概要

本論文では、μITRON 仕様の RTOS を利用したマルチタスク組込みソフトウェアの設計・検証手法を提案し、実事例に対してその手法を適用することで提案手法の評価を行った。また、μITRON のメールボックスシステムコールをエミュレートするライブラリも作成した。

目次

図目次

第１章 はじめに

1.1 背景

近年、組込みソフトウェアの開発規模は拡大しており、品質の確保が難しくなって来ている。そのため、ソフトウェアの信頼性を確保するためにモデル検査を利用しようとする試みが活発になってきている。モデル検査は、状態遷移系のモデルの取り得る状態を自動的・網羅的に探索し、そのモデルの振舞いの正しさを検証する技術である。この網羅性が、現在のレビュー、テストベースの方法では保証しにくくなって来ている部分であり、モデル検査の持つ非常に大きな魅力である。

しかし、実用にあたっては問題点もある。１つ目は状態爆発の問題である。計算機の性能の向上によって従来に比べてかなり大きなモデルを一度にモデル検査する事が可能になっている事は確かであり、これがモデル検査を実際のソフトウェア開発に利用することの可能性を大きく広げたことも確かである。ただ、それでもソフトウェアのモデルの取り得る状態数は非常に大きいため、現在のところ、最終的な実装レベルの情報が全て詰め込まれたモデルを一度にモデル検査する事は不可能に等しい。2 つ目の問題として、検証ノウハウの不足が挙げられる。モデル検査の組込みソフトウェア開発への適用は新しい試みであるため、一般的な検証手法はまだ存在していない。そのため、事例検証を行い、その結果を元に検証手順を整理する必要がある。これが、将来的な一般的な検証手法の確立に繋がる。

また、近年の組込みソフトウェアの開発においては RTOS を利用する事も多くなっている。RTOS の上で動作するアプリケーションを検証する場合、その振る舞いをまったく意識せずにシステム全体の検証をすることは難しい。モデル検査においてもこれは同じである。しかも、モデル検査においては RTOS のスケジューリングポリシーを考慮したモデルを作成する事が状態数の抑制にも繋がる。実際の組込みソフトウェア開発を意識してという意味でも、モデル検査の際の状態数を考慮してという意味でも、RTOS を利用することを意識してモデル検査手法を体系化することは有効である。

他に、近年の盛んな研究により、ソフトウェアのモデル検査のためのツール・ライブラリなども開発されている。ソフトウェアの検証に有効活用して行くためには、これらを用いた事例検証も行わなければならない。

1.2 目的

本研究の目的は、大きく２つである。１つ目は、組込みソフトウェアのモデル検査手法を提案することである。２つ目は、提案手法を実事例へ適用することである。

本研究で目指す「組込みソフトウェアのモデル検査手法」は、μITRON 使用の RTOS を利用するマルチタスクのソフトウェアを対象とする検証手法である。提案する検証手法の特徴は、設計モデルから検証モデルを生成する際の抽象化の仕方にある。抽象化の際には、ディスパッチの発生するタイミングと割込み許可区間を意識する。これらを意識することで、検証する意味のある妥当な大きさの検証モデルを作成し易くなる。

また、本研究では事例検証の過程で、μITRON のシステムコールの１種であるメールボックス機能のエミュレーションライブラリを作成した。事例検証によって提案手法の評価を行うと共に、このメールボックスライブラリの検証から事例本体の検証までの流れの中で得られたモデル検査において注意すべき点などの知見も整理する。

1.3 検証対象事例

本研究で検証対象とする組込みソフトウェアの種類・モデル検査環境の概要について以下に示す。

対象アプリケーション種類：	カーオーディオ
使用言語：	C
使用リアルタイム OS：	μITRON 仕様 OS
タスク数：	７つ
タスク構成：	
	ユーザー入力イベントハンドルタスク
	CD 制御タスク
	チューナー制御タスク１
	チューナー制御タスク２
	電子ボリューム制御タスク
	ディスプレイ制御タスク１
	ディスプレイ制御タスク２
ソースコード規模：	全体　-> 約 35000 行
	検証対象範囲　-> 約 15000 行
モデル検査器：	SPIN
モデル検査支援ツール：	UMLchecker
モデル検査用ライブラリ：	RTOS エミュレーション PROMELA ライブラリ（μITRON4.0 仕様 OS 対応）
	RTOS メールボックスシステムコールライブラリ（上記ライブラリと連係動作）

1.4 結論

本研究では、まずμITRON仕様のOSを利用したマルチタスクの組込みソフトウェアを検証する際の検証手法を提案した。次に、事例検証の過程で必要であったメールボックスシステムコールのエミュレーションライブラリを作成し、作成したライブラリがμITRONの仕様に合っているかどうかの検証を行った。最後に、提案した手法をカーオーディオの事例に対して適用し、提案手法の評価を行った。提案した手法を用いる事で、設計モデルをモデル検査可能な大きさで、かつ意味のある検証モデルにする事ができた。

1.5 本論文の構成

本論文では、まず2章において本研究で利用している技術を、第3章では本研究で提案するマルチタスク組込みソフトウェアの設計検証手法について述べる。第4章では本研究で作成したメールボックスライブラリについて記す。第5章で、カーオーディオの事例を提案手法を用いて検証した例を示す。最後に、第6章でまとめとする。

第 2 章 関連技術

2.1 μITRON

μITRON はリアルタイム OS の仕様である。μITRON 仕様の OS は、現在日本の組込みソフトウェアで利用されている OS のうちで、主要なものの１つである。他に組込みソフトウェアで良く利用されている OS としては、LINUX 系、Windows 系のものがある。

2.1.1 タスク

「タスク」は、μITRON における並行動作の単位である。つまり、１つのタスク中のプログラムは順番に実行されるが、異なるタスク中のプログラムは並行して実行されることを意味している。ただし、ここでの並行実行とは、実時間で同時に２つのタスク中のプログラムが実行される事は意味しておらず、RTOS によって決められた順序に従って実行するタスクを切り替えながら、それぞれのタスクの中のプログラムを実行して行くことを意味している。

2.1.2 タスクの状態

μITRON のタスクの取り得る状態は、大きく分けて５つの状態（未登録状態、休止状態、待ち状態、実行可能状態、実行状態）です。

2.1.3 スケジューリング方式

「スケジューリング」とは、実行するタスクを切り替える順番を決定する事である。μITRON には、スケジューリングの際の判断基準が 2 段階存在する。最も優先される基準は「タスクの優先度」である。2 番目の判断基準は「実行可能状態になった順番」である。

μITRON のスケジューリングのアルゴリズムでは、はじめに「実行可能状態」のタスクの中で、「最も優先度の高いタスク」を探す。ここで、「最も優先度の高いタスク」が１つの場合は、このタスクが次に実行されるタスクだと決定される。複数ある場合（複数のタスクが同一の優先度を持っており、それらのタスクの優先度が、実行可能状態のタスクの中で最高である場合）には、それらのタスクの中で「実行可能状態になった順番が最も早いタスク」が次に実行されるタスクとなる。基本的には一度実行権を得たタスクは、それよりも優先度の高いタスクが実行可能状態になるか、自タスクが待機状態になるまでは、ずっと動き続ける。

実行するタスクの切り替え（ディスパッチ）が発生する可能性のあるタイミングは、システムコールが発行された時か、割込みハンドラの処理から復帰する時である。

2.2 SPIN

SPIN（Simple Promela INterpreter）は、PROMELA（PRotocol Meta LAnguage）で記述されたモデルをモデル検査するためのツールである（インタープリタとあるが、単純なインタープリタ部分のみを指した名称ではないため「ツール」と表現している)。このツールは、フリーで提供されているものである。現在では、ソフトウェアのモデル検査に利用できるモデル検査ツールの代表格として、その立場を確立している。

2.2.1 モデル検査とは

モデル検査とは、一言で言うと「有限状態のモデルを自動的に全網羅探索しながら、定義した性質を満たしているかどうかを調べる技術」である。この技術のメリットは、「自動的」、「全網羅」の２つである。

「自動的」とは、モデル検査においては、「検証するモデルの記述」と「検証性質の記述」が終れば、後は人の手を介さずに検証結果（記述した性質を満たしているかどうか）を得る事ができるという事である。現在のソフトウェアの品質保証のための主な手法は「テスト」である。「テスト」においては、検証したい性質が１つあった時にその性質を検証するためには、「複数のテストケース」を生成する必要がある。この「テストケース」を生成する必要が無いのがモデル検査である。ただし、モデル検査を行うにはその状態空間が閉じている必要がある。このため、「システムへの入力にはどのような種類があって、それらがどのような順番で入ってくるのか」をモデル化する必要がある。つまり、モデル検査の際には、「検証したいシステムのモデル」と「検証したいシステムに対して入力を与える外部のモデル」を記述する必要がある。本論文では、前者のモデルを「システムモデル」、後者を「外部モデル」と呼ぶことにする。

