユーザモードでは変更できない仕様であったため，状態確認 API 同様，trap 命令でユーザモードから例外を発生させ，特権に切り替えた上で，意図した処理を実行する API を用意した．なお，テスト結果を出力するための UART レジスタは，テストプログラムの全範囲で使用するため，MPU の領域を 1 つ使用して，UART レジスタを共有することで対応した．

5.6　評価

本節では，商用 A-OS である JulinarOs に対して，AKTG を使用してテストした結果について評価する．本研究では，AKTG による開発コスト，テスト実行時間削減の評価のために，JulinarOs が提供する API に対して，5.3.2 節で述べた 23,434 件のテストを実施した．テスト実行環境は以下である．

- A-OS: APTJ [17] JulinarOs V1.2.0
- コンパイラ: GHS(Version 2015.1.7)
- プロセッサ: Renesas RH850/P1M
- 評価ボード:
  Renesas RH850/P1x-100-pin PiggyBack board + RH850/X1x MainBoard
- デバッガ: Renesas E1 Debugger
- 開発環境: Renesas CS+
- ホスト PC: Windows 8.1 Pro

5.6.1 開発コストの削減

図 5.5 の TESRY データの行数は，26 行である．この TESRY データを入力として，AKTG でテストプログラムを生成すると，テストプログラム，arxml ファイル，Os_MemMap.h の総行数は 1,693 行となった．本研究のテスト実施環境では，プロセッサの ROM/RAM サイズの制約から，1 回に実行可能なテストケース数は 500 件程度であった．1 件〜500 件まで，段階的にテストケース数を変化させた場合の TESRY データの行数と，生成ファイルの総行数の比較を図 5.9 に示す．なお，テストケースにより，TESRY データの行数，テストプログラムの行数にばらつきがあるため，テストケース数分をランダムに選択したテストケースで測定している．対象とするテストケース数が大きくなるほど，TESRY データの行数に対して生成するファイルの総行数が増えていることから，実施するテスト件数が大きいほど，開発コスト削減の効果が高くなることが分かる．なお，23,342 件すべてのテストケースを実行するためのファイル総行数は，約 600 万行であり，スクラッチで開発することは現実的ではなく，本アプローチでなければ，実施できない規模である．TESRY データは，可読性や保守性が高いことに加え，生成したテストプログラムは，属人性がないため，レビュー時間も短縮可能である．

5.6.2 テスト実行時間の削減

5.3.4 節で述べたシャットダウンを伴うテストケースである 92 件を，1 件から段階的に実行した際の，強制マージ使用時と個別に実行する場合のテスト実行時間の比較を図 5.10 に示す．個別に実行した場合，デバッガへの接続，ロード，実行，切断がテストケース 1 件ごとに発生するため，実行件数が多くなるほど，実行時間が長くなっている．強制マージの効果は，十分に高いことが分かる．

5.6.3 有効性

本テストでは，数多くの不具合を検出したが，特定の処理単位でしか発生しない不具合が存在しており，テストを削減するのではなく，効率的に実施するアプローチの有効性を確認した．

本研究では，いくつかのプロセッサに JulinarOs のポーティングを実施したが，プロセッサに依存した AKTG の設定ファイルおよびテストライブラリを修正することで簡単に全テストを実施可能であることを確認した．ポーティングする度に

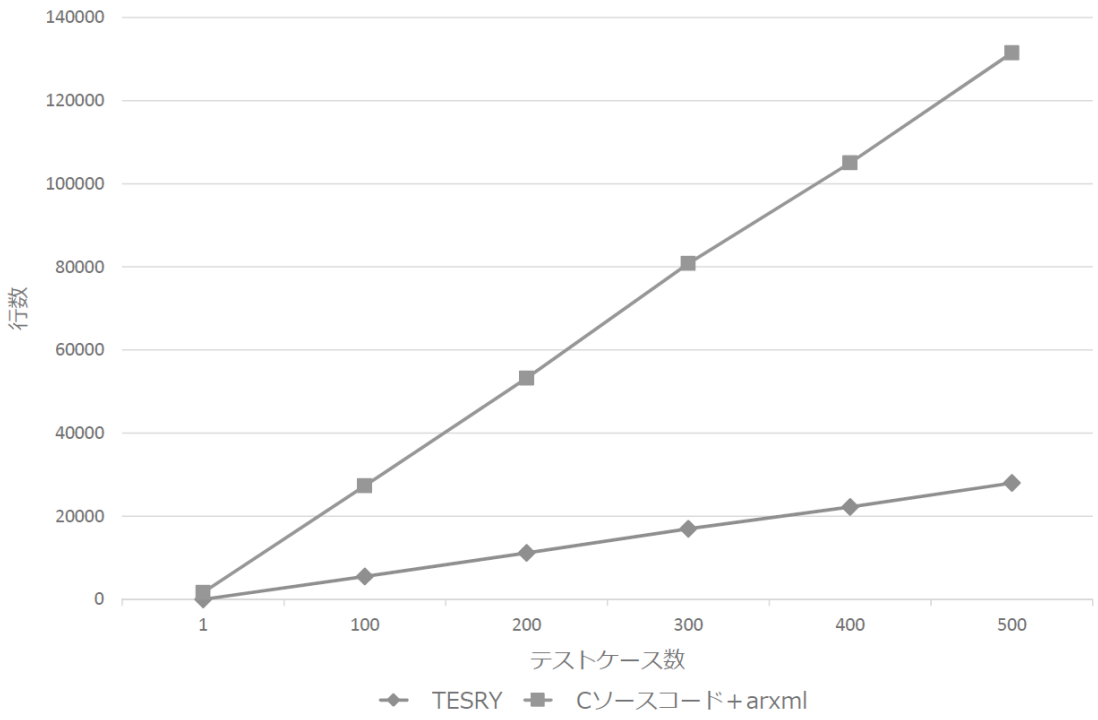


図 5.9: TESRY データの行数と C ソースコード+arxml の行数の比較結果

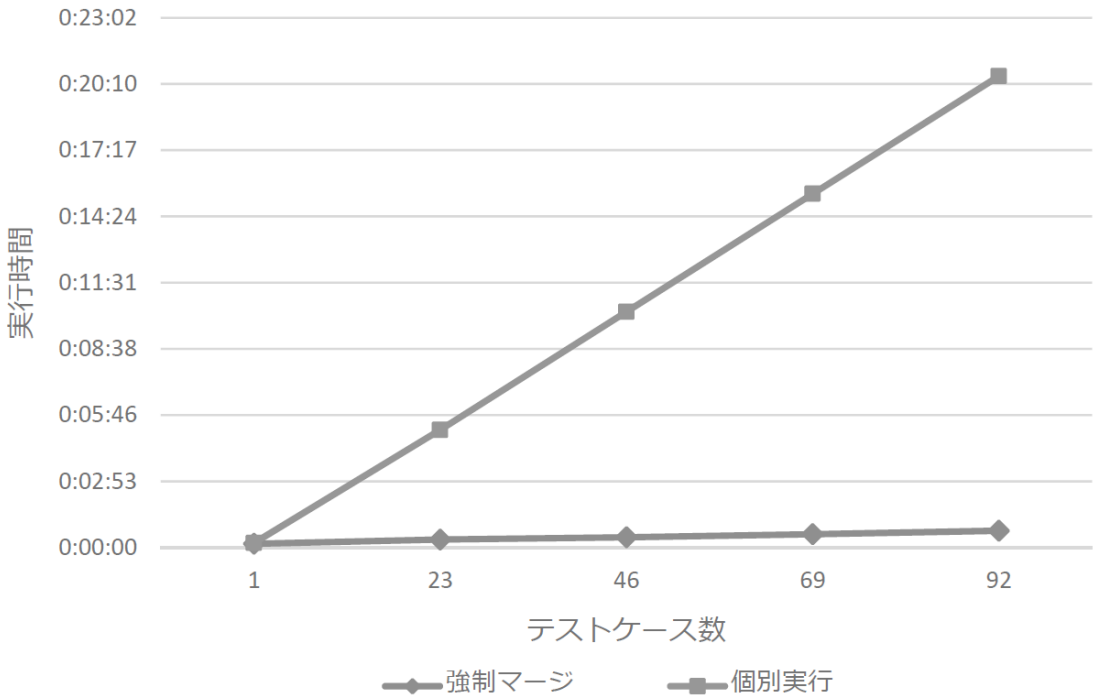


図 5.10: 強制マージ使用時と個別実行時のテスト実行時間の比較結果

リグレッションテストが容易に実施可能であることは，RTOS 開発において非常に有用である．

また，テストのメトリクス評価のために，カバレッジを取得する際，カバレッジ取得に必要なコードをテストプログラムに埋め込む必要があったが，AKTG を修正することで容易に対応可能であった．さらに，JulinarOs だけでなく，オープンソースの A-OS である TOPPERS/ ATK2[14] に対しても，AKTG を使ってテスト可能であることを確認した (第 6 章を参照のこと)．

以上から，本研究のアプローチは再利用性や拡張性の面でも優れていると言える．

5.7 まとめ

本章では，メモリ保護機能に対応した A-OS のテスト実施における課題を明確にし，その解決手法を提案した．テストプログラム開発コストおよび，テスト実行時間の問題に対してはテストプログラム生成ツールを開発し，大幅な削減を実現した．非信頼 OSAP に対する制限事項に対しては，CPU 例外を使用したテストライブラリを独自に開発することで，解決した．

実際に，商用 A-OS である JulinarOs に対して，本研究のアプローチを適用し，その有効性を確認した．さらに，複数のプロセッサ，複数の A-OS の実装に対しても適用可能であることを確認し，再利用性や拡張性の高さを確認した．

第6章　AUTOSAR OS仕様ベースのRTOSに対するテスト

6.1　概要

近年，自動車に搭載されるECUの高度化・複雑化が進み，車載ソフトウェアの開発コストが急増している．また，1つのECUで実現する機能の数も増えており，複数のアプリケーションを1つのプロセッサ上に混在させたいニーズが増え，アプリケーション間での故障や不具合の伝搬を防止するために，メモリ保護機能の必要性も高まっている．ECUのような組込みシステムでは，汎用OSとは異なり，リアルタイム性の観点から，メモリマネジメントユニット(MMU)ではなく，メモリプロテクションユニット(MPU)を用いて保護を実現することが一般的である．MPUは，MMUとは異なりアドレス変換を行わないため，TLB(Translation Lookaside Buffer)を必要とせず，リアルタイム性の保証がしやすい．MPUは，アドレス空間をいくつかの領域に分け，その領域ごとにアクセス権を設定可能である．

欧州では，AUTOSAR[2]というECU向けのソフトウェアプラットフォーム仕様が策定・公開されている．ソフトウェアプラットフォームの標準化により，車載ソフトウェア開発コストの削減を目指しており，国内でもAUTOSARが普及しつつある．AUTOSARでは，車載ソフトウェアで共通的に使用されるRTOSや，通信スタック，メモリスタックなどのソフトウェア仕様が規定されている．AUTOSAR仕様のRTOS(以下，A-OS)[4]には，メモリ保護機能が規定されており，この機能により，アプリケーション間での故障や不具合の伝搬防止を実現する．

TOPPERSプロジェクトでは，A-OS仕様をベースとしたRTOSであるATK2[14]を開発，公開している．A-OS仕様では，MPUを用いてメモリ保護機能を実現するように規定されているが，MPUを使った具体的なコンフィギュレーションの方法は規定されていない．そこで，ATK2は，MPUを用いたメモリ保護機能を有するμITRON仕様[3]ベースのRTOSである，HRP2カーネル[15]の設計[16]を参考に，未規定の仕様を定め，メモリ保護機能が実装されている．

