修 士 論 文

# ソフトウェアプロダクトライン開発における<br>コア資産の設計検証手法に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

朝倉　功太

2009 年 3 月

修 士 論 文

# ソフトウェアプロダクトライン開発における コア資産の設計検証手法に関する研究


指導教官　岸　知二　特任教授

審査委員主査　岸　知二　特任教授
審査委員　落水　浩一郎　教授
審査委員　青木　利晃　特任准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0610201 朝倉　功太



提出年月: 2009 年 2 月

**概 要**

ソフトウェアプロダクトライン開発は、プロダクトラインアーキテクチャとコア資産（再利用資産）に基づく体系的な開発形態であり、製品系列を持つソフトウェアの開発を行うことの多い組込み分野を中心にその有効性が注目されている。しかし、ソフトウェアプロダクトライン開発を通じて再利用されるコア資産に不具合があるとその影響は大きく、その設計・検証手法が重要な課題となっている。そこで本稿では、特にアーキテクチャ上の想定に関わる問題に注目し、コア資産設計時のアーキテクチャ上の想定をモデル化する手法と、その想定を含めた設計の検証手法を提案する。

# 目 次

**第 1 章 はじめに** **1**
1.1 背景 . . . . . 1
1.1.1 ソフトウェアプロダクトライン開発への関心の高まり . . . . . 1
1.1.2 コア資産開発の難しさ . . . . . 1
1.2 目的 . . . . . 2
1.3 本稿の構成 . . . . . 2

**第 2 章 アーキテクチャ上の想定について** **3**
2.1 アーキテクチャ不整合 . . . . . 3
2.2 アーキテクチャ上の想定の分類 . . . . . 3
2.3 想定とコア資産の設計検証 . . . . . 4
2.4 想定の明示化に対する取り組み . . . . . 4
2.4.1 岸らの想定モデリング . . . . . 5
2.4.2 Trew らの設計上の意図をドキュメント化する取り組み . . . . . 5

**第 3 章 コア資産の設計検証手法** **6**
3.1 提案 . . . . . 6
3.1.1 想定のテンプレート化 . . . . . 6
3.1.2 想定のモデル化 . . . . . 7
3.1.3 想定を含めた検証 . . . . . 7
3.2 アーキテクチャ上の想定のモデル化と検証の流れ . . . . . 7

**第 4 章 想定のテンプレート化** **9**
4.1 概要 . . . . . 9
4.2 想定モデルテンプレートの構築 . . . . . 10
4.2.1 モデル化の流れ . . . . . 10
4.2.2 システム基盤にて提供されるメカニズム . . . . . 10
4.2.3 標準化された分類 . . . . . 11
4.2.4 想定モデルテンプレートの階層構造 . . . . . 11
4.2.5 想定モデルテンプレートと MARTE の対応 . . . . . 11
4.2.6 構築された想定モデルテンプレート . . . . . 17

**第 5 章 想定のモデル化** **20**
5.1 概要 . . . . . 20
5.2 設計想定モデルの構築 . . . . . 21
5.2.1 例題の説明 . . . . . 21

5.2.2 モデル化の流れ . . . 21
5.2.3 設計想定モデルの階層構造 . . . 22
5.2.4 想定モデルテンプレートの絞り込み . . . 22
5.2.5 プロダクトラインアーキテクチャ上の基盤に対する想定のモデル化 . . . 23

**第 6 章 想定を含めた検証 27**
6.1 概要 . . . 27
6.2 検証の流れ . . . 27
6.3 想定を含めた検証方法 . . . 28
6.3.1 検証性質 . . . 28
6.3.2 検証想定モデルの構築 . . . 28
6.3.3 設計検証モデルの構築 . . . 30
6.3.4 検証性質を確認する仕組みの検討 . . . 31
6.3.5 検査コードの作成 . . . 31
6.3.6 モデル検査 . . . 33

**第 7 章 適用例 34**
7.1 例題設定の説明 . . . 34
7.2 想定のモデル化 . . . 35
7.2.1 設計想定モデル . . . 35
7.3 想定を含めた検証 . . . 37
7.3.1 検証性質 . . . 37
7.3.2 検証想定モデル . . . 37
7.3.3 検証性質を確認する仕組み . . . 38
7.3.4 検査コード . . . 39
7.3.5 モデル検査の結果 . . . 39
7.3.6 不具合の概要 . . . 40
7.3.7 優先度に対する想定の再検討 . . . 41
7.4 例題を通じた手法の評価 . . . 42

**第 8 章 議論 43**
8.1 設計想定モデルを設計意図を表すドキュメントとして活用する . . . 43
8.2 検証想定モデルを検証を行った範囲を表すドキュメントとして活用する . . . 43
8.3 ツールによる想定のモデル化支援 . . . 43

**第 9 章 おわりに 44**
9.1 まとめ . . . 44
9.2 今後の課題 . . . 44
9.3 謝辞 . . . 44

**付 録 A テンプレート化したμ ITRON4.0 仕様 スタンダードプロファイルのメカニズム 46**

**付 録 B ステートマシン図（ストップウォッチ SPL） 48**

**付 録 C 設計想定モデル (ストップウォッチ SPL) 52**

**付 録 D 検証想定モデル (ストップウォッチ SPL)** **62**

**付 録 E 検査コード (ストップウォッチ SPL)** **66**

# 第1章　はじめに

## 1.1　背景

### 1.1.1　ソフトウェアプロダクトライン開発への関心の高まり

組込みソフトウェアが様々な機器に内蔵され、社会のいたるところで使われるようになり、多機能化、短納期化に加え、多様な要求に対応するために多くのバリエーションを持った製品を開発する必要性が高まってきた。そうした中、製品系列として開発される組み込まれるソフトウェアに対して、共通したソフトウェアアーキテクチャ（プロダクトラインアーキテクチャ）と再利用可能なソフトウェアコンポーネント（コア資産）に基づく、体系的な開発形態であるソフトウェアプロダクトライン開発の有効性が注目されている。

### 1.1.2　コア資産開発の難しさ

製品系列内のソフトウェアはプロダクトラインアーキテクチャに基づき、様々なコア資産や製品に特化したコンポーネントが複雑に組み合わされることで開発される。そのため、多くの製品開発を通じて十分な信頼性が確保されたと思われていたコア資産と特定のコンポーネントとの結合時に予期せぬ不具合が発生することがある。

例えば、Trew らは高性能テレビのソフトウェアプロダクトラインの開発経験において、特定の製品で使用実績のあるコンポーネントを他の製品開発において、再利用しようとしたが結合時に様々な不具合が発生し、プロダクトライン開発に期待していた開発効率の向上や品質の改善が十分に得られないという問題に直面した [2]。不具合の具体例としては、2 種類のコンポーネントをひとつの製品に実装したところ、単独では動作していたコンポーネントが、互いの優先度の関係から動作しなくなるというものであった。彼らは、複数の開発経験で得た約 900 件の障害票に基づき、それらの不具合原因を調査した。その結果、以下のようなクラス結合時の不具合原因が明らかになった。

1. 設計と実行時のパラメータ範囲の不整合
2. パラメータ値の解釈の不一致
3. パラメータ順序の不一致
4. 共有されたグローバルデータの使い方の不一致
5. 予期せぬ状態イベントの組み合わせ
6. 予期せぬ再入可能性
7. 競合状態
8. 保護されていないクリティカルセクションやデッドロック

9. 不明瞭なモジュール間責務の割り当て

1, 2, 3, 4 に関してはデータの扱い、5, 6, 7, 8 に関しては動的な振る舞い、9 に関してはモジュール分割の問題であると言える。つまりこれら不具合の原因は、各コンポーネントの提供する機能的な側面にあるのではなく、各コンポーネント設計時に想定しているアーキテクチャ設計と再利用先のそれとが一致していなかったことにあると考えられる。

## 1.2 目的

Trew らの経験からも明らかなように、コア資産を設計ならびに検証する際、アーキテクチャ上の想定が適切に設定されていなければ、コア資産の再利用による生産性の向上は期待できない。

よって、本稿ではコア資産を含むプロダクトラインアーキテクチャ設計におけるアーキテクチャ上の想定を体系的かつ明示的にモデル化し、その想定モデルに基づいた設計の検証を行う手法を提案することを目的とする。

## 1.3 本稿の構成

本稿の構成は以下の通りである。

2 章では、アーキテクチャ上の想定に関連する研究を紹介する。3 章では、提案するコア資産の設計検証手法の概要について述べる。4 章では、想定のテンプレート化の概要を述べた後、システム基盤に対して考慮すべき想定をテンプレート化する方法を示す。5 章では、想定のテンプレートに基づきプロダクトラインアーキテクチャ上の想定をモデル化する方法について述べる。6 章では、プロダクトラインアーキテクチャ設計上の想定を表すモデルに基づく、モデル検査技術を用いたアーキテクチャ上の想定を含む検証方法について述べる。7 章では、コア資産の設計検証手法を事例に適用し、その評価を行う。8 章では、関連する議論を行う。

# 第2章　アーキテクチャ上の想定について

## 2.1　アーキテクチャ不整合

Garlan らはアーキテクチャ設計を行う際、明示的あるいは暗黙的に設定される想定のことをアーキテクチャ上の想定と呼び、その重要性をアーキテクチャ不整合の議論の中で指摘している [3]。アーキテクチャ不整合とは再利用を困難にしている理由を分析した結果の観測である。これは、再利用資産は設計される時点で、それ自身が利用されるアーキテクチャ設計に対する想定を持っており、使う側がそれと整合したアーキテクチャを持っていないとその利用が困難になるという捉え方である。また表 2.1 は、彼らが指摘したアーキテクチャ不整合に関わるアーキテクチャ上の想定を表にまとめたものである。

| カテゴリ | 項目 | 内容 |
|---|---|---|
| コンポーネント | 基盤 | 再利用資産が稼動する基盤の特性に対する想定 |
| | 制御モデル | 制御スレッドのあり方に対する想定 |
| | データモデル | 扱うデータのモデルに対する想定 |
| コネクタ | プロトコル | コンポーネント間の協調方法に対する想定 |
| | データモデル | 交換されるデータのモデルに対する想定 |
| 全体アーキテクチャ | — | コンポーネント構成に対する想定 |
| 構築プロセス | — | コンポーネントの生成順序に対する想定 |

表 2.1: アーキテクチャ不整合に関わるアーキテクチャ上の想定

## 2.2　アーキテクチャ上の想定の分類

岸らは、コア資産検証上の課題についての議論の中で、アーキテクチャ上の想定を以下のように分類し、典型的な想定を例示している [5]。

- システム外部に対する想定：

  システムがどのような環境の中で、どのような使われ方をするのかという想定。例えば、センサーを監視する場合、そこで検知されうるイベントの種類やその順序に関する想定など。

- システム内部に対する想定：

  システムの持つ個々の処理が、どのような内部状況の中で動作するかという想定。例えば、意味的には独立した処理であっても、それらが同時に動作することで共有データを介して設計上の干渉を持つことがありうる。そうしたシステム内部で同時に起こりうる処理に関する想定など。

- システム基盤に対する想定：

  制御モデルやデータモデルといったシステム基盤が提供するメカニズムに関する想定。例えば、スケジューリングポリシーや通信モデルなど。

また、彼らは Garlan らがアーキテクチャ不整合の議論 [3] の中で指摘しているアーキテクチャ上の想定は、主に上記のシステム基盤に対する想定に関わると分析している。

## 2.3 想定とコア資産の設計検証

コア資産開発手法の例として、Klaus らが過去 8 年間に得られた製品系列開発における知見に基づき開発した、ソフトウェアプロダクトライン開発向けフレームワーク（SPLE フレームワーク）を紹介する [4]。

彼らのフレームワークにおいて、ドメイン成果物（コア資産に相当）は、製品系列の参照アーキテクチャ（プロダクトラインアーキテクチャに相当する可変性を含めたシステム全体の構成）に基づき、コンポーネントの単位で詳細設計、実装、検証が行われる。ここでは、コンポーネント設計におけるインタフェース定義の重要性や可変性の実現方法など、成果物の再利用化のための議論がアプリケーションに近いレベルを中心に行われている。しかし、彼らのフレームワークではハードウェアや OS により提供される機構など、コンポーネントが動作する基盤について十分議論されているとは言えない。よって、このフレームワークに従い、再利用されるコンポーネントを設計したとしても、アーキテクチャ上の想定が明示化されていないことに起因するアーキテクチャ不整合が発生する可能性があると言える。

Klaus らのフレームワークと同様に現実の設計においても、アーキテクチャ上の想定は必ずしも明示的に記述されない。例えば、システム外部に対する想定はその分野における常識的な知識とみなされるかもしれないし、システム内部に対する想定は、システム開発上のノウハウであると考えられる場合もある。あるいは、システム基盤に対する想定は、実装環境の問題であると考えられることもある。