「全網羅」とは、記述した状態遷移モデルの全ての遷移系列を網羅する事を意味している。唐突であるが、ソフトウェアの設計をして行く上で処理の目的ごとに並行処理の単位に分割して行くことは良くあることである。しかし、このような設計を行った時に問題になるのが、「複数の状態遷移モデルを合成して得られるシステム全体を表す状態遷移モデルは直感的にはイメージしづらく、並行動作の数が多くなれば多くなるほど、全体の振る舞いを正しく理解するのが難しくなる」という点にある。この原因は、「実行する並行動作」の切り替えがどの場所で発生したら問題があり、どの場所では発生しても問題が無いのかと言う事を判断する難しさにある。これを解決するために、従来は経験豊富な技術者が設計段階で十分と思われるレビューを行い、最終的に長期にわたる実機テストを行う事で品質の向上に努めてきていた。しかし、近年のソフトウェアの巨大化にともない、「十分と思われる」、「長期の」といった点が問題化して来ている。つまり、「十分と思われるという判断だけでは限界」に達しているという事と、巨大化したソフトウェアを検査するには「不十分なテスト期間」しか取る事ができなくなって来ている点にあると考えられる。

そこで、モデル検査が有効になってくる。多くのモデル検査器では、並行動作（SPIN ではプロセスを単位とする）を記述することを許している。そして、許されている全てのタイミングで「実行する並行動作」の切り替えが起こる可能性を考慮して、全網羅探索してくれる。

つまり、モデル検査を行う意味は、「十分だと思われる」だった物を、「この条件でこの性質に関しては絶対に大丈夫」だと断言できる点にあると言える。

2.2.2 SPIN によるモデル検査の手順

SPIN によるソフトウェアの設計モデル検査の手順を説明する。細かい部分は差があるかもしれないが、一般的に検証の大きな流れは以下の様になる。

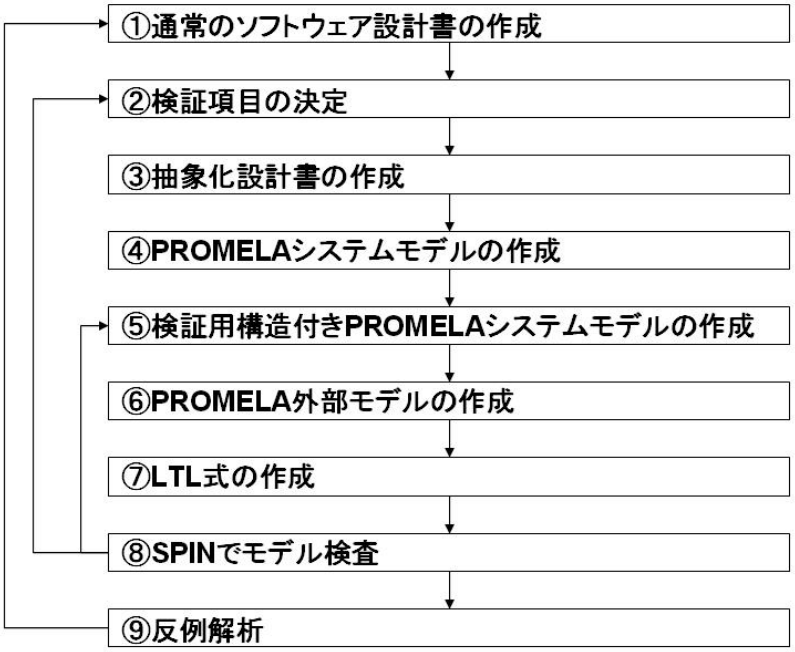


図 2.1　一般的なソフトウェアモデル検査の手順

それぞれのステップについて、捕捉して行く。

①：このステップで作成する設計書は、モデル検査を意識しない一般的な設計書である。

②：厳密には、ここで決定するのは検証項目群である。このソフトウェアが満たすべき性質で検証する必要のあるものを出来る限り洗い出し、同じモデルで検証できるものごとにグループ化する。

③：②の検証項目のグループの中から検証するものを 1 つ選び、それらを検証するのに必要な構造を残して、①の設計書を抽象化する。

④：③で抽象化した設計書を PROMELA のモデルにマッピングする。

⑤：③で選択した検証項目グループの中で、全く同じ PROMELA モデルで検証できる項目を選択し、それらを検証するのに必要な検証用の構造があれば、④で作成した PROMELA システムモデルに追加する。
⑥：③で選択した検証項目を検証するのに必要な、PROMELA 外部モデルを作成して、PROMELA システムモデルに追加し、このモデル全体を PROMELA 検証モデルとする。
⑦：検証する上で LTL 式により検証性質を記述した方が検証しやすければ、⑤で選択した検証項目群の中の該当するものから、LTL 式を作成する。この時点で、LTL 式がその検証項目を検証するのに適切である事が自明で無い場合には、LTL 式の正しさを確認する。LTL 式の正しさが確認できたら、SPIN の LTL 式を never claims に変換する機能を用いて、never claims を生成し、PROMELA 検証モデルに追加する。
⑧：⑦で作成した PROMELA 検証モデルから、SPIN により検証器 pan.c（pan は Protcol ANalyzer の略）を生成して、コンパイル・実行する。検証結果が期待値と一致して、現在の PROMELA システムモデルで検証する性質が他に残っていれば⑤に戻り、この PROMELA システムモデルで検証するべき性質が残っていなければ③に戻って次に検証する検証項目グループを選択する。もう他に検証するべき検証項目が残っていなければ検証終了し、この設計モデルは仕様を満たしていると判断する。検証結果が期待値と一致しない場合に、検証結果の期待値が真であった場合には、⑨で反例を解析し、不備の原因を分析し、①の設計書を修正する。期待値が偽であった場合には、可能であれば LTL 式を変更して問題の原因の絞込みを行い、不備の原因を分析し、①の設計書を修正する。

2.3 UMLchecker

この章では、本研究で部分的に利用したツールの１つである。「UMLchecker」について述べる。

このツールの主な特徴を以下に述べる。

開発元：

文部科学省主導のプロジェクトである「e-Society」(プロジェクト期間：2003-2007)において、開発された成果物。現在（2007.2）開発途中。

実行環境など：

Eclipse のプラグインとして開発されている。他に、Omondo 社が開発している EclipseUML を UML エディタとして利用している。そのため、以下の２つが必要になる。

・Eclipse3.1.x

・EclipseUML2.1.x

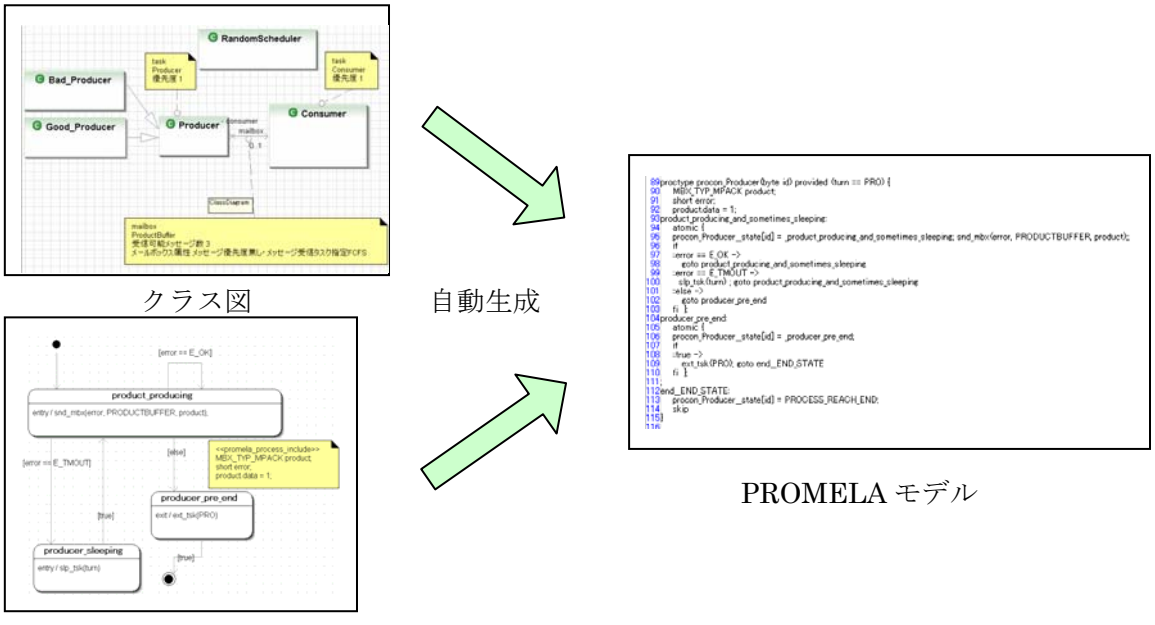
また、検証部分には、SPIN を利用しているため SPIN もインストールされている必要がある。

