# 15-B-13

# モデル検査とテストによる車載オペレーティングシステムの検証 [1]

## 1. はじめに

車載ソフトウェアの安全性や信頼性に関する問題は、社会において非常に大きな関心となりつつある。車は、従来は機械的に制御されてきたが、近年、コンピュータ制御技術の発展と利便性や性能の追求により、多くの部品の電子化が進んできている。これにより、車載ソフトウェアの規模の急速な増大と複雑化がもたらされ、主に、電子制御部分の安全性と信頼性に関する問題が取り上げられつつある。世界標準においては、機能安全に関する標準が一般の電子システムだけでなく[3]、車載システムに特化されたものが策定されている[2]。また、実社会においては、2010 年に発生したトヨタ車の急加速問題において、電子スロットル制御システムの検証が NHTSA[2]と NASA により実施された[1]。

我々は、このような車載ソフトウェアの安全性や信頼性の問題を背景に、車載オペレーティングシステムの検証手法の研究と実践を行っている。対象としている OS は、OSEK/VDX[4]に準拠するものである（以降、OSEKOS）。OSEK/VDX は ECU のアーキテクチャの業界標準を作成することを目的とした団体であり、1993 年に設立された。標準化の対象には様々なものがあるが、それらの中の 1 つに OS に関するものがある。現在は、AUTOSAR[5]により標準化活動が引き継がれているが、実際使われているのは OSEK/VDX に準拠しているものが多く見受けられる。以降、AUTOSAR に準拠した OS を AUTOSAROS と記述する。AUTOSAROS は、OSEKOS に保護機能やマルチコア機能などの拡張がされており、スケジューリングなどの基本的な機能は OSEKOS と同じである。よって、OSEKOS の検証法は、AUTOSAROS にも、同様に適用できるものと考えている。

OS は車載ソフトウェアの基盤であり、安全性の評価の際、非常に重要な位置づけとなる。そこで、世界標準において、高い安全性を目標とする際に推奨されている形式手法を採用し、社会的にも現実的にも品質の高い OS を提供することが動機である。我々は、2006 年から北陸先端科学技術大学院大学（以降、JAIST）と株式会社デンソー（以降、デンソー）により共同研究を開始した。デンソーは、対象 OS を用いた車載システムの開発を行っているため、主に、OS のユーザの立場から、検討を行った。その後、実際に車載システムに組み込まれている製品に適用するため、それを開発しているルネサスマイクロシステム株式会社（以降、

[1] 事例提供: 国立大学法人 北陸先端科学技術大学院大学　青木 利晃 氏
[2] National Highway Traffic Safety Administration 米国運輸省道路交通安全局

ルネサスマイクロシステム）が、2009 年から加わった。現在は、ルネサスシステムデザイン株式会社（以降、ルネサスシステムデザイン）が開発を行っている。検証対象の OS は、すでに製品化されており、従来の手法で検査が行われているものである。以降、この OS を RELOS と記述する。

このプロジェクトの目的は、RELOS を次世代の車に組み込むため、形式手法を適用することにより、さらに高い品質を達成することである。一方、形式手法を現実的なシステム開発に適用するためには、様々な工夫が必要であり、一筋縄では形式手法の産業応用ができないことも多々ある。そこで、ルネサスエレクトロニクス株式会社（以降、ルネサスエレクトロニクス）、デンソー、早稲田大学、JAIST と共同で、形式手法の実践課題を解決しながら、車載 OS の検証を行った。

## 2. 問題点と解決のためのアプローチ

### 2.1. 解決すべき問題点

RELOS は、すでに製品化されており、従来の手法で検査が行われている。その際、いくらかの誤りを発見および修正したと聞いている。それでも、RELOS の正しさに確信が持てなかったようである。アドホックなテストやテストケースを大きく間引いて行っている限りは、正しさの確信を持つことは難しい。信頼度成長曲線を用いることも考えられるが、統計的な指標であるため、それのみで正しさを確信することはできない。正しさを確信するための、なんらかの技術的な解決策が必要である。

### 2.2. アプローチ

図 15-B-13-1 に我々がとったアプローチの概要を示す。大きく分けて、設計検証とテストの 2 つのフェーズがある。設計検証では、RELOS の設計を仕様記述言語 Promela で記述し、モデル検査ツール Spin[10]を用いて検証を行う。これにより、設計モデルの正しさを十分に確認する。この設計検証により、正しさが十分に確認されたので、テストでは、設計モデルは正しいという前提を置く。すなわち、設計モデルを、いわゆる、golden モデル、とみなすのである。テストでは、RELOS の実装が設計モデルと適合していることをテストする。そのために、設計モデルから、網羅的なテストケースと期待値、および、それを確認するためのテストプログラムを自動生成する。

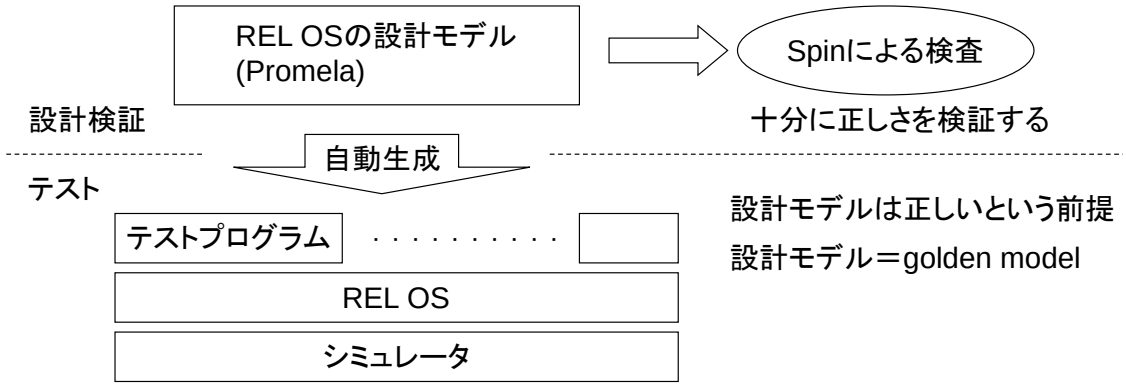


**図 15-B-13-1　RELOS の正しさを確信するためにとったアプローチの概要**

このプロジェクトの狙いは、RELOS のバグを発見することではなく、その正しさを確信することである。前述したが、検証対象の RELOS は、すでに実際に車載システムに組み込まれている。その際、デンソー、および、ルネサスにより、従来手法により検証が行われており、高い品質が確保されている。それでも、正しさの確信には至っていないため、形式手法を含む先進的なソフトウェア工学により、RELOS の正しさを確信することが目的である。正しさの確信のため、我々がとったアプローチは、形式手法により golden モデルを作成すること、および、網羅的なテストを行うことである。

## 2.3. 解決のためのアイデア：テストによる正しさの確信

RELOS の正しさを確信するためには、RELOS 自体を実際に実行して動作を確認することが望ましいと考えている。他の方法としては、検証可能な記述に変換したり、信頼がおけるモデルから自動生成したりすることが考えられる。しかしながら、ハードウェアを含めた振る舞いを検証することは困難である。また、このような方法で検証しても、一度も実行せずに、製品として出荷することは考えにくい。最終的には、実際に製品を実行して確認することが必要となるのである。そこで、我々は、テストにより RELOS 自体を実際に実行して、正しさの確信を行うことにした。この場合、実機やシミュレータにより実行するため、ハードウェアの振る舞いをモデル化する必要はない。しかしながら、実機やシミュレータの実行状況はモニタリングして解析する必要があるであろう。モデル化では、必要な振る舞いに注目して、抽象化したり抽出したりする。実機やシミュレータでは、そのような注目する振る舞いは、モニタリングして確認する必要がある。また、我々のアプローチでは、テストにおいて、シミュレータを用いている。このシミュレータは OS だけではなく、他のシステム開発でも多く使われており、十分に実績があるものである。確信を獲得するという目的では、実機の方が望ましいが、十分に実績があるシミュレータで代用している。