しかしながら、これらの想定はコア資産を設計・検証する上で、極めて重要な情報であると言える。なぜならば、コア資産は複数の製品に再利用されるが、こうした想定は製品毎に変化しうるからである。例えば、製品毎に利用される環境が異なるかもしれない。また、実装される機能の種類や数は製品毎に異なる可能性もある。もしくは、製品によって利用する OS やハードウェア構成などのシステム基盤が異なる場合もあるからである。

コア資産が様々な製品に使われうるためには、そのコア資産はそれを再利用する製品群の持つ想定の多様性に対応した設計がなされていなければならない。別な言い方をすれば、プロダクトラインアーキテクチャは、その上で作られる製品群の持つ想定の広がりを受け止められる設計になっていなければならない。

ゆえに、コア資産の設計検証において想定の広がりを明示的に記述することが、再利用性を高める上で重要であると言える。

## 2.4 想定の明示化に対する取り組み

ソフトウェアプロダクトライン開発におけるアーキテクチャ上の想定の明示化という観点で、関連研究について議論する。

### 2.4.1 岸らの想定モデリング

岸らは、コア資産の設計ならびに検証を行う上で、コア資産の再利用のされ方に対する想定を明示化することの重要性を指摘し、コア資産が実際に再利用される製品とフィーチャ、クラスの対応関係に着目した想定モデルとその構築方法を提案した [5]。彼らの示した想定モデルとは、コア資産がどのような利用のされ方をするのかについて、明示的に示すものであった。また彼らは、コア資産を再利用するプロダクト群、それに含まれるプロダクトから導出される設計モデルと段階的に導出し、更に設計導出が可能な範囲で制約を付与することで、想定モデルを構築する手法を提案した。

彼らの提案する想定モデルとその構築方法を用いることで、コア資産の設計ならびその検証をする際に考えるべき想定の範囲を明示的に認識する効果や、現実的な検証量とするためにコア資産の利用方法の検討する材料になることが期待できる。しかし、彼らも指摘しているように提案された想定モデルは、プロダクトラインのモデルから導出されるため、そこに含まれていない情報は直接的に得ることはできない。特に基盤に関する想定は、他の想定以上に暗黙的に扱われることが多く、これに対する対応は彼らにとって課題であった。

### 2.4.2 Trew らの設計上の意図をドキュメント化する取り組み

Trew らは、ソフトウェアプロダクトライン開発において遭遇した不具合に関して原因分析を行った結果、コア資産の結合不具合は再利用コンポーネントのテスト方法に問題があったのではなく、アーキテクチャや設計に原因があったとの観測を示した [2]。彼らは、この分析結果から非機能要求に影響を与える設計上の意図を経験に基づき整理し、その設計意図を守るために、設計を行う上でのルール（ポリシーの集合）を自然言語で記述したドキュメントを作成した。これは、再利用コンポーネントを設計・開発する際、開発者にドキュメント化されたルールを順守させることで、結合不具合の原因を減少させることを目的とした取り組みであった。

この取り組みは、コア資産の開発者たちでそれぞれ異なっていた設計ポリシーを、一貫したルールとして明示化した点に大きな意義があった。しかし、この順守されるべきルールはあくまで自然言語で記述されたものであり、そこに含まれる曖昧性を完全に除去することは困難であると考えられる。ゆえに、この提案のままでは結果的にルール策定者とコア資産開発者との間で誤解が生じ、結合不具合につながってしまう可能性があると言える。

# 第3章　コア資産の設計検証手法

## 3.1　提案

2.3 で指摘したように、再利用性の高いコア資産の開発を行うためには、設計時にそれが対応可能なアーキテクチャ上の想定の広がりが明示化され、その広がりの範囲内で十分に検証がなされていることが重要である。そこで本研究では、コア資産の開発において想定すべき内容をテンプレートとして予めモデル化し、コア資産が再利用されるプロダクトラインアーキテクチャにおける想定をそのテンプレートに基づきモデル化・検証を行う手法を提案する（図 3.1）。

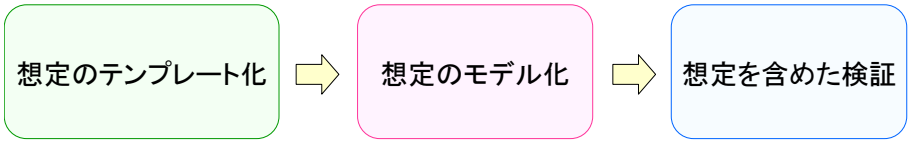


図 3.1: コア資産の設計検証手法の全体像

### 3.1.1　想定のテンプレート化

コア資産の持つアーキテクチャ上の想定を明示化するためには、考慮すべき想定が明示化されている必要がある。つまり、システム外部に対する想定を明示化するにはシステムが利用される環境が、システム内部に対する想定を明示化するには処理を行う際の内部状況が、システム基盤に対する想定を明示化するためには基盤が提供するメカニズムが予めモデル化されている必要がある。

また、製品の属するドメインごとに考慮すべきアーキテクチャ上の想定には類似性があり再利用可能であると考えた。例えば、電子ポットのようにヒーターを制御することで水温を一定に保つような製品ドメインを考えた場合、そのドメインに属する製品は同じ製品系列かどうかに関わらず、システム外部に対しては操作ボタンの入力順序やヒーターの状態変化など、システム内部に対しては UI やヒーターを制御するタスクの処理や相互作用など、システム基盤に対してはソフトウェア開発となっている RTOS やハードウェア構成などのアーキテクチャ上の想定について共通して考慮する必要がある。しかし、全て共通かというとそうとは限らず、RTOS が提供可能な機構など詳細な項目については製品群により異なる可能性がある上、特定の RTOS を使用できるか否かの決定には CPU の性能やメモリの容量に依存する可能性もある。よって、考慮すべきアーキテクチャ上の想定をモデル化するためには、そのような共通性や可変性、依存関係を記述可能であることが必要だと考える。

そこで、本研究では要素の選択に関して必須や排他的・非排他的選択、依存関係をモデルの記法として提供しているフィーチャモデルの表記法 [6] を応用して、考慮すべきアーキテクチャ上の想定をテンプレートとしてモデル化する方法を提案する。

### 3.1.2　想定のモデル化

2.3 で指摘したように、一般にコア資産が再利用されるプロダクトラインアーキテクチャにおいて、想定が明示化されているとは限らない。そこで、本研究では 3.1.1 でモデル化した想定のテンプレートを用いて、プロダクトラインアーキテクチャ上でコア資産の持つ想定を明示的に意識することで抽出し、フィーチャモデルの記法を応用したモデル化手法を提案する。

この際、アーキテクチャ上の想定は必ずしもフィーチャモデルのように選択・代替の表現のみを用いて記述できるとは限らない。例えば、タスクなどの優先度やその起動順序を想定として扱いたいとした時、選択や代替ではそのような想定を表現することは困難である。そこで本研究では、システムが正常に動作するために守られるべき想定を制約という形でモデル化することを提案する。

### 3.1.3　想定を含めた検証

ソフトウェアプロダクトライン開発において、コア資産に不具合が含まれると開発の効率は著しく低下する。ゆえに、コア資産が想定する範囲において、十分に検証がなされることが重要であると考えられる。そこで、本研究では想定を含めた検証において以下の提案を行う。

- モデル上での想定範囲の絞り込み：

  アーキテクチャ上の想定に関して検証を行う際、広い範囲で想定が定義されていた場合においてその想定範囲をすべて検証するのは現実的ではない。よって、検証を行うためには現実的に検証可能な範囲まで想定範囲を絞り込む必要がある。そこで、本研究では検証を行うためにモデル上で想定範囲を絞り込み、検証した想定範囲を示すモデルとして示す方法を提案する。

- モデル検査技術の利用による想定範囲内の網羅的な検証：

  コア資産は想定される範囲を明示する必要があり、同時に検証を行った範囲を明示しておく必要がある。検証を行った範囲が明示されてなければ、再利用する際に現在開発中のシステムにおいて再利用可能かどうかの判断ができないからである。また、検証範囲を行った範囲を明示したならば必ずその想定内では明示された想定が満たされていることを検証すべきである。一般に、設計の検証手法として行われることの多いレビューでは、あらゆる想定下での検証をしようとするとコストがかかりすぎると共に考慮漏れが発生する可能性があり、それのみに頼り検証を行うことは現実的ではない。よって、本研究では先に提案した想定の制約方法とモデル化された範囲において網羅的な検証可能形式が可能であるモデル検査技術を利用し、明示した想定範囲内での検証方法を提案する。

## 3.2　アーキテクチャ上の想定のモデル化と検証の流れ

提案における具体的なモデル変換の流れについて、概念図（図 3.2）に基づき述べる。

本研究では、コア資産の持つアーキテクチャ上の想定をモデル化の基礎として、想定モデルテンプレートを用いる。想定モデルテンプレートは、開発が行われているドメインにおいてコア資産が考慮すべきアーキテクチャ上の想定がフィーチャモデルの記法を応用した形でモデル化したものである。一般に、プロダクトラインアーキテクチャ（可変性を含んだ設計モデルに相当）において想定は必ずしも明示的に記述されているとは限らない、そこで想定モデルテンプレートに基づきプロダクトラインアーキテクチャ上の想定を設計想定モデルとして明示化する。つまり設計想定モデルは、テンプレート上で示された考慮すべきアーキテクチャ上の想定を、実際のプロダクトラインアーキテクチャ上の想定で具体化する作業であると言え、

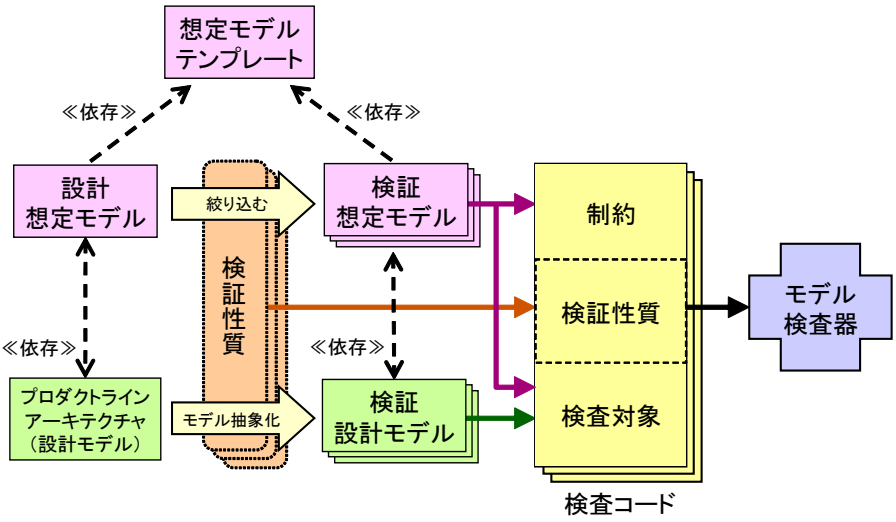


図 3.2: 提案の概念図

そこには明示化の有無に関わらずプロダクトラインアーキテクチャ上で考慮すべき想定のインスタンスがすべてモデル化されていることをねらっている。ゆえに、これら設計想定モデルとプロダクトラインアーキテクチャは対として扱われ、コア資産の設計者・利用者間で設計意図を共有するためのドキュメントとしての役割を果たすことをねらっているといえる。

また、コア資産を再利用した開発を効率的に進めるために、コア資産は想定された範囲内で確実に検証が行われている必要がある。そこで、プロダクトラインアーキテクチャ上のあらゆる想定がモデル化された設計想定モデルから検証性質（設計において満足されていることを確認したい性質）に関係のある想定のみに絞り込むことで考慮すべき想定の範囲を明示化し、その想定の範囲内で検証を行う。この時、設計想定モデルから検証性質により絞り込んだ考慮すべき想定を検証想定モデルという形で明示化する。つまりこの検証想定モデルには、コア資産の持つプロダクトラインアーキテクチャ上での想定のうち検証性質に関係のあるもののみがモデル化された形になっているといえる。よって、この検証想定モデルに基づき設計モデルであるプロダクトラインアーキテクチャから、検証に関係のある設計モデルのみに抽象化したものを検証設計モデルとして明示化する。

本研究では、想定の範囲内で性質の検証を行うためにモデル検査を用いる。そこで、モデル検査器への入力となる検査コードを作成するために、検証したい性質と先に明示化した検査想定モデル・検証設計モデルを用いる。なお、検証設計モデルからは検証対象が、検証想定モデルからは設計では明示化されていなかった検証対象のプロパティや制約が導出される。

# 第4章　想定のテンプレート化

## 4.1　概要

2.2 において、アーキテクチャ上にはシステムの外部・内部・基盤に対する想定が存在することを紹介した。よって、コア資産の持つ想定を明確にする上で、考慮すべきアーキテクチャ上の想定を予めモデル化しておく必要がある。これにはコア資産が含め開発の対象となっているシステムの外部・内部・基盤に対する想定といった観点におけるアーキテクチャ上の想定が含まれる。しかし 2.3 でも指摘した通り、これらシステムの外部・内部・基盤に対する想定は必ずしも明示的に記述されているとは限らず、一般にプロダクトラインアーキテクチャなどの設計モデル上からは得がたいものである。