機能：

・最も大きな機能は、UML のクラス図とステートマシン図を記述すると、そこから PROMELA のモデルを自動生成できる点にある

・他に、LTL 式・検証時のオプション・検証対象クラス指定などを１まとめとして検証項目として、これを管理するための機能が付加されている

操作画面：



クラス図

ステートマシン図

図 2.2 UMLchecker への入力 UML モデルと出力された PROMELA モデル

2.4 μITRON エミュレーションライブラリ

本研究で全体を通して利用している「μITRON エミュレーションライブラリ」[2]（以下μITRON ライブラリ）について概説する。

このライブラリは、μITRON 仕様 OS のスケジューリング機構・システムコールを模擬するための PROMELA で記述されたライブラリである。一般に、RTOS を利用するソフトウェアを SPIN でモデル検査しようとすると、抽象度の差は生じるかもしれないがカーネルの振る舞いを模擬する PROMELA コードを自前で記述する必要がある。しかし、このライブラリを利用する事で、それらの部分を記述する必要がなくなる。さらに、ライブラリ自体もモデル検査によって検証されているため、自前で作ったものよりも信頼度が高いと言える。

また、通常の SPIN のプロセス定義で記述された並行動作するモデルの検査よりも、μITRON ライブラリを利用して、タスクとしてモデルを作成した方が多くの場合に探索空間が狭くなる。これは、SPIN のスケジューリングでは全ての最少処理単位ごとにディスパッチする可能性があるのと比較して、μITRON のスケジューリング規則に固定しているため、ディスパッチの発生箇所がかなり制限されていることに関係している。これにより、RTOS 以外のソフトウェア部分のモデルをより大きく記述できるようになる。

さらに、通常の SPIN のプロセスとして並行動作単位を記述したモデルでは、本来は起こり得ないようなエラーを検知することになる。このようなエラーが検証結果に混じってくると、検証結果の反例から、エラーが発生した原因をその都度判断してやる必要が発生し、検証手順が複雑化する。それに比べて、μITRON のライブラリを利用してタスクとして記述した場合は、実際に発生し得るエラーしか検知しないため、エラーの原因が RTOS 以外のソフトウェア部分にある事が明確になる。そのため分析が比較的容易になる。

本研究で提案する検証手法は、このような RTOS の振る舞いを模擬するための PROMELA モデルが既に作成されている事を前提とする。本研究では、μITRON ライブラリを利用した。また、このライブラリと連携利用できるμITRON 仕様に基づいたメールボックスライブラリを作成した。

第 3 章　マルチタスクソフトウェアのモデル検査手法の提案

3.1 提案手法の対象範囲

本研究で提案するマルチタスクソフトウェアのモデル検査手法の想定する検証対象・検証環境・検証性質を以下に示す。

検証対象

対象ソフトウェア：　RTOS を利用したマルチタスクの組込みソフトウェア

対象 RTOS：　μITRON 仕様 OS

スケジューリング方式：　イベント駆動型のスケジューリング/ディスパッチ処理

検証環境

モデル検査器：　SPIN／PROMELA

（モデル検査支援ツール）：RTOS のスケジューリング・システムコールをエミュレートする PROMELA ライブラリ

検証性質：

システムコールの発行にともなうタスクのデッドロックの有無

メールボックスシステムコールを利用したメッセージの到達性・到達順序の検証

3.2 提案手法で想定するモデル検査の手順

本手法では、ソフトウェアの段階的な設計と同時並行にモデル検査を行うことを想定している。この理由は、「システムの不備をなるべく早く発見するため」と「ソフトウェアの設計者になるべく抽象化の手間を取らせたくないため」という 2 つである。想定しているモデル検査の手順を以下に示す。

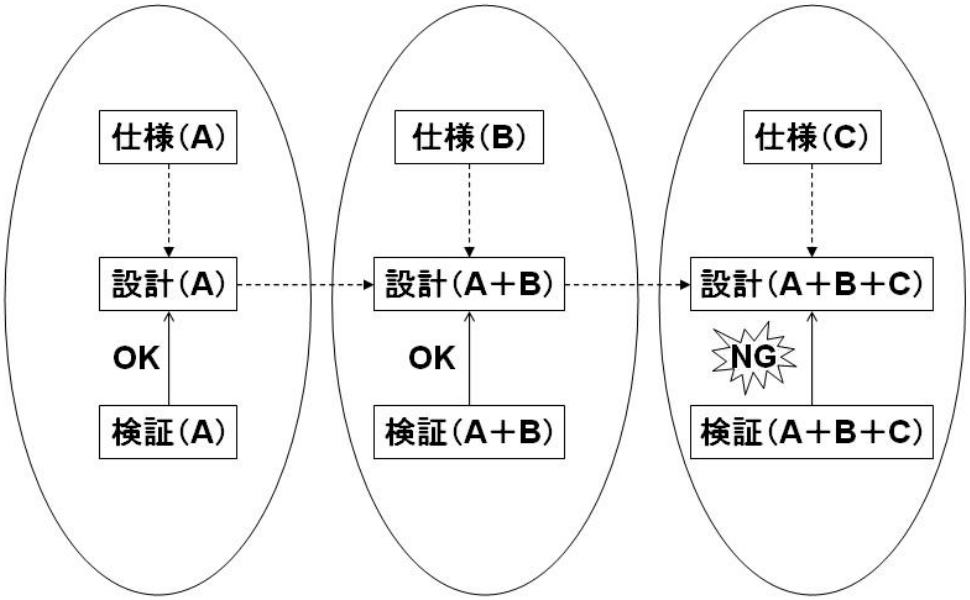


図 3.1　段階的な設計とモデル検査の併用

図 3.1 について説明する。上記の図は、設計を順番に仕様 A、仕様 B、仕様 C を満たす様に変更しながら、設計が変更されるたびにモデル検査を適用して行く過程を示している。

上記の例では、仕様 C を満たすような設計に変更したところで検証エラーが発生している。つまり、設計（A+B+C）と設計（A+B）の差分によってエラーが発生している事が簡単にわかる。エラーの原因を特定するのが容易になるという意味でこの手順は優れている。

また、モデル検査という観点から見た時にもメリットがある。上記の手順を取ることにより、必然的に最初に作成する PROMELA モデルは比較的小さなものになる。つまり、状態爆発の可能性が下がり、モデル検査可能な大きさのモデルに納まる可能性が高くなるにも関わらず、抽象化が現れていない点がメリットである。

さらに、徐々に検査モデルが大きくなっていくので、設計にどのような変更を加えた時に検証不可能な大きさになったかを容易に発見できる。この様な場合の 1 つの選択しとしては、「上手く抽象化することで状態数を抑制すること」が挙げられる。しかし、これは一般のソフトウェア技術者にとっては、難しい作業である。そこで、もう 1 つ考えられるのは、「設計の順序を変更することで、抽象化を行わないで良い範囲でできる限り有効にモデル検査を利用する」ことである。つまり、設計中のシステムには満たすべき複数の仕様があり、それらには優先順位があるはずである。さらに、その中にはモデル検査で検証可能なものとそうでないものがあるはずである。この優先順位が高くモデル検査可能な仕様から順番に設計に反映して行くことで、できる限り抽象化に頼らず、できる限りモデル検査を有効に使うことができる。全ての場合にこれで対応できる訳ではないが、モデル検査に関する知識を極力意識せずに、その恩恵を受ける事ができるとゆう意味では、一定の効果が期待できる。

3.3　タスクの振る舞い設計方針

本研究では、3.2 で示したモデル検査の手順を前提として、設計段階からモデル検査の適用しやすさ、モデル検査の効果の得やすさを考慮して段階的に設計を行ってゆくための指針をまとめた。以下で、その方針について述べる。

3.3.1 提案する設計方針で対象 RTOS を限定している理由

3.1 にも記したように、本研究で提案する設計・検証手法が対象とするソフトウェアは、イベント駆動型のスケジューリング･ディスパッチを行うμITRON 使用の RTOS を利用した組込みソフトウェアである。ただし、タスクの設計方針に限れば、最も重要なのは「イベント駆動型のスケジューリング・ディスパッチを行う」ことである。「イベント駆動型のスケジューリング・ディスパッチを行う RTOS」は、ソフトウェアの設計者の視点で見ると、「ディスパッチが発生する場所を予測しやすい RTOS」であると言える。これが、「タイムシェアリング型のスケジューリング・ディスパッチを行う RTOS」であれば、一定時間ごとにディスパッチャが呼び出されることになるが、ソフトウェアの設計者から見て実際の実行時間は見えないものなので、「ディスパッチが発生する場所を正確に特定する事がほぼ不可能な RTOS」であると言える。これらの RTOS を利用したソフトウェアの設計に対して、SPIN でのモデル検査を適用することを考えると、前者は「制限された場所でしか実行するプロセスの切り替えがおこらないモデル」であり、後者は「ほとんど全ての場所で実行するプロセスの切り替えが起こり得るモデル」である。そのため、探索する状態数にかなりの差が出てくる。つまり、「イベント駆動型のディスパッチを行う RTOS」の方がモデル検査との相性が良い。これが、本研究での対象を限定した理由である。