車載ソフトウェアは，機能安全規格 (ISO26262) や Automotive SPICE で規定されているように，高い安全性と品質保証が求められる．ATK2 は，車載ソフトウェアの中核をなし，メモリ保護機能自体が正しく動作しないと，システム全体の安全性が保証できなくなるので，ATK2 のメモリ保護機能自体のテストは重要である．また，MPU は，プロセッサによって設定可能な属性や領域数，アライメント制約が異なるため，メモリ保護機能に対応した RTOS のポーティングは複雑である．したがって，ポーティング後に，メモリ保護機能が正しく振る舞うことを網羅的かつ効率的にテストできることが望ましい．

RTOS のテスト手法に関する研究として，並列性のあるタスクに関する仕様に対して，モデル検査により網羅的に検証する手法が提案されている [55][56][57][58][59]．一方，本研究が対象とするメモリ保護機能に対するテストは，モデル検査による状態の網羅ではなく，アクセスするメモリ領域やアクセス時の条件の膨大な組み合わせに対応した手法が必要である．また，過去の研究では，タスクの振る舞いに特化していることや，テストケースの生成のみでテストプログラムの生成が行えないことから，本研究には適用できない．

そこで本章では，第 3 章，第 4 章で述べた，μITRON ベースの RTOS に対して研究開発したテストスイートを活用し，ATK2 に対するテスト手法を確立し，実施した．本章の貢献は，MPU を用いたメモリ保護機能を有する RTOS をテストするためのテスト手法を確立し，テストの効率性と有用性を示すことである．また，A-OS 仕様の RTOS およびメモリ保護機能を有する RTOS に対するテストの規模を明らかにすることである．そして，本研究の成果を用いて，メモリ保護機能を持つ RTOS の品質を向上することも，目的の 1 つである．

本章の構成は以下の通りである．6.2 節で，テスト対象とした ATK2 のメモリ保護機能について述べ，6.3 節で TTSP について述べる．6.4 節で確立したメモリ保護機能に対するテスト手法について述べ，6.5 節でテスト実施結果について評価を行う．6.6 節で関連研究について述べ，6.7 節でまとめる．

6.2 ATK2 のメモリ保護機能

本節では，HRP2 カーネルを参考に実装した ATK2 のメモリ保護機能について説明する．A-OS 仕様には，4 つのスケーラビリティクラス (以下，SC) が定義されているが，本章では，メモリ保護機能に対応したスケーラビリティクラス 3(以下，SC3) のみを対象とする．A-OS 仕様における API 保護，およびアクセス保護につ

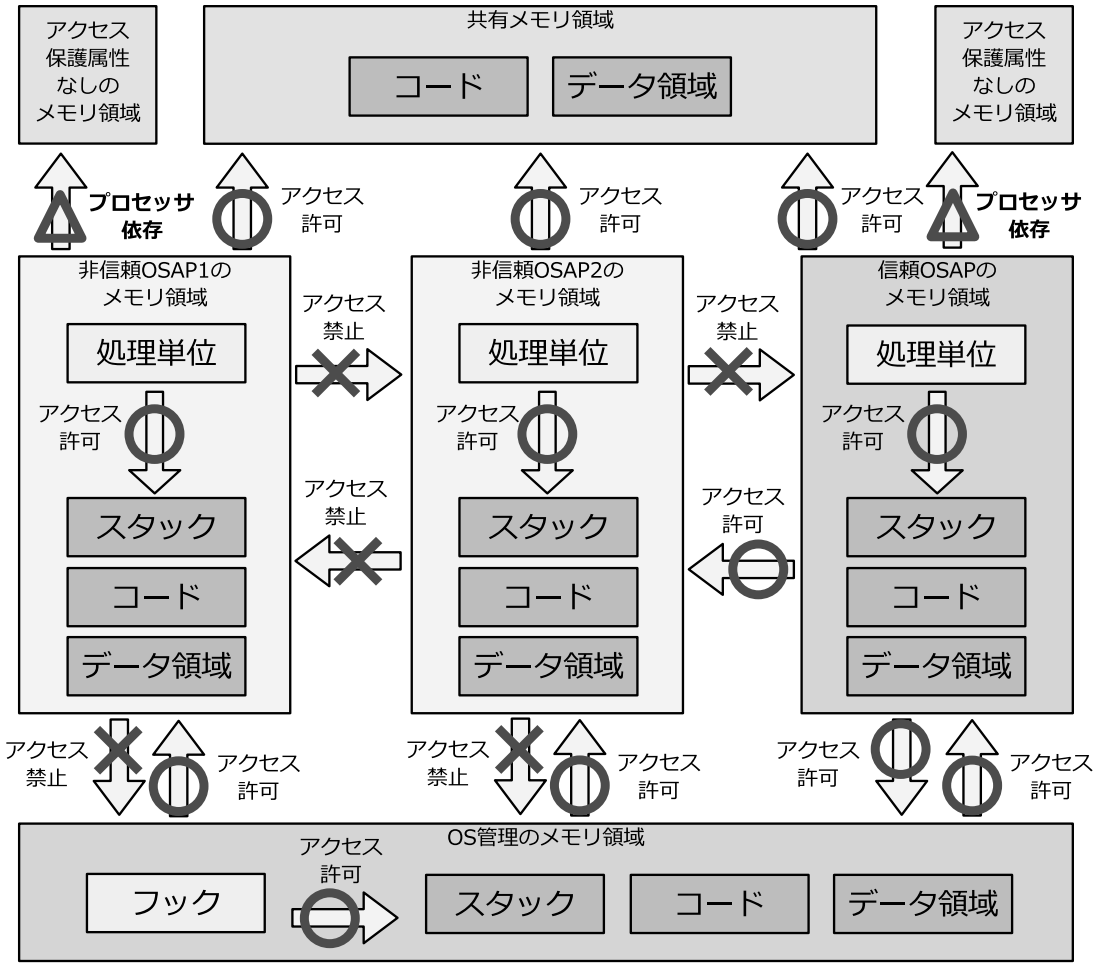


図 6.1: ATK2 によるメモリ保護設定の例

いては，5.2.2 節を参照のこと．

ATK2 によるメモリ保護設定の例を図 6.1 に示す．図 6.1 の例では，1 つの信頼 OSAP と 2 つの非信頼 OSAP (OSAP1 と OSAP2) を設定している．他に OS が使用する領域として，各 OSAP に含まれる OS オブジェクトやメモリ領域以外にも，OS 管理のメモリ領域，共有メモリ領域，アクセス保護属性なしのメモリ領域も存在する．非信頼 OSAP は，自身に所属する OS オブジェクトやメモリ領域はアクセスできるが，それ以外に対してはアクセスが禁止される．信頼 OSAP は，無制限にいずれのアクセスも許可される．なお，アクセス保護属性を設定していない，あるいはプロセッサの仕様上，アクセス保護属性を設定できないメモリ領域に対してアクセスを行った場合に，プロセッサがどのように振る舞うかは，プロセッサによって異なる．

6.2.1 A-OS 仕様と ATK2 実装の差異

A-OS 仕様で規定されているメモリ保護機能仕様には，実装する上での課題があり，ATK2 では実装していない仕様がある．具体的には，以下の 3 点がある．

- 信頼 OSAP からの他の OSAP に所属する OS オブジェクトへのアクセス
  A-OS 仕様では，信頼 OSAP はアクセス保護が適用されないが，API 保護は適用されるように規定されている．アクセス保護が適用されないのに対し，API 保護が適用されることは，OS 仕様としての対称性が低いことから，ATK2 では，信頼 OSAP に対しては API 保護を適用していない．つまり，信頼 OSAP に所属する OS オブジェクトは，すべての OSAP に所属する OS オブジェクトに対して，API を発行することが可能である．

- 非信頼 OSAP の ISR 所属制限
  A-OS 仕様では，非信頼 OSAP に ISR が所属することができる．ISR が非信頼 OSAP に所属するということは，ISR がメモリ保護違反を引き起こした場合に，その ISR を強制終了する可能性があるということである．一般に，ISR は早い応答性が求められる処理を実行する処理単位であるので，レジスタの退避，復帰やスタック切り替えは最小限の処理に留めるべきである．また，割込み優先度に応じて，複数の ISR が多重に発生して起動することがあるので，すべての ISR 用にスタックを 1 つだけ用意して，スタックを切り替えずに，すべての ISR を実行したほうが効率的である．しかし，非信頼 OSAP に所属する ISR は，メモリ保護違反などによって強制終了する可能性があるので，前述の処理の効率化を適用できず，割込み発生時のオーバーヘッドが大きくなってしまう．以上の理由から，ATK2 では，ISR は信頼 OSAP にのみ所属する実装とした．

- プロテクションフックの戻り値制限
  A-OS 仕様において，プロテクションフックの戻り値として指定できるのは以下の 5 つである．

  - PRO_SHUTDOWN
    OS をシャットダウンする
  - PRO_TERMINATETASKISR
    アクセス保護違反を引き起こしたタスク/ISR を強制終了する

- PRO_TERMINATEAPPL
  アクセス保護違反を引き起こしたタスク/ISRが所属するOSAPを強制終了する
- PRO_TERMINATEAPPL_RESTART
  アクセス保護違反を引き起こしたタスク/ISRが所属するOSAPを再起動する
- PRO_IGNORE
  無視する(アクセス保護違反は無視できないためシャットダウンされる)

上記の中で，PRO_TERMINATEAPPLとPRO_TERMINATEAPPL_RESTARTは，動作中のOSAPを強制的に停止させるものである．OSAPには，リソースやカウンタといったOSオブジェクトも所属しているが，OSAPが停止した場合に，これらのOSオブジェクトに関連する他のOSオブジェクトがどのように振る舞うかが，A-OS仕様には規定されていない．例えば，非信頼OSAPに所属しているリソース[1]を，他のOSAPに所属しているタスクが取得している状態で，非信頼OSAPが強制終了された場合，取得中だったリソースがどう処理されるかが規定されていない．排他制御に使用中のリソースが，意図せず開放されてしまうのは問題であるが，開放しないままでいると，タスクが開放しようとした際に，リソースの所属OSAPが停止状態のためエラーとなってしまう．このように，A-OS仕様では，OSAPが停止する場合の仕様が不明確であるので，ATK2では，PRO_TERMINATEAPPLとPRO_TERMINATEAPPL_RESTARTに対する処理を未実装とした．また，ISRが非信頼OSAPに所属できないことから，PRO_TERMINATETASKISRで強制終了する対象は，タスクのみとなる．

6.2.2 API保護の実現方法

ATK2では，各OSオブジェクトが所属するOSAPをOSが管理し，API発行時に，呼び出し元の処理単位が所属するOSAPを確認し，APIの引数に与えられたOSオブジェクトやアドレスのチェックを行う．

あるタスクから別のタスクの状態を取得するAPIであるGetTaskStateを用いて，API保護について説明する．GetTaskStateの引数は，状態を取得したいタス

[1]タスク/ISR間の排他制御に用いるOSオブジェクト．

ク ID(TaskID) と，取得した結果を格納する変数へのポインタ (State) の 2 つである．

例として，非信頼 OSAP1 に所属する TASK1 が，異なる OSAP(非信頼 OSAP2) に所属する TASK2 に対して，GetTaskState を呼び出すこととする．State には，通常呼び出し元の TASK1 が指定した変数へのポインタが与えられる．State に不正なアドレスを指すポインタが指定される可能性があるので，TASK1 から GetTaskState が呼び出されると，まず State が指し示すアドレスへ，TASK1 が書き込み可能かを，API の中で OS がチェックする．