またこれらアーキテクチャ上の想定には、明確に分類することは困難であるが、開発の目的やドメイン、製品によって重要な想定が存在し、それらは個々の開発に依存せず共通して考慮すべき想定が存在すると考える。

そこで本研究では、これらシステムの外部・内部・基盤に対する想定に対し、標準化された分類を適用しモデル化する（図 4.1）。さらに、その分類の構造に対してフィーチャモデルの表記法を応用し、考慮すべきアーキテクチャ上の想定を階層的なグラフ構造を持ったモデル（想定モデルテンプレート）を作成する。具体的には、フィーチャモデルの表記法のうち必須、選択、代替、依存関係の表記を用いる。

なお、標準化された分類を用いることで、異なる製品系列で開発されたコア資産を再利用しようと考えた場合も、共通の分類のもとで想定を比較することで再利用の可否の判断が可能になると考えた。

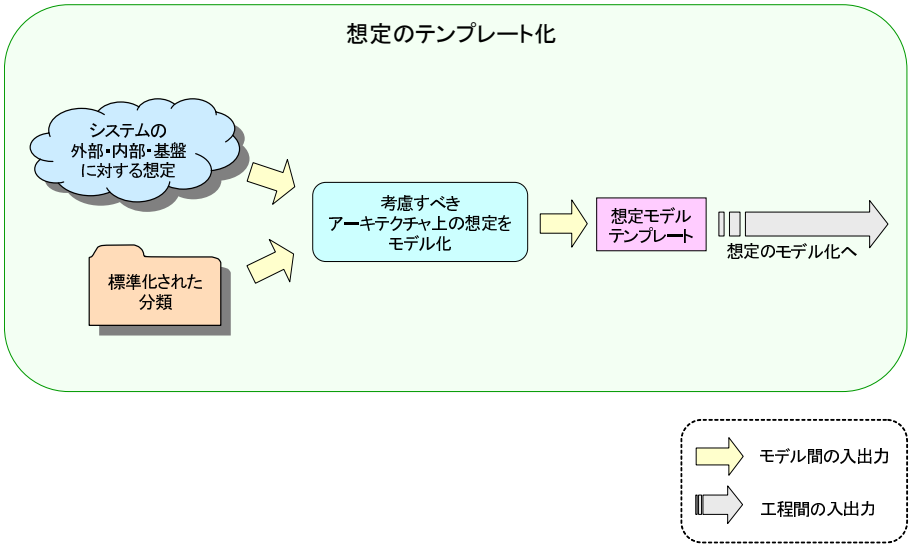


図 4.1: 想定のテンプレート化の流れ

## 4.2　想定モデルテンプレートの構築

想定モデルテンプレートの具体的な構築方法の一例としてシステム基盤が提供するメカニズムに着目し、そのモデル化の方法を示す（図 4.2）。

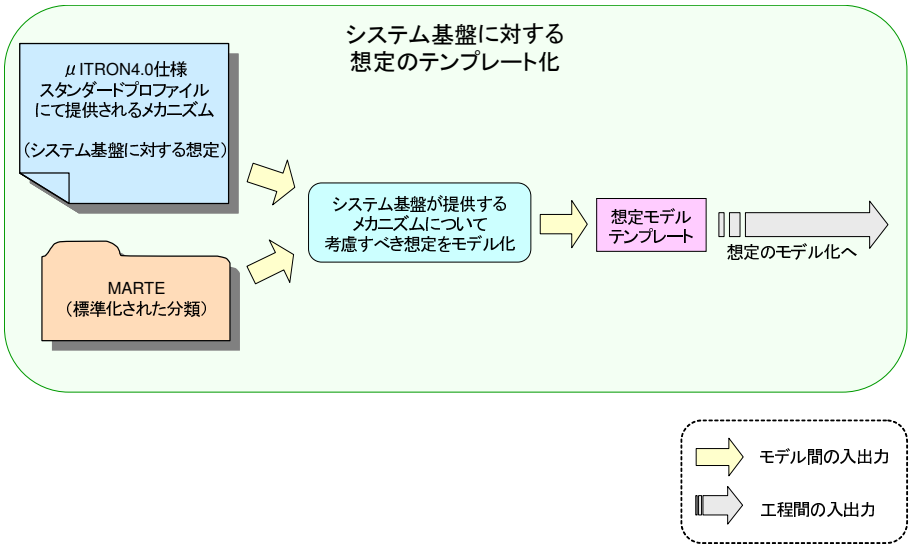


図 4.2: システム基盤に対する想定のテンプレート化

### 4.2.1　モデル化の流れ

想定モデルテンプレートにおいて最上位に位置するのが、フィーチャモデルの表記法を利用したグラフ構造のルートにあたる部分で、フィーチャモデルの表記法で定義されている概念に相当する。つまり想定モデルテンプレートにおいて、システム基盤が提供するメカニズムに対する想定をテンプレートとして作成するために、システムの仕様としてシステム基盤の仕様を、想定の分類としてシステム基盤の標準的な分類を用いる。ここで示すシステム基盤が提供するメカニズムとは主に OS が管理・提供するメカニズムを意図しており、それら管理の詳細はその仕様にて定義されると考えた。ここでは、システム基盤の具体例として $\mu$ ITRON4.0 仕様 (Ver.4.03.00) スタンダードプロファイル（以下、$\mu$ ITRON と呼称）[7] を考える。またその標準化された分類として、OMG (Object Management Group) により、組込み・リアルタイム向け UML プロファイルとして標準化が進められている MARTE (UML Profile for Modeling and Analysis of Real-time & Embedded System)[8] を参照する。

そこで、MARTE にて定義されているパッケージやステレオタイプ、タグ付き値の分類に基づき、$\mu$ ITRON にて提供されているメカニズムに対する想定を、フィーチャモデルの表記法を応用してモデル化したものを想定のテンプレートとして作成する。

### 4.2.2　システム基盤にて提供されるメカニズム

本研究ではコア資産の設計・検証を対象としている。よって、$\mu$ ITRON が提供するメカニズムのうちコア資産の振る舞いに影響を与えるアーキテクチャ上の想定をテンプレート化する（テンプレート化したメカ

ニズム一覧は、付録 A 参照)。

### 4.2.3 標準化された分類

$\mu$ ITRON にて提供されているメカニズムを分類するために、MARTE で定義されている GRM (Generic Resource Modeling) と SRM (Software Resource Modeling) の 2 つのプロファイルを用いる。GRM はリアルタイム組込みシステムを実行するための一般的なプラットフォームの概念を提供し、SRM はマルチタスクでの実行を支援するソフトウェアのアプリケーションインタフェース（API）を記述することを意図して策定されている。つまり、GRM では並行実行のためのメカニズム（スケジューラなど）を提供する基盤に対して、SRM ではメカニ想定モデルテンプレートにおいて最上位に位置するのが、フィーチャモデルの表記法を利用したグラフ構造のルートにあたる部分で、フィーチャモデルの表記法で定義されている概念に相当する。つまり想定モデルテンプレートにおいて、メカニズムの分類やそれを利用するための API に対して付与するステレオタイプやタグ付き値が定義されている。

### 4.2.4 想定モデルテンプレートの階層構造

想定モデルテンプレートはフィーチャモデルの表記に加え、階層構造を持たせたものである。そこで、図 4.3 に想定モデルテンプレートの階層構造を示す。

なお、図 4.3 中の、(M) や (I) はどこで定義された分類かを示す意図で付与したもので (M) は MARTE を、(I) は $\mu$ ITRON で定義された分類を示している。

想定モデルテンプレートにおける各層の説明は以下通りである。

- 概念 : フィーチャモデルの表記法における概念が登録される
- メカニズムの分類 : MARTE にてメカニズムを特徴付けるために定義されたステレオタイプの分類が登録される
- メカニズムの名称 : $\mu$ ITRON にて定義されているメカニズムの名称が登録される
- メカニズムの設定や API の分類 : MARTE にてメカニズムを特徴付けるステレオタイプが登録される
- メカニズムの設定や API の種類 : MARTE にてメカニズムを特徴付けるステレオタイプにて、メカニズムの設定や API を定義するためのタグが登録される
- メカニズムの設定値や API のインスタンス : MARTE にて定義されたメカニズムの設定や API に相当した、$\mu$ ITRON の設定や API が登録される
- メカニズムの設定値や API のインスタンスに対する制約 : フィーチャの選択に対する制約が登録される

### 4.2.5 想定モデルテンプレートと MARTE の対応

$\mu$ ITRON で提供されているメカニズム（メカニズムの種類や設定、提供する API）と MARTE で定義された分類（プロファイル、ステレオタイプ、タグ付き値）の対応関係の例として、$\mu$ ITRON メカニズムのひとつであるセマフォの想定モデルテンプレートの構築例を図 4.4 に示す。尚、図 4.4 において、左側が MARTE のプロファイル定義を表しており、右側がメカニズム（セマフォ）の想定モデルテンプレートである。

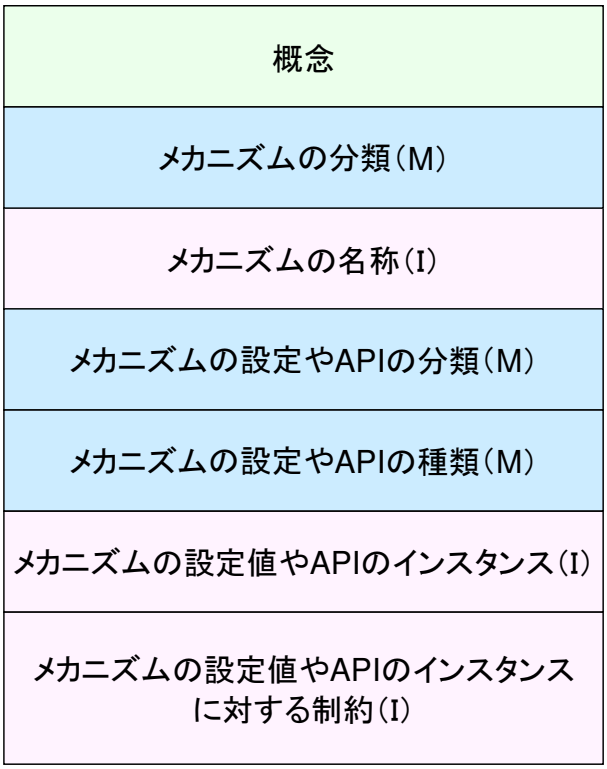


図 4.3: 想定モデルテンプレートの階層構造

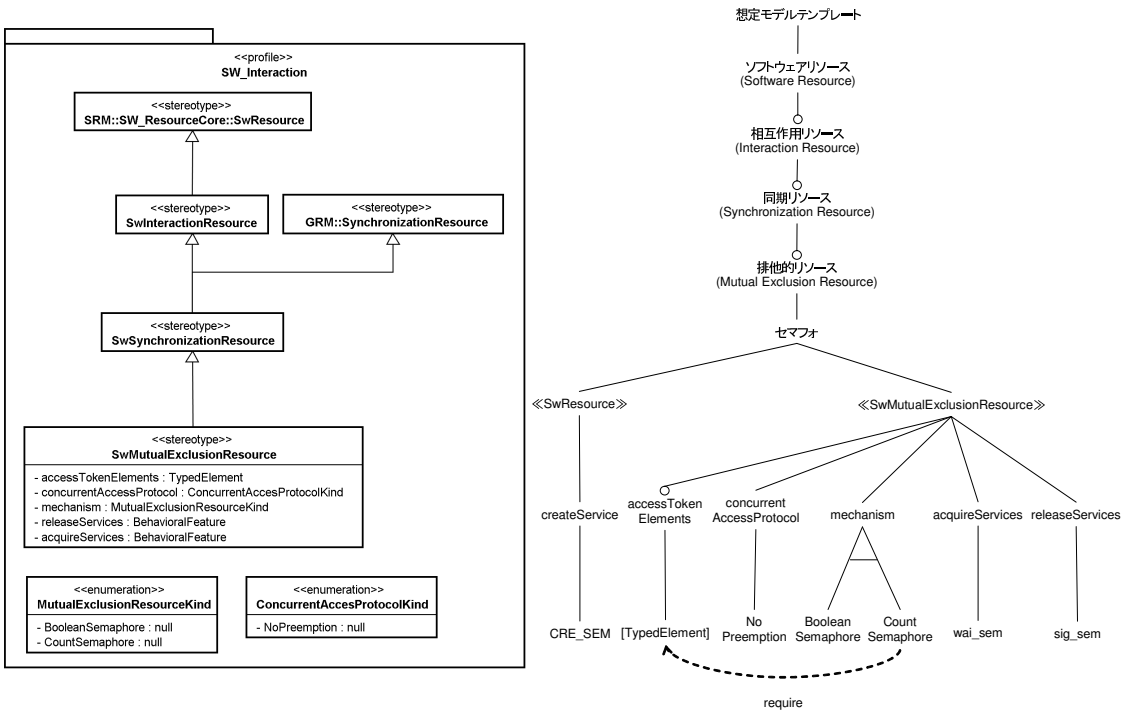


図 4.4: 想定モデルテンプレートの構築例

### フィーチャの階層構造とステレオタイプの継承関係の対応

$\mu$ ITRON の提供するメカニズムであるセマフォの持つ想定の明示化を行うために、MARTE におけるステレオタイプ定義の継承関係を用いる（図 4.5）。MARTE においてセマフォを特徴付けるステレオタイプは SwMutualExclusionResource であり、これは SRM のサブプロファイルである SW_Interaction にて定義される。SW_Interaction プロファイルの定義は、同じく SRM のサブプロファイルである SW_ResourceCore にて定義された SwResource をインポートすることが元となる。そこで、本研究で提案する想定モデルテンプレートという概念の子として、ソフトウェアリソース（Software Resource）というメカニズムの分類が定義される（図 4.5 中 1-A 参照）。