下の図は、ディスパッチが発生するタイミングの違いを表した図である。この例では、２つのタスクは、同一の優先度を想定したものである。イベント駆動型の例は、一度実行権を得たタスクが、ブロックするか、割り込みにより自タスクよりも優先度の高いタスクが実行可能状態にならない限りは、実行権を保持し続けることを想定したものである。タイムシェアリング型は、一定時間ごとにディスパッチャが呼び出され、この時に現在実行中のタスクを除いた実行可能状態のタスクに実行権を移すことを想定している。また、システムコールを発行しているタイミングだけを記述しているが、それ以外の場所でもシステムコールの発行以外のなんらかの処理が行われているものとする。

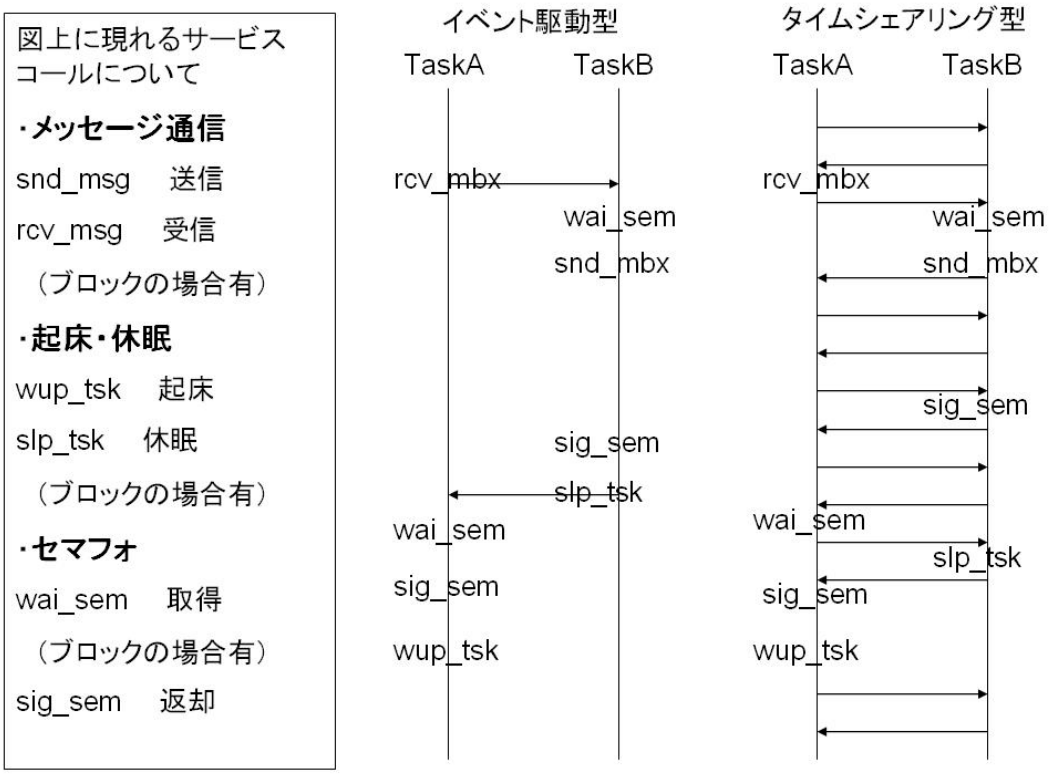


図 3.2 イベント駆動型とタイムシェアリング型のディスパッチタイミングの違い

3.3.2 タスクの設計方針の概要

RTOS を利用するソフトウェアの設計検証を行うには、3.2 で示したソフトウェアのモデル検査手法に加えて、さらに別のアイディアが必要である。そこで、注意すべき性質の１つである「システムコールの発行にともなうタスクのデッドロック・ライブロックの検出」を可能にするために、タスク設計に関して指針を定めた。

具体的には、どのような仕様を優先的に設計に反映して行くと上記の性質を検証し易いかを決定した。それらの仕様を設計に反映するためのモデル検査手順を以下で説明して行く。

主な手順は下記の様になる。詳細については次の章から順次説明して行く。

１．タスク分割
２．タスク間関連の明確化
３．タスクの振る舞い設計
４．状態の生成
５．モデル検査

3.3.3 タスク分割

本手法では、タスクの分割に関しては、特に制約事項は無い。ただし、むやみにタスク数を増やすことは、モデル検査の探索空間を広げること繋がる。あまりにタスク数が少なければモデル検査を適用しなくても、比較的容易にバグを発見することができるかもしれないが、あまりにタスク数が多すぎても、それぞれのタスクを現実的な粒度でモデル化する事ができなくなる。モデル検査の行いやすさを考えると、必要以上にタスクの数が多くない方が無難である。

3.3.4 タスク間関連の明確化

タスクの分割に続いて、タスク間の関連を明確に定義する。ここでの「明確化」は、「タスク間に RTOS の管理するオブジェクトを利用した通信があるかどうか」、「システムコールの発行により他のタスクの状態を変更したり、されたりすることがあるのか」、「複数のタスクが RTOS の管理するオブジェクトではない共有変数などにより影響し合っているかどうか」をはっきりさせるという意味である。関連がある場合、どのシステムコールによって関係しているのかを、わかりやすい形で整理する必要がある。必須ではないが、例として図 3.3 の様なものを想定している。

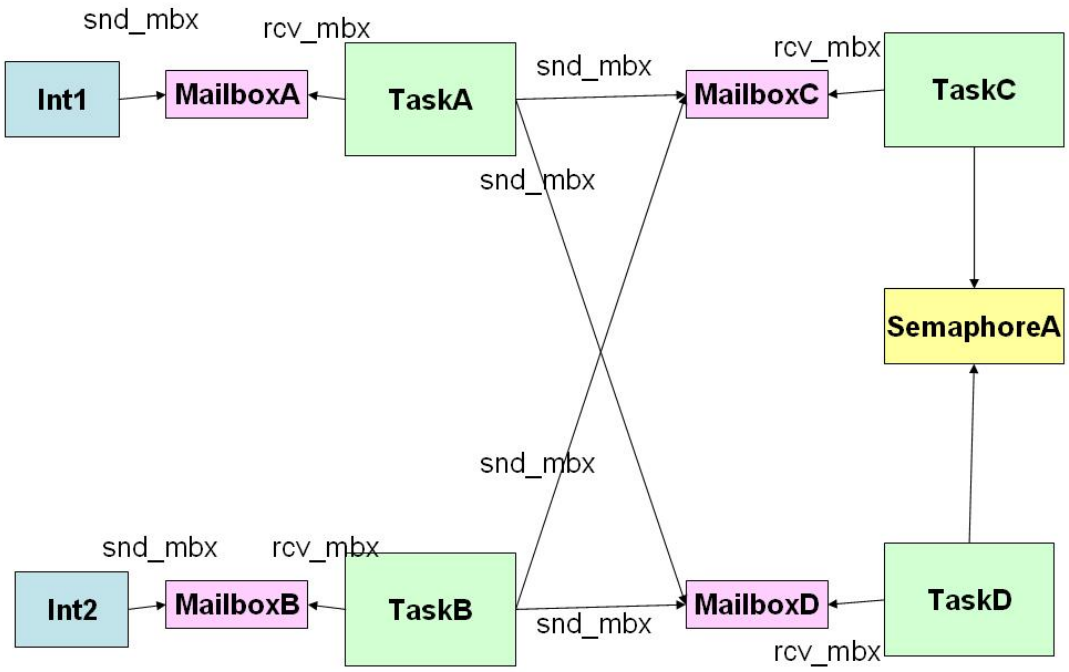


図 3.3　タスク関連図記述例

3.3.5 タスクの振る舞い設計

タスク間の関係が決定したら、次はタスクの振る舞いを設計して行く。このステップには、提案手法の特徴的なポイントが１つ含まれている。それは、「振る舞いを構成する要素を出来る限り、システムコールだけにする」という点である。「出来る限り」と言っているのは、この設計方針の目的がスケジューリングが正しく行われと保証されたモデルを作成することであるため、場合によっては、それを回避するためにシステムコールの呼び出し条件を記述する必要があるからである。このような制約をかけるのは、タスクのスケジューリングに関わらない部分を排除することでモデル検査の探索空間を狭くする狙いがある。

3.3.6 状態の生成

このステップでは、上記の設計から検証用の状態遷移モデルを生成する。このステップに提案手法の２つ目の特徴が含まれる。それは、「状態遷移モデルにおけるそれぞれの状態の中で行われる処理を全て不可分処理として定義する」という点である。この時、状態を作る際の基準も整理したため、以下に示す。

基準となる処理は３種類存在する。

１．ブロックを引き起こす可能性のあるシステムコール

２．プリエンプションを引き起こす可能性のあるシステムコール

３．グローバル変数などの共有資源への参照・定義のための処理

これらの基準を元に、さらに以下のルールに従って状態を生成して行く。

ルール：全ての状態は、「0 個以上の基準以外の処理ではじまり最後に１つの基準１か基準２を含む、または、基準３に該当する処理のみを含む」という形にする。

3.3.7 モデル検査

上記の状態遷移モデルに従って PROMELA のモデルを生成し、検証を行う。唯一の注意点は、「状態」を PROMELA の「d_step シーケンス」に変換するという事である。