ATK2 では，各 OSAP がアクセス可能なメモリ領域の情報を，コンフィギュレーションに応じて生成するメモリ保護情報ファイルに定義された内容から参照することできる．メモリ保護情報ファイルについては，6.2.4 節で説明する．タスクの状態を確認する変数のサイズは 1byte であり，State が指し示すアドレスから 1byte 分のメモリ領域に対して，TASK1 が書き込み可能かを OS がチェックし，書き込み不可であれば，エラーとして後続の処理を実行しない．

TASK1 から State が指し示すアドレスへアクセス可能である場合，OS は次に，非信頼 OSAP1 が TASK2 に対してアクセス可能かをチェックする．OS はまず，呼び出し元の TASK1 が所属する OSAP(非信頼 OSAP1) を取得し，TASK2 にアクセス可能な OSAP のリストに，非信頼 OSAP1 が含まれているかをチェックする．リストに含まれていなければ，エラーとして後続の処理を実行しない．

6.2.3 アクセス保護の仕様設計

本節では，A-OS 仕様で未規定であるアクセス保護に対する ATK2 の仕様について説明する．

MPU は，アドレス空間をいくつかの領域に分け，その領域ごとにアクセス権を設定可能であるが，MPU で扱うことのできる領域の数は有限であり，多くの場合，8 領域程度である [16]．そのため，同じアクセス権を設定する必要のあるコードやデータを，連続した番地に配置するように，静的にメモリ配置を行い，MPU に設定するメモリ領域の数を，MPU が扱うことのできる領域数に収まるようにする必要がある．これを踏まえ，ATK2 では，HRP2 カーネルの設計を参考に，以下の設計とした．

- MPU に関する操作 (初期化や設定の書換え，有効化・無効化) は，OS の内部処理で行う．

- 同じアクセス権を設定する必要のあるコードやデータを，連続した番地に配置するように，OS が静的にメモリ配置を行う．このとき，MPU に設定するメモリ領域を，MPU の制限に合わせて適切にアライメントする．
- OS のユーザは，MPU に設定する具体的な番地やサイズを知ることなく，セクションやオブジェクトファイル単位でアクセス保護属性の設定ができる．ここで，セクションとは，リンカスクリプトが扱う単位であり，同じメモリ配置の属性を持ったコードやデータのまとまりである．
- 信頼 OSAP は，プロセッサの CPU コアを特権モード (アクセス保護が無効な状態) で実行し，非信頼 OSAP は，ユーザモード (アクセス保護が有効な状態) で実行する．
- 非信頼 OSAP から，OS が提供する API を呼び出した場合，ユーザモードから特権モードへ切り替えて処理を行い，API からリターンする際にユーザモードへ切り替える．

また，ATK2 が管理対象とするメモリ領域は，以下の 4 つの領域に分類した．

- 非信頼 OSAP のメモリ領域
- 共有メモリ領域
- OS 管理のメモリ領域
- アクセス保護属性が設定されていないメモリ領域

非信頼 OSAP のメモリ領域，共有メモリ領域，OS 管理のメモリ領域は別々のセクションに分かれ，各セクションにアクセス保護属性が設定される．信頼 OSAP のメモリ領域は，OS 管理のメモリ領域と区別する必要がないため，実質的に OS 管理のメモリ領域に包含される．アクセス保護属性が設定されていないメモリ領域は，非信頼 OSAP からの読み出し，書き込み，実行アクセスが禁止される．各メモリ領域は表 6.1 に示したアクセス保護属性を設定することができる．

6.2.4 アクセス保護の実現方法

本節では，ATK2 のアクセス保護の実現方法について説明する．実現方法の概要図を図 6.2 に示す．

表 6.1: メモリ領域に対するアクセス保護属性

非信頼 OSAP のメモリ領域	
	スタック (自処理単位からのみ読み出し，書き込み可能)
	コード (自 OSAP からのみ実行可能)
	専有リードオンリーデータ (自 OSAP からのみ読み出し可能)
	共有リード専有ライトデータ (自 OSAP からのみ書き込み可能，他 OSAP からは読み出しのみ可能)
	専有リードライトデータ (自 OSAP からのみ読み出し，書き込み可能)
共有メモリ領域	
	コード (全 OSAP から実行可能)
	共有リードオンリーデータ (全 OSAP から読み出しのみ可能，書き込み不可能)
	共有リードライトデータ (全 OSAP から読み出し，書き込み可能)
OS 管理のメモリ領域	
	スタック (OS，信頼 OSAP からのみ読み出し，書き込み可能)
	コード (OS，信頼 OSAP からのみ実行可能)
	専有リードオンリーデータ (自 OSAP からのみ読み出し可能)
	専有リードライトデータ (自 OSAP からのみ読み出し，書き込み可能)

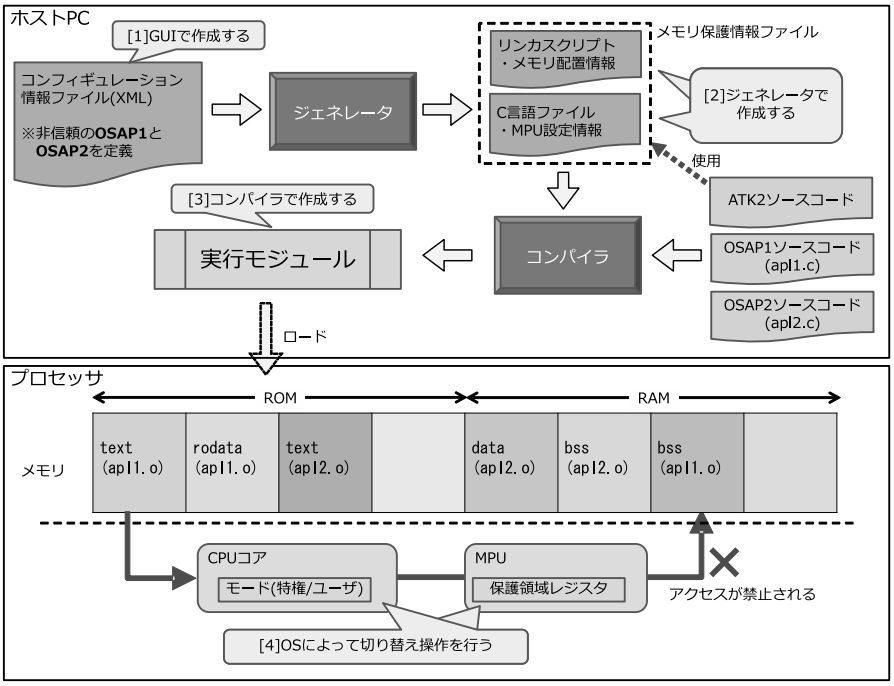


図 6.2: アクセス保護の実現方法

まず，ユーザは実現したい OS オブジェクトや OSAP の設定を含めたメモリ保護に関する設定を記述したコンフィギュレーション情報ファイルを作成する (図 6.2[1])．AUTOSAR では，各モジュールのコンフィギュレーション情報を XML で記述する仕様となっており，通常は GUI ツールを用いてコンフィギュレーションを行い，XML を出力する．5.2.2 節で述べた通り，アクセス保護設定に関するコンフィギュレーションパラメータに関しては，A-OS 仕様では定義されていないため，ATK2 では HRP2 カーネルの静的 API によるコンフィギュレーションをベースに，独自に拡張を行った．

XML で記述したコンフィギュレーション情報ファイルを入力として，ジェネレータと呼ぶツールにより，C 言語ファイル，リンカスクリプトを生成する (図 6.2[2])．C 言語ファイルには，OSAP を管理，制御する上で必要な情報や，アクセス権の情報が，ATK2 ソースコードから使用できる形式で出力される．リンカスクリプトには，6.2.3 節で述べた通り，有限である MPU の領域数を節約するために，最適なメモリ配置を実現するためのコードが出力される．これらの生成したファイルをメモリ保護情報ファイルと呼ぶ．

メモリ保護情報ファイルと，ATK2 ソースコード，各アプリケーションを実装したファイルを，コンパイラによってビルドして実行モジュールを作成する (図 6.2[3])．

実行モジュールがプロセッサにロードされ，非信頼 OSAP の処理単位が実行されると，MPU を使用したアクセス保護が機能する (図 6.2[4])．図 6.2 では，非信頼 OSAP1 の処理単位から非信頼 OSAP2 のデータ領域へはアクセスができないコンフィギュレーションとなっている．したがって，OSAP1 の処理単位において，OSAP2 の bss 領域へのアクセスを CPU コア上で実行しようとしても，MPU がアクセス違反を検知し，CPU 例外を発生させることで，OSAP2 の bss 領域が保護される．

CPU コアが特権モードとユーザモードのどちらで動作するか，および MPU に設定する保護領域レジスタの情報を，OS が各 OSAP の実行状況に応じて，適切に切り替えることによって，システム全体としてのアクセス保護を実現する．

6.3 TTSP の概要

本研究では，これまでに μITRON ベースの RTOS である，ASP カーネル [7] と FMP カーネル [8] に対応したテストスイートを開発する過程で，RTOS に対するテストプロセスやテスト手法に関して研究を行った (第 4 章を参照のこと)．開発したテストスイートは TTSP(TOPPERS Test Suite Package) と命名した．この研究成果を踏まえ，ATK2 に対するテストは，TTSP を A-OS 仕様に適合させ，さらにメモリ保護機能に対するテストを追加して実施することにした．本節では，TTSP の概要について説明する．

6.3.1 API テスト

TTSP が対象とするテストは，主に RTOS がアプリケーションに提供する API に対するテストである．これは，RTOS の API に不具合や仕様に反する処理が存在すると，RTOS 上で動作するアプリケーションが誤作動する原因となりうることや，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 のテストにおいて，最も重要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 OS の状態や各 OS 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，API テストでは，あるシステム状態で API を発行し，仕様通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，テストプログラムにて，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．この一連のテストの流れをテストシナリオと呼ぶ．

RTOS の仕様書から，テストで確認する API 発行に対する RTOS の振る舞いをテストケースとして抽出し，テストケース毎にテストシナリオを作成する．テストケースの例を図 6.3 に示し，図 6.3 のテストケースに対応するテストシナリオの例を図 6.4 に示す．

6.3.2　TESRY 記法

次節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 6.4 のテストシナリオを TESRY 記法で記述した例を図 6.5 に示す．

6.3.3　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．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 OS 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 OS 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 OS 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，第 3 章を参照のこと．

低優先度の実行状態のタスクから，高優先度の
休止状態のタスクに対して act_tsk を発行すると，
対象タスクが実行状態になること．

図 6.3: テストケースの例 (図 3.1 の再掲)