## 2.4. 解決のためのアイデア：golden モデルを介した確信の獲得

我々は、RELOS が正しいということを確信することが目的である。この場合、RELOS のソースコードや振る舞いに頼って、テストを実施することはできない。例えば、RELOS

の内部的な制御構造から、分岐を網羅するテストケースを求めることは、この目的の達成のためにはすべきではない。なぜならば、RELOS が正しいかどうか分かっていないので、それに基づいてテストケースを作成しても､正しいかどうか確信することはできないのである。もちろん、バグを効果的に見つけるという目的では、ある局面では有効に働くであろうが、正しいことを確信するためには適切ではない。

そこで、正しさを確信するためには、正しさの基準となる golden モデルを作成する必要があると考えた。我々のアプローチでは、golden モデルは正しい、RELOS は golden モデルと同じ振る舞いをする、よって、RELOS は正しい、と結論づけるのである。一方で、このアプローチでは、golden モデルが正しいと確信する必要がある。ここで、形式手法の出番である。形式手法により厳密で正しいモデルを作成し、golden モデルとするのである。そして、golden モデルと RELOS が同じ振る舞いがすることを、網羅的なテストにより保証するのである。

ここでは、正しさを確信するためのアプローチの全体像を述べたが、golden モデルを獲得するために設計検証と、golden モデルと RELOS の一致を確認するためのテストでは、解決しなければならない様々な技術的な問題が生じる。それらを解決するための技術要素の全体像を図 15-B-13-2 に示す。以下で「3.設計検証」において設計検証、「4.テスト」においてテストについて詳細を述べる。

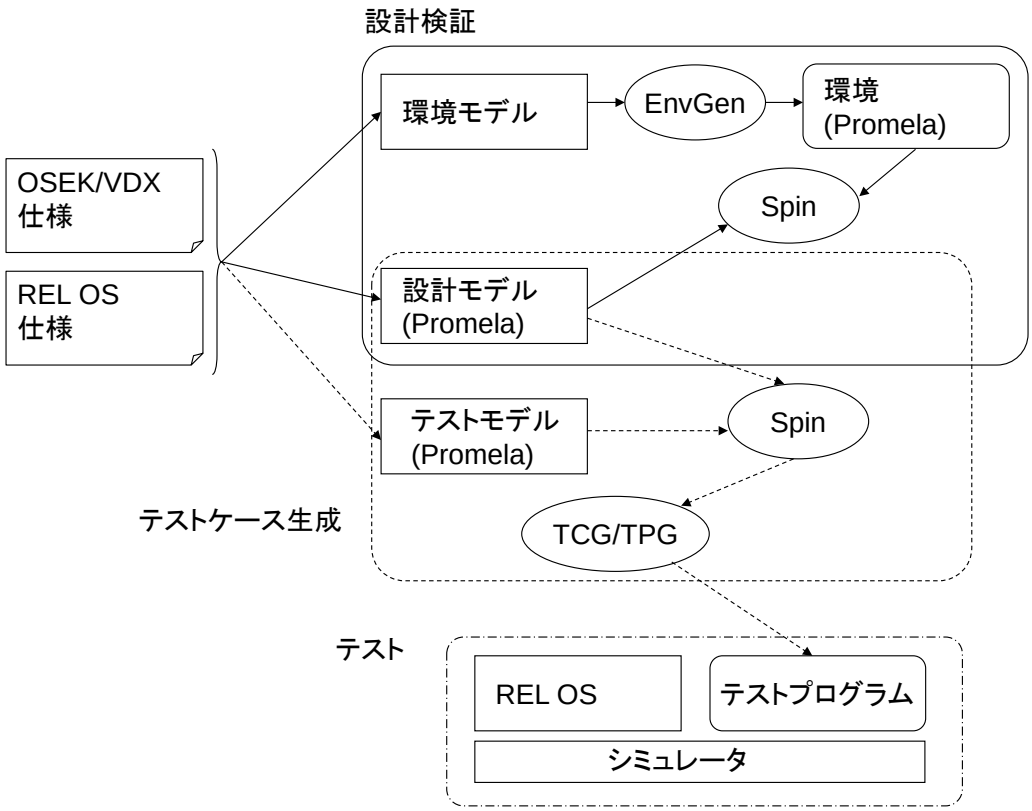


**図 15-B-13-2　技術要素の全体像**

# 3. 設計検証

OSEKOS の主な機能はタスクや ISR(InterruptServiceRoutine)を制御すること、すなわち、スケジューリングである。実際の OS では、キューやテーブルなどのデータ構造を用いて情報が管理され、それに基づいて実行するタスクや ISR を計算することにより実現される。このようなスケジューリングの実現は複雑である。様々なタスクや割り込みの組み合わせが考えられるため、どのような組み合わせでも仕様どおり動作することを保証することは難しい。そこで、本プロジェクトでは、RELOS のスケジューラが、スケジューリングを正しく行っていることを検証する。

## 3.1. 設計モデル

スケジューリングの仕組みを分析するため、RELOS の設計モデルを作成し、検証を行った。検証はモデル検査ツール Spin を用いて行った。Spin は、並行動作するチャネル通信オートマトンに基づいた振る舞いを対象に、LTL[3]などで表現された性質を自動的に検証する。検証対象のモデルは Promela と呼ばれる仕様記述言語で記述される。Promela は、並行プロセス単位で振る舞いを記述し、個々の並行プロセスは、ガードコマンドに基づいて手続き的に記述する。配列やレコード型など典型的なデータ型も使うことができ、スケジューリングのメカニズムを直接的に取扱いやすい。また、現場のエンジニアにとっては馴染みやすい言語であり、コミュニケーションをとりやすい。設計モデルは Promela で記述した。スケジューラは、レディキューと呼ばれる、タスクの起動順序や優先度などを記憶するデータ構造と、関連する情報を管理するテーブル、各種条件を表現するフラグなどから構成される。実行するタスクや ISR は、これらのデータに基づいて決定される。また、サービスコールが呼び出されると、データの内容を更新することにより、次に実行するものが決定される。これらのデータ構造と操作は、Promela で直接的に取り扱うことができる。作成した設計モデルの規模は約 2,800 行である。

## 3.2. モデル検査による設計検証と環境

OSEKOS は、タスクや ISR からのシステムサービスの呼び出しを受けて動作をするオープンシステムである。すなわち、タスクや ISR が無いと動作しないのである。設計モデルも同様で、実行可能なくらい詳細に記述はされているが、タスクや ISR からシステムサービスを呼び出さないと動作しない。そこで、Spin による検証を行うためには、RELOS の設計モデルとは別に、タスクや ISR などを表現する外部の記述が必要である。このような記述は環境と呼ばれている。

環境には、検証対象の記述と併せて閉じた記述となるよう、検証対象の機能の呼出し、検

[3] Linear Temporal Logic 線形時相論理

証対象からの呼び出し、入出力などについて記述する。このような環境には、すべての機能呼出や入出力を非決定的に行うものや、それらを特定の範囲で行うものがある。前者の環境は universalenvironment と呼ばれている[14]。universalenvironment は、網羅的ではあるが、検査する性質を時相論理式などで記述する際に、前提を明確に記述しなければならない[13]。そのため、検査する性質が書きづらくなりがちである。また、状態爆発問題も発生しやすい。後者の場合は、検査する性質の前提を環境で表現することができるため、性質自体の記述が単純になり、検査する範囲も限定することができる。