また MARTE において、SwResource を継承する形で SwInteractionResource が定義される。これに対応した形で想定モデルテンプレートにおいても、ソフトウェアリソースの子として相互作用リソース（Interaction Resource）がメカニズムの分類が登録される（図 4.5 中 1-B 参照）。以降、SwSynchronizationResource と同期リソース（Synchronization Resource）が、SwMutualExclusionResource と排他的リソース（Mutual Exclusion Resource）が MARTE における定義の継承関係と対応して、メカニズムの分類として定義される（図 4.51-C, 1-D 参照）。

先ほど述べたように MARTE においてセマフォを特徴付けるステレオタイプの定義は SwMutualExclusion-Resource である。よって想定モデルテンプレートにおいてセマフォは、排他的リソース（Mutual Exclusion Resource）の子として分類されたことになる。

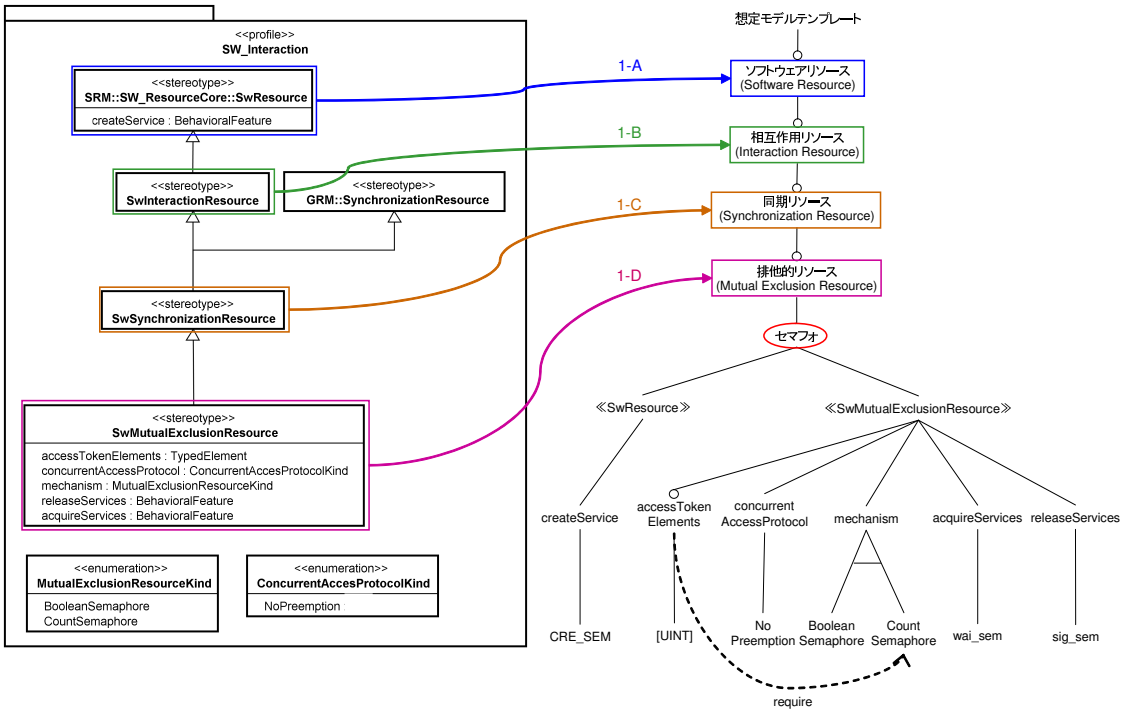


図 4.5: フィーチャの階層構造とステレオタイプの継承関係の対応

### メカニズムの設定や提供する API の分類

MARTE ではステレオタイプと共にタグ付き値が定義される。そこで、本研究ではこのタグ付き値を用いてメカニズムの提供する API を分類すると共にその想定を明示化する。具体的にはメカニズムの設定や生成するための API などが分類や想定の対象となる。

MARTE にてセマフォを特徴付けるステレオタイプである SwMutualExclusionResource が継承するステレオタイプにて定義されるタグ付き値より、$\mu$ ITRON においてセマフォとして提供するサービスに対応するものを抽出し、そのステレオタイプをメカニズムの API やその設定の分類としてセマフォの子とする。具体的には、メカニズムの生成のための API に対応する createService が定義されている≪ SwResource ≫がセマフォの子となり（図 4.6 中 2-A 参照）、同様にメカニズムの設定に対応する concurrentAccessProtocol などが定義されている≪ SwMutualExclusionResource ≫がセマフォの子となる（図 4.6 中 2-B 参照）。

### メカニズムの設定や提供する API の明示化

MARTE において、メカニズムが特徴付けるステレオタイプにて定義されているタグと値を用いて、メカニズムの設定や提供する API の明示化を行う。

図 4.7 中 3-A のように、SwResource ステレオタイプにて定義されている createService タグに対応した形で想定モデルテンプレートにおいても createService という API の分類を≪ SwResource ≫の子として分類する。また図 4.7 中 3-B のように、MARTE において createService タグに付く値として BehavioralFeature（振る舞い特性）と定義されている。これは UML のスーパーストラクチャにて定義されており、一般に API に対応する。よって、$\mu$ ITRON においてセマフォというメカニズムを生成するサービスの API であ

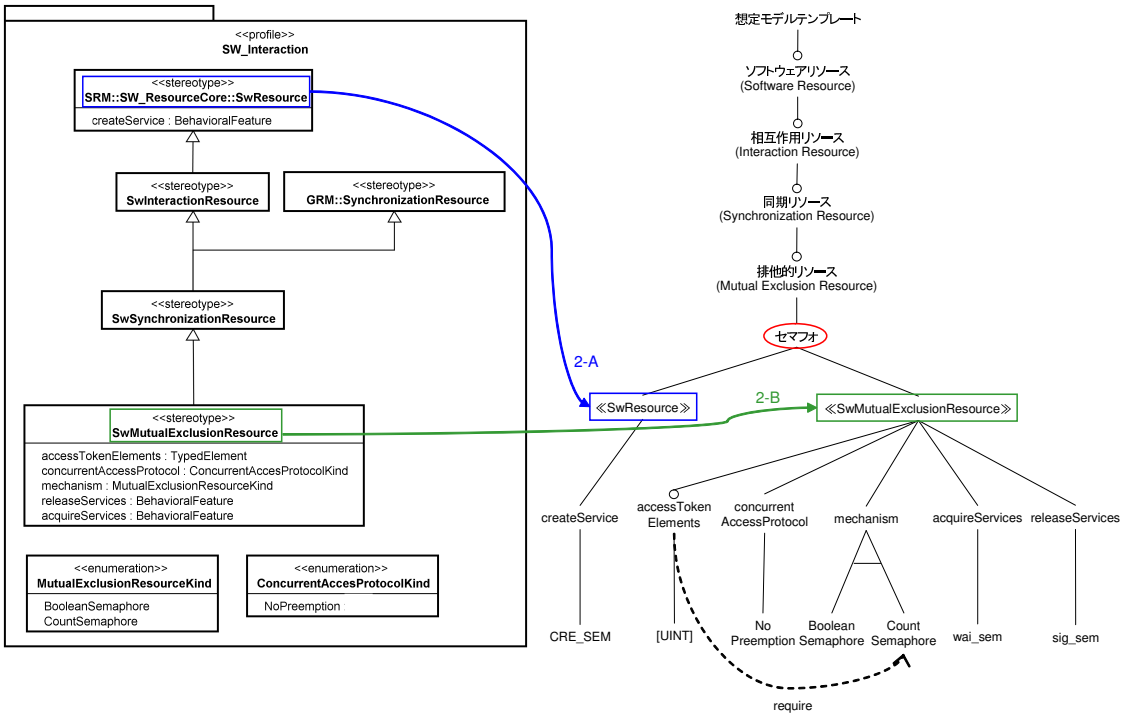


図 4.6: メカニズムの設定や提供する API の分類

る CRE_SEM が BehavioralFeature のインスタンスとなるため、createService の子として想定モデルテンプレートにおいて分類する。

また、図 4.8 中 4-A の SwMutualExclusionResource にて定義されている mechanism のように想定される設定が選択的な場合がある。この選択肢として MARTE では、図 4.8 中 4-B のように enumeration の形で排他的に選択可能な設定が示されている。よって、本研究では図 4.8 中 4-C のように、MARTE において enumeration で定義されている排他的な選択肢を、フィーチャモデルの表記法である代替を用いて表す。つまり、想定モデルテンプレートにおいてはセマフォの仕組み（mechanism）の設定として、BooleanSemaphore か CountSemaphore のいずれかが排他的に選択可能であることを明示化したことになる。

MARTE においてタグとして定義されているメカニズムの設定や提供される API が必ずしも $\mu$ ITRON を基盤とした開発において想定される訳ではない。例えば、図 4.9 中 5-A のように accessTokenElements というタグは、排他的共有資源への同時アクセス可能な数を明示化するために MARTE にて定義されているが、mechanism として BooleanSemaphore が選択された場合には accessTokenElements は必ず 1 となるため定義は不要となる。つまり、accessTokenElements が定義するためには mechanism として CountSemaphore が選択されることが求められるといえる。そこで本研究では、accessTokenElements のように定義されるかどうか不定または選択的なものをフィーチャモデルの表記法における選択で表すとともに、accessTokenElements が選択されるか否かは CountSemaphore が選択されることに要求することから CountSemaphore に対してフィーチャモデルの表記法において要求の依存関係を表す require を付与し選択を制約する（図 4.9 中 5-B 参照）。

なお、この依存関係についてはメカニズム間でも想定モデルテンプレートにおいて定義可能である。具体的には、図 4.10 中 6-A のように、タスクや周期ハンドラはタスクスケジューラに依存していることが表