前状態
　優先度（低）の TASK1 が実行状態
　優先度（高）の TASK2 が休止状態
処理
　TASK1 が act_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 6.4: テストシナリオの例 (図 3.2 の再掲)

```
pre_condition:
  TASK1:
    type   : TASK
    tskpri : LOW
    tskstat: running
  TASK2:
    type   : TASK
    tskpri : HIGH
    tskstat: dormant

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

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

図 6.5: TESRY 記法の例 (図 3.4 の再掲)

6.4 APIテストのA-OS仕様への適合とアクセス保護テスト

6.3.1節で述べた通り，ATK2においても，APIテストが重要であると考え，TTSPをA-OS仕様に適合し，APIテストを実施することにした．しかし，TTSPが対象とするRTOSは，メモリ保護機能に対応していないため，TTGは非信頼OSAPに所属する処理単位が受ける制約を考慮していない．例えば，メモリ保護機能がないRTOSのテストプログラムでは，システム状態確認APIや前状態を実現するためのAPIは，どの処理単位からでも発行が可能である．しかし，メモリ保護機能がある場合，非信頼OSAPに所属する処理単位は，アクセスできるOSオブジェクトやメモリ領域に制限があるので，TTGのテストプログラム生成処理に対応が必要である．

また，テストプログラムをRTOS仕様の違いに対応するだけでなく，API保護に対するテストケースを，APIテストに追加する必要がある．例えば，アクセス権のないOSオブジェクトを引数に与えてエラーとなることを確認するテストケースや，アクセス権のないメモリ領域へのポインタを引数に与えてエラーとなることを確認するテストケースなどの追加が必要となる．さらに，アクセス保護は，許可されていないアクセスを監視，検出する機能であるので，APIテストだけを実施しても，アクセス保護に対するテストはできない．

そこで，APIテストでは，API保護に対するテストケースを追加し，APIテストとは別に，アクセス保護が正しく動作することを確認するテスト(以下，アクセス保護テスト)を新たに実施した．本節ではAPIテストのA-OS仕様への適合と，新規に実施したアクセス保護テストについて説明する．なお，ATK2向けのテストスイートは，AKTSP(Automotive Kernel Test Suite Package)と命名した．

6.4.1 APIテスト

APIテスト[2]では，TESRY記法，TTGに対してA-OS仕様への適合を行った．

[2]AKTSPでは，仕様書の表記に合わせてAPIテストをシステムサービステストと呼称する．

TESRY 記法の適合

TESRY 記法は，μITRON 仕様に対するテストシナリオの記述方法と同様に，A-OS 仕様で定められた OS オブジェクト毎に必要なパラメータを規定した．図 6.5 と同等のテストシナリオを，A-OS 仕様の TESRY 記法で記述した例を図 6.6 に示す．

A-OS 仕様では，タスク種別 [ttype] として基本タスクと拡張タスクが指定できる．また，タスク毎にスケジューリングポリシー [spolicy] として，フルプリエンプティブかノンプリエンプティブかを指定できる．また，排他制御の仕組みにより，タスク起動時に優先度が変化することがあるので，現在優先度 [curpri] と初期優先度 [inipri] を区別する必要がある．さらに，タスクが所属する OSAP[osap] を指定する必要がある．

μITRON 仕様で，タスクを起動する API として発行していた act_tsk に相当する A-OS 仕様のタスクを起動する API は ActivateTask である．μITRON 仕様では，API の戻り値のデータ型は 1 種類であるが，A-OS 仕様では複数のデータ型が戻り値になるので，データ型 [rettype] も指定する必要がある．

TTG の適合

TTG を A-OS 仕様に適合させ，AKTG(Automotive Kernel Test Generator) と命名した．AKTG は，構成や生成テストプログラムの構造は TTG を踏襲し，前節で述べた A-OS 仕様に対応した TESRY データを入力として動作する．A-OS 仕様は，μITRON 仕様と同じく優先度ベーススケジューリングを行う RTOS であるので，TTG が生成するテストプログラムの基本構成を，そのまま使用可能であった．

メモリ保護機能のテスト実現のために，AKTG では主に以下の設計を行うことが必要であった．

- 非信頼 OSAP からのシステム状態確認 API 発行
  システム状態確認 API は，OS 管理のメモリ領域に配置されるデータを参照する必要があるため，システム状態確認 API を発行する処理単位が非信頼 OSAP に所属していた場合，アクセスができない．したがって，OS が提供する API と同様に，非信頼 OSAP からシステム状態確認 API を呼び出した場合，特権モードへ切り替えて実行することが必要である．

- 非信頼 OSAP からの前状態の実現
  前状態の実現のために，テストシナリオに登場する処理単位から，いくつか

```
pre_condition:
  TASK1:
    type   : TASK
    curpri : LOW
    tstat  : RUNNING
    ttype  : AK_BASIC
    spolicy: AK_SCHFULL
    osap   : NON_TRUSTED_OSAP1
  TASK2:
    type   : TASK
    inipri : HIGH
    tstat  : SUSPENDED
    ttype  : AK_BASIC
    spolicy: AK_SCHFULL
    osap   : NON_TRUSTED_OSAP2
    accosap:
      - NON_TRUSTED_OSAP1

do:
  id     : TASK1
  syssrv : ActivateTask(TASK2)
  rettype: StatusType
  retval : E_OK

post_condition:
  TASK1:
    tstat: READY
  TASK2:
    tstat  : RUNNING
    curpri : HIGH
```

図 6.6: A-OS 仕様に対応した TESRY 記法の例 (図 5.5 の再掲)

の API を発行する必要があるが，呼び出し元の処理単位が非信頼 OSAP の場合，呼び出し先の OS オブジェクトによっては，API 保護によりアクセス権エラーとなる可能性がある．そこで，前状態実現の処理に限り，信頼 OSAP が提供する信頼関数という機能を利用し，非信頼 OSAP からでも任意の OS オブジェクトに対して API を発行できるようにする必要がある．6.2.2 節で述べた通り，OS が提供する API は，呼び出し元の所属 OSAP からアクセス可否を判定するので，システム状態確認 API とは異なり，特権モードで動作させても，アクセス権がなければエラーとなってしまう．したがって，前状態を実現するための API では，呼び出し元の所属 OSAP を信頼 OSAP に切り替える信頼関数を使用する必要がある．なお，信頼関数は，ユーザにも提供される機能であるため，本ケースに合致するシステム状態は，A-OS 仕様上，実現可能である．

- スタック領域の分離
  メモリ保護機能がない RTOS の場合，タスクが使用するスタック領域に区別がないので，テストシナリオに登場するタスクの数だけスタック領域を用意すればよい．信頼 OSAP のタスクは，タスクの数だけスタック領域を用意すればよいが，非信頼 OSAP のタスクは，各非信頼 OSAP からアクセス可能なメモリ領域に，それぞれの非信頼 OSAP に所属するタスクの数だけ，スタック領域を確保する必要がある．

- 非信頼 OSAP からの OS シャットダウン
  テストプログラム実行中に何らかのエラーが発生した場合，不正な処理が続行しないように，OS シャットダウンによりテストプログラムを終了させる必要がある．しかし，A-OS 仕様では，非信頼 OSAP の処理単位から OS シャットダウンを実行できない．そこで，前状態を実現するための API と同様に，OS シャットダウンを行う信頼関数を用意し，エラー発生時に限り，非信頼 OSAP からも OS シャットダウンを実行可能とする必要がある．

なお，TTG では，TESRY データに記載された 1 パラメータを，テストプログラムでも 1 行ずつ記述してチェック処理を実行していたのに対し，AKTG のテストプログラムでは，全パラメータチェックする関数呼び出し 1 行でチェックするように変更した．前節で述べたタスクのように，A-OS 仕様は μITRON 仕様と比べて，設定できるパラメータが多いので，1 パラメータを 1 行ずつ記述してチェックすると，1 つの関数のコードサイズが大きくなってしまう．コンパイラによっては，

1つの関数のコードサイズが大きいとコンパイルできなくなることがあるので，これを避けるため，全パラメータをチェックする関数呼び出し1行で実行するように変更して，コードサイズを削減した．

テストケース数増加への対応

A-OS仕様でテスト対象としたAPIの数は57個であり，μITRON仕様の103個と比べて半分程度しかなく，APIテストのテストケース数は，TTSPよりも少なくなるものと思われた．しかし，2.1.3節で述べたタスクのように，A-OS仕様はμITRON仕様と比べて，設定できるパラメータが多いことに加えて，信頼OSAPと非信頼OSAPの組み合わせや，API保護に対するテストケースの追加が必要である．これらの理由により，TTSPで1,629件であったテストケース数が，AKTSPでは40,000件以上になることが判明した．各パラメータの設定値が変わることで，APIの振る舞いも変わることから，パラメータの組み合わせを減らすことはできず，全組み合わせをテストする必要がある．しかし，40,000件という膨大なTESRYデータを手作業で作成するのは，保守性の面でも困難であるので，何らかの組み合わせツールを用いて，TESRYデータ作成の効率化を行うべきと考えた．