OSEKOS の設計検証では、スケジューリングが仕様どおりであることを確認したい。そのためには、実行されることが期待されるタスク、もしくは、ISR を、検証する性質として記述しなければならない。しかしながら、これらは、タスク構成やシステムサービスの呼び出し順に依存するため、非決定的なパラメータの設定やシステムサービスの呼び出しに基づいて記述することは困難である。そこで、我々は、universalenvironment を用いるのではなく、環境のモデル化を行うことにした。タスクの数、資源の数、優先度の割り当て方、資源の使用関係などをクラス図と OCL(ObjectConstraintLanguage)によりモデル化を行い、システムサービスの呼び出し列を拡張したステートチャート図と OCL を用いて記述する。さらに、期待する性質も、それらのモデルの中に記述する。このようなモデルのことを、環境モデルと呼んでいる。そして、環境モデルから、特定の範囲で環境を生成するツールを実装した[6、9、8]。環境モデルは OSEK/VDX の仕様と RELOS の仕様に基づいて、タスク構成と呼び出し列、それらに対して、期待する性質を記述する。そして、提案した環境生成ツール EnvGen により、Promela で表現された環境を自動生成する。この環境と設計モデルを組み合わせて Spin で検証を行う。

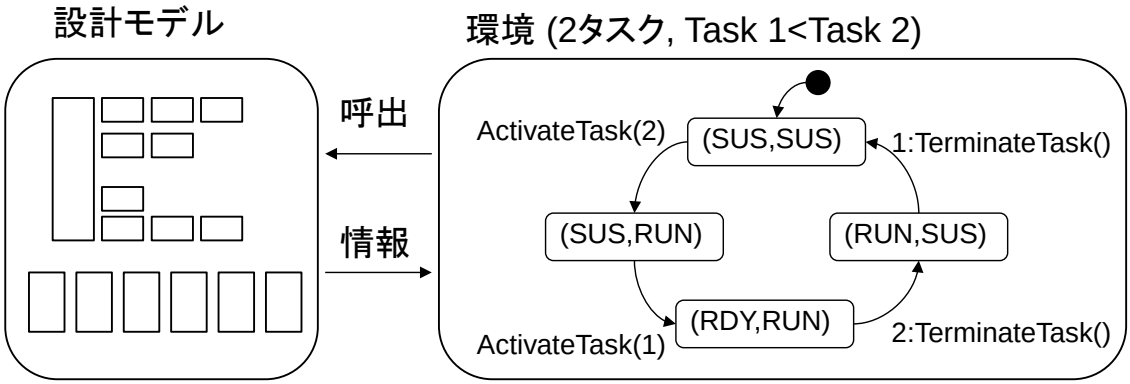


**図 15-B-13-3　環境（タスクや ISR などを表現する外部の記述）**

## 3.3. 環境モデリングと環境の自動生成

前述したように､設計検証では､環境モデルに検証する振る舞いと期待する性質を記述し、設計モデルと組み合わせて Spin により検証を行う。

環境の例を図 15-B-13-3 に示す。これは、2 つのタスク Task1 と Task2 があり、Task2 の優先度が Task1 の優先度より高い場合の環境である。環境は状態遷移モデルにより表現されており、状態には期待する性質、遷移には呼び出す RTOS のシステムサービスが書かれている。初期状態では、2 つのタスクは両方とも SUSPENDED 状態にあることが期待されている。そして、Task2 をシステムサービス ActivateTask により起動すると、Task2 の状態だけ RUN 状態になることを期待している。その後、Task1 をシステムサービス ActivateTask により起動すると、Task1 が READY 状態になり、Task2 は RUN 状態であり続けることを期待している。これは、Task2 の優先度が Task1 の優先度より高いからである。そして、Task2 がシステムサービス TerminateTask により、その実行が終了し、Task1 に実行権が移った後、同様にその実行が終了すると、初期状態に戻る。このように、環境では、システムサービスの実行列の集合と、それにより期待するタスクの状態遷移が書かれている。

図 15-B-13-3 は、2 つのタスク Task1 と Task2 があり、Task2 の優先度が Task1 の優先度より高いという特定の場合の環境である。タスクの個数と優先度の割り当て方、資源の優先度と使用関係のバリエーションを考慮すると、数多くの場合が存在する。その個々の場合の環境を手作業で作成するのは、とてもコストがかかる。そこで、環境のバリエーションをモデル化した、環境モデルを提案した[6]。環境モデルは拡張したクラス図とステートチャート図から構成される。

図 15-B-13-4 は環境モデルの例である。図の左側のクラス図では、環境の構成のバリエーションをモデル化している。この環境モデルでは、タスクと資源の個数と使用関係を記述している。クラス Task がタスクをクラス Resource が資源の集合を表現している。クラス Task は属性 pr と states を持ち、それぞれ、タスクの優先度とタスクの状態を表現している。クラス Resource は属性 pr と states を持ち、それぞれ、シーリング優先度と資源の状態を表現している。タスクと資源は複数存在することができ、それらの間の関係は多重度により表現されている。多重度の上限は変数 M、N、P、Q で表現されており、任意であることを意味している。環境の構成には様々な制約がある。例えば、タスクが資源を使う場合には、その資源のシーリング優先度はタスクの優先度より高くなければならない、などである。このような制約は OCL により記述されている。図 15-B-13-4 の右側のステートチャート図は、タスクによるシステムサービス呼び出しと期待する性質が記述されている。状態遷移に付加されている ActivateTask や TerminateTask はシステムサービスの呼び出しである。それぞれの状態には、タスクの期待する状態が記述されている。このステートチャート図では、クラス図のクラス Task により実体化されるインスタンスの動作がまとめて記述されており、そのために、いくらかの拡張を行っている。例えば、|Rdy->Run という記述が状態 Run から状態 Sus への状態遷移に付加されている。これは、同期遷移と呼んでおり、他のタスクの動作への副作用を引き起こす。その意味は、あるタスクがシステムサービス TerminateTask を呼び出して Run から Sus に遷移する際、残りのタスクで Rdy 状態のものがあれば、そのタスクが Run 状態になることである。

以上の環境モデルから図 15-B-13-3 にあるような環境を自動生成するツール EnvGen を提案した[6]。このツールでは、まず、環境モデルのクラス図から、オブジェクトを実体化し、オブジェクトグラフを作成する。オブジェクトグラフは唯一に決まるものではなく、多重度や OCL で記述した制約を満たす範囲で複数存在する。EnvGen は可能なオブジェクトグラフをすべて生成する。図 15-B-13-3 のクラス図では、多重度の上限が変数 M、N などになっているが、生成の際、それらの変数に特定の値を与える。つまり、上限を与えて、その範囲でオブジェクトグラフを生成するのである。また、生成の際、OCL で記述した制約を満たすオブジェクトグラフを発見する必要がある。そこで、制約を充足するようなオブジェクトグラフを効率的に列挙するため、ツールの内部で、SMT ソルバ Yices[20]を用いている。次に、列挙したそれぞれのオブジェクトグラフに対して、環境を構成する。まず、環境モデルのステートチャート図に基づいて、それぞれのオブジェクトの状態モデルを実体化する。そして、それらの状態モデルを合成して、環境の振る舞いを獲得し、Promela 記述へと翻訳する。ここで、検査する性質は表明として、Promela 記述に挿入されている。ツール EnvGen では、このようにして、与えられた範囲で、可能な環境を漏れなく列挙し、対応する Promela 記述を自動生成する。生成された Promela 記述は、設計モデルと組み合わせて、Spin によりモデル検査が実施される。