第４章 μITORN 仕様メールボックスシステムコールの PROMELA ライブラリの作成・検証

本研究では、2.4 で概説した「PROMELA のμITRON エミュレーションライブラリ」を利用させて頂く。ただし、本研究で扱う事例においては、μITRON 仕様 OS のサービスコールの１種である「メールボックス」が頻出している。しかし、この機能に関しては、現状では提供されていない。この機能がなければ、スケジューリングに関する性質を検証する事ができなかった。そのため、「PROMELA のμITRON エミュレーションライブラリ」と連携利用するための「メールボックスライブラリ」を仕様書を元に PROMELA で実装し、ライブラリ化した。また、仕様が正しく反映されているかどうかを検証し、それを用いて例題の検証を行った。

4.1 メールボックスの仕様と PROMELA モデル作成にあたっての抽象化方針

まず、μITRON におけるメールボックスの仕様について簡単に述べる。下記の仕様は、μITRON の仕様書からの抜粋ではなく、そこから整理して再構成したものである。

メールボックスは、タスク間でメッセージのやり取りをするための非同期通信機構の１種である。メールボックスの属性で代表的なものは、「メールボックス ID」、「メールボックス属性」、「メッセージキュー」、「待ちタスク行列」である。メールボックスのシステムコールのうちで基本的なものは、「生成」、「送信」、「受信」の３つである。

「メールボックス ID」は、メールボックスを一意に識別するためにつけられる識別番号である。したがって、同じ「メールボックス ID」を持つメールボックスが、同時に複数存在する事があってはならない。

「メールボックス属性」は、メッセージを振り分けるルールを指定するための属性である。μITRON のメールボックスでは、４つのルールから選択可能である。以下で述べる「生成」のシステムコールの発行時に指定する。

「メッセージキュー」は、現在送り先が確定していないメッセージを一時的に保存しておくための機構である。

「待ちタスク行列」は、現在受信可能なメッセージがないので、新しいメッセージが入ってくるのを待っているタスクを管理するための機構である。
※「メッセージキュー」と「待ちタスク行列」に同時に何かが登録されている状態にはならない（メッセージが余っているなら、登録されているタスクに渡してやる事ができるはずである)。

「生成」は、「あるメールボックス ID、メールボックス属性を持ったメールボックスを利用できるようにする」ためのシステムコールである。メールボックスの他のシステムコールが呼ばれる前に、必ずこのシステムコールが呼ばれていなければならない。このシステムコールが発行されていないメールボックスを指定して、これ以外のメールボックスのシステムコールが発行されたら、「メールボックスが存在しない」といった意味のエラーコードが返され、他には何の処理もされない。

「送信」は、「あるメールボックス ID のメールボックスにメッセージを登録する」ためのシステムコールである。

「受信」は、「あるメールボックス ID のメールボックスからメッセージを取得する」ためのシステムコールである。
※送信と受信の振る舞いについては詳細を述べないが、下記の状態遷移図に「送信・受信の発行」とそれにともなう「メールボックスの状態の変化」を示す。

尚、C 言語の API では、それぞれ下記の様に記す。尚、返り値、引数の型は省略している。

生成： cre_mbx(mbxid, mbxatr)
送信： snd_mbx(mbxid, pk_msg)
受信： rcv_mbx(mbxid, pk_msg)

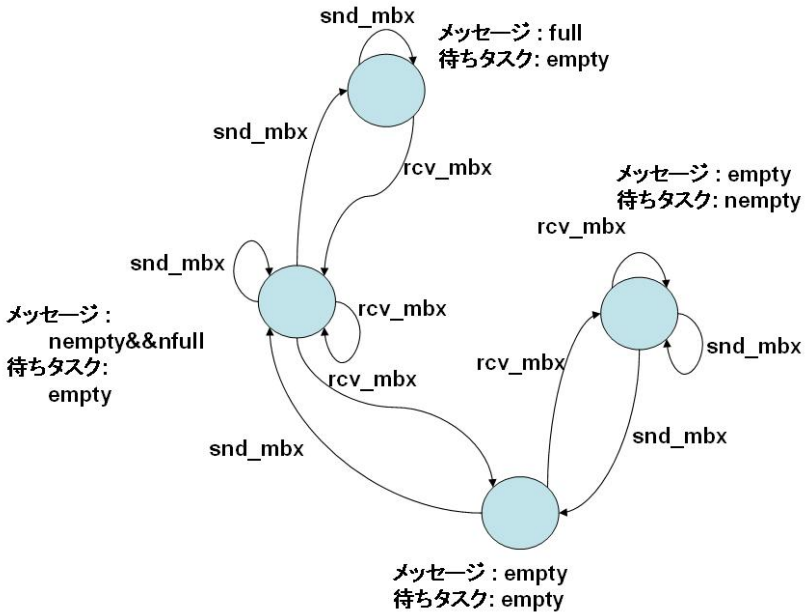


図 4.1 メールボックスの状態遷移図

以下では、PROMELA モデルの作成にあたって大きく２つの点を抽象化した。どのような理由で、どのような抽象化を行ったかを記す。

上記の図は厳密には、メールボックスの仕様とは、異なる点がある。それは、メッセージが一杯で、待ちタスクが空の状態が存在することである。この状態は、起こり得ない状態である。元の仕様では、メールボックスで扱えるメッセージの数に上限は無いことになっている。しかし、実際の使用状況を考えると、メモリプールの管理機能と併用して使うため、メモリプールからのメモリ確保が失敗した時点で、「送信」を実行する事ができなくなる。よって、本当に無限長という事はあり得ない。そのため、本研究で作成したメールボックスライブラリにおいては、送信時の「メモリプールからメッセージ用の領域を取得してから、メールボックスへのメッセージの送信」という流れと、受信時の「メールボックスからメッセージを受信してから、メッセージ用の領域をメモリプールへ返却する」という流れを、メールボックスの機能として一まとめにして抽象化した。メッセージキューが一杯になった時の解釈としては、「メッセージ用に使えるメモリ領域が残っていない」という事になる。

また、メールボックスの仕様では、メールボックスで扱えるデータ型が任意である（1つのメールボックスで異なる型のメッセージを扱うことができる)。これは、メッセージの値ではなく、ポインタをメールボックスで管理しているからである。しかし、PROMELA においては直接的にポインタを扱うことができない。そのため、PROMELA モデルでは、データ型は固定とし、アドレスではなく値を管理することにした。

4.2 メールボックス機能の検証

4.2.1 検証項目

メールボックスライブラリの検証に関しては、大きく２つの項目に関して検証を行った。

１．4.1 の状態遷移図の通りにメールボックスの状態が変わっているか？

２．メールボックスが、常に「メッセージ有り∧待ちタスク有り」にはならないか？

4.2.2 検証と外部モデル

4.2.1 の検証項目に関してシステムコール snd_mbx と rcv_mbx を検証するために、簡単な外部モデルの作成を行い、システムコールに検証のための構造を追加した。尚、検証には「UMLchecker」を利用した。

検証に当たって、初めに下記のモデルを作成した。モデルの概説をする。タスク構成は、メッセージ送信タスク１つ、メッセージ受信タスク１つで、優先度は等しい。それぞれのタスクは送信・受信をただひたすら繰り返しているだけである。タスクのほかに、スケジューラーに該当するプロセスを１つ準備した。このスケジューラープロセスは任意のタイミングで動く事ができ、現在実行中のタスクの優先度キューをローテーションさせる inline 関数を呼び続ける。したがって、全ての可能なタイミングでプリエンプションが発生する可能性があるモデルであると言える。

初めは、外部モデルを作成して、ただちに以下の LTL 式によって検証を行っていた。

```
/*rcv_mbx の発行直前にメールボックスの状態が、待ちタスクのある状態だった*/
#define precon_e_ne (precondition == WTSK_NEMPTY)
/*rcv_mbx の内部のブロックポイントの直後で、待ちタスクのある状態だった*/
#define poscon_e_ne (poscondition == WTSK_NEMPTY)
/*rcv_mbx の内部のブロックポイントの直後で、メッセージも待ちタスクも無かった*/
#define poscon_e_e (poscondition == MBUF_EMPTY_WTSK_EMPTY)
/*
 *      Receive プロセスの状態 Receiving に到達したときに、
 *      1 回前の rcv_mbx の発行によるメールボックスの状態の遷移が、
 *      正しく行われているかどうかを検証する。
 *      この場合は、常に
 *      事前状態が、      「メッセージ無し∧待ちタスク有り」ならば、
 *      事後状態が、      「メッセージ無し∧待ちタスク有り」か
 *                        「メッセージ無し∧待ちタスク無し」になる
 *      ということを検証する。
 *      ![]((Receiver@Receiving && precon_e_ne) -> (poscon_e_ne || poscon_e_e))
 */
```