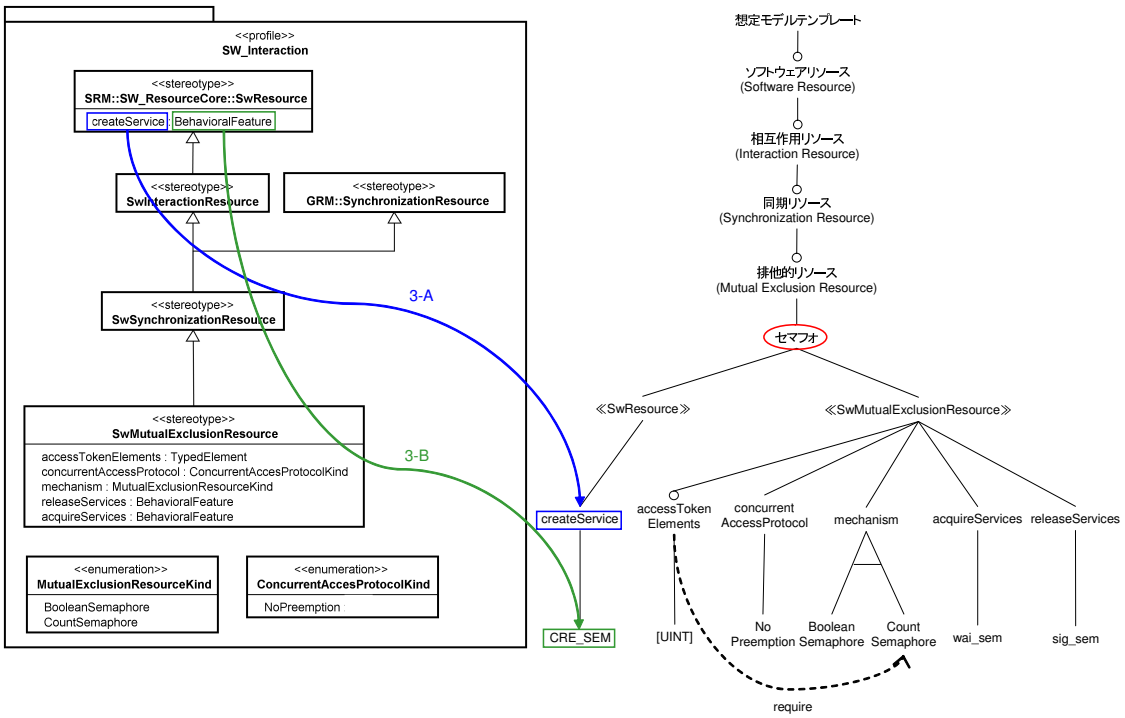


図 4.7: メカニズムの設定や提供する API の明示化

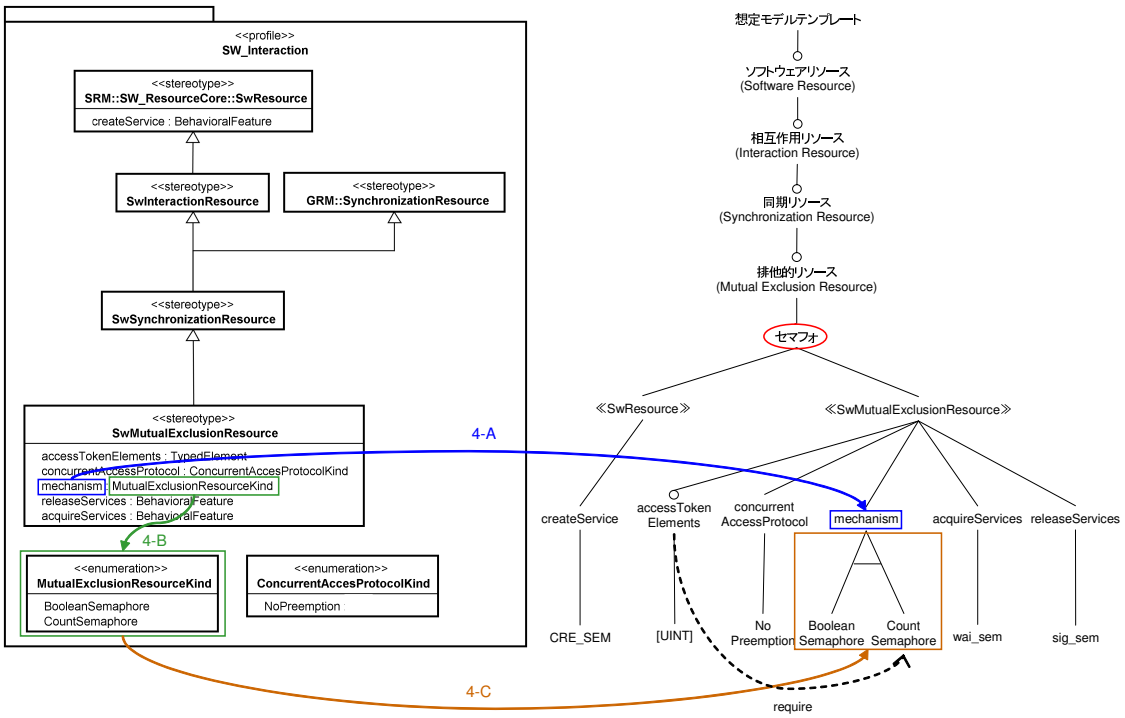


図 4.8: 排他的なメカニズムの設定

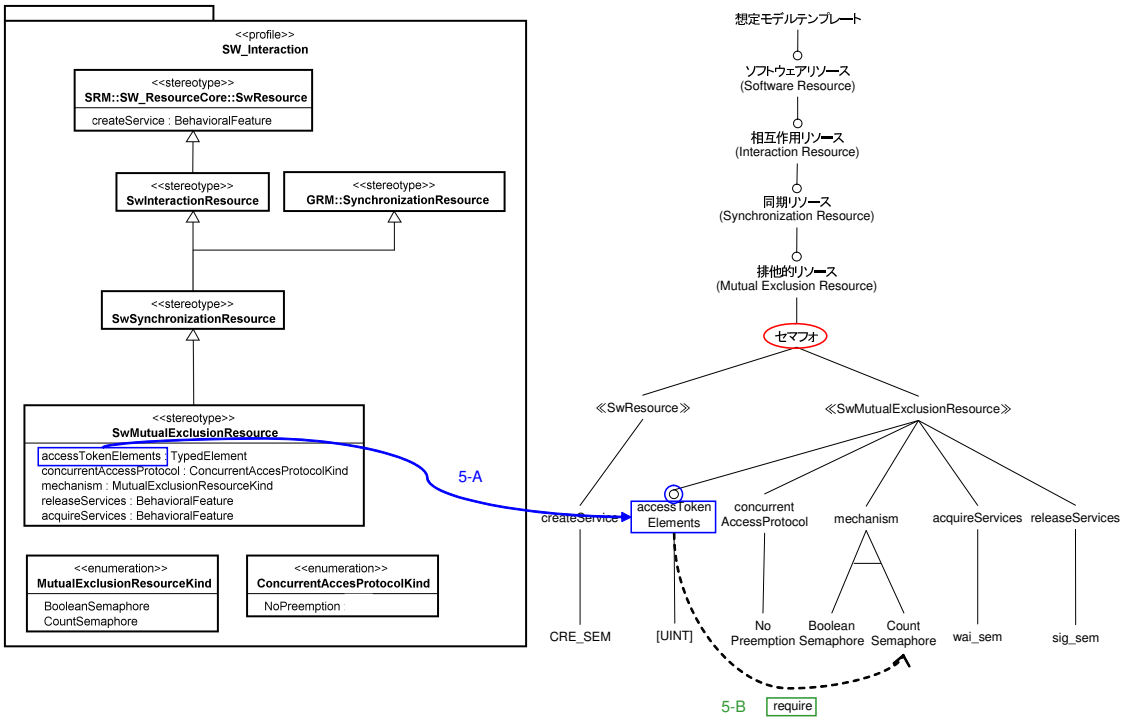


図 4.9: 選択的なメカニズムの設定と依存関係の明示化

現可能である。また、図 4.10 中 6-B のようにメールボックスがメカニズムとして機能するためには固定長メモリプールが定義されていることが必要であるという意図も同様に表現することが可能である。

### 4.2.6 構築された想定モデルテンプレート

$\mu$ ITRON4.0 仕様 (Ver.4.03.00) スタンダードプロファイルにおいて考慮すべき想定をテンプレートとして明示化した、想定モデルテンプレートを図 4.11 に示す。

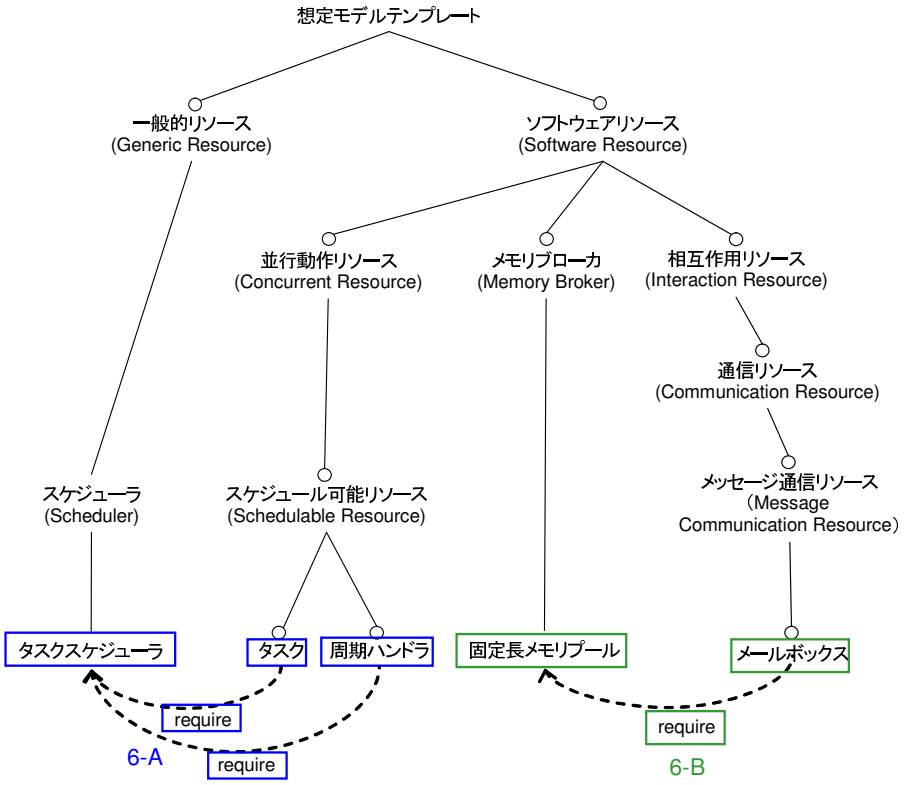


図 4.10: メカニズム間の依存関係の明示化

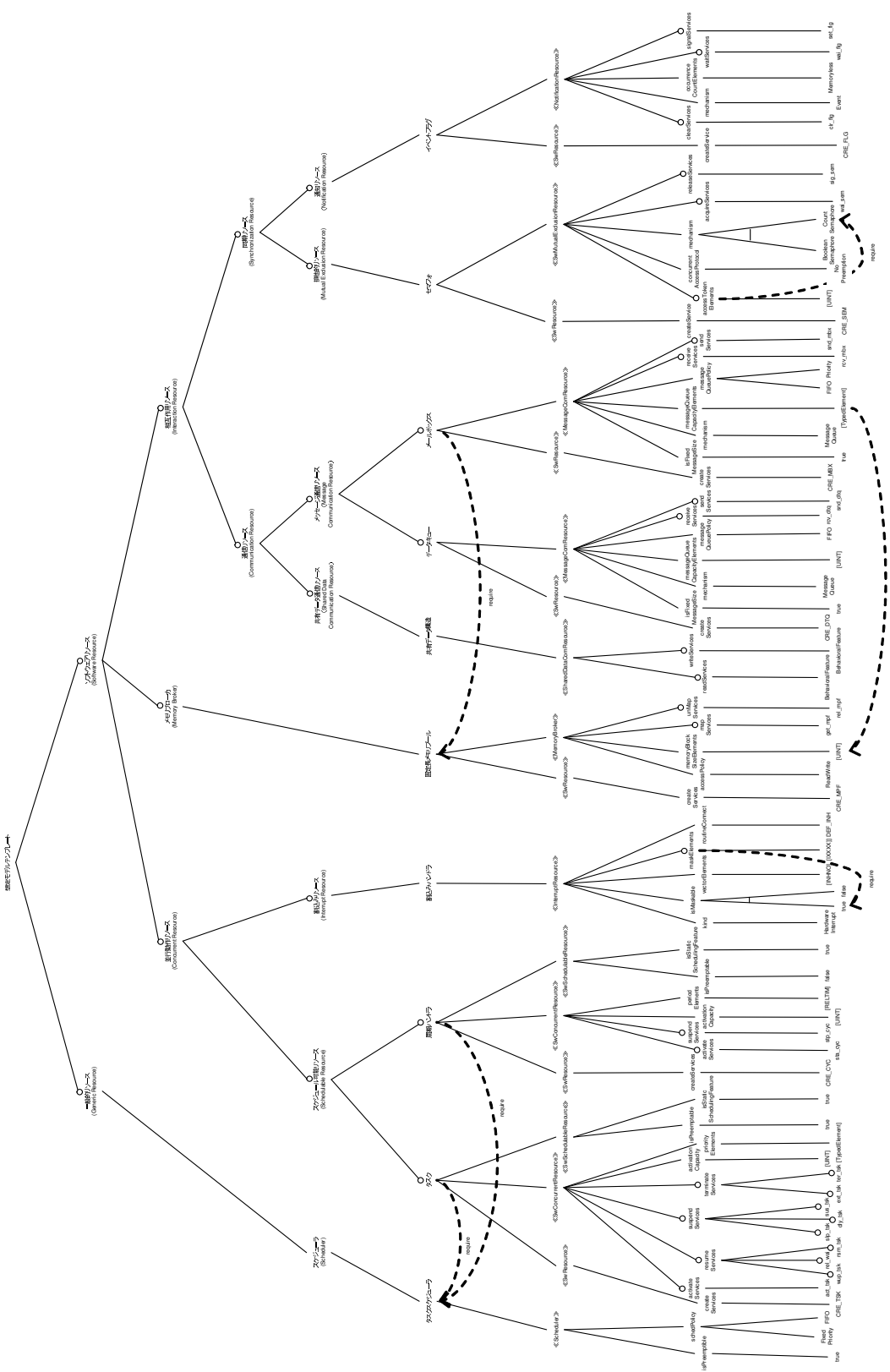

図 4.11: 構築された想定モデルテンプレート

# 第5章　想定のモデル化

## 5.1　概要

コア資産は、製品系列において開発される個々の製品のアーキテクチャを包括するアーキテクチャであるプロダクトラインアーキテクチャおいて再利用されることを前提に開発されることが多い。しかし、プロダクトラインアーキテクチャ設計に対してコア資産の持つアーキテクチャ上の想定は、その設計モデル中に明示化されることは少ない。現実の開発においては、これらコア資産の持つアーキテクチャ上の想定は暗黙的に扱われることが多く、それが多くの場合アーキテクチャ不整合の原因となる。