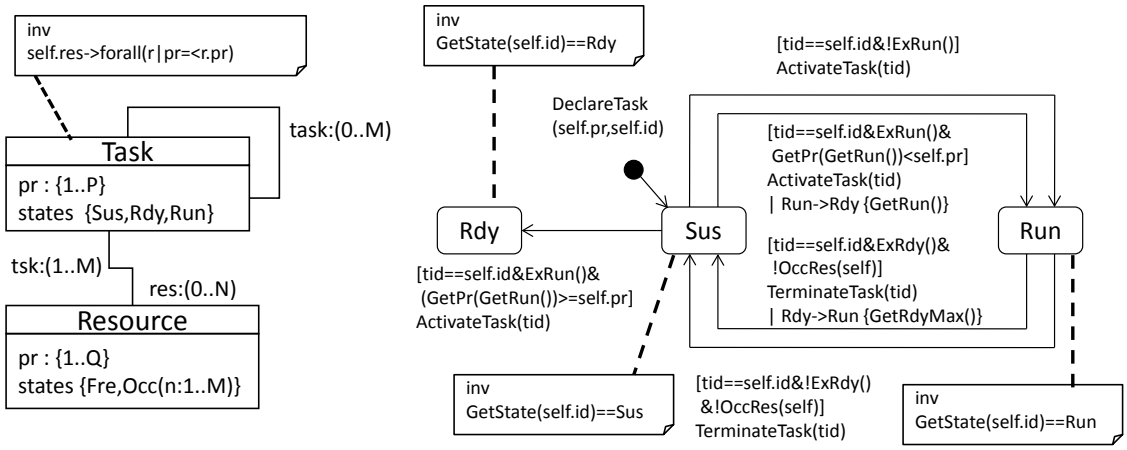


**図 15-B-13-4　環境モデル**

## 3.4. 分割検証

環境モデルは、検証対象である設計モデルへのシステムサービスの呼び出し列の集合を状態モデルとして表現し、それにより期待する性質を併記したものとなっている。ある意味、任意長のテストケースを状態モデルにより表現しているのである。期待する性質は、RELOS のスケジューラの動作を確認するものであり、期待するタスクの状態変化を記述している。ここで、任意のタスク、資源などの構成を考慮した環境モデルを作成すると、設計モデルと同様な構造になることが判明した。つまり、タスクの状態変化を正確に表現するためには、

タスクの優先度や起動順番を記憶しておく必要があり、そのためには、設計モデルで使われているタスクキューと同じデータ構造と操作が必要なのである。よって、任意の構成に関して設計モデルを検査しようとすると、検査対象と同様な環境モデルになってしまう。環境モデルも設計モデルと同じ複雑さを持っているので、環境モデルの信頼性は設計モデルと同等であり、検査の効果が上がらない。望ましくは、シンプルな環境モデルを作成して、設計モデルと比べて相対的に信頼性を高くすべきである。

ここで、2 つのアプローチ、環境の過大近似と過小近似が考えられる。前者では、実際に期待する挙動より広い範囲の環境を作成する。例えば、優先度が同じ複数のタスクが存在し、それらがすべて READY 状態である時、それらの中のどれか 1 つが RUN 状態である、という性質の記述を行う。このようにすれば、キューのような設計モデルに出現するデータ構造が必要なくなり、相対的にシンプルな環境モデルを作成することができる。しかしながら、この方法では、環境モデルにおいてタスクがどの状態にあるか正確に把握することができないため、意味の無い表明になってしまいがちである。また、システムサービスの呼び出しに多くの非決定性を含むため、偽反例も多くなってしまう。そこで、我々は、2 つ目のアプローチである環境の過小近似を行うことにした。この手法では、実際に期待する挙動より狭い範囲の環境を作成する。具体的には、場合分けを行い、それぞれの場合において環境モデルを作成する。例えば、複数のタスクが存在し、それらの優先度がすべて異なる場合と、すべて同じ場合という場合分けを行う。1 つ目の場合では、タスクキューは必要なく、優先度の比較だけで、期待するタスクの状態を決定できる。2 つ目の場合は、1 つのバッファを使うだけで、期待するタスクの状態を決定できる。このようにして、場合分けを行い、相対的にシンプルな環境モデルを作成した。

### 3.5. 検証結果

以上でも述べたが、我々は、場合分けを行い、環境モデルを作成した。表 15-B-13-1 に、その場合分けの結果を示す。過小近似のアプローチでは、場合分けの網羅性が問題となるため、慎重に場合分けを行わなければならない。そこで、まず、検証すべき機能を洗い出した。そして、すべての機能を検証する環境モデルを作成すると複雑になりすぎるため、これらの機能を分類し、分類毎に環境モデルを作成した。

まず、タスクの基本的なスケジューリングの検証を行う。これは、Acti-vateTask、ChainTask、TerminateTask といったシステムサービスの確認を行うのである。表 15-B-13-1 では、1~4 が、これらに相当する。次に、優先度のバリエーションにより場合分けを行う。環境モデルを複雑にする大きな要因の 1 つに優先度のバリエーションがある。任意の優先度の割り当てを考慮すると、検証対象と同様なキューが必要となる。そこで、異なる優先度を割り当てる場合と、同じ優先度を割り当てる場合で分けることにした。前者の場合は、環境モデルにタスクキューは必要でない。後者の場合は、キューは必要であるが、優先度毎にキューは必要ではない。表 15-B-13-1 は、奇数が前者で、偶数が後者である。次に、タスクの

多重起動、資源、イベント、ISR に関する機能の検証を行う。これらの機能は、タスクの基本的なスケジューリングと組み合わせて検証を行う。タスクの起動や終了といったスケジューリングの合間に実行されるものだからである。表 15-B-13-1 では、5、6 が多重起動、7、8 が資源、9、10 がイベント、11、12 が ISR に関する機能を検証するための環境モデルである。

表 15-B-13-1　場合分け

| No. | 名前 | 検証の目的 | 制約 |
|---|---|---|---|
| 1 | TaskDiff | タスクの検証 | 異なる優先度 |
| 2 | TaskEq | タスクの検証 | 同一優先度 |
| 3 | CtDiff | ChainTask の検証 | 異なる優先度 |
| 4 | CtEq | ChainTask の検証 | 同一優先度 |
| 5 | MultDiff | 多重起動の検証 | 異なる優先度 |
| 6 | MultEq | 多重起動の検証 | 同一優先度 |
| 7 | ResDiff | 資源の検証 | 異なる優先度 |
| 8 | ResEq | 資源の検証 | 同一優先度 |
| 9 | EvDiff | イベントの検証 | 異なる優先度 |
| 10 | EvEq | イベントの検証 | 同一優先度 |
| 11 | IsrDiff | ISR の検証 | 異なる優先度 |
| 12 | IsrEq | ISR の検証 | 同一優先度 |

作成した個々の環境モデルから環境を生成するためには、タスクや資源の個数など、生成範囲を決めなければならない。そこで、検証する機能がどのように設計モデルにおいて実現されているか分析を行い、範囲を決定した。取り扱えるタスクや資源の上限のチェックを除けば、基本的な振る舞いに関しては、比較的少数のタスクや資源で検証が行えることがわかった。例えば、タスク 1 つでは、タスクキューへの登録、削除、空の場合の動作が確認でき、タスク 2 つでは基本的なタスクの preemption の確認ができる、などである。このような分析を行った結果、個々の環境モデルにより異なるが、タスク数 1~3、資源数 1~3、ISR 数 1~2 の範囲内で環境の生成を行った。そして、生成された環境と設計モデルを組み合わせて Spin により検証を行った。検証結果を表 15-B-13-2 に示す。生成した環境の行数、状態、遷移は、環境の平均である。環境は合計で 786 個、生成時間は、合計で、81.4 秒である。よって、環境あたりの生成時間は、約 0.1 秒である。EnvGen では、SMT ソルバを用いて数え上げを行っているので、効率的に生成できていることがわかる。モデル検査にかかった時間は、それぞれの場合で生成された、すべての環境を使ってモデル検査を行った時間の合計である。これらすべての環境、すなわち、786 個の環境を検査するために、8819.3 秒かかっており、平均すると、1 個あたり約 11 秒である。そのうちのほとんどは、pan.c をコンパイルする時間であった。本手法では、すべての環境について検査が完了しており、状態爆発問題が回避で