検証結果は「OK」であったが、この段階で、前提条件が成立する事を確認していないことに気がついたため、検証をやり直した。

まず、このメールボックスシステムコールを呼び出す外部モデルが、想定している振る舞いをしているかどうかをモデル検査した。今回この外部モデルに対する要求は、「4.1 で示したメールボックスが取る可能性のある 4 つの状態それぞれにおいて、snd_mbx も rcv_mbx も呼ばれる可能性のあるモデルである事」である。外部モデルの検証の結果、「メッセージ無し∧待ちタスク有り」の状態で rcv_mbx が発行される可能性が無いことがわかった。結果が出てしまうと原因は自明であった、メッセージ受信タスクが 1 つしかないので、自分が待ち状態に入っている時には、rcv_mbx を発行するタスクは他にいないのである。つまり、検証用の外部モデルが仕様を満たしていなかったと言える。この例はかなり単純な部類の例であるが、より複雑な外部モデルを生成しなければならない事も当然あるはずである。しかし、ここで重要な事は「バグが発見し易いものであるか？発見し難いものなのか？」とか、「外部モデルの複雑さがどの程度か？」という事では無い。

この例が示している事は、検証には取るべきステップが存在し、そのステップをとらずに検証を行うと、意味のある検証結果を得られなくなる場合があるという事である。

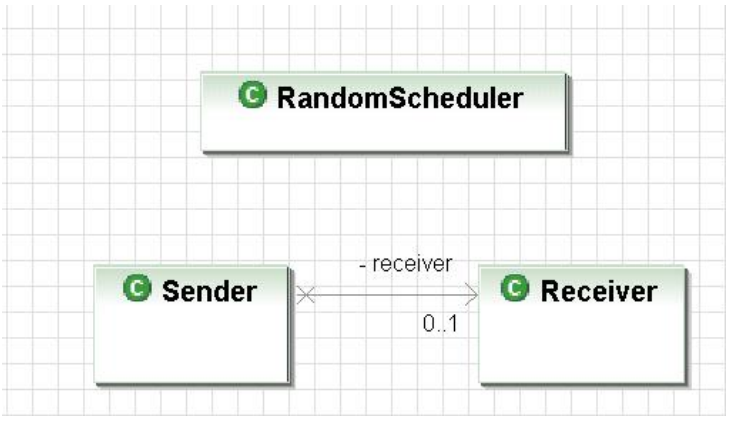


図 4.2 検証用外部モデル 1 （クラス図）

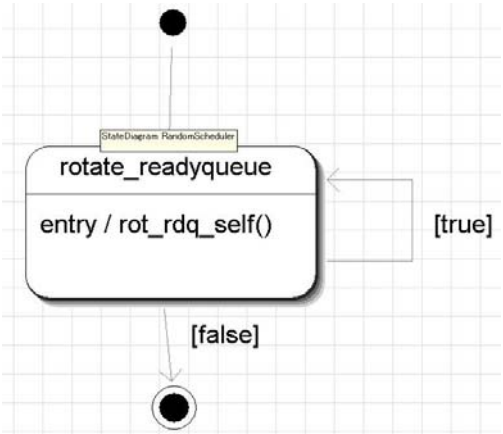


図 4.2 検証用外部モデル 1 （ランダムスケジューラーのステートマシン図）

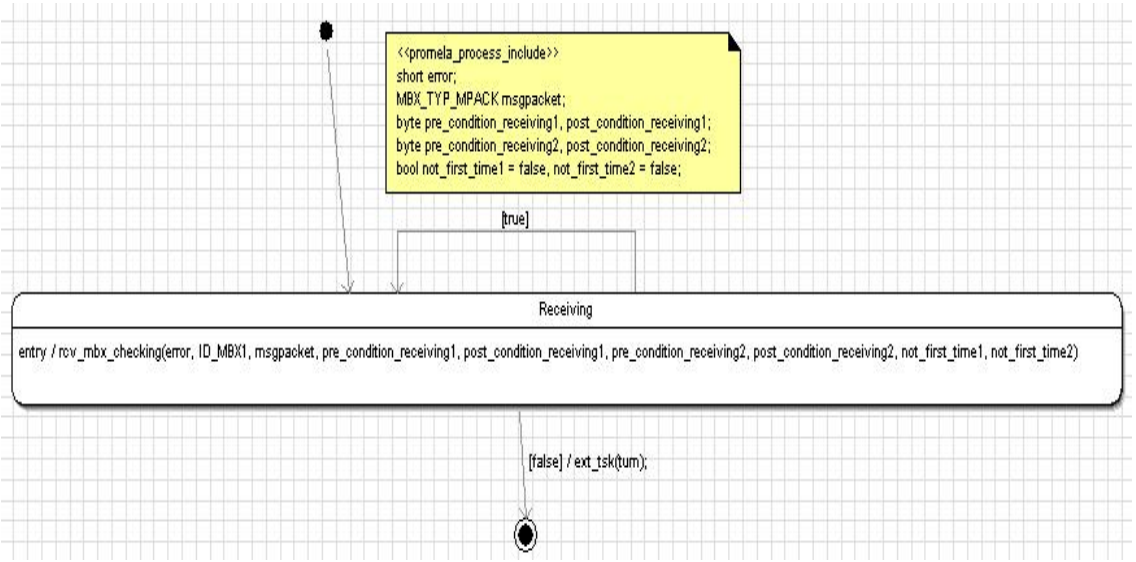


図 4.2 検証用外部モデル１（受信タスクのステートマシン図）

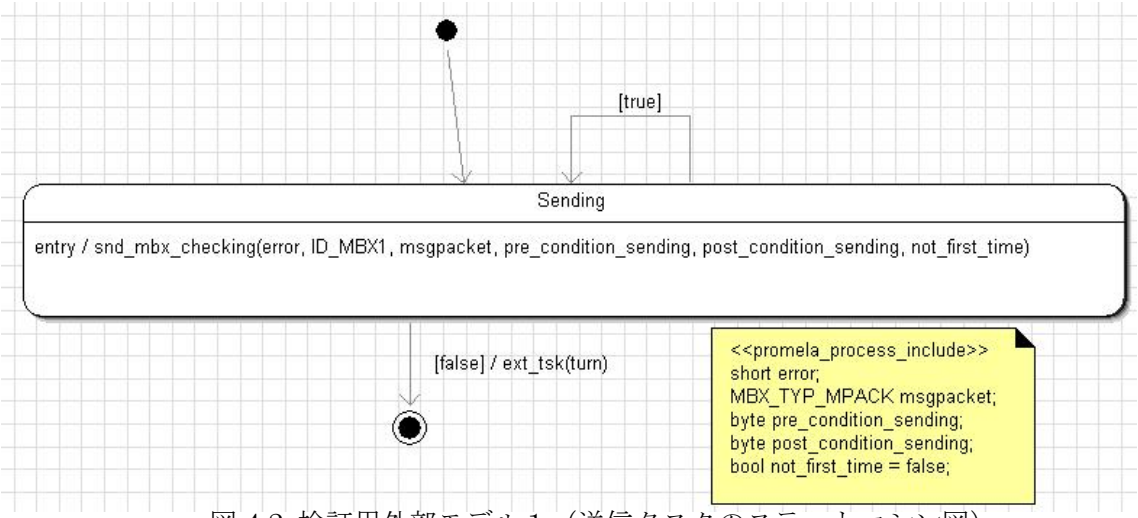


図 4.2 検証用外部モデル１（送信タスクのステートマシン図）

最終的には、受信タスクを２つに増やして、検証を行い、メールボックスの状態遷移については、仕様を満たしている事を検証できた。また、メールボックスが既定されている 4 状態以外の状態にならないことも検証できた。

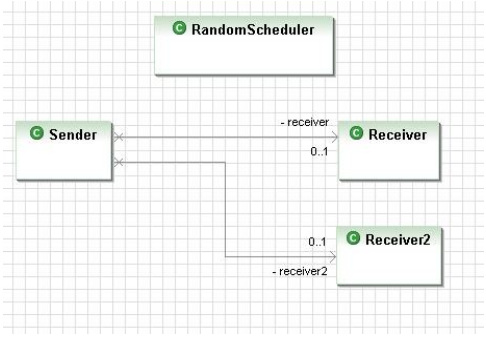
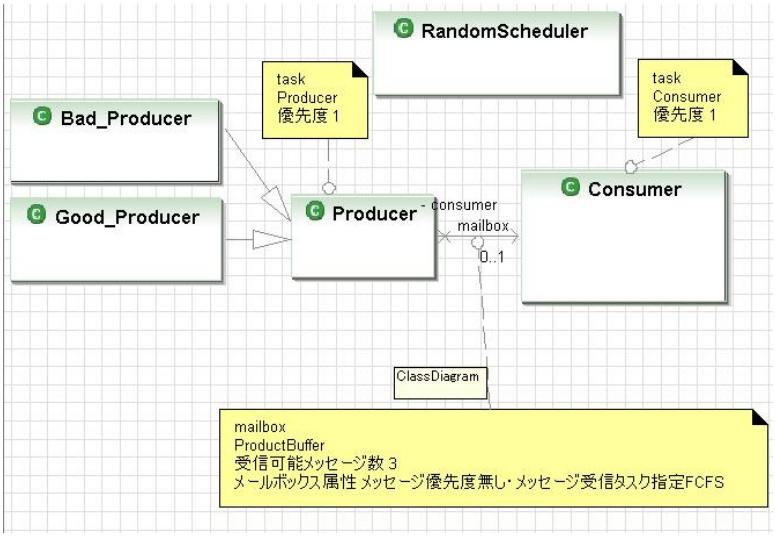