そこで、本研究では4章にて示した想定モデルテンプレート（考慮すべきアーキテクチャ上の想定を包括したモデル）に基づき、プロダクトラインアーキテクチャ設計上の想定を設計想定モデルとして明示化する手法を提案する（図5.1）。つまり、設計想定モデル上にはコア資産がプロダクトラインアーキテクチャに対して持つ想定が、設計モデルと対になる形で別に明示化されたモデルであるといえる。

なお、ここで提案した設計想定モデルはプロダクトラインアーキテクチャに関して想定を含めた検証を行う際に、その想定を検証する基礎として用いることをねらっている。

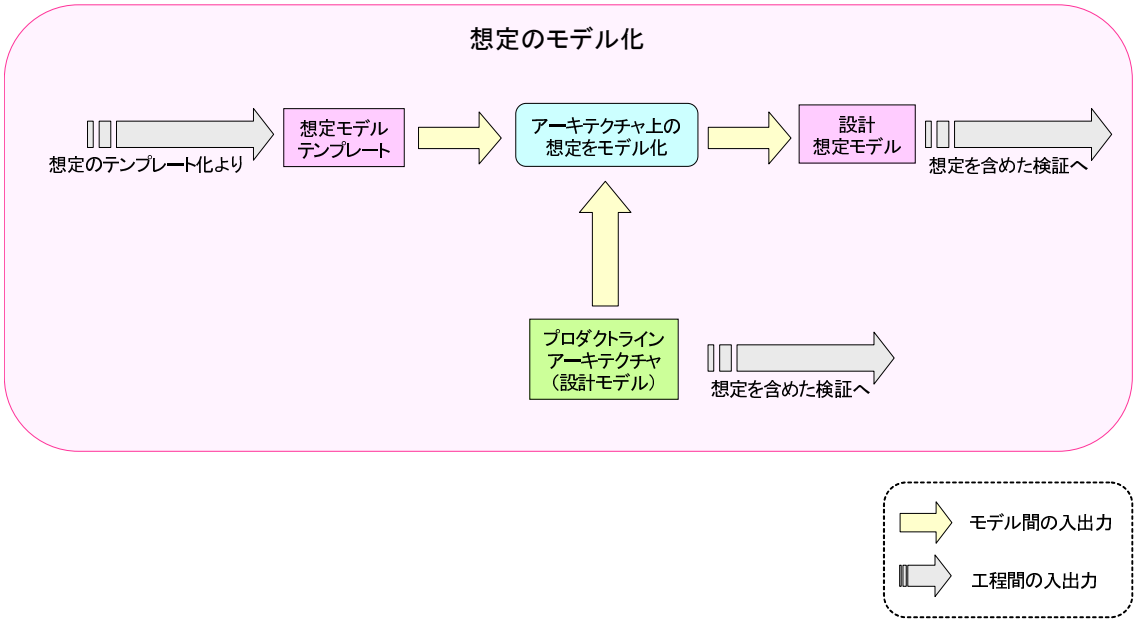


図 5.1: 想定のモデル化の流れ

## 5.2 設計想定モデルの構築

設計想定モデルの具体的な構築方法の一例として、$\mu$ ITRON をシステム基盤として設計されたプロダクトラインアーキテクチャにおいて、タスクの持つアーキテクチャ上の想定に着目し、そのモデル化の方法を示す。

### 5.2.1 例題の説明

図 5.2 に、この例におけるプロダクトラインアーキテクチャに相当する可変性を持った設計モデル（クラス図・ステートマシン図）を示す。この例では、クラス図における可変性（設計におけるクラスの選択）を表ために Gomaa の拡張 [9] を用いた。具体的には、≪ kernel ≫というステレオタイプがアーキテクチャ設計上の必須のクラスを表し、≪ optional ≫というステレオタイプが選択的なクラスであることを表している。また、T1 と T2 の 2 クラスは$\mu$ ITRON 上で動作するタスクであり、≪ kernel ≫ステレオタイプが付与された T1 をコア資産とみなし、T1 が持つアーキテクチャ上の想定をモデル化する。なおこの例では論点を絞るために、T1 と T2 の間の関連には多重度の関係しか設けておらず、特に意味がある関連ではない。

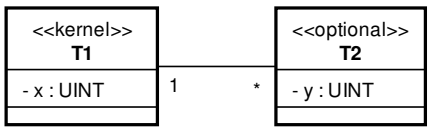


図 5.2: 例題のプロダクトラインアーキテクチャ

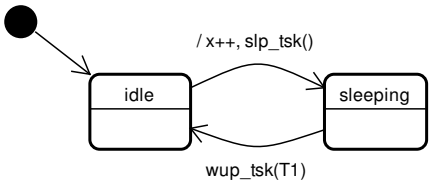


図 5.3: T1 のステートマシン図（プロダクトラインアーキテクチャ）

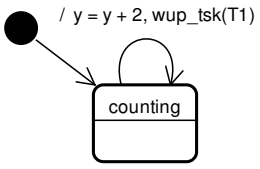


図 5.4: T2 のステートマシン図（プロダクトラインアーキテクチャ）

### 5.2.2 モデル化の流れ

図 5.1 の想定のモデル化の流れに従い、想定モデルテンプレートに基づいたプロダクトラインアーキテクチャ上の想定のモデル化を行う。本例題は$\mu$ ITRON を開発の基盤としている。よって、4.2 にて構築した$\mu$ ITRON の想定モデルテンプレート（図 4.11）を用いて、図 5.2 のプロダクトラインアーキテクチャ上の想定を設計想定モデルとして明示化する。

### 5.2.3　設計想定モデルの階層構造

設計想定モデルの階層構造を図 5.5 に示す。

想定モデルテンプレートに基づいた設計想定モデルの作成は、想定モデルテンプレートにて明示化されたメカニズムをインスタンス化する作業に等しい。よって設計想定モデルを作成する過程で、図 5.5 のように想定モデルテンプレートで定義されたメカニズムの名称の子として、メカニズムのインスタンスが登録される。

なお、設計想定モデルにおける概念は設計想定モデルとなり、設計想定モデルのグラフ構造のルートとなる。


図 5.5: 設計想定モデルの階層構造

### 5.2.4　想定モデルテンプレートの絞り込み

一般に、想定モデルテンプレートは考慮すべきアーキテクチャ上の想定を全て含むことを意図しているため、実際の設計モデルのモデル要素に基づき、テンプレートから設計モデルにおいて考慮すべきアーキテクチャのみに絞り込む。この作業は、プロダクトラインアーキテクチャにおいて利用されるメカニズムを明示的に意識するためのものであるといえる。

具体的に例題を用いて絞り込みの作業を示す。図 5.2 のプロダクトラインアーキテクチャに含まれるクラスは T1 と T2 であり、これらは 5.2.1 にて述べた通りタスクである。タスクは $\mu$ ITRON によって提供されるメカニズムの名称なので、$\mu$ ITRON の想定モデルテンプレート（図 4.11）より、メカニズムの名称がタスクであるフィーチャを発見する。この時、タスクとタスクスケジューラというメカニズムの名称の間に

は要求の依存関係を表す表記があることから、タスクの想定を行うためにはタスクスケジューラについても想定を行う必要があることがわかる。つまり、図 5.2 のプロダクトラインアーキテクチャにおいて、システム基盤に対する想定に限ればタスクとタスクスケジューラという 2 つのメカニズムについてのみ想定を行えばよいことがわかった。そこで、この 2 つのメカニズムの全ての親と子のみを残し、それ以外の全てのフィーチャを除くことで、図 5.6 のような想定モデルテンプレートに絞り込める。

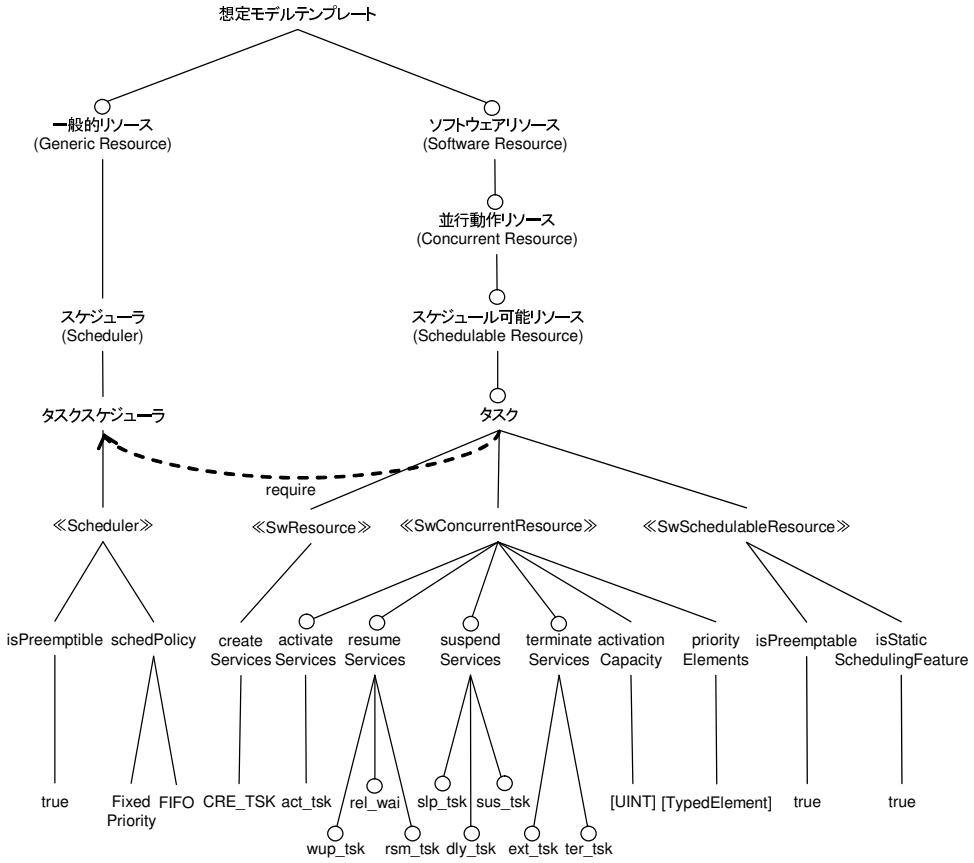


図 5.6: 絞り込まれた想定モデルテンプレート

### 5.2.5 プロダクトラインアーキテクチャ上の基盤に対する想定のモデル化

プロダクトラインアーキテクチャにおいて利用されるメカニズムのみに絞り込まれた想定モデルテンプレート（図 5.6）を用いて、プロダクトラインアーキテクチャ上の基盤に対する想定を明示化する。この作業は、プロダクトラインアーキテクチャにおいて利用されるメカニズムの想定を明示化しているといえる。

図 5.7 に例題において構築された設計想定モデルの例に説明用の注釈を加えたものを示す。以降、図 5.7 の設計想定モデルの構築方法を示す。

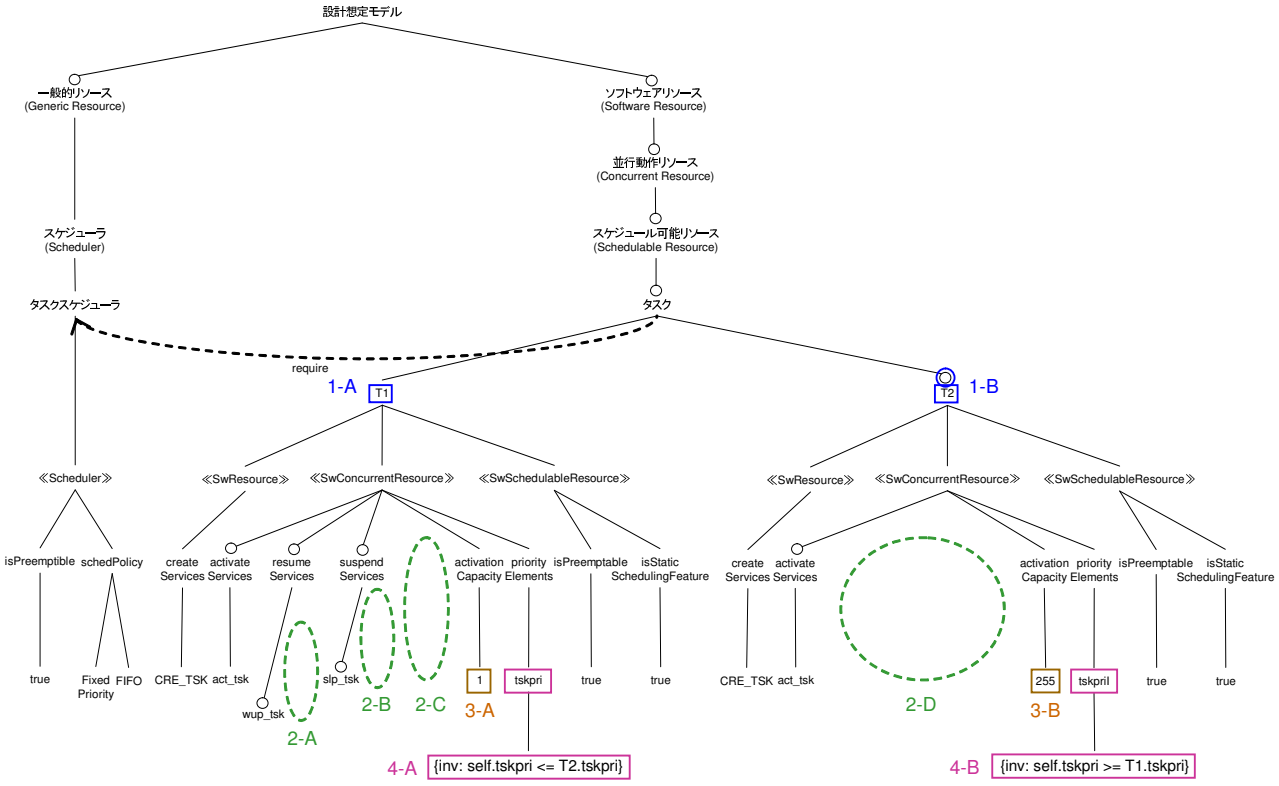


図 5.7: 設計想定モデルの構築例

#### メカニズムのインスタンスの登録

5.2.4 でも述べたように、図 5.2 のプロダクトラインアーキテクチャにおいて利用されている $\mu$ ITRON のメカニズムはタスクとタスクスケジューラであり、具体的なインスタンスとしてはタスクが T1 と T2 であり、タスクスケジューラはそれ自身がインスタンスであると言える。よって、これらを図 5.5 の階層構造に従い、タスクという $\mu$ ITRON にて定義されたメカニズム（の名称）の子として T1 と T2 を登録する（図 5.7 中 1-A、1-B）。なお、T2 に関しては 5.2.1 でも述べた通り、選択的なクラスである。よって、図 5.7 中 1-B のように、選択的なフィーチャとなる。