きていることがわかる。無条件の非決定的なシステムサービスの呼び出しでは、状態爆発が発生し、モデル検査が完了しないことは確認している。それと比べて、本手法では、上記のように比較的短時間でモデル検査が完了し、検査結果が得られているので、探索状態を十分に現実的な範囲に抑制できていると言える。

このような場合分けによる検証は、すべてのシステムサービスを非決定的に呼び出す方式と比べて網羅性は劣る。しかしながら、前述したように、無条件の非決定的なシステムサービスの呼び出しは、正確に検証性質を記述することを困難にしてしまう。また、そのような呼び出しでは、状態爆発問題も発生する。いずれにしても、なんらかの限定が必要なのである。本手法では、検証の目的に応じて場合分けをすることにより、状態爆発問題を回避しつつ、特定の機能を重点的に、限られた範囲で網羅的に検証を行うことができる。

設計検証は以上のような環境モデリングによるモデル検査を用いた検査とレビューにより行った。JAIST 側が作成しており、事前の検証では数多くの誤りを指摘できた。ここでは、設計モデル構築における試行錯誤段階の検査であるため、正確な数は計測していない。モデル検査は自動的に結果が返ってくるので、環境モデルなど準備が整っていれば、手軽に検査を実施できる。よって、設計モデル構築の試行錯誤の段階からモデル検査を適用して、早期に誤りを発見修正することができた。次に、RELOS の実装と整合を取るために、デンソー、ルネサス側でレビューを行い、それを踏まえて、JAIST 側が検査を行った。この段階で、さらに、設計モデルの誤りを 5 箇所指摘できた。仕様の勘違い、スケジュールのタイミングの誤りや、エラー処理に関する誤りなどが主な原因である。

**表 15-B-13-2　モデル検査による検査結果**

| No. | 環境 | 生成した環境 | | | | | モデル検査 |
|---|---|---|---|---|---|---|---|
| | モデル | 数 | 時間(s) | 行数 | 状態 | 遷移 | 時間(s) |
| 1 | TaskDiff | 26 | 0.5 | 153 | 4 | 9 | 113.8 |
| 2 | TaskEq | 14 | 0.4 | 273 | 10 | 19 | 70.7 |
| 3 | CtDiff | 26 | 0.6 | 183 | 4 | 17 | 116.3 |
| 4 | CtEq | 14 | 0.5 | 343 | 10 | 37 | 76.4 |
| 5 | MultDiff | 26 | 0.7 | 199 | 8 | 19 | 119.1 |
| 6 | MultEq | 14 | 1.6 | 597 | 50 | 99 | 106.3 |
| 7 | ResDiff | 248 | 38.3 | 878 | 44 | 110 | 2904.0 |
| 8 | ResEq | 98 | 15.4 | 1079 | 53 | 136 | 1702.0 |
| 9 | EvDiff | 26 | 1.2 | 336 | 15 | 47 | 143.0 |
| 10 | EvEq | 14 | 2.3 | 1271 | 78 | 251 | 253.6 |
| 11 | IsrDiff | 182 | 10.3 | 502 | 17 | 49 | 1249.7 |
| 12 | IsrEq | 98 | 9.6 | 903 | 40 | 117 | 1964.4 |

# 4. テスト

## 4.1. 設計モデルに基づいたテスト

検証した設計モデルに基づいて実装する際、設計検証で保証した性質は、実装後も成立していなければならない。よって、検証した結果をソフトウェア実装後も保証する仕組みが必要である。このような仕組みには 2 つのものが考えられる。1 つ目は、設計モデルからソースコードを生成する手法である。この手法では、検証した設計モデルに基づいて、保証した性質を崩さないように詳細化したり、複数の設計モデルを合成し、実装と同型な実装モデルを作成する。そして、作成した実装モデルからソースコードを自動生成するのである。2 つ目は、設計モデルに基づいて、人手で最適化やプラットフォームへの適合を行い、その後、設計モデルと整合しているかどうか検査を行う。我々は、この 2 つ目の手法を採用した。検証対象の RELOS はすでに実装済みであることが主な理由である。また、車載 OS には、高いパフォーマンスを実現することが要求されている。そのため、アセンブラで実装されていたり、対象チップの特性を考慮した工夫がされている。これらを考慮した自動生成は非常に困難であることが予想される。

RELOS が設計モデルと整合していることを確認するために、設計モデルからテストケースを自動生成し、RELOS をテストすることにした。我々は、設計モデルや環境モデルのレビューを行い、モデル検査により設計モデルを検証した。設計モデルの妥当性を確認するのに、大きな労力を割いているのである。これらの活動により、相対的に設計モデルの品質は高いはずである。そこで、設計モデルをテストオラクルと見なし、テストケースを抽出するのである。

これまでに様々なオートマトンに基づいた整合テストが提案されているが[11]、完全な整合性を保証するためには、多くの前提が必要なことがわかっている。これは、実装を完全にブラックボックスとして取り扱い、入出力のみ観測できるという前提によるものが大きい。そこで、我々は、ブラックボックスではなく、テストにおいて、積極的に、実装の状態を観測することにした。テストでは、設計モデルから抽出する状態の期待値と実装の状態を比較している。そして、設計モデルには、実装における振る舞いを決定する状態を、すべて出現させることにした。ここでの状態は、例えば、レディキューやテーブル、フラグなどである。一方、OS は様々な使われ方をするため、そのような状態を網羅することは困難である。そこで、我々は、モデル検査ツールを用いて、状態を網羅するテストケースを生成することにした。このように設計モデルを作成してテストすることにより、以下の 3 つの前提の下で、設計と実装が一致することが言える。1)振る舞いを決定する状態がすべて設計モデルに出現している。2)実装の振る舞いも、それらの状態だけから決まっている。3)実装の振る舞いは決定的である。これらの前提は、現実からかけ離れているものでは無いであろう。前述したように、設計モデルは、それ自身では動作しない。よって、テストケース生成のためにも、環境が必要である。この環境はテストケースを抽出するためのものでありテストモデルと呼

ぶことにする。テストモデルには、テストしたい状況や範囲を記述する。また、テストケースが多く生成されると、期待値を決めるのが困難である。本手法では、設計モデルがテストオラクルであり、正しく振る舞いを表現しているという前提である。そこで、設計モデルの状態を参照して、期待値を求めることができる。

我々は、Spin を用いてテストケースを自動生成するツール TCG を実装した。TCG は、設計モデルとテストモデルを Spin により状態を網羅的に探索し、その際出力されるログを解析してテストケースを生成する。テストケースはシステムサービスの呼び出し列と期待値から構成される。また、このようなテストケースに基づいて RELOS のテストを行うテストプログラムを自動生成するツール TPG も実装した。TCG と TPG を用いることにより、設計モデルとテストモデルを入力として、自動的に RELOS のテストを行うことが可能となる。実際のテストは、シミュレータを用いて行った。

## 4.2. テストケース生成

設計モデルは、前節で示した検証手法、および、レビューにより十分に確認を行った。そこで、この設計モデルは正しいという前提を置いて、テストオラクルと見なし、テストケースを自動生成することにした[7]。この手法では、Promela で記述された設計モデルに環境を与えて Spin により到達性解析を行い、その際出力されるログを解析してテストケースを生成する。Spin は、状態探索を行う際、深さ優先探索に関する探索(Down)やバックトラック(Up)といった情報を出力することができる。さらに、埋め込み C コードの機能を使うことにより、特定の状態に到達するときにログを出力することができる。これらの出力された情報を用いて、検査における探索木を復元し、それを走査することにより、テストケースを生成するのである。我々は、このような仕組みに基づいてテストケースを生成するツール TCG の実装を行った。