図 4.3 検証用外部モデル２（クラス図）

4.3 メールボックス機能を利用した例題の検証

本研究では、メールボックスが正しく機能している事と、目的としているディスパッチのタイミングに関わる問題を発見することが可能である事を確認するために、メールボックスライブラリを用いて簡単な例題の検証を行った。本研究で扱った例題は、並行動作するプログラムの例題として良く出る「生産者・消費者問題」である。

2.3 で概説した「UMLchecker」と 2.4 で概説した「RTOS エミュレーションライブラリ」,「本研究で作成したメールボックスライブラリ」を連携利用して、モデル検査を行った。結果、デッドロックを検出できた。



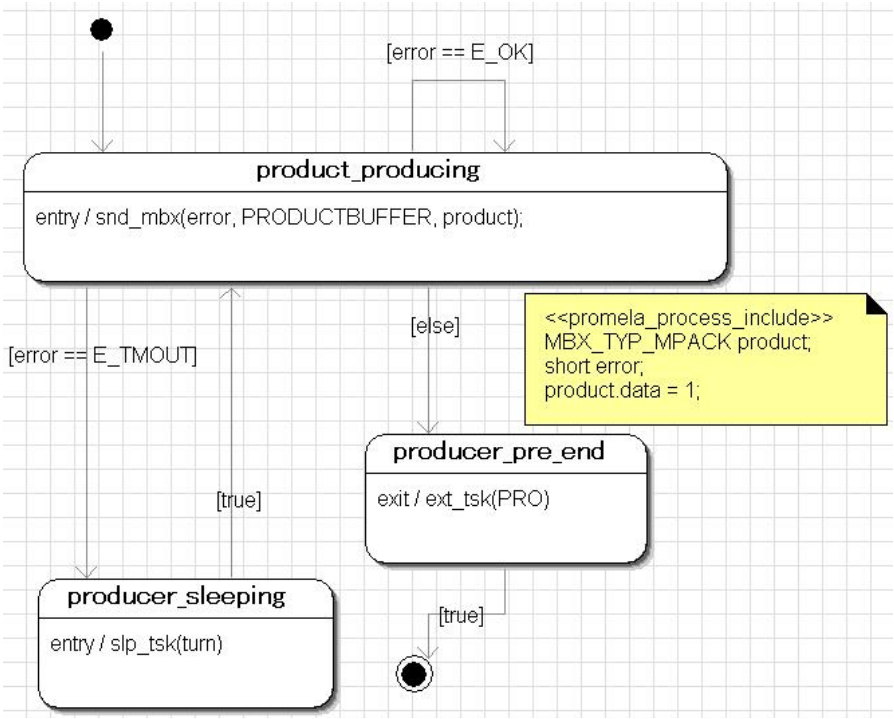


図 4.4　生産者消費者問題

第 5 章 実事例への提案手法の適用

5.1 提案手法による設計モデルの生成

以下に、分析設計資料とソースコードを元に洗い出したタスク関連図とステートマシン図を示す。尚、システムコールの呼び出し箇所については、メールボックスに関わる場所と EventMake タスクのイベント再送構造で rot_rdq を使用している所に着目して抽出しているため、元々の事例と等価なモデルには成っておらず、事例を元に再構成した別の設計モデルと見る方が適切かもしれない。

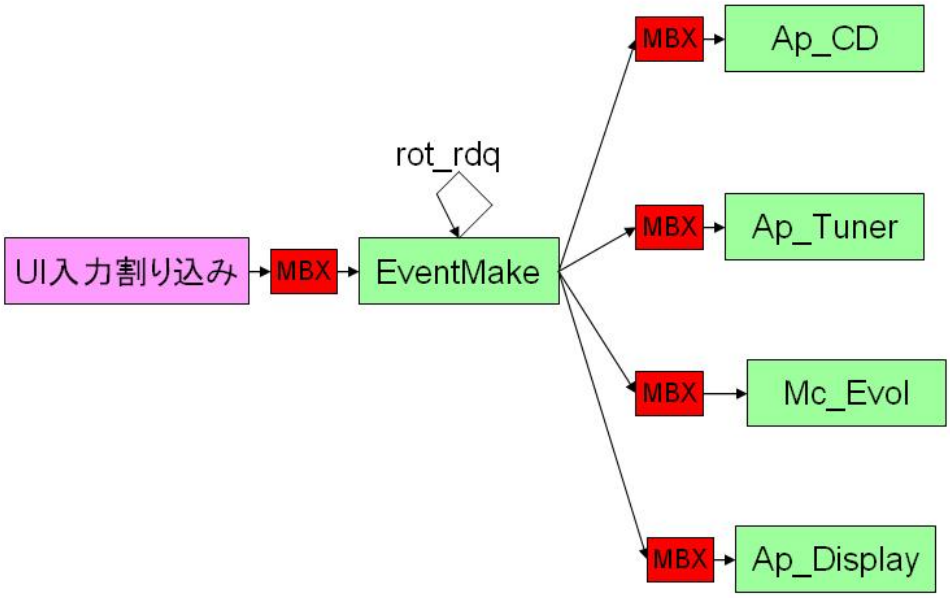


図 5.1 カーオーディオのタスク関連図

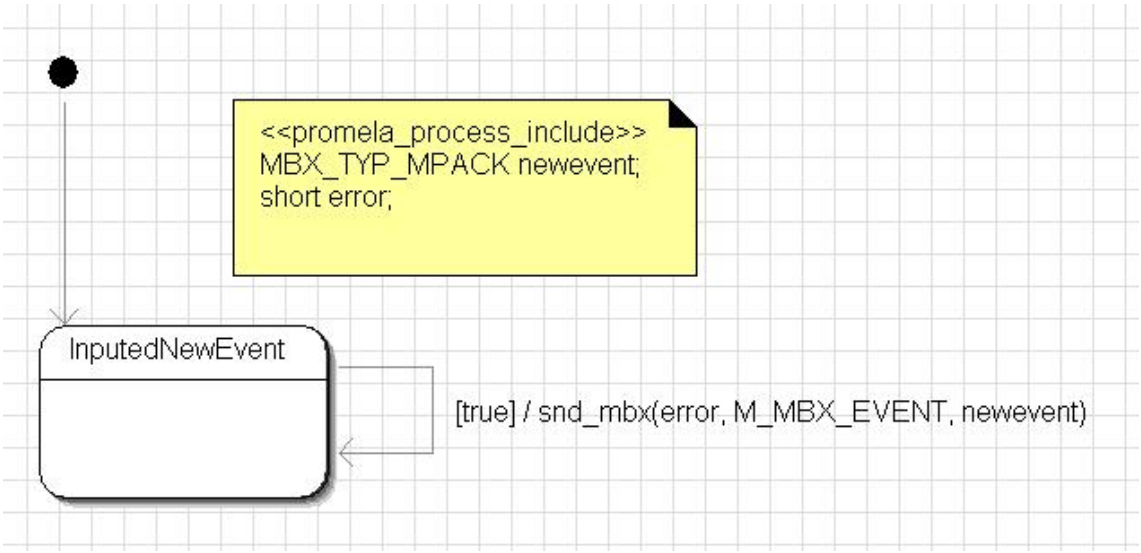


図 5.2 UI 入力割り込み外部モデルのステートマシン図

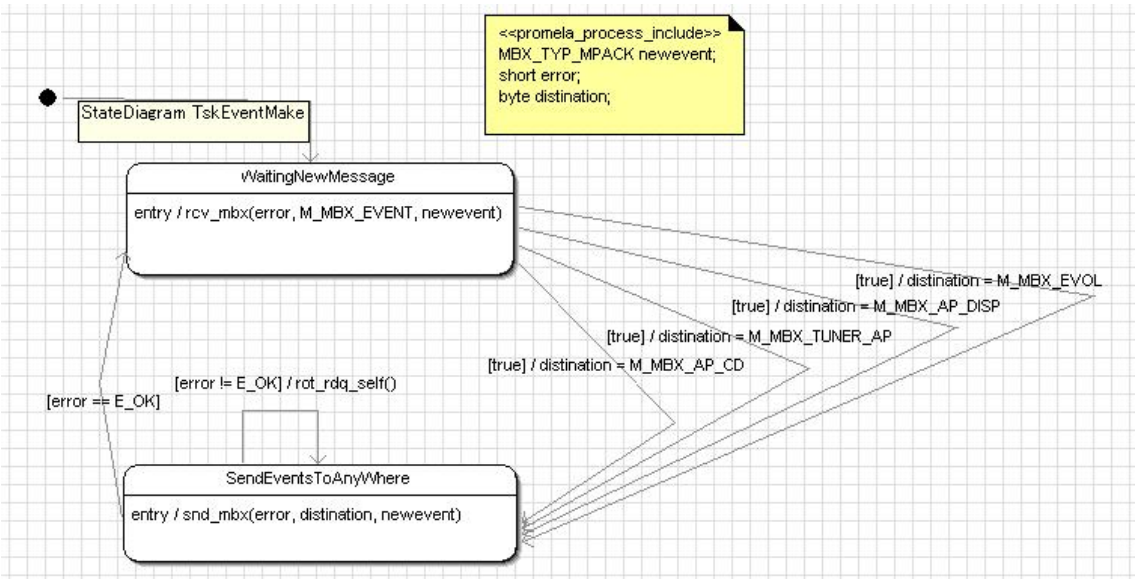


図 5.3　EventMake タスクのステートマシン図

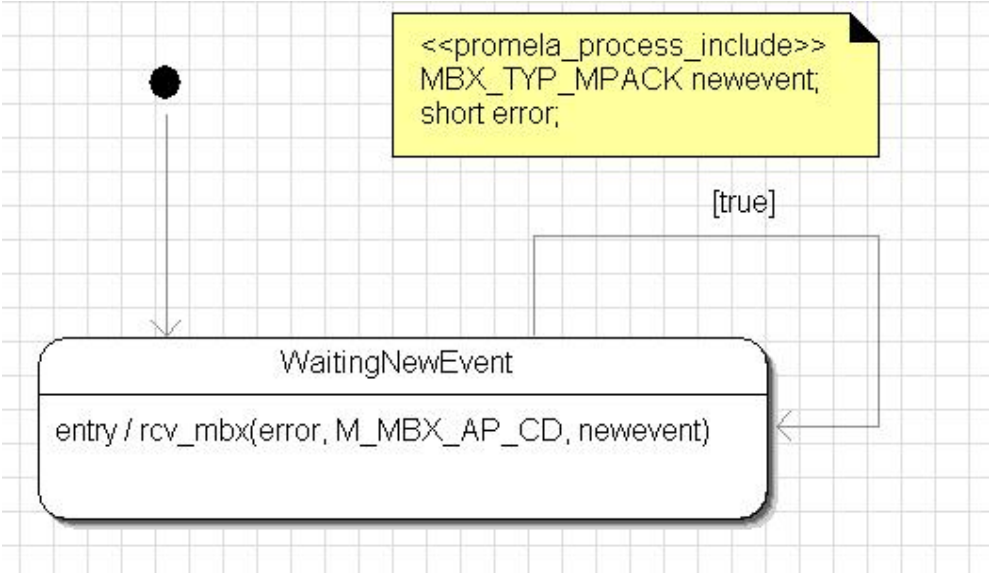


図 5.4　Ap_CdController タスクのステートマシン図

タスク分割は、図 5.1 に示す通りである。ユーザーインターフェイスからの入力は、直接それぞれのコントローラタスクに配分されるのではなく、一度 EventMake タスクがすべのユーザー入力イベントを受信し、その内容を分析し、それぞれのコントローラタスクへ振り分ける構造になっている。また、EventMake タスクからのコントローラタスクは、送信失敗した場合には無限に再送処理を繰り返す構造にした。図上からは読み取れないが、タスクの優先度は全て等しいものとする。

次に、それぞれのステートマシン図について説明する。まずは、図 5.2 に示すユーザーインテーフェイスからの入力割り込みのステートマシン図である。割り込みは、当然かもし

れないが、タスクとしてではなく、その発生タイミングが制御されないプロセスとしてモデル化する。また、これは基本的に全ての割込みのモデルに言えることだが、状態数は1つ、遷移は自己遷移のみとする。この理由は、基本的に割り込み処理は、多重割り込みで無い限りは分割されて実行される事が無いため、発生したら必ず最後まで実行し終えなければならないからである。本研究で提案している手法では、多重割り込みは考慮外である。多重割り込みを実現したければ、割り込みプロセスの宣言にも、provided 修飾節を加えて割り込み同士の間で優先度をもうける必要がある。

図 5.3 は、EventMake タスクのステート図である。前述した様に、このタスクはユーザーインターフェイスからの入力を全て一度このタスクで受けて、そのイベントの内容に応じてそれぞれのハードウェアコントローラタスクにイベントを振り分けるのが仕事である。今回の設計では、状態を 2 つに分けることになった。これは、3.3.6 で述べた状態の生成ルールに従っている。つまり、rcv_mbx のシステムコールの特性上、その中で待機状態になることを考慮すると、「ユーザーインターフェイスからの入力要求をしてからブロックするまで」、「ユーザーインターフェイスからのイベントを受信してブロック解除してから、そのイベントの中身を分析してどのハードウェアコントロールタスクに通知するかを決定するまで」、「ハードウェアコントローラへのイベント通知を試行するまで」の 3 状態があり、割り込みを許可するのは、それぞれの状態の間の遷移上だけである。尚、このステートマシン図では、遷移上にアクションが記述されているが、これは UMLchecker に依存した記述であり、解釈としては、「記述されている遷移の遷移元の状態で内部状態 exit の処理が実行される直前に実行されるアクション」となる。また、送信状態において snd_mbx の実行後エラーコードが設定されていた場合に、rot_rdq が発行されているのは、snd_mbx がエラーを返すという事がそのシステムコールの抽象化方針から解釈すると、「メモリプールからメッセージ用の領域を確保できなかった」となるため、「rot_rdq を発行する事によりいずれかのタスクが実行状態になることを期待して、さらにそのタスクが rcv_mbx を発行しているタスクであればメッセージ受信後にメモリプールにメモリブロックが返却されるのを期待している。」といった意味がある。これと似たような構造は、実際の事例の実装にも現れている。

図 5.4 は CD ドライブ制御タスクである。ただし、この設計では大分抽象的な記述しかしていないため。ここも、rcv_mbx の内部ブロックを考慮して次のような解釈になる。「イベント受信待ち状態」と「イベント処理状態」の 2 状態からなり、イベント処理状態ではいかなる割り込みも受け付けない設計である。

5.2 検証項目

現在のところ、特定の性質の検証は行えていない。SPIN のデフォルトである「不正な終了状態がないかどうかの検証」のみを行った。

5.3 検証結果

現在のところ、上記の性質しか検証できていない。ただし、特定の性質は検証してはいないが、「不正な終了状態がないかどうかの検証」を行った。結果としては、図 5.1 全体を同時に検証することができた。

5.4 考察

提案手法を元に、カーオーディオ事例の検証を行った。対象範囲全体を同時に検証する事が出来た。５つのタスクを同時に動かしながら検証することが出来た。この５つのタスクと１つの割り込みプロセスを同時並行に動かしたモデルは決して大きなシステムとは言えないが、特定の提案手法のガイドラインにそって抽象化を行い検証可能な大きさのモデルを作成出来た事には意味があると考える。

第６章 おわりに

本研究では、μITRON 仕様 OS のシステムコールの発行が原因となるシステムの望ましくない状態を設計モデルから排除することを目的として、組込みソフトウェアの設計検証手法を提案した。また、μITRON エミュレーションライブラリと併用できるメールボックスライブラリを作成した。最後に、提案手法に基づいた設計検証プロセスで、作成したライブラリを利用して事例の検証を行った。

提案手法では、組込みソフトウェアの設計検証において、設計モデルから、抽象化によって検証可能な大きさの意味のある検証モデルを作成するためのガイドラインを示した。本手法に従って、カーオーディオの事例検証を行った結果、検証可能な大きさのモデルを作成する事ができた。本研究で提案した手法は、設計の段階的な詳細化を想定したものである。しかし、現段階では最初のステップだけしか考慮していない。設計を詳細化して行き、全ての設計情報を含んだ１つのモデルによってモデル検査を行おうとすれば、いずれ状態爆発が起きるのは自明である。それは、5 章で検証したような単純な例でも該当する。

従って、今後、モデル検査を用いた設計・検証法を確立して行く上では、モデルを切り分けて検証するための手法とその手法を適用するためにはどのような制約があるかを整理する必要があると考えている。

謝辞

本研究を行うにあたり、ご指導賜りました岸知二特任教授、片山卓也教授に深く感謝いたします。また本研究を進めるにあたりアドバイスをして下さった青木利晃特任助教授に厚く御礼申し上げます。

参考文献

[1] Gerard J.Holzmann, THE SPIN MODEL CHECKER: Primer and Reference Manual, Pearson Education.

[2] 青木利晃,片山卓也,RTOS に基づいたソフトウェアのためのモデル検査ライブラリ,組込みソフトウェアシンポジウム２００５論文集,2005.

[3] μITRON4.0 仕様 Ver.4.02.00,（社）トロン協会 ITRON 部会.