#### 登録されたメカニズムの提供する API に対する想定の明示化

メカニズムのインスタンスが提供する API に対する想定を明示化する。つまり、5.2.5 にて登録されてたメカニズムのインスタンスが提供する API のうち、プロダクトラインアーキテクチャにて利用されていないものを除去する。

具体的には図 5.7 中 2-A、2-B、2-C、2-D のように、図 5.2 のプロダクトラインアーキテクチャで利用されるメカニズムの API のみを残し、無関係なメカニズムの API の分類やインスタンスを除く。この時、想定モデルテンプレート上の選択的なフィーチャの記述に基づき、絞込みを検討するフィーチャを探し、除去を行うかどうかの判断はプロダクトラインアーキテクチャにて利用されているか否かに基づく。

ここで除かれた API の例を挙げると、T1 の terminateServices に分類される ext_tsk や ter_tsk などで共にプロダクトラインアーキテクチャに含まれるメカニズムにおいて利用されない API である。またこの絞込みの際、子フィーチャを全て除かれた親フィーチャも同時に除かれるものとする。

ただし、タスクの起動を行うサービスは通常ステートマシン図には API の呼び出しとして明示されない。よって、activateServices に分類される act_tsk に関しては、ステートマシン図が記述可能なもの（少なくとも初期遷移以外の遷移が 1 つ以上存在する）は、タスクの起動が行われたとみなし、設計想定モデルに残すこととする。具体的には、図 5.3 や図 5.4 のステートマシン図にて、T1 と T2 のタスクの振る舞いが記述されていることから、タスクの起動を行うサービスが利用されたとみなす。

#### 起動されるタスク数に対する想定の明示化

図 5.2 のプロダクトラインアーキテクチャにおいて、T1 と T2 の多重度の関係は 1 対 N である。つまり、設計モデル上では実際にいくつのタスクが起動されるか決まっていない。しかし、現実には開発基盤となる RTOS、具体的には $\mu$ ITRON 仕様の RTOS の実装において起動可能なタスクの最大数は制限される。よって、ここでは例題において開発基盤となる RTOS の実装では同時起動可能なタスク数の最大値を 256 と決定されているものとし、それタスク起動数に関する想定として明示する。具体的には、SwConcurrentResource ステレオタイプにて定義されている activationCapacity に具体的な想定されるタスクの起動数を指定する。この例の場合、図 5.7 中 3-A、3-B のように T1 が起動される数として 1、T2 が起動される数として 255 が想定されている。

#### タスク優先度に対する想定の明示化

図 5.2 のプロダクトラインアーキテクチャにおいて、優先度に対する想定は明示されていない。しかし、実際の開発において $\mu$ ITRON を使用する場合には優先度を必ず定義しなくてはならず、振る舞いに関しても大きな影響がある設定である。また、優先度は同一のシステム基盤上で動作するタスク間の相対的な上下

関係で定義されるものであり、定量的に明示することは困難である。よって、本研究では優先度を表すモデル要素に対し、各タスクの優先度間の相対的な制約を OCL[10] の記述を用いて与えることで明示化する。

図 5.6 の想定モデルテンプレートにおいてタスクに対する優先度は、SwConcurrentResource ステレオタイプの子である priorityElements として、優先度の想定を持つモデル要素を指定する形（TypedElement）で定義されている。そこで、ここでは tskpri という任意の名称を優先度に与え、OCL の不変条件を表す記述を与えることで制約する。具体的には、5.7 中 4-A のように、T1 をコンテキストとして T1 の優先度を表す tskpri は常に T2 の優先度を表す tskpri 以下であるということを示す。つまり、$\mu$ ITRON では優先度を示す値が小さい方が高優先度であることが規定されているので 5.7 中 4-A の記述では、常に T1 のタスク優先度の方が T2 のタスク優先度以上であることが制約されているといえる。また、5.7 中 4-B に関しても同様に常に T2 のタスク優先度の方が T1 のタスク優先度以下であることが制約されているといえる。

# 第6章　想定を含めた検証

## 6.1　概要

コア資産とプロダクトラインアーキテクチャに基づく製品系列の開発において、コア資産が想定された範囲内で満足すべき性質が確実に検証されていないと、コア資産の再利用による生産性の向上は期待できない。よって、検証を行う際に想定した範囲を明示し、それに基づく検証を確実に行う必要があるといえる。

そこで本研究では、コア資産の想定の対象となるプロダクトラインアーキテクチャとその設計に対する想定モデル（設計想定モデル）に加え、その設計において満たすべき検証性質に基づき、モデル検査技術を用いることで想定した範囲内での検証性質が満足されていることを検証する手法を提案する（6.1）。

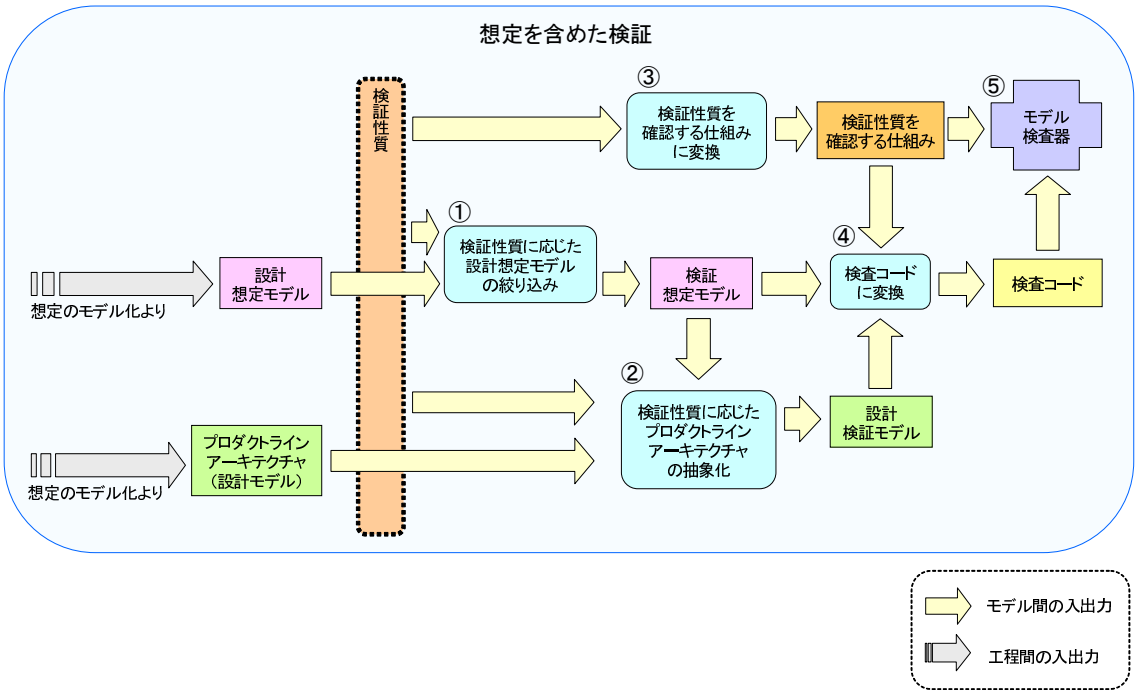


図 6.1: 想定を含めた検証の流れ

## 6.2　検証の流れ

想定を含めた検証は以下の流れで行われる。

1. アーキテクチャ設計上の想定をモデル化した設計想定モデルを検証性質とプロダクトラインアーキテクチャに基づき絞り込み、検証を行う上で想定すべきアーキテクチャ上の想定をモデル化した検証想定モデルを作成する。つまりこの作業は、検証する上で考慮すべき想定の絞り込みに相当する。

2. 設計モデルであるプロダクトラインアーキテクチャを、上記 1 にて作成した検証想定モデルに基づき抽象化し、検証性質を検証する上で必要な構造を含んだ設計モデル（設計検証モデル）を作成する。つまりこの作業は、検証する上で考慮すべき想定の構造や振る舞いの抽象化に相当する。

3. 本手法はモデル検査技術を用いて検証することを前提としている、そのためモデル検査器上で検証性質を確認する仕組みに変換する。

4. 上記 1, 2, 3 にて作成された検証想定モデル、設計検証モデル、検証性質を確認する仕組みに基づきモデル検査器に入力するための検査コードを作成する。

5. 上記 4 にて作成された検査コードをモデル検査器にて検証することで、コア資産が持つプロダクトラインアーキテクチャ上の想定を含めた性質の検証を行う

## 6.3 想定を含めた検証方法

想定を含めた具体的な検証の一例として、5.2.1 で示した $\mu$ ITRON をシステム基盤として設計されたプロダクトラインアーキテクチャにおける、タスクの持つアーキテクチャ上の想定を含めた方法を示す。尚、ここではモデル検査器のひとつである SPIN[11] とシステム基盤となる $\mu$ ITRON 上での振る舞いを検証するために $\mu$ ITRON RTOS 用ライブラリ for PROmela/spiN ( $\mu$ IPRON)[12] を用いた検証を示す。