## 4.3. テストモデル

設計モデルはシステムサービスを呼び出さないと動作しないので、設計検証の時と同様、テストケース生成においても環境が必要である。この環境は設計検証の時のものとは異なる。設計検証における環境は、設計モデルの正しさ確認するためのものであり、テストケース生成においては、実装、すなわち、RELOS の正しさを確認するためのものである。よって、テストする振る舞いは、実装の観点から決められる。そこで、テストケース生成のためのモデルである、テストモデルを作成した。テストモデルでは、タスクや資源などの構成、システムサービスの呼び出し方、そして、その構成と呼び出し結果の期待値を記述する。

テストモデルを作成するために、まず、テストすべき機能を洗い出した。そして、実装のレビューを行い、それらの機能をカバーできる、タスクや資源の構成を設定した。例えば、基本的なタスクスケジューリングのテストでは、タスクの数を 3 に設定した。実装におけるタスクキューの実現を考慮すると、タスクの優先度の違いによる実行順番の確認では 2 個のタスクがあれば十分である。しかしながら、それに影響しないはずのタスクが入っても、実

行順番は変わらないことを確認することも基本的なタスクスケジューリングの機能であり、そのためには、3 個のタスクが必要である。このような機能の洗い出しと、その機能をカバーする構成について検討を行った。

テストモデルにおいて、それぞれのシステムサービスの機能をテストする際に必要となる事前条件を記述し、それらを非決定的に呼び出すことにした。このように記述すると、Spin により到達性解析を行うことにより、可能な呼び出し列をすべて探索し、網羅的なテストケースを生成することができる。事前条件の記述では、設計モデルの内部状態を参照できる。例えば、システムサービス TerminateTask でタスクが正常終了するかテストするためには、実行中のタスクにおいて、そのシステムサービスを呼び出さなければならない。しかしながら、実行中のタスクを計算するのは大変である。そこで、設計モデルにおいて実行中のタスクの情報が格納されている変数があるので、それを参照して、TerminateTask の呼び出しの事前条件を記述する。このように、設計モデルの情報を参照することにより、事前条件を非常に容易に記述できた。なお、参照して良い理由は、設計モデルがテストオラクル、すなわち、正しいという前提を置いているからである。期待値についても、同様に、設計モデルの情報を参照して、出力することにした。例えば、実行中のタスクを期待値として求めるためには、設計モデルの中の、それが格納されている変数の値を出力するだけでよい。通常、期待値の定義は大変な作業であるが、本手法では、設計モデルをテストオラクルとみなしているので、それに基づいて容易に定義することができる。

## 4.4. テスト結果

テストケースの生成結果を図 15-B-13-5 に示す。生成に用いた構成は、タスク数:3、資源数:2、イベントマスク数:1、ISR 数:1 である。これらの値は、前述した検討の結果、導いたものである。図 15-B-13-5 では、3 つのタスクをタスク A～C、2 つの資源を、リソース A とリソース B として表現している。また、それらへの優先度の割り当てが 5 行に渡って書かれている。例えば、図 15-B-13-5 の左上の部分は、タスク A～C の優先度が、それぞれ、1、2、3 であることを示している。さらに、リソース A とリソース B には、それぞれ、(1、2)、(1、3)、(2、3)などの優先度が割り当てられている。ここで、優先度のバリエーションに関しては、対称性を考慮して、削減してある。6 行目には、それぞれの優先度の割り当てにおいて、生成されたテストケース数が書かれている。合計で、742、748 個のテストケースが生成された。前述したとおり、テストケースの生成は、Spin による到達性解析と、それにより生成されたログを解析してテストケースを生成する TCG の実行に分かれる。前者にかかった時間は、図 15-B-13-5 の pan 実行時間の行に、後者にかかった時間は、TCG の実行時間の行に示している。単位は秒である。なお、生成に使用した計算機の仕様は、Intel(R)Core2DuoCPU3.00GHz、メモリ 1Gbyte である。

生成されたテストケースは、システムサービスの呼び出し列と期待値により構成される。このテストケースに従って、実際にシステムサービスを呼び出し、その結果と期待値を比較

するテストプログラムが必要である。本研究では、そのようなテストプログラムを自動生成するツール TPG を実装した。TPG により生成されたテストプログラムは RELOS と組み合わせてシミュレータ上で実行することができ、デバッグ用のインターフェースを使って期待値を比較することができるようになっている。これにより、テストケース生成からテストの実施まで、完全に自動化を行うことができた。TPG によるテストプログラム生成時間は、図 15-B-13-5 の TCG 実行時間の行に示している。なお、使用した計算機は、上記したものと同じである。

生成したそれぞれのテストプログラムをコンパイルしてシミュレータ上で実行し、すべてのテストを実施した。それぞれのテストプログラムによるテストは、独立に実施できるため、原理的には、並列にテストを行うことができる。しかしながら、シミュレータを動作させるためにはライセンスが必要であり、本研究で使えるライセンスは限られている。そのため、基本的には 1 台の計算機でテストを実施し、複数ライセンスが使える夜中などは、複数台でテストを実施した。その結果、テストをすべて完了するのに、約 3 ヶ月かかった。例えば、テストケース 26,489 個を実行するのに 169.75 時間(内、コンパイル 42.5 時間、実行 127.25 時間)、テストケース 44,723 個を実行するのに、265 時間(内、コンパイル 80.25 時間、実行 184.75 時間)かかった。すべてのテストの実施にかかった時間の詳細は計測できなかったが、これらのデータから推測すると、のべ約 4,535 時間、すなわち、約 189 日かかることになる。このことから、夜中に複数台で並行して実施することにより、テストの実施期間が短縮できていることがうかがえる。また、テストを実施した結果、すべてのテストケースにおいて、RELOS の問題は見つからなかった。

| | 優先度 | | | | | | 優先度 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| タスクA | 1 | | | | | | 1 | | | | | |
| タスクB | 2 | | | | | | 1 | | | | | |
| タスクC | 3 | | | | | | 1 | | | | | |
| リソースA | 1 | 1 | 2 | 3 | 2 | 1 | 1 | 1 | 2 | 3 | 2 | 1 |
| リソースB | 2 | 3 | 3 | 3 | 2 | 1 | 2 | 3 | 3 | 3 | 2 | 1 |
| テストケース数 | 12483 | 15077 | 26373 | 37127 | 25035 | 8495 | 26489 | 26489 | 66361 | 66361 | 66361 | 13301 |
| pan実行時間 | 12.9 | 16.8 | 26.0 | 38.7 | 26.7 | 10.7 | 27.6 | 30.6 | 66.3 | 66.9 | 68.3 | 14.6 |
| TCG実行時間 | 19.0 | 24.9 | 67.9 | 73.1 | 58.5 | 12.7 | 81.8 | 93.7 | 289.2 | 287.2 | 282.6 | 27.2 |
| TPG実行時間 | 176.8 | 174.4 | 508.5 | 522.8 | 433.9 | 95.0 | 548.4 | 626.9 | 2267.4 | 2694.1 | 2692.2 | 232.5 |

| | 優先度 | | | | | | 優先度 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| タスクA | 1 | | | | | | 1 | | | | | |
| タスクB | 1 | | | | | | 2 | | | | | |
| タスクC | 2 | | | | | | 2 | | | | | |
| リソースA | 1 | 1 | 2 | 3 | 2 | 1 | 1 | 1 | 2 | 3 | 2 | 1 |
| リソースB | 2 | 3 | 3 | 3 | 2 | 1 | 2 | 3 | 3 | 3 | 2 | 1 |
| テストケース数 | 17151 | 22427 | 44723 | 60707 | 39457 | 10331 | 13011 | 20707 | 33117 | 56281 | 25459 | 9425 |
| pan実行時間 | 19.0 | 22.7 | 45.0 | 60.5 | 40.2 | 11.5 | 13.7 | 21.9 | 33.4 | 55.6 | 25.5 | 9.9 |
| TCG実行時間 | 35.6 | 44.7 | 117.6 | 179.7 | 99.4 | 16.8 | 21.8 | 38.8 | 78.3 | 161.8 | 55.1 | 14.1 |
| TPG実行時間 | 290.7 | 320.2 | 799.8 | 1353.5 | 694.4 | 138.9 | 175.2 | 317.9 | 546.8 | 1486.5 | 442.8 | 125.3 |

**図 15-B-13-5　テストケース生成結果**

# 5. 考察と知見

## 5.1. 検証結果とテストの意義

実際に設計モデルのレビューと検証にかかった時間は 6 ヶ月程度である。設計モデルの検証の実施は JAIST 側が、設計モデルのレビューはデンソーとルネサスマイクロシステム側が行った。そして、1 ヶ月に 1 度の打ち合わせの際、レビューの結果の報告を行い、それを踏まえて、JAIST 側が環境モデルを作成し、モデル検査による検証と確認を行った。いずれの側も、検証やレビュー作業のみを行っているわけではなく、通常業務と掛け持ちなので、実質的な時間は、もっと少なくなるであろう。その詳細は計測していない。一方、RELOS は、同様の過去の OS の開発を含めると十数年開発が続けられている。また、実際の車載システムで使われており、これまでに、何度もバグの修正が行われており、十分に洗練されたものである。よって、テスト結果により、RELOS のバグが発見されなかったことは、驚くことではない。ここで興味深いのは、6 ヶ月かけて検証した設計モデルが、RELOS と同じくらいの品質を持っているということである。テストを実施する前、テストケースがパスしない場合があるとすれば、設計モデルの間違いであろうと予想していた。しかしながら、そのような場合は見つからなかった。

また、生成されたテストケースは、通常はしないような OS の使い方を含んでいる。テストモデルでは、システムサービスを可能な限り非決定的に呼び出しているためである。これにより、ソフトウェアの、いわば、加速テストを行うことができる。通常のアプリケーションではあまりやらないが、将来的には、そういった使い方がされるかもしれない部分をテストしているのである。さらに、OS の使用実績を積むことができるとも言える。生成したテストプログラムは、それぞれが、OS の使用例、すなわち、アプリケーションなのである。

## 5.2. 設計モデルの正しさの確信

本プロジェクトの目的は、効率的に誤りを検出することではなく、正しさを確信することである。そのために、golden モデルを作成する必要があった。ここで作成した設計モデルが、その golden モデルの役割を担う。設計検証における正しさの確信のポイントは、以下の 3 つであると考えている。

- 過小近似によるシンプルな環境モデルの作成。記述が複雑になると、その正しさの確信度は下がっていく。シンプルであればあるほど、記述したものに対する確信度は高いはずである。設計モデルでは、RELOS において、スケジューリングの判断に用いられるデータと条件は、すべて含まれるように記述したため、その複雑さは、プログラムと、ほぼ、同等である。そこで、シンプルな振る舞いと突合せることにより、正しさを確信することにした。シンプルな振る舞いは、より、確信が置けるのである。我々のアプローチでは、環境モデルが、その役割を担っており、シンプルに記述する必要があった。そのため、過小近似（場合わけ）により、設計モデルと比べて、相対的にシンプルにする工夫をした。シンプルな振る舞いは、レビューもしやすい。

- レビューによる場合わけと範囲の絞込み。モデル検査により検証を行うため、コンピュータで自動的に検査できるくらいの状態に抑える必要がある。RELOS を用いたシステムでは、膨大な数のタスクを扱うため、その範囲でモデル検査を実施することは不可能である。定理証明を用いることも考えられるが、膨大な労力がかかることが容易に想像できる。そこで、レビューにより場合わけと範囲の絞り込みを行った。実際、例えば、3 個のタスクで大丈夫であれば、それ以上タスク数が増えても同じ振る舞いになる、という感触があるものである。それ以上であれば、50 個のタスクでも、51 個のタスクでも振る舞いは同じであろう、ということである。そこで、同様の考え方で、厳密にレビューを行った。振る舞いのバリエーションを整理し、典型的な実行系列を抽出して、以上のような大丈夫であろうタスクの個数を割り出した。
- モデル検査による網羅的探索による確信。以上のようなレビューによる範囲の絞り込みを行い、その範囲において、網羅的に探索する。すなわち、レビューにより、この範囲で大丈夫であれば正しいであろうと目鼻をつけ、本当に分析が難しい振る舞いにのみに絞り込んだ後に、その範囲を網羅的に探索するのである。確信を獲るという目的では、エンジニアの考えや経験で納得しやすい形で絞り込み、残りは網羅的に探索することが良いであろう。網羅的な検査無しで、タスク数 1~3、資源数 1~3、ISR 数 1~2 における任意の振る舞いで正しさを確信するのは難しいが、絞り込みに関しては納得しやすく確信しやすいはずである。

## 5.3. golden モデルベースのテストによる正しさの確信

本プロジェクトでは、golden モデル（設計モデル）を作成するためにモデル検査を用いている。ここでの考え方は、golden モデルは正しい、RELOS は golden モデルと同じ振る舞いをする、よって、RELOS は正しい、と結論づけることである。1)正しさの基準として golden モデルを作成し、2)RELOS がそれと同じ振る舞いをすることを確認するのである。1 では、モデル検査を用いて、golden モデルになっていることを確信した。2 では、網羅的なテストにより、同じ振る舞いであることを確信した。網羅性(モデル検査)と対象そのものを実行する(テスト)、という 2 つの特性は、正しさを確信するには適していると考えている。前者は、漏れが無いということで安心し、後者は実物の実動作ということで安心するのである。

また、バグを発見するための手法と正しさを確信するための手法は異なるはずである。例えば、テストにおいては、前者に関しては、効率的にバグを発見したいので、なるべく効果的な少量のテストに絞り込む必要がある。一方、後者に関しては、テストの絞り込みを行うと、その妥当性を確認する必要があり、本当に絞り込んで大丈夫だったか不安が残る。無闇に絞り込むよりは、可能であれば、必要なテストを全部行うのが望ましい。我々のアプローチでは、与えられた OS オブジェクトの数の範囲で網羅的にテストケースを生成する。範囲の割り出しで人手によるレビューは入るが、その範囲では、必要なテストを全部行っている。範囲の絞り込みは、複雑な振る舞いの分析を伴わないため、確信を獲やすくなっている。

テスト結果の分析も重要である。期待値の一致だけでなく、カバレッジの計測、さらには、実行時の状況をモニタリングしたり視覚化することが望ましい。これらにより、予期したとおり、対象が十分に鍛えられているか確認するのである。

## 5.4. 形式的に記述および検証したモデルの再利用

本プロジェクトで作成したような形式的に記述および検証されたモデルを獲得するコストは高い。検証して、意図どおりに記述されていることを確認するだけでは、もったいない。積極的に再利用すべきである。本プロジェクトでは、設計モデルからテストケースを自動生成することにより、設計モデルを再利用している。この手法では、設計モデルを作成したり検証する作業は、製品のテストにつながっており、設計検証の重要性を認識しやすく、安心して、設計モデルの検証に重点を置いて時間をかけることができるというメリットもある。形式手法を導入する動機付けになるであろう。また、自動生成などの自動化により、形式手法の適用にはコストがかかるが、その他の部分でコストを削減することができ、全体として、コストを現実的な範囲、もしくは、削減することができるであろう。形式手法を適用したモデルの積極的な再利用は、形式手法の実践において、キーとなる考え方である。

## 5.5. 動機付けとエンジニアとのコミュニケーション

本プロジェクトでは、詳細設計に相当する振る舞いをモデル検査ツール Spin により検証することにした。Spin では、Promela と呼ばれる仕様記述言語で振る舞いを記述する。その文法は、エンジニアにとってわかりやすいものであり、意志疎通がしやすい。また、プロジェクト開始当初、実際に OS を開発しているエンジニアには、Spin/Promela を使いこなす十分な知識は無かった。一方、大学側は、Spin/Promela の知識があったが、対象 OS の実装に関する知識は無かった。そこで、大学側で、想像で OS の振る舞いを Promela で記述し、それを、エンジニアにレビューしてもらった。この作業を繰り返すことにより、最終的な設計モデルを獲得することができた。現在では、エンジニアが、OS の設計モデルを記述できるようになっており、それを大学側がレビューを行っている。

教科書的には、設計モデルを作成して、設計者の意図を確認することが設計検証の目的である。モデル検査や形式手法を用いて設計検証を行うことを考えると、それなりの時間がかかることが容易に想像できる。そうすると、設計検証の目的は意図を確認することである、という主張だけでは、エンジニアには受けいれられにくい。そこで、Promela で記述した設計モデルから、テストケース、および、テストプログラムを自動生成する仕組みを作成した。これにより、網羅的なテストが可能になっただけでなく、Promela でモデルを作成する動機づけになった。設計モデルを記述すると、実際の製品のテストにつながるので、設計モデルの記述と検証に安心してコストをかけることができるのである。

## 5.6. 形式手法適用の強弱

すべての工程に形式手法を導入して厳密に検証することが望ましいが、実際のシステム開

発では、コストの問題から現実的では無い場合が多い。本手法では、設計工程に焦点を当て、この工程で時間をかけて形式手法を適用することにより、信頼性の高い設計モデルを作成している。そして、それにより、設計モデルを golden モデルとみなす根拠としているのである。このように、集中すべき工程やアーティファクトを限定し、それを有効活用することにより、コストを削減し、形式手法を現実的に適用することが可能になる。設計工程に注目した理由は、OSEKOS のような RTOS は、手続き的な記述で、仕様やメカニズムを表現しやすく、検討しやすいからである。実際、従来から、キューなどのデータ構造を使って説明されていることがよくある。そのため、本手法のような手続き的な設計モデルと相性が良い。

### 5.7. レビューと形式手法の使い分け

本手法では、過小近似により設計モデルの検証を行っているため、すべての振る舞いの検査が尽くされていない。また、相対的にシンプルな環境モデルにより性質を記述しているが、それが誤っているかもしれない。このように、様々な不完全さがあるが、基本的には、完全な正しさの保証は実践的には不可能であり、なんらかの前提であったり、検証を打ちきる基準が存在する。本手法では、設計モデルのレビューにより、分割したそれぞれの場合において誤りが見つかれなければ大丈夫である、という確信を得たので、この検証作業で十分という判断を行った。また、環境モデルは相対的にシンプルなので、正しいという前提を置いた。そして、その前提や範囲では、テストやレビューではなく、モデル検査などの手法を用いて正しいということを証明したのである。テストにおいても、同様である。もちろん、それらの確信をさらに上げるための活動も考えられる。例えば、OS の形式仕様を作成して環境モデルがその形式仕様に適合していることの検証や、モデル検査ではなく、対話的証明による設計モデルの正しさの検証などである。それらの適用は、実践的には、コストと信頼性のトレードオフを勘案して決めなければならないであろう。

## 6. まとめ

本プロジェクトでは、RELOS の正しさを確信するために、モデル検査とテストを組み合わせて、検証を実施した。結果として、製品のバグは発見されなかったが、モデル検査と網羅的なテストにより、RELOS は正しいという確信を持つことができた。形式手法は、品質向上のための有望な手法として期待されながらも、普及が進んではいない。様々な原因が考えられるが、本プロジェクトのような実践が行われ、知見や経験を共有することにより、普及を促進することができるのではないかと考えている。本原稿が、そのための情報を提供できたのであれば、幸いである。

参考文献

[1] Technical Assessment of Toyota Electronic Throttle Control (ETC) Systems, NHTSA, 2011.

[2] ISO 26262 Road vehicles - functional safety, 2011.

[3] IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems, 1998.

[4] OSEK/VDX Operating System Specication 2.2.3., 2005.

[5] Specification of Operating System 4.0.0, AUTOSAR, 2009.

[6] Kenro Yatake and Toshiaki Aoki: Automatic Generation of Model Checking Scripts based on Environment Modeling, The 17th Inter- national SPIN Workshop on Model Checking of Software, pp.58-75, 2010.

[7] J.Chen and T.Aoki: Conformance Testing for OSEK/VDX Operating System Using Model Checking, APSEC, pp.274-281, 2011.

[8] 矢竹健朗, 青木利晃: UML に基づく RTOS 設計検証のための環境自動生 成法, 日本ソフトウェア科学会 学会誌 コンピュータソフトウェア, Vo.29, No.3, pp.121-142, 2012.

[9] Kenro Yatake, Toshiaki Aoki: SMT-based Enumeration of Object Graphs from UML class diagrams, 5th International workshop UML and Formal Methods, ACM SIGSOFT Software Engineering Notes, 37(4), pp.1-8, 2012.

[10] G.J.Holzmann: The Spin Model Checker - Primer and Reference Man- ual. 2004.

[11] D.Lee and M.Yannakakis: Principles and Methods of Testing Finite State Machines - a Survey, Proceedings of the IEEE, vol. 84, no. 8, pp.1090–1123, 1996.

[12] O. Tkachuk, et.al:Automated environment generation for software model checking, ASE, 2003.

[13] M. Dwyer and C. Pasareanu: Filter-based model checking of partial systems, FSE, pp.189–202, 1998.

[14] John Penix, et.al: Verifying Time Partitioning in the DEOS Scheduling Kernel, Formal Methods in System Design, Vol.26, No.2, pp.103–135, 2005.

[15] G. Klein: Operating System Verification - An Overview, S¯adhan¯a , 34(1), pp.26–69, 2009.

[16] L.Zhu, et.al:Formalizing Application Programming Interfaces of the OSEK/VDX Operating System Specification, TASE, pp.27–34, 2011.

[17] Y.Huang, et.al:Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP, TASE, pp.142–149, 2011.

[18] E.Cohen, et.al: VCC: A Practical System for Verifying Concurrent C, TPHOLs, pp.23–42, 2011.

[19] PAT, Process Analysis Toolkit 2.9 User Manual. Software Engineering Lab, School of

Computing, National University of Singapore, 2007.
[20] Bruno Dutertre and Leonardo de Moura: The YICES SMT Solver, 2006.
[21] G.Klein, et.al: seL4: Formal verification of an OS kernel, ACM Sym- posium on Operating Systems Principles, pp.207–220, 2009.
[22] G.Klein, et.al: Comprehensive formal verification of an OS microkernel ACM Transactions on Computer Systems, Volume 32 Issue 1, pp.1–70, 2014.
[23] Tobias Nipkow: Interactive Proof: Introduction to Isabelle/HOL. Soft- ware Safety and Securitym pp.254-285, 2012.
[24] Anthony Fox and Magnus O. Myreen. A Trustworthy Monadic For- malization of the ARMv7 Instruction Set Architecture. In Interactive Theorem Proving (ITP), 2010.