テストケースの組み合わせ生成ツールはいくつか存在するが，テストケースの組み合わせだけ生成できても，膨大なテストケースに対して1つ1つTESRYデータを作成するのは現実的ではない．そこで，テストケースの組み合わせだけでなく，生成したテストケース毎の期待値(テスト結果)も論理的に生成可能であるPictMaster[52]を選択した．PictMasterは，Excelベースで使用できるツール(VBAで実装)であり，VBAのコードがオープンソースで自由に拡張できることも，PictMasterを選択した理由である．

PictMasterは，組み合わせるパラメータとそれぞれのパラメータが取りうる値，各パラメータ同士の関係により発生する制約，組み合わせ毎に得られる結果という3つのフィールドに分けてデータを入力し，入力内容に応じたすべてのパラメータの組み合わせと結果を出力する．PictMasterの標準の出力形式がExcelファイル形式であったため，PictMasterの出力をTESRYデータに変換しやすいように，csvファイル形式で出力するように，PictMasterのVBAの修正を行った．その他にも，利便性向上のためにVBAを修正したが，全体の修正量としては，実プログラム行数で50行程度であった．

ActivateTaskの正常系テストケースに対して作成したPictMasterファイルの例

(抜粋) を図 6.7 に示す．

まず，ActivateTask のテストケースにおいて考慮すべきパラメータと，その組み合わせ対象値を定義する．例えば，ActivateTask を発行する処理単位 (call_origin) や，ActivateTask を発行される側のタスクの状態 (other_tstat) などである．

続いて，各パラメータを組み合わせる上での制約表を定義する．網掛け (色付け) したセルが制約条件となり，無地のセルが制約を受ける対象となる．仕様制約 1 では，ActivateTask を発行される側のタスクの状態 (other_tstat) が実行可能状態 (READY) の場合，かつ ActivateTask を発行するタスクがフルプリエンプティブ (AK_SCHFULL) の場合，ActivateTask を発行される側のタスクの現在優先度 (other_curpri) は高 (TSK_PRI_HIGH) にはならない (”#”は否定の意) という制約を設定している．この例では，呼び出し元タスクの優先度 (org_curpri) を中 (TSK_PRI_MID) で固定している．ここで，org_curpri が TSK_PRI_MID であるのに対し，other_curpri が TSK_PRI_HIGH であり，かつ呼び出し先のタスクの状態 (other_tstat) が READY であると，優先度の高いタスクが実行可能状態であるにも関わらず，優先度が低いタスクが動いている状態になってしまう．さらに，A-OS 仕様では，スケジューリング方式をタスク毎に決めることが可能で，呼び出し元タスクのスケジューリング方式 (org_spolicy) がフルプリエンプティブ (AK_SCHFULL) の場合にのみ，この問題が起きるので，この制約も必要となる．このように，A-OS 仕様上，発生し得ない組み合わせを，制約表により排除している．

そして，TESRY 生成表で，TESRY データの元データとなる csv ファイルに出力するデータを定義する．1，2 行目は，pre_condition と，TASK1 の状態 (tstat) が実行状態 (RUNNING) で，初期優先度が中 (TSK_PRI_MID) という定義を，無条件ですべてのテストケースで生成している．3，4 行目では，TASK1 のタスク種別 (ttype) が，基本タスク (AK_BASIC) か拡張タスク (AK_EXTENDED) のどちらかが定義される．同様に 5，6 行目では，TASK1 のタスクのスケジューリングポリシー (spolicy) が，フルプリエンプティブ (AK_SCHFULL) かノンプリエンプティブ (AK_SCHNON) のどちらかが定義される．これらの定義を行い，生成を行うと，制約条件に合致した，すべての組み合わせの TESRY データ用 csv ファイルが生成される．

この csv ファイルを，Ruby による簡単なスクリプトを用いて TESRY データへ変換し，AKTG への入力データとした．PictMaster を用いた処理フローを図 6.8 に示す．

パラメータ	組み合わせ対象値
call_origin	TASK,C2ISR
org_curpri	TSK_PRI_MID
org_ttype	AK_BASIC,AK_EXTENDED
org_spolicy	AK_SCHFULL,AK_SCHNON
other_tstat	SUSPENDED,READY
other_curpri	TSK_PRI_LOW,TSK_PRI_MID,TSK_PRI_HIGH

制約表

パラメータ	仕様制約1
call_origin	
other_tstat	READY
other_curpri	#TSK_PRI_HIGH
other_ttype	
other_evt_flg	
org_spolicy	AK_SCHFULL
other_actcnt	

TESRY生成表

TESRY_param	call_origin	org_ttype	org_spolicy	other_tstat
&pre_condition,	TASK,C2ISR			
&TASK1,type,TASK,tstat,RUNNING,inipri,TSK_PRI_MID,	TASK,C2ISR			
&ttype,AK_BASIC,	TASK,C2ISR	AK_BASIC		
&ttype,AK_EXTENDED,	TASK,C2ISR	AK_EXTENDED		
&spolicy,AK_SCHFULL,	TASK,C2ISR		AK_SCHFULL	
&spolicy,AK_SCHNON,	TASK,C2ISR		AK_SCHNON	

図 6.7: PictMasterファイルの例

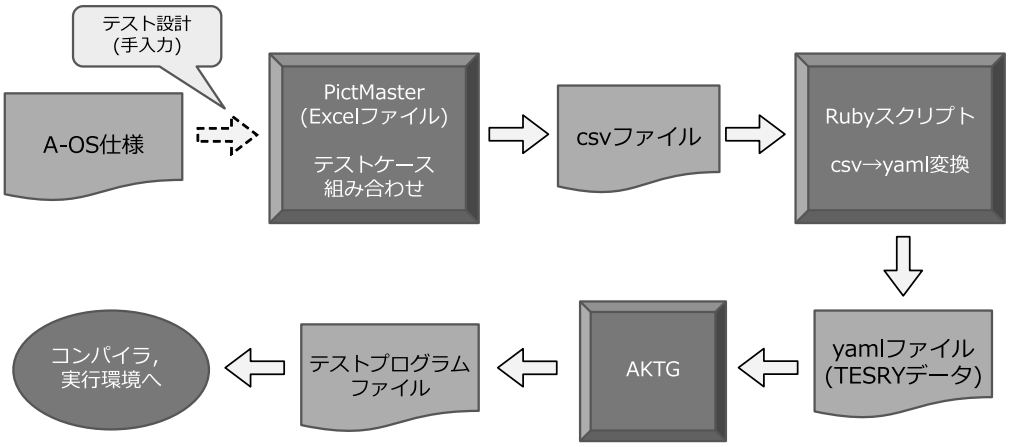


図 6.8: PictMaster を用いた処理フロー (図 5.4 の再掲)

6.4.2 アクセス保護テスト

アクセス保護テストでは，API テストの仕組みを応用し，網羅的に，かつ効率的にアクセス保護テストを実施する手法を考案した．また，様々なプロセッサに対してアクセス保護テストを実施できるよう，プロセッサに依存した部分を明確に分離し，ポータビリティの向上を図った．本節では，アクセス保護に対するテストについて，まずテスト対象の処理とテストシナリオの例について述べ，実現方法について説明する．

テスト対象の処理

本研究では，アクセス保護テストの対象とする処理を以下の 2 つと考えた．

(i) ユーザが設定したコンフィギュレーション情報から正しいメモリ保護情報ファイルが生成されるか (図 6.2[2])

(ii) 生成された情報を用いて，OS による意図したアクセス保護が行われるか (図 6.2[4])

(i) は，OS に付属するジェネレータに対するテストである．ジェネレータのような入力パラメータの組み合わせが無数に存在し，包括的な検証が困難なツールに対する検証は，出力結果のレビューやバリデーションツールによって正当性，妥当性を担保することが現実的な解決手法と言える．そこで，(i) に対するテストは，レビューで行うものとし，表 6.1 に示したすべてのメモリ領域がジェネレータの生

成情報に含まれるコンフィギュレーション情報ファイルを使用して，生成されたメモリ保護情報ファイルが正しく設定されることをレビューした．また，APIテストでは，テストケース毎のコンフィギュレーションに応じたファイルが，ジェネレータによって生成され，ビルド・実行されるため，間接的にジェネレータのテストは実施されていると考えられる．なお，ジェネレータが生成したコードに対する検証手法や十分性に関しては，本研究の対象とはせず，今後の研究対象とする．

(ii) は，5.2.2節で述べたアクセス保護に対するテストであり，本研究ではこれをアクセス保護テストの対象とした．

テストシナリオの例

前節での述べたアクセス保護テストのテストシナリオの一例を図6.9(a) に示す．

まず，前状態から，後状態1において，信頼OSAP1から別の非信頼OSAP2へ処理単位を切り替える．これにより，CPUコアは特権モードからユーザモードに切り替わり，MPUは非信頼OSAP2のアクセス可能なメモリ領域が設定される．つまり，後状態1で，OSによって非信頼OSAP2に与えられた正しいアクセス権となるように，CPUコアとMPUが設定されているはずである．

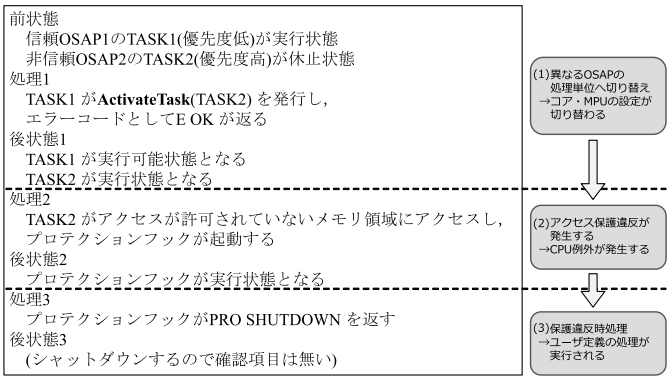
続いて，MPUの設定が正しいことをテストするために，処理2で，実行中の非信頼OSAP2のタスクから，非信頼OSAP2がアクセスを許可されていないメモリ領域へのアクセスを行う．MPUによりアクセス違反が検知され，CPU例外が発生し，OSがプロテクションフックを起動する．

続いて，プロテクションフックから，OSをシャットダウンする指示であるPRO_SHUTDOWNを返すことにより，OSはシャットダウンする．

このように，アクセス保護テストは，(1) 処理単位の切り替わり，(2) メモリアクセス，(3) 保護違反時処理/正常処理という流れで行うことにした．(1) は，処理単位の切り替えパターンとして，タスクディスパッチや割込み発生時などCPUコアとMPUの設定変更が発生するすべての処理単位が切り替わる組み合わせを確認する．(2) は，メモリアクセスパターンとして，アクセスが禁止されたメモリ領域へアクセスした場合に，アクセス保護違反が検知されることに加え，アクセスが許可されたメモリ領域へアクセスした場合に正しくアクセスできることも確認する．これを，表6.1に示したすべてのメモリ領域と，アクセス保護属性なしのメモリ領域[3]に対して確認する．(3) は，保護違反時処理パターンとして，アクセス保護違

[3]プロセッサによってはアクセス時の振る舞いが不定となるので，テストできない可能性がある．

(a)



(b)