### 6.3.1 検証性質

意味ある検証を行うためには、検証対象においてどのような性質が満たされるべきなのか明示的に意識する必要がある。

よって、ここでは 5.2.1 で示した例題における検証性質として、「タスク T1 の進行性」を確認することとする。

### 6.3.2 検証想定モデルの構築

検証想定モデルは、設計想定モデル（アーキテクチャ設計上の想定を明示化したモデル）を検証性質に応じて絞り込みことで構築される。

具体的に構築された検証想定モデル（図 6.2）を用いてその構築方法を示す。なお、図 6.2 には説明用の注釈がつけられている。

#### 検証想定モデルの階層構造

設計想定モデルの階層構造と変わらず、図 5.5 の階層構造となる。

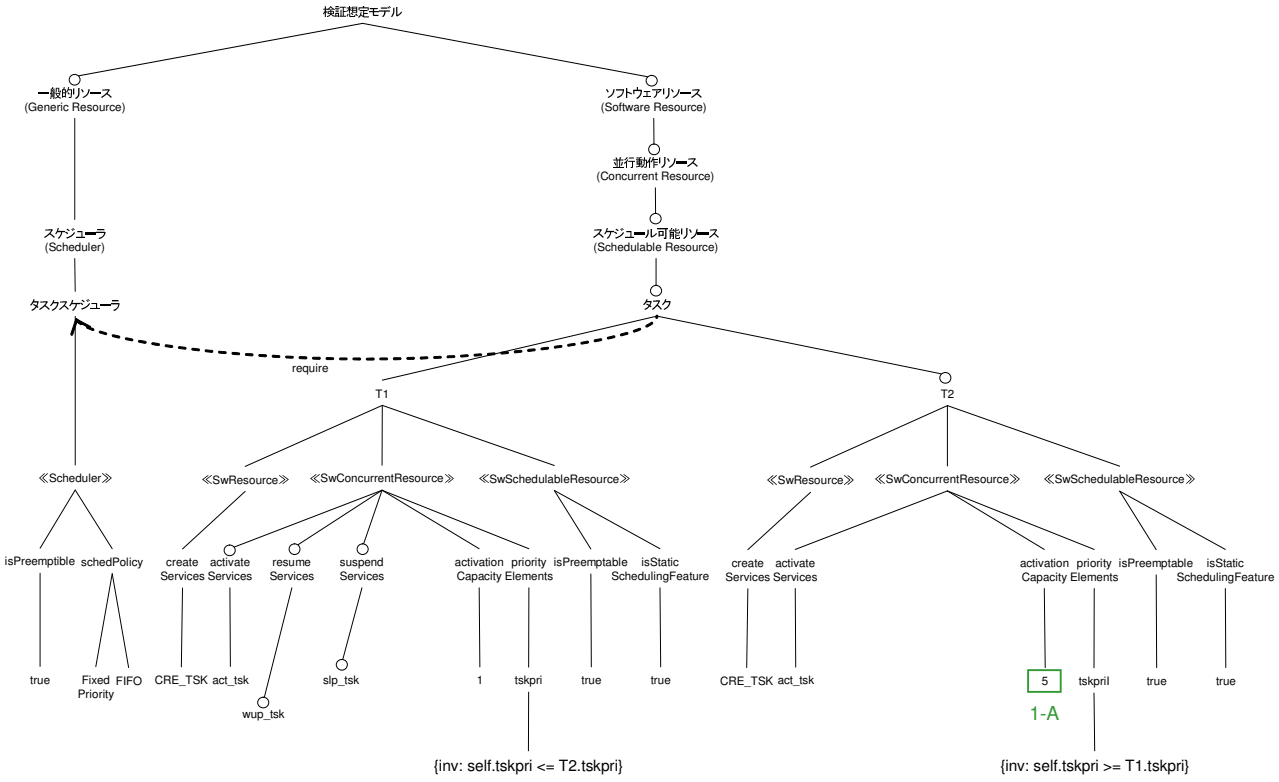


図 6.2: 検証想定モデルの構築例

#### 検証において想定される範囲の絞込み

検証において、現実に利用されることのない範囲を検証しようとするとコストは高くなり、無意味な場合な場合が多いと考えられる。つまり、検証を行う範囲を絞り込みそれを明示化した上で、その範囲内で検証性質が満たされていることを確認することに意味があると考える。よって、本研究では検証を行う上で妥当な範囲に想定を絞り込んだ上で、その範囲内を検証することとする。

具体的には設計想定モデルにおいてはタスク T2 が起動可能な最大数として 255 という数が想定されていた。しかし、実際に開発される製品において起動されるタスクの数が最大で 5 であったとする。この場合、図 6.2 中 1-A のように、起動されるタスクの最大数を表す activationCapacity は 5 というように、想定の範囲を絞り込むことが可能である。

### 6.3.3 設計検証モデルの構築

設計検証モデルは、プロダクトラインアーキテクチャに含まれるメカニズム間の関係や振る舞いを、検証想定モデルで明示化された検証性質に関係のあるアーキテクチャ上の想定と検証性質によって抽象化することで構築される。

図 6.3 に設計検証モデルの構築例を、図 6.4 と図 6.5 に設計検証モデルにおけるタスク T1 とタスク T2 のステートマシン図を示す。以降、ここで例示した設計検証モデルの構築方法を示す。

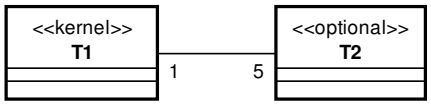


図 6.3: 設計検証モデルの例

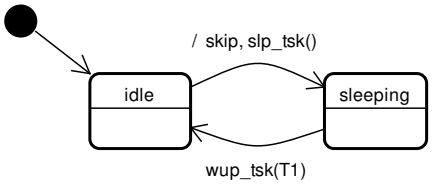


図 6.4: T1 のステートマシン図（設計検証モデル）

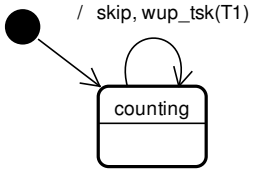


図 6.5: T2 のステートマシン図（設計検証モデル）

#### タスク間の多重度関係の絞り込み

6.3.2 にて起動されるタスク T2 の最大数は 5 であるという想定の下で構築された検証想定モデル（図 6.2）に基づき、プロダクトラインアーキテクチャ（5.2）の多重度を絞り込む。

この絞り込みの過程を改めて説明すると、図 6.3 のように元々設計時はタスク T1 とタスク T2 の多重度の関係は 1 対 N と定義されていたが、$\mu$ ITRON を基盤とする想定モデルテンプレートを構築する際に RTOS の実装上の制約から T1 に対する T2 の多重度は 255 を想定範囲が絞られた。更に、実際にシステムが稼動する状況を検証想定モデルにて想定したところ、タスク T1 はシステムにおいて最大で 1 タスク起動され、

タスク T2 は最大で 5 タスク起動されることが明らかになった。よって、ここではプロダクトラインアーキテクチャにおけるタスク T1 とタスク T2 の多重度の想定を 1 対 5 であると絞り込み、図 6.3 のように多重度の関係を表す。

#### 設計モデルの抽象化

一般に、モデル検査を行う際と同様に、検証性質に基づきプロダクトラインアーキテクチャ（5.2）を構成するタスクの属性ならびに振る舞いを抽象化する。つまり、ここでは検証性質である「タスク T1 の進行性」に影響がない要素は意味を持たない操作に抽象化する。

具体的には、タスク T1 がアクションとして行っている「x++」という操作や、タスク T2 がアクションとして行っている「y=y+2」という操作はタスク T1 の進行性に影響がないことは明らかである、よってモデル検査器 SPIN における意味を持たない操作である「skip」に抽象化する（図 6.4、図 6.5）。また、この抽象化に伴い図 5.2 のプロダクトラインアーキテクチャ上でクラス T1 と T2 の属性としてモデル化されている、「x」や「y」をプロダクトラインアーキテクチャより除く。

### 6.3.4 検証性質を確認する仕組みの検討

モデル検査において、検証性質である「タスク T1 の進行性」を確認するための仕組みを検討する。

本例題において使用するモデル検査器として SPIN を前提としている。よって、ここでは SPIN において進行性を検証するための仕組みである「progress ラベル」を用いることとする。

### 6.3.5 検査コードの作成

検証想定モデル、設計検証モデル、検証性質を確認する仕組みに基づき、検査コードへの変換の方針を示す。ここでは本研究の主眼である、アーキテクチャ上の想定をどのように検査コードに変換するのかという点に絞りその変換方針を示す。

#### 検証性質を確認する仕組みの検査コードへの変換

で検討された確認する仕組みを検査コードに変換する。

具体的には、例題の検証性質である「タスク T1 の進行性」を確認するために、progress ラベルを図 6.4 中の T1 の振る舞いを検査コードに変換した検査コードの skip に対して付与する。

#### タスクの生成数に対する想定の検査コードへの変換方針

本例題において用いるモデル検査器 SPIN では非決定的な代入を行うことができる。そこで、各タスクのインスタンスを生成するためのコードを記述する、もしくは何もしない、という非決定的な代入を行うことにより、タスクの生成数に対する想定の範囲を検査コード上で表現可能である。

T1 タスクが 0 または 1 つ生成される可能性がある場合の具体的な例をリスト 6.1 に示す。

List 6.1: タスク生成に関する想定の検査コードへの変換例

```
1  #define T1 1
2
3  proctype T1() provided (turn == T1) {
4  }
5
6  init {
7    bool T1_cre_tsk_flg = 0;
8
9    /* 非決定的にタスクの生成をするかしないかが選ばれる */
10   if
11     ::cre_tsk(1, T1); T1_cre_tsk_flg = 1;
12     ::else;
13   fi;
14
15   /* フラグが立っている場合のみタスクの起動が行われる */
16   if
17     ::(T1_cre_tsk_flg == 1) -> act_tsk(T1); run T1();
18   fi;
19 }
```

### 優先度に対する想定の検査コードへの変換方針

本研究において、優先度はプロダクトラインアーキテクチャ上で動作するタスクごとに他タスクとの相対的な上下関係を OCL の制約を用いることで想定される。よって、ここではこの優先度に関する想定の変換方法について方針を示す。

本例題において、基盤としている $\mu$ ITRON4.0 スタンダードプロファイルでは少なくとも 1〜16 の 16 段階のタスク優先度を扱うことができる。また、優先度の段階数は生成されるタスクの総数と等しいと言える。よって、ここではモデル検査器 SPIN で実行可能な非決定的な代入を用いる。

以下の方針を検査コード上の初期化プロセスにて実装することで、優先度の想定範囲内での検証が可能である。

1. 各タスクインスタンスの優先度を表す変数を定義する
2. 各タスクインスタンスの優先度に対し、1 からタスク総数まで非決定的に代入を行う
3. 検証想定モデル上で OCL により定義されたタスク間の優先度の制約により、全てのタスクインスタンスが制約を満たす場合のみ、2. で代入された優先度を持つタスクとして生成する

具体例をリスト 6.2 に示す。この例では、タスク総数が 3、T1 タスクの優先度対する制約が 2 以下であるという想定の範囲内での変換例となる。

List 6.2: 優先度に関する想定の検査コードへの変換例

```
1  #define T1 1
2
3  proctype T1() provided (turn == T1) {
4  }
5
6  init {
7    byte T1_tskpri = -1;
8    bool T1_cre_tsk_flg = 0;
9
10   /* タスクの総数分だけ非決定的な代入を行う */
11   /* この例ではタスク総数は3と仮定している */
```

```
12    if
13      ::T1_tskpri = 1;
14      ::T1_tskpri = 2;
15      ::T1_tskpri = 3;
16    fi;
17
18    /* 優先度に関する制約を満たす場合のみタスクを生成し、フラグを立てる */
19    /* この例では T1 の優先度に関する制約は 2 以下であることと仮定している */
20    if
21      ::(T1_tskpri <= 2) -> cre_tsk(T1_tskpri, T1); T1_cre_tsk_flg = 1;
22      ::else;
23    fi;
24
25    /* フラグが立っている場合のみタスクの起動が行われる */
26    if
27      ::(T1_cre_tsk_flg == 1) -> act_tsk(T1); run T1();
28    fi;
29  }
```

### 6.3.6 モデル検査

6.3.5 の方針で変換された検査コードとで検討した検証性質を確認する仕組みを用いてモデル検査を行う。

具体的には、progress ラベルの挿入された検査コードをモデル検査器に入力し進行性の検証を行うことで、想定を含めた検証性質が満たされているか否かを確認することができる。

# 第7章　適用例

## 7.1　例題設定の説明

$\mu$ ITRON をシステム基盤としたソフトウェアプロダクトライン開発に対して、提案手法を適用しその評価を行う。

今回使用する例は、$\mu$ ITRON 上で動作させることを前提に設計されたストップウォッチの製品系列（以降、ストップウォッチ SPL と呼称）である。図 7.1 にストップウォッチ SPL の開発イメージを、図 7.2 にフィーチャモデル、図 7.3 にユースケース図を示す。なお、ユースケース図の拡張は Eriksso[13] による拡張を一部修正したものである。

本例題は、2 つの製品からなる製品系列で、製品 1 は必須の機能である時間計測機能と表示機能として 7 セグ LED 表示機能のみを持ち、製品 2 は必須の機能に加え選択的な機能であるスプリット計時機能とキャラクタ LCD 表示機能を持つ。

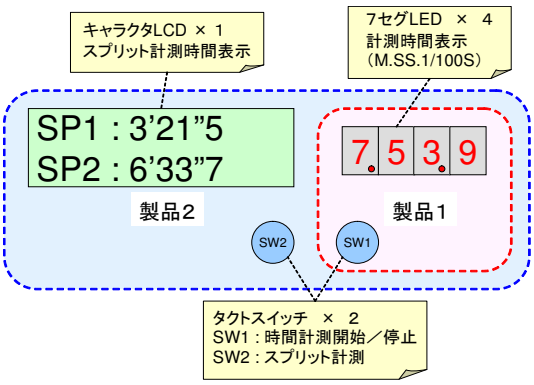


図 7.1: 開発イメージ

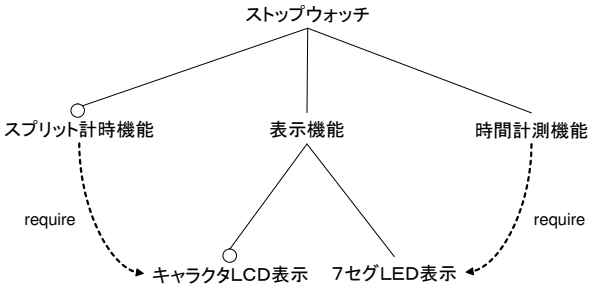


図 7.2: フィーチャモデル

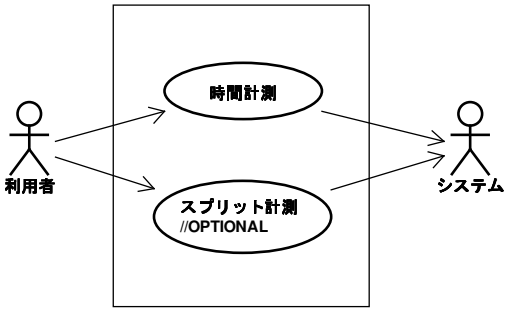


図 7.3: ユースケース図

図 7.4 にストップウォッチ SPL のプロダクトラインアーