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シームレスな検証

青木利晃[†] 佐藤信[†††]
谷充弘[††] 矢竹健朗[†]

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．最近では，車載システムに特化された機能安全の世界標準も策定されており，実社会では，トヨタ車の電子スロットル制御システムの検証が NASA により実施されたという事案も生じている．このような問題を背景に，我々は，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．我々が対象としている OS は，OSEK/VDX に準拠するものである．本論文では，モデル検査とテスト手法を組み合わせて，設計検証から実装のテストまでシームレスに検証を行う手法，および，実際の製品への適用について紹介する．

Seamlessly Model Checking and Testing Automotive Operating System

Toshiaki Aoki,[†] Makoto Satoh,[†††] Mitsuhiro Tani [††]
and Kenro Yatake[†]

The safety and reliability of automotive systems are becoming a big concern in our daily life. Actually, a functional safety standard which is specilized in automotive systems has been proposed by ISO. In addition, electorical throttle systems have been inspected by NASA due to the unintended acceleration problem of Toyota's cars. To follow such recent circumstance, we are studying about practical vericiation of automotive operating systems. The operating system which we focus on is the one conforming OSEK/VDX standard. In this paper, we show an approach to seamlessly connect design verification with implementation testing based on model checking, and its application to a real product.

1. はじめに

1.1 背　景

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，車載システムに特化されたものが策定されている．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 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[2)] に準拠するものである．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 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．

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 JAIST と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

† 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology
†† (株) デンソー
DENSO CORPORATION
††† ルネサスマイクロシステム株式会社
Renesas Micro Systems Co., Ltd.

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が，2009 年から加わった．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．その OS 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が目的である．

1.2 OSEK/VDX

OSEK/VDX OS(略して，OSEK OS) は，優先度ベースのスケジューリングを採用している．タスクの管理をするために，タスクを起動する ActivateTask, 終了する TerminateTask, タスクのチェインを実現する ChainTask といったサービスコールが準備されている．また，共有資源を管理するため，Resource と呼ばれる概念があり，それをサービスコール GetResource により取得したり，サービスコール ReleaseResource により開放することにより，排他制御を実現できる．また，この Resource には，優先度逆転問題を回避するため，Priority Ceiling Protocol(PCP) が採用されている．その他にも，イベントやアラームといった機能が定義されている．

OSEK OS のスケジューリングの例を図 1 に示す．縦方向は優先度を表現していて，上に行くほど優先度が高いことを意味している．横方向は時間の経過を表現しており，右に行くほど時間が経過していることを意味している．OSEK OS では，Full Preemptive と Non Preemptive のタスクが混在できる．図 1 では，TASK H(P) と TASK M(P) は Preemptive Task, TASK L(NP) は Non Preemptive タスクである．RES M は資源を表現しており，INT H と INT L は割り込みが発生した時に呼び出されるサービスルーチン (ISR) である．実行順は以下のとおりである．

1. 初期状態では，TASK L(NP) が起動されている．2. このタスクが RES M を取得すると，PCP により優先度が一時的に引き上げられる．3. INT H の割り込みが生じると，それに対応する ISR が起動され，その中で TASK H(P) が起動される．4. その ISR の処理が終了すると，TASK L(NP) に制御が戻る．5. TASK L(NP) が実行中に INT L が発生しても，優先度が低いので，その ISR は実行されない．6. RES M を開放すると，オリジナルの優先度に戻るので，INT L の ISR が実行される．7. ISR の実行が終わると TASK L(NP) に制御が戻る．ここで，このタスクは Non Preemptive なので，TASK H(P) が起動されているが，その実行は行わない．8. TASK L(NP) が終了すると，TASK M(P) が実行される．9. このタスクが RES M を取得すると，PCP により優先度が一時的に引き上げられる．10. このタスクが TASK H(P) を起動する．11. TASK M(P) が RES M を開放すると，もとの優先度に戻る．ここで，このタスクは Preemptive であり，TASK H(P) がすでに起動されているため，TASK H(P) に制御が移る．12. TASK H(P) が終了すると，TASK M(P) が実行される．

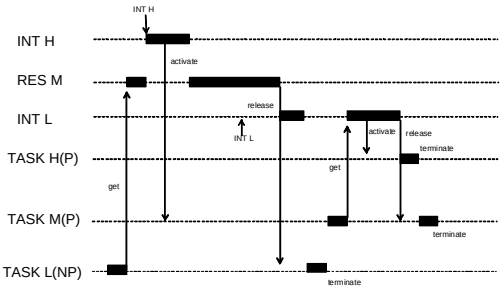

図 1 OSEK OS におけるタスクスケジューリング

OSEK OS では，このようなスケジューリングを行う機能がメインである．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(Interrupt Service Routine) 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，本共同研究では，OS のスケジューラにより，このようなタスクの管理が正しく行えていることを検証する．また，対象とした OS は，ルネサスエレクトロニクス社の RX-OSEK850 であり，OSEK/VDX に準拠したものである．

2. アプローチ

我々のアプローチの概要を図 2 に示す．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．設計検証に関する範囲とフローは実線で，テストに関する範囲とフローは点線で表現している．

2.1 設計モデル

スケジューリングの仕組みを分析するため，RX-OSEK850 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[8] 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 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 2300 行である．

2.2 モデル検査による設計検証と環境モデリング

OSEK OS 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，RX-OSEK850 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 *universal environment* と呼ばれている[12]．universal environment 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[11]．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．

OSEK OS 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

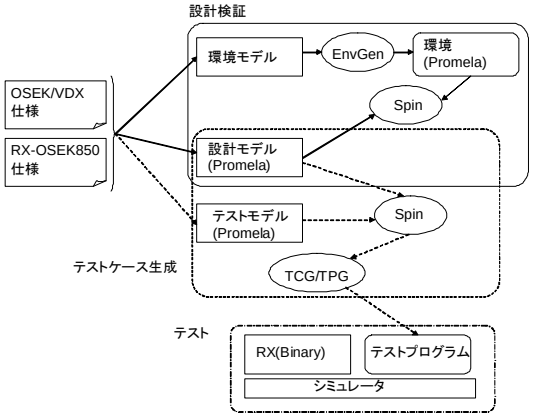

図 2 全体像

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，universal environment 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(Object Constraint Language) 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[4),6),7)]．環境モデルは OSEK/VDX の仕様と RX-OSEK850 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．

2.3 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 RX-OSEK850 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．

RX-OSEK850 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，RX 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[9]，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．これらの前提を満たすのは困難であり，現実的ではない．我々は，完全な整合性を目指すのではなく，設計で考慮した状態を網羅することを目指すことにした．そして，もしテストしたい状態があるのであれば，それが含まれるように設計モデルを作成したのである．本手法では，RX-OSEK850 における状態 (キューやテーブルなどに保存されている値) が異なるのであれば，設計モデルでも異なるように作成した．これにより，設計モデルにおける状態は，実装において注目している状態をカバーしていることになる．それでも，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．

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 RX-OSEK850 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 RX-OSEK850 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．

3. 関 連 研 究

形式手法において，検証という言葉は，定理証明を用いた演繹的手法に用いられる場合が多く，その意味で OS の検証を行った代表的な事例は文献[13] にまとめられている．

モデル検査に基づいたものとしては， The Honeywell Dynamic Enforcement Operating System(DEOS) は航空宇宙向けの OS であり，そのカーネルの時間分割機能の設計検証を行った研究がある[12]．この研究では，Spin を用いて設計モデルの作成と検証を行っているが，前述したとおり環境の作り方が我々の手法と異なる．また，この研究では，設計検証のみを行っているが，我々は，設計検証だけでなく，それに基づいたテストまで行っている．

OSEK/VDX を対象とした検証に関する研究として[14],[15] がある．[14] では，C ソースコードを検証するツール VCC[16] を用いて，システムサービスの基本的な性質を検証している．この研究では，OSEK/VDX のシステムサービスの形式化に焦点を当ててあり，検証に関しては，アドホックな手法で VCC を適用しており，単純な性質のみ取り扱っている．[15] では，OS のソースコードから CSP のモデルを作成して，モデル検査ツール PAT[17] で検査している．手作業でモデルを作成しており，モデルとソースコードの整合性については確認をしていない．

4. 設 計 検 証

4.1 環境モデル

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 3 に示す．これは，2 つのタスク Task 1 と Task 2 があり，Task 2 の優先度が Task 1 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

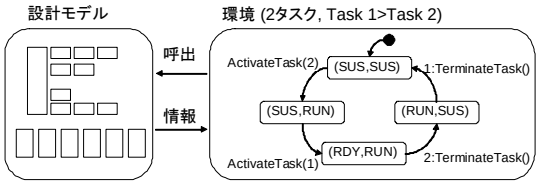

図 3 環境

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，Task 2 をシステムサービス ActivateTask により起動すると，Task 2 の状態だけ RUN 状態になることを期待している．その後，Task 1 をシステムサービス ActivateTask により起動すると，Task 1 が READY 状態になり，Task 2 は RUN 状態であり続けることを期待している．これは，Task 2 の優先度が Task 1 の優先度より高いからである．そして，Task 2 がシステムサービス TerminateTask により，その実行が終了し，Task 1 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 3 は，2 つのタスク Task 1 と Task 2 があり，Task 2 の優先度が Task 1 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[4]．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 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 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．

4.2 環境自動生成

以上の環境モデルから図 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[4]．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 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 ソルバー yices[18] 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．

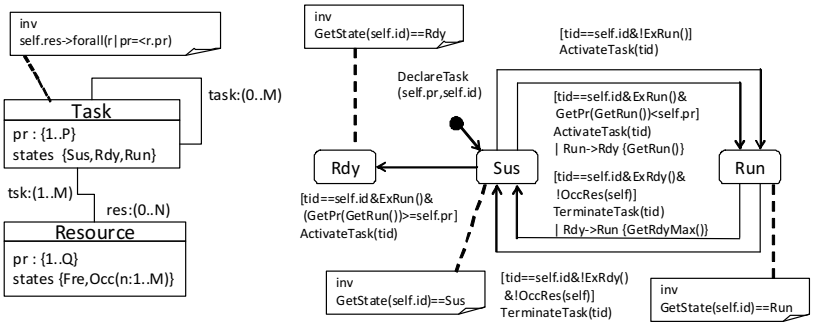

図 4 環境モデル

4.3 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，RX-OSEK850 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．

4.4 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．図 5 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， ActivateTask, ChainTask, TerminateTask といったシステムサービスの確認を行うのである．図 5 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．図 5 では，奇数が前者で，偶数が後者である．

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

図 5 場合分け

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

図 6 モデル検査による検査結果

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．図 5 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．

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．検証結果を図 6 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，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．

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，状態爆発問題を引き起こしやすい．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．次に，RX-OSEK850 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．

5. テ ス ト

5.1 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[5]．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．

5.2 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，RX-OSEK850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．

5.3 テスト結果

テストケースの生成結果を図 7 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 7 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 7 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 7 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)Core2Duo CPU 3.00GHz，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 RX-OSEK850 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 7 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，

	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 7 テストケース生成結果

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 4535 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，RX-OEK850 の問題は見つからなかった．

6. 考　　察

本研究の成果として，設計検証から実装のテストまでシームレスに実施できたことが挙げられる．ここでの考え方は，設計モデルの検証に重点を置き，十分に検証された設計モデルをテストオラクルとみなしてテストケースを自動生成することである．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い．よって，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た．

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，設計モデルをテストオラク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，OSEK OS 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．

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，RX-OSEK850 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，RX-OSEK850 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，RX-OSEK850 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．

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．

7. ま と め

本論文では，モデル検査とテスト手法を組み合わせて，設計検証から実装のテストまでシームレスに検証を行う手法，および，実際の製品への適用について紹介した．結果として，製品のバグは発見されなかったが，モデル検査や網羅的なテストにより，設計と実装のトレーサビリティを確保しながら，設計と実装の妥当性を確認したことは，RX-OSEK850 の高い品質の証拠になったと考えている．今後は，複雑なハードウェア割り込み処理を伴うものや，マルチコア上に実装されたものを対象に，同様の手法の適用を試みる予定である．

参 考 文 献

1) Technical Assessment of Toyota Electronic Throttle Control (ETC) Systems, NHTSA, 2011.
2) OSEK/VDX Operating System Specication 2.2.3., 2005.
3) Specification of Operating System 4.0.0, AUTOSAR, 2009.
4) Kenro Yatake and Toshiaki Aoki: Automatic Generation of Model Checking Scripts based on Environment Modeling, The 17th International SPIN Workshop on Model Checking of Software, pp.58-75, 2010.
5) J.Chen and T.Aoki: Conformance Testing for OSEK/VDX Operating System Using Model Checking, APSEC, pp.274-281, 2011.
6) 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.
7) 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.
8) G.J.Holzmann: The Spin Model Checker - Primer and Reference Manual. 2004.
9) 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.
10) O. Tkachuk, et.al:Automated environment generation for software model checking, ASE, 2003.
11) M. Dwyer and C. Pasareanu: Filter-based model checking of partial systems, FSE, pp.189–202, 1998.
12) 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.
13) G. Klein: Operating System Verification - An Overview, Sādhanā , 34(1), pp.26–69, 2009.
14) L.Zhu, et.al:Formalizing Application Programming Interfaces of the OSEK/VDX Operating System Specification, TASE, pp.27–34, 2011.
15) Y.Huang, et.al:Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP, TASE, pp.142–149, 2011.
16) E.Cohen, et.al: VCC: A Practical System for Verifying Concurrent C, TPHOLs, pp.23–42, 2011.
17) PAT, Process Analysis Toolkit 2.9 User Manual. Software Engineering Lab, School of Computing, National University of Singapore, 2007.
18) Bruno Dutertre and Leonardo de Moura: The YICES SMT Solver, 2006.