```
pre_condition:
  TASK1:
    type    : TASK
    tstat   : RUNNING
    curpri  : TSK_PRI_LOW
    osap    : TRUSTED_OSAP1
  TASK2:
    type    : TASK
    tstat   : SUSPENDED
    curpri  : TSK_PRI_HIGH
    osap    : NON_TRUSTED_OSAP2
  PROTECTION_HOOK1:
    type    : PROTECTION_HOOK
    hookstat: AK_STOP
do_1:
  id     : TASK1
  syssrv : ActivateTask(TASK2)
  rettype: StatusType
  retval : E_OK
post_condition_1:
  TASK1:
    tstat: READY
  TASK2:
    tstat: RUNNING
do_2:
  id    : TASK2
  syssrv: <アクセス保護違反を発生させる関数>
post_condition_2:
  PROTECTION_HOOK1:
    hookstat: AK_ACTIVATE
do_3:
  id    : PROTECTION_HOOK1
  syssrv: return(PRO_SHUTDOWN)
post_condition_3:
```

図 6.9: アクセス保護テストのテストシナリオと TESRY 記法の例

		切り替え先			
		信頼 OSAP	非信頼 OSAP1	非信頼 OSAP2	OS管理
切り替え元	信頼 OSAP	-	パターン1	-	-
	非信頼 OSAP1	パターン2	パターン3	パターン4	パターン5
	OS管理	-	パターン6	-	-

仕様上，切り替え不可能な
組み合わせはパターンから除外

		非信頼OSAP2			
		タスク	OSAP固有の スタートアップフック	OSAP固有の シャットダウンフック	OSAP固有の エラーフック
非信頼 OSAP1	タスク	○	-	○	-
	OSAP固有の スタートアップフック	○	○	○	-
	OSAP固有の シャットダウンフック	-	-	○	-
	OSAP固有の エラーフック	○	-	○	-

図 6.10: 処理単位の切り替えパターン

反が発生した場合に，ユーザが設定した通りの保護違反時処理が実行されることを確認する．アクセス保護違反が発生しないテストケースの場合，メモリアクセスを行った処理単位の実行が継続することを確認する．

各パターンの組み合わせ

処理単位の切り替えパターンを，図 6.10 に示す．

パターン 1，パターン 6 は，それぞれ信頼 OSAP，OS 管理の処理単位から非信頼 OSAP1 の処理単位に切り替わるパターンである．これは，CPU コアが特権モードで動作している状態から，ユーザモードへ切り替わるのに加え，MPU が非信頼 OSAP1 のアクセス権の状態へ切り替わる．パターン 2，パターン 5 は，それぞれ非信頼 OSAP1 の処理単位から，信頼 OSAP，OS 管理の処理単位に切り替わるパターンである．これは，CPU コアがユーザモードで動作している状態から，特権モードへ切り替わる．パターン 3 は，同じ非信頼 OSAP1 に所属する処理単位でも，異なるタスクの場合に，保護するスタック領域が切り替わるパターンである．これは，MPU のスタック領域に関する設定だけ切り替わる．パターン 4 は，非信頼 OSAP1 の処理単位から異なる非信頼 OSAP2 の処理単位に切り替わるパターンで

ある．これは，MPU の非信頼 OSAP 毎の設定が切り替わる．

さらに，各パターンにおいて，切り替わる処理単位の組み合わせを考慮する．図 6.10 で拡大したパターン 4 を例に説明する．非信頼 OSAP には，タスク，OSAP 固有のスタートアップフック，シャットダウンフック，エラーフックが所属できる．ただし，各処理単位が動作している状態で，異なる非信頼 OSAP の処理単位へ遷移する可能性があるかどうかは，A-OS 仕様から判断できるため，仕様上，切り替え不可能な組み合わせはパターンから除外することができる．パターン 4 では，図 6.10 に示した 8 つの処理単位の切り替えがテスト対象となる．

テストに使用したプロセッサにおけるメモリアクセスパターンを，図 6.11 に示す．ROM 領域は 8 種類，RAM 領域は 19 種類のメモリ領域がある．RAM 領域には，スモールデータや書き込み可否，スタックなどの種別が増えるため，ROM 領域より多くの領域が存在する．他に，メモリ保護の設定がされていない領域があるが，テストに使用したプロセッサでは，メモリ保護の設定ができない領域であり，アクセス時の振る舞いが不定であるので，テスト対象外とした．

信頼 OSAP と OS 管理の処理単位は，特権モードで動作するため，アクセスできないメモリ領域は存在しない．また，図 6.10 で想定した処理単位から区別されるメモリ領域は以下の 6 つである．

(A) すべての OSAP，処理単位からアクセスできるメモリ領域

(B) 非信頼 OSAP1 からのみアクセスできないメモリ領域

(C) 非信頼 OSAP2 からのみアクセスできないメモリ領域

(D) 全ての非信頼 OSAP からアクセスできないメモリ領域

(E) 対象スタックを使用する処理単位からアクセスできるスタック領域へのアクセス

(F) 対象スタックを使用する処理単位からアクセスできないスタック領域へのアクセス

つまり，これら 6 つのメモリ領域に分割して，各処理単位からアクセスすることで，すべての組み合わせを網羅することができる．テストに使用したプロセッサでは，読み出し・書き込み・実行のアクセス保護属性を設定可能であったので，各メモリ領域に対しては，読み出しの先頭アドレス，終端アドレス，書き込みの先

メモリ領域の種類		アクション番号				
		読出/先頭	読出/終端	書込/先頭	書込/終端	実行
ROM	OSの専用コード領域	0	1	2	3	4
	OSの専有リードオンリーデータ領域	5	3	7	8	9
	非信頼OSAP1の専有コード領域	0	1	10	11	2
	非信頼OSAP1の専有リードオンリー領域	3	4	12	13	14
	非信頼OSAP2の専有コード領域	0	1	15	16	2
	非信頼OSAP2の専有リードオンリー領域	3	4	17	18	19
	共有コード領域	0	1	20	21	2
	共有リードオンリーデータ領域	3	4	22	23	24
RAM	OSの専有リードライト領域（スモールデータ）	25	26	27	28	29
	非信頼OSAP1の専有リードライトデータ領域(スモールデータ)	5	6	7	8	30
	非信頼OSAP2の専有リードライトデータ領域(スモールデータ)	5	6	7	8	31
	共有リード専有ライトデータ(スモールデータ)(共有リード領域)	5	6	32	33	34
	共有リード専有ライトデータ(スモールデータ)(非OSAP1の専有ライト)	7	8	9	10	35
	共有リード専有ライトデータ(スモールデータ)(非OSAP2の専有ライト)	9	10	9	10	36
	共有リードライトデータ(スモールデータ)	11	12	13	14	37
	OSの専有リードライト領域	38	39	40	41	42
	OSの専有リードライト領域(スタック)	43	44	45	46	47
	非信頼OSAP1の専有リードライトデータ領域	11	12	13	14	48
	非信頼OSAP2の専有リードライトデータ領域	11	12	13	14	49
	共有リード専有ライトデータ領域(共有リード領域)	15	16	50	51	52
	共有リード専有ライトデータ領域(非OSAP1の専有ライト)	17	18	15	16	53
	共有リード専有ライトデータ領域(非OSAP2の専有ライト)	19	20	15	16	54
	共有リードライトデータ領域	21	22	23	24	55
	非信頼OSAP1のタスク1スタック	0 0	1 1	2 2	3 3	4
	非信頼OSAP1のタスク2スタック	0 0	1 1	2 2	3 3	4
	非信頼OSAP2のタスク1スタック	0 0	1 1	2 2	3 3	4
	非信頼OSAPのフックのスタック	0 0	1 1	2 2	3 3	4
その他	メモリ保護の設定がされていない領域	×	×	×	×	×

すべてのOSAP，処理単位からアクセスできるメモリ領域へのアクセス	対象スタックを使用する処理単位からアクセスできるスタック領域へのアクセス
非信頼OSAP1からのみアクセスできないメモリ領域へのアクセス	対象スタックを使用する処理単位からアクセスできないスタック領域へのアクセス
非信頼OSAP2からのみアクセスできないメモリ領域へのアクセス	アクセス保護属性なしのメモリ領域
全ての非信頼OSAPからアクセスできないメモリ領域へのアクセス	

図 6.11: メモリアクセスパターン (抜粋)

頭アドレス，終端アドレス，先頭アドレスの実行の 5 パターンでアクセスを行うことにした．

図 6.11 に示した通り，各メモリ領域に定義可能なメモリ領域の種類が，先に挙げた 6 つのメモリ領域のどれに該当するかを色分けし，メモリ領域毎に 0 から連番で番号を付与する．この番号をアクション番号と呼ぶ．テストプログラムにおけるメモリ領域へのアクセスは，6 つのメモリ領域毎に，アクセス用の関数 (以下，メモリアクセス関数) を用意しておき，メモリアクセス関数の引数にアクション番号を与える実装とする．ただし，(E)(F) のスタック領域に関しては，アクセス対象のスタック毎に，アクセスするメモリ領域を切り替えるため，アクション番号はそれぞれの領域に対して 0 から連番となる．

ATK2 では，保護違反時処理パターンは，以下の 5 つとなる．

(I) プロテクションフックの戻り値が PRO_SHUTDOWN (OS をシャットダウンする)

(II) プロテクションフックの戻り値が PRO_TERMINATETASKISR (アクセス保護違反を引き起こしたタスクを強制終了する)

(III) プロテクションフックの戻り値が PRO_IGNORE (アクセス保護違反は無視できないため，シャットダウンされる)

(IV) プロテクションフックの戻り値が規定外 (規定外の戻り値の場合，シャットダウンされる)

(V) プロテクションフックなし (アクセス保護違反発生時点で，OS をシャットダウンする)

アクセス保護違反が発生しないテストケースでは (V) のみとする．

AKTG，PictMaster の使用

アクセス保護テストは，各パターンの組み合わせにより，テストケース数が膨大となるので，API テスト同様，AKTG によるテストプログラムの生成と，PictMaster によるテストケースの生成を用いた効率化が求められる．

TESRY 記法は，前状態，処理，後状態の組み合わせだけでなく，後続する処理，後状態を続けて記述することができる．したがって，図 6.9 に示したようなアク

セス保護テストのテストシナリオを，TESRY 記法で記述すれば，AKTG によってテストプログラムを生成することが可能である．図 6.9(a) のテストシナリオを，TESRY 記法で記述した例を図 6.9(b) に示す．

前節で述べた，処理単位の切り替えパターンと保護違反時処理パターンについては，PictMaster により，容易に組み合わせが可能であり，メモリアクセスパターンを除いたパターンを組み合わせた TESRY データは，API テストと同様の手法で生成することができた．この TESRY データは，プロセッサに依存しない．

プロセッサに依存する部分の明確化

メモリアクセスパターンは，プロセッサや MPU によって定義できるメモリ領域の種類が異なり，読み出し・書き込み・実行の区別が付かない可能性もある．つまり，アクション番号の上限値がプロセッサ毎に異なる．したがって，前節で述べたプロセッサに依存しない TESRY データに対して，何らかの方法で，プロセッサに依存するメモリアクセスパターンを組み合わせる必要がある．

そこで，まず前節で述べた PictMaster による組み合わせの段階では，プロセッサに依存しない TESRY データの基本パターンのみ生成しておく．具体的には，各メモリアクセス関数の引数を「xxx」といった識別子にしておき，アクション番号による組み合わせは行わない．テスト対象のプロセッサが確定し，メモリアクセスパターンが確定した段階で，メモリアクセス関数毎の上限値を設定したシェルスクリプトによって，基本パターンの TESRY データを，必要な数分の TESRY データへ複製し，メモリアクセス関数の引数を数値に変換することにした．

アクセス保護テスト設計のデータフローを図 6.12 に示す．この設計により，新たなプロセッサに対してアクセス保護テストを実施する際は，メモリアクセスパターンを，プロセッサに応じて作成し，シェルスクリプトにアクション番号上限値情報を入力するのみで対応できる．なお，メモリアクセス関数も，引数に応じたアクセス処理を行う必要があるため，プロセッサ毎に実装が必要となる．

6.5　評価

本研究では，6.4 節で述べた手法を用いて，ATK2 に対してテストを実施した．ATK2 の開発，テストには，Terasic 社の Altera DE2-115 Development and Education Board(以下，評価ボード) を使用した．評価ボードには，Altera 社の

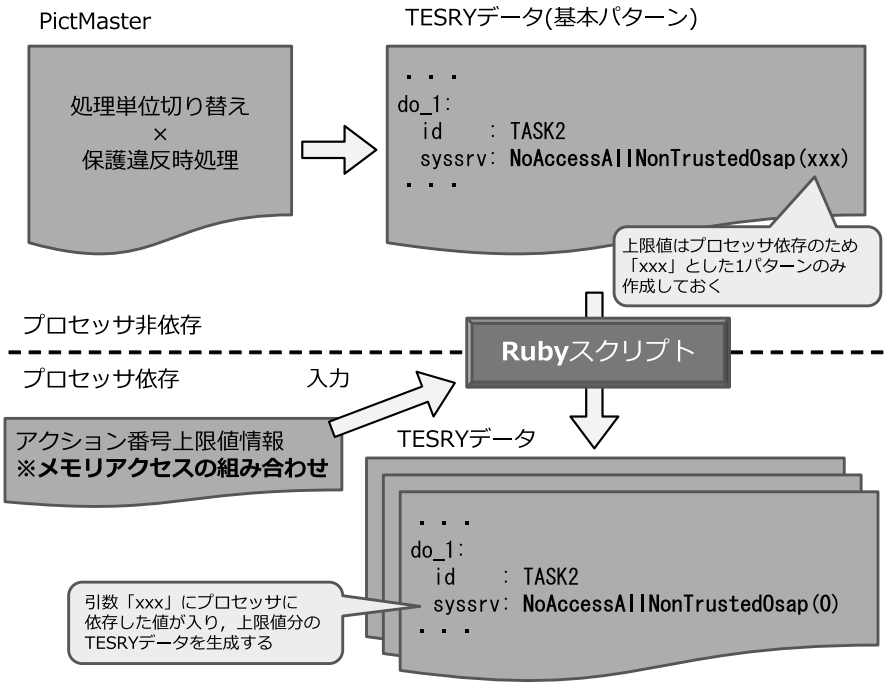


図 6.12: アクセス保護テスト設計のデータフロー

FPGA 向けの Nios2 プロセッサが搭載されている．Nios2 の MPU では，コード領域用の領域を 32 個，データ領域用の領域を 32 個と使用でき，読み出し，書き込み，実行のすべてを区別してアクセス制御できる．

本節では，AKTSP を用いて，ATK2 をテストした結果について評価する．表 6.2 に，テスト実施結果の一覧を示す．TTG では，TESRY データの行数に対するテストプログラムの行数が約 3.7 倍であったのに対し，AKTG では約 2 倍となっているのは，6.4.1 節で述べたコードサイズ削減の影響である．なお，TTG，AKTG ともに Ruby で開発し，TTG の総行数は 19,328 行であったのに対し，AKTG は 39,182 行であった．コード増加の理由は，メモリ保護機能への対応である．

6.5.1 API テスト

テストケース数

まず，A-OS 仕様の SC の中で，メモリ保護機能に対応していないスケーラビリティクラス 1(SC1) に対する API テストと，ASP カーネルに対する API テストについて比較を行う．API の数は，ASP カーネルに比べて SC1 には 3 分の 1 程度し

表 6.2: テスト実施結果一覧

	ASP カーネル	SC1	SC3
OS 仕様			
API 数	103	35	57
メモリ保護機能	なし	なし	あり
API テスト			
PictMaster シート数	-	112	223
テストケース数	1,675*	7,334#	40,633#
TESRY データ行数	60,648*	431,834#	2,837,186#
テストプログラム行数	224,719#	949,757#	5,884,643#
アクセス保護テスト			
PictMaster シート数	-	-	185
テストケース数	-	-	43,244#
TESRY データ行数	-	-	5,715,216#
テストプログラム行数	-	-	12,731,538#

*:手動で作成，#:ツールによる生成

か存在しないが，テストケース数は約 4.4 倍となっている．6.4.1 節で述べた通り，A-OS 仕様は組み合わせるパラメータが多いことで，メモリ保護機能がない場合でも，μITRON 仕様に比べてテストケース数が大幅に増加することが確認できた．

さらに，SC1 と SC3 では，API の数は約 1.6 倍であるが，テストケース数は約 5.5 倍になっている．これは，メモリ保護機能があることにより，タスクなどの OS オブジェクトに信頼，非信頼の組み合わせが増えることが原因である．メモリ保護機能への対応により，追加が必要となるテストの規模を明らかにすることができた．

実施工数

PictMaster の Execl ファイルは，API 毎に作成したが，各ファイルには制約表や TESRY 生成表の可読性を向上するために，正常系，異常系などのカテゴリー毎に，別シートで作成した．1 つのシート作成は，レビューを含めても 1 人日程度であるので，SC1 全体で約 5.6 人月，SC3 全体で約 11 人月規模の作業である．一方，SC1 と SC3 で合計約 327 万行規模の TESRY データを人手で開発，保守することは現実的でない．1 人日で 50 件の TESRY データを作成できたとしても，SC1 と SC3 のすべての TESRY データを作成するには，約 48 人月要するため，API テストの実施工数を大幅に削減できたと言える．

組み合わせ結果の正当性

PictMaster を用いて，意図した組み合わせが生成されていることは，レビューで確認した．csv ファイルは，Excel で開くことにより，行のフィルタ機能や並べ替えが使用可能となるので，組み合わせが網羅されているか，容易に確認することができる．また，AKTG によって生成したテストプログラムを実行した結果として，各 API の実行回数とソースコードカバレッジが想定通りの結果となることを確認した．

品質の向上

本研究の目的の 1 つは，メモリ保護機能を持つ RTOS の品質を向上することである．本手法を用いることにより，特定の処理単位から呼び出した場合にのみ発生する不具合を検出できた．これにより，ATK2 の品質向上に貢献し，この目的を達

成したと考えられる．人手で実施した場合，テストケース削減のために実施されない可能性もあるため，PictMaster により，すべての処理単位から呼び出すように組み合わせた効果があったと言える．なお，API テストにより，検出した ATK2 の API に関する不具合は 7 件であった．

6.5.2　アクセス保護テスト

実施工数

PictMaster の Execl ファイルは，図 6.10 に示した 6 つのパターン毎に作成したが，プロテクションフックの有無などによって 1 つのテストプログラムとして実行できないテストケースの組み合わせを分け，別シートで作成した．API テストと異なり，アクセス保護テストの PictMaster ファイルは，基本構成が同じであるので，1 つのシート作成は，レビューを含めても 0.3 人日程度であり，全体で約 2.8 人月規模の作業である．API テスト同様，アクセス保護テストも実施工数を大幅に削減できたと言える．

品質の向上

アクセス保護テストにおいても，特定の処理単位切り替えの組み合わせのみで発生する不具合を検出できたことから，メモリ保護機能を持つ RTOS の品質向上を達成したと考えられる．不具合の中には，特定の処理単位切り替えの組み合わせのみ検出されるものも存在したため，API テスト同様，網羅的な組み合わせの効果があったと言える．なお，アクセス保護テストにより，検出した ATK2 のアクセス保護に関する不具合は 5 件であった．

また，この 5 件とは別に，ルネサスエレクトロニクス社の RH850/F1L など Nios2 以外のプロセッサへ ATK2 をポーティングした際に，ATKSP のアクセス保護テストにより不具合を検出した事例が 7 件報告されている．主に以下の 3 つのカテゴリーに分類される不具合である．

- 処理単位切り替え時のスタックポインタ設定ミス
- MPU に設定する情報のミス
- プロセッサ，コンパイラが持つ制約の誤解

いずれも，プロセッサ，コンパイラに依存した不具合である．アクセス保護のポーティングは，プロセッサやコンパイラの仕様に依存する要素が多く，難易度も高いため，不具合を生みやすい．ポーティング後に，アクセス保護に対して網羅的にテストを実施することに有用性があると言える．

また，アクセス保護テストの手法は，アクセス保護機能だけでなく，テストシナリオの基本パターンがあり，条件を組み合わせる必要のあるテスト全般に対して有効な手法である．

6.5.3 A-OS 仕様への対応

6.2.1 節で述べた通り，ATK2 には，実装していない A-OS 仕様が存在する．これらの仕様を実装した場合に，提案したテスト手法を適用する方法について考察する．

非信頼 OSAP に ISR が所属する場合，処理単位の切り替えパターンに，非信頼 OSAP に所属する ISR を追加するだけよい．OSAP の強制終了，再起動に対応する場合，保護違反時処理パターンに，PRO_TERMINATEAPPL と PRO_TERMINATEAPPL_RESTART を戻り値とする組み合わせを追加するだけでよい．

以上から，A-OS 仕様への対応は，各パターンの組み合わせを追加するのみでよく，容易に実施可能であると考えられる．加えて，AKTG に，非信頼 OSAP に所属する ISR に対するプログラム生成処理を追加する必要があるが，技術的な課題はない．

6.6 関連研究

RTOS のテスト手法に関する研究はいくつか存在する [55][56][57][58][59][60]．文献 [60] では，RTOS に適用可能なソフトウェアテスト手法について述べているが，具体的なテストプログラムの実装には言及していない．文献 [55][56][57][58][59] は，モデル検査によりテストケースの組み合わせ生成を行っているが，本研究が対象とするメモリ保護機能に対するテストケースは，組み合わせツールの PictMaster で生成可能であり，モデル検査の必要性は低いと考えている．また，一般的なソフトウェアに対する研究として，テストケースの組み合わせ手法や，テストプログラムを生成するツールに関するいくつかの研究が存在する [38][39][61][62][63]．いず

れの研究も，特定の機能に特化しているものや，メモリ保護機能を持つ RTOS に対するテストプログラムの生成には対応していないものであり，本研究には適用できない．

メモリ保護機能に対応した RTOS のテストに関しては，μITRON 仕様のメモリ保護機能拡張 (PX 仕様) に対応した TOPPERS/HRP を対象とした，高信頼化技術のハンドブックが提案されている [49]．しかし，メモリ保護機能に対して，具体的にどのような手法でテストを実施するかまでは言及されていない．

メモリ保護機能に対するテストとしては，Linux カーネルの W⊕X プロテクション機能に対してモデル検査を行う研究がある [64]．しかし，モデル検査により，W⊕X プロテクション機能の解析，問題抽出を行っているのみで，テストケースやテストプログラムの生成には言及していない．

6.7　まとめ

本章では，メモリ保護機能に対応した RTOS である ATK2 に対するテスト手法を確立し，API テスト，およびアクセス保護テストを実施した．これにより，ATK2 の不具合を 19 件検出し，ATK2 の品質向上に貢献した．

国内の AUTOSAR のニーズの高まりを受け，ATK2 が利用されるケースも増加している．本テスト手法の確立以降，ATK2 は，様々なプロセッサにポーティングされ，AKTSP が使用されており，アクセス保護テストの有用性が評価されている．

第7章　結論

7.1　まとめ

本研究では，高信頼性システム向けRTOSの品質を向上するためのツールやテスト手法に関する4つの研究を実施した．

1つ目の研究では，μITRONベースのRTOSに対するテストを分析し，テストプログラム生成ツールの必要性について述べた．さらにテストプログラム生成ツールであるTTGを開発し，実際にテストスイートの開発・テストの実施を行い，ツールの評価を行った．TTGで生成したテストプログラムを実行することで，仕様と実装に相違がある不具合や，ターゲット依存部の実装不具合を検出した．

2つ目の研究では，μITRONベースのマルチプロセッサ向けRTOSであるFMPカーネルに対して，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ーを策定し，プロセッサ間同期制御ライブラリ，拡張ISSを開発し，マルチプロセッサ特有の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．

3つ目の研究では，AUTOSAR OSを対象にAPIテストを実施するため，TTGをベースとして，AUTOSAR OS向けにテストプログラム生成ツールAKTGを開発した．AUTOSAR OS仕様のRTOSに対するテストでは，テストプログラムとは別に，RTOSのコンフィギュレーション情報に応じて，arxmlファイルとOs_MemMap.hも用意する必要があるため，入力するテストシナリオのデータに応じて，AKTGでこれらのファイルも生成可能とし，テストに必要なファイルの開発コストを大幅に削減した．テスト実行時間が膨大になる問題に対しては，連続して実行することが困難なテストケースを強制的にマージして実行する手法を考案した．実際に，商用AUTOSAR OSであるJulinarOsに対して本テスト手法を適用し，テス

トプログラム開発コストおよび，テスト実施時間の大幅な削減を実現した．

4 つ目の研究では，メモリ保護機能に対応した AUTOSAR OS である ATK2 に対して API テスト，およびアクセス保護テストを実施した．AUTOSAR OS 仕様は，μITRON 仕様と比べて，OS オブジェクトに設定可能なパラメータが多く，テストケース数が膨大となる問題に対して，テストケースの組み合わせツールを用いたテストケース生成を行った．アクセス保護テストでは，アクセス保護機能に対するテストパターンを導出し，テストのポータビリティ向上のため，プロセッサに依存する部分の明確化を行った．これにより，ATK2 の不具合を 19 件検出し，ATK2 の品質向上に貢献した．

以上の 4 つの研究成果により，提案したツール，テスト手法が RTOS の品質向上に有効であることを示した．また，開発したツール，テストスイートの保守性，再利用性，拡張性の高さを示した．本テスト手法の確立以降，様々なプロセッサに ASP カーネル，FMP カーネル，ATK2，JulinarOs がポーティングされ，本テスト手法が使用されており，有用性が評価されている．

7.2　今後の課題

TTG，AKTG の開発により，テストケースに変更が発生した場合でも，TESRY データの修正のみを行えばよいため，テストプログラムの保守性が高いことは確認できた．しかし，現状，TESRY データはテキストファイルで管理しているため，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．今後，さらなる保守性の向上のため，仕様書，ソースコード，TESRY データ間でのトレーサビリティを向上させるツールや手法についても検討が必要である．

また，今後の取り組みとして，表 2.2 に挙げた残りのテストの実施，および非機能要件のテストが挙げられる．特にジェネレータが生成したコードは，実際に高信頼性システムで使用されるため，OS の静的コードと同様に品質確保が必要である．本研究で示したテスト手法により，多くのコンフィギュレーションの組み合わせがテストされるため，ジェネレータの大部分はテストされていると考えられるが，生成されるコードの組み合わせは無限にあるため，ジェネレータ自体の品質を確保するためのテスト手法を確立することが有用であると考えられる．

謝辞

本論文に関する研究活動を通じて，御指導と御助言を頂きました名古屋大学大学院情報学研究科情報システム学専攻の高田広章教授，本田晋也准教授に，心より御礼申し上げます．本研究を進めるに当たり，多くの御提案，御指摘を頂き，研究成果の有用性向上に協力して下さった，名古屋大学大学院情報学研究科附属組込みシステム研究センター，およびコンソーシアム型共同研究の皆様に謹んで感謝の意を表します．また本博士論文をまとめるにあたり，多くの有益な御助言を賜りました，名古屋大学大学院情報学研究科情報システム学専攻の関浩之教授にも感謝致します．本博士課程入学の機会を与えて頂きました，富士ソフト株式会社にも感謝致します．

最後に，本論文の完成に至るまで，あたたかく支え続けてくれた家族に心から感謝します．

参考文献

[1] 経済産業省 2010 年版組込みソフトウェア産業実態調査報告書. http://www.meti.go.jp/policy/mono_info_service/joho/downloadfiles/2010software_research/index.html

[2] AUTOSAR. https://www.autosar.org/

[3] 坂村健/監修, 高田広章/編. μITRON4.0 仕様 Ver.4.02.00. トロン協会, 2004.

[4] Specification of Operating System. https://www.autosar.org/

[5] TOPPERS プロジェクト. http://www.toppers.jp/

[6] TOPPERS 新世代カーネル統合仕様書 Release 1.2.0. http://www.toppers.jp/docs/tech/ngki_spec-120.pdf

[7] TOPPERS/ASP カーネル. http://www.toppers.jp/asp-kernel.html

[8] TOPPERS/FMP カーネル. http://www.toppers.jp/fmp-kernel.html

[9] RTEMS. http://www.rtems.com/

[10] AMP T-Kernel. http://www.tron.org/download/index.php?route=product/category&path=25_29

[11] VxWorks. https://www.windriver.com/japan/products/vxworks/

[12] SMP T-Kernel. http://www.tron.org/download/index.php?route=product/category&path=25_28

[13] A. Zahir, P. Palmieri. OSEK/VDX-operating systems for automotive applications. OSEK/VDX Open Systems in Automotive Networks, IEE Seminar, 1998.

[14] TOPPERS/ATK2. http://www.toppers.jp/atk2.html

[15] TOPPERS/HRP2 カーネル. http://www.toppers.jp/hrp2-kernel.html

[16] 石川拓也, 本田晋也, 高田広章. 静的なメモリ配置を行うメモリ保護機能を持ったリアルタイム OS. コンピュータ ソフトウェア, Vol.29,No.4, pp.161–181, 2012.

[17] APTJ Co., Ltd., http://www.aptj.co.jp/

[18] Specification of Memory Mapping. https://www.autosar.org/

[19] Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler. The art of software testing, John Wiley & Sons. Inc, 2004.

[20] 玉井哲雄. ソフトウェア工学の基礎. 岩波書店, 2004.

[21] 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.

[22] TOPPERS/TTSP. http://www.toppers.jp/ttsp.html

[23] 小林隆志, 沢田篤史, 山本晋一郎, 野呂昌満, 阿草清滋. On the Job Learning: 産学連携による新しいソフトウェア工学教育手法. 電子情報通信学会 信学技報 SS2009-28, vol.109, no.170, pp.95–100, 2009.

[24] 独立行政法人 情報処理推進機構 高信頼化ソフトウェアのための開発手法ガイドブック. https://www.ipa.go.jp/sec/softwareengineering/reports/20100915.html

[25] IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, 1990.

[26] IEEE Standard for Software Test Documentation. IEEE Std 829-1998, 1998.

[27] IBM developerWorks 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. https://www.ibm.com/developerworks/jp/java/library/j-cq01316/

[28] YAML. http://yaml.org/

[29] Sourcery G++ Lite. https://www.mentor.com/embedded-software/sourcery-tools-services/

[30] SyeEye. https://sourceforge.net/projects/skyeye/

[31] GCC GCOV. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[32] Linux Test Project. https://github.com/linux-test-project/ltp

[33] Subrata Modak, Balbir Singh. Building a Robust Linux Kernel piggybacking The Linux Test Project. Linux Symposium (OLS), 2008

[34] FreeRTOS. https://www.freertos.org/

[35] D.Clarke, T.Jeron, V.Rusu, E.Zinovieva, STG: A symbolic test generation tool. Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference (TACAS 2002), vol.2280 of LNCS, pp.470–465, 2002.

[36] Wilkerson L.Andrade, Patricia D.L.Machado, Everton L.G.Alves, Diego R.Almeida, Test Case Generation of Embedded Real-Time Systems with Interruptions for FreeRTOS. Formal Methods: Foundations and Applications, pp.54–69, 2009.

[37] V.Rusu, L.du Bousquet, T.Jeron, An approach to symbolic test generation. International Conference on Integrating Formal Methods, vol.1945 of LNCS, pp.338–357, 2000.

[38] A.Belinfante, J.Feenstra, R.de Vries, J.Tretmans, N.Goga, L.Feijs, S.Mauw, Formal test automation: a simple experiment. International Workshop on the Testing of Communication Systems (IWTCS'99), pp.179–196, 1999.

[39] T.Jeron and P.Morel, Test generation derived from model-checking. Computer Aided Verification (CAV'99), vol.1633 of LNCS, pp.108–122, 1999.

[40] D.Clarke, I.Lee, Automatic Test Generation for the Analysis of a Real-Time System: Case Study. IEEE Real-Time Technology and Applications, pp112–124, 1997.

[41] D.Clarke, I.Lee, H.Xie, VERSA: A Tool for the Speci cation and Analysis of Resource-Bound Real-Time Systems. Journal of Computer and Software Engineering, 3(2), 1995.

[42] 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, 2012.

[43] TOPPERS プロジェクト／適用事例. http://www.toppers.jp/applications.html

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 「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. http://www.nces.is.nagoya-u.ac.jp/press/nces-release-1105.pdf

[49] 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.

[50] Specification of ECU Configuration. https://www.autosar.org/

[51] Green Hills Software. https://www.ghs.com/products/compiler.html

[52] PictMaster. https://ja.osdn.net/projects/pictmaster/

[53] Shai Rubin, Moshe Levinger, Randall R. Pratt, William P. Moore. Fast Construction of Test-Program Generators for Digital Signal Processors. IEEE International Conference On Acoustics, Speech, and Signal Processing, 1999.

[54] Francisco X. Duarte, José C. Alves, José Machado da Silva, Gabriel Pinho, and José S. Matos. A High Level Test Processor and Test Program Generator. Conference on Design of Circuits and Integrated Systems, 2005.

[55] Jiang Chen, Aoki T. Conformance Testing for OSEK/VDX Operating System Using Model Checking. Software Engineering Conference (APSEC), pp.274–281, 2011.

[56] Yunja Choi, Taejoon Byun. Constraint-based test generation for automotive operating systems. Software & Systems Modeling, pp.1–18, 2015.

[57] Ling Fang and Kitamura T., Thi Bich Ngoc Do, Ohsaki H. Formal Model-Based Test for AUTOSAR Multicore RTOS. Software Testing, Verification and Validation (ICST), pp.251–259, 2012.

[58] 青木利晃, 山崎真吾. モデル検査によるリアルタイムオペレーティングシステムの設計検証. 組込みシステムシンポジウム 2008 論文集, pp.159–166, 2008.

[59] 加藤淳, 神武直彦, 春山真一郎, 狼嘉彰. モデル検査を用いて組込みシステムにおけるソフトウェアとハードウェアの協調動作に関する要求仕様の不整合を検出する手法. 組込みシステムシンポジウム 2009 論文集, pp.65–70, 2009.

[60] Tsoukarellas M.A., Gerogiannis V.C., Economides K.D. Systematically testing a real-time operating system. Micro, IEEE, pp.50–60, 1995.

[61] 丹野治門, 張暁晶, 星野隆. 結合テストにおけるテスト項目自動生成手法の提案と評価. 電子情報通信学会技術研究報告, Vol.110, No.227, pp.37–42, 2010.

[62] 土屋達弘, 菊野亨. ペアワイズテスト:ソフトウェアテストの効率化を求めて. 電子情報通信学会論文誌 D, Vol.J90-D, No.10 , pp.2663–2674, 2007.

[63] 秋山浩一 ソフトウェアテストの最新動向：3. 組合せテストの設計. 情報処理学会誌, Vol.49, No.2, pp.140–146, 2008.

[64] Siarhei Liakh, Michael Grace, Xuxian Jiang. Analyzing and improving Linux kernel memory protection: a model checking approach. ACSAC '10 Proceedings of the 26th Annual Computer Security Applications Conference, pp.271–280, 2010.

研究業績

学術論文

- 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, Mar. 2012.
- 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, Apr. 2012.
- 鴫原一人，一場利幸，本田晋也，高田広章，“μITRON ベースのマルチプロセッサ向け RTOS のテスト”，情報処理学会論文誌，Vol. 53, No. 12, pp.2682–2701, Dec. 2012.
- 鴫原一人，石川拓也，本田晋也，高田広章，“メモリ保護機能を持つ AUTOSAR 仕様ベースのリアルタイム OS に対するテスト”，情報処理学会論文誌，Vol. 57, No. 8, pp.1737–1751, Aug. 2016.

国際会議論文（査読あり）

- Kazuto Shigihara, Shinya Honda and Hiroaki Takada, ”Test Program Generator for AUTOSAR OS,” 2017 13th European Dependable Computing Conference (EDCC2017), pp. 79–86, Sep. 2017.

国内会議発表論文（査読あり）

- 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 (JaSST’10 Tokyo), Jan. 2010.
- 金ハンソル，浅見侑太，阿部真也，金スンヨプ，金榮柱，金賢敏，竹谷美里，木村貴寿，鴫原一人，森孝夫，山本雅基，本田晋也，高田広章，“組込みリア

ルタイム OS 向けテストツールのマルチプロセッサ拡張”，ソフトウェアテストシンポジウム 2011 (JaSST’11 Tokyo), Jan. 2011.

- 森孝夫，鴫原一人，本田晋也，高田広章，“コンソーシアム型共同研究によるマルチプロセッサ向け RTOS のテストスイート開発”，組込みシステムシンポジウム 2011 (ESS2011), Oct. 2011.
- 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 (JaSST’12 Tokyo), Jan. 2012.

研究会発表論文（査読なし）

- 鴫原一人，森孝夫，本田晋也，山本雅基，高田広章，“RTOS のテスト自動生成システムに関する一考察”，情報処理学会研究報告 組込み技術とネットワークに関するワークショップ (ETNET2010), Mar. 2010.
- 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 組込み技術とネットワークに関するワークショップ (ETNET2010), Mar. 2010.
- 鴫原一人，眞弓友宏，本田晋也，高田広章，“マルチプロセッサ対応 RTOS を対象としたテストシナリオ記述法とテストプログラム生成ツール”，情報処理学会研究報告 第 18 回組込みシステム研究会, Aug. 2010.
- 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 組込み技術とネットワークに関するワークショップ (ETNET2011), Mar. 2011.
- 阿部真也，鴫原一人，本田晋也，山本晋一郎，高田広章，“テストプログラム生成ツールのフロントエンドプロセッサの開発”，情報処理学会研究報告 組込み技術とネットワークに関するワークショップ (ETNET2011), Mar. 2011.
- 金榮柱，金スンヨプ，金ハンソル，金賢敏，竹谷美里，木村貴寿，鴫原一人，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 に

対するAPIテストの実施”，情報処理学会研究報告 組込み技術とネットワークに関するワークショップ(ETNET2011), Mar. 2011.

- 一場利幸，高瀬英希，鴫原一人，本田晋也，高田広章，“マルチプロセッサ環境におけるタイミング依存のシナリオを実行可能なシミュレーション機構”，情報処理学会研究報告 組込み技術とネットワークに関するワークショップ(ETNET2011), Mar. 2011.
- 風間佳之，作道直樹，段慧，木戸脇有美，平橋航，鴫原一人，海上智昭，本田晋也，高田広章，“メモリ保護対応RTOSにおけるメモリ保護機能のテスト”，情報処理学会研究報告 組込み技術とネットワークに関するワークショップ(ETNET2013), Mar. 2013.
- 廉正烈，鴫原一人，海上智昭，本田晋也，高田広章，“リアルタイムOSに対するMC/DC計測手法”，情報処理学会研究報告 組込み技術とネットワークに関するワークショップ(ETNET2014), Mar. 2014.
- 鴫原一人，本田晋也，高田広章，“メモリ保護対応RTOSにおける非信頼関数の実現手法”，情報処理学会研究報告 組込み技術とネットワークに関するワークショップ(ETNET2018), Mar. 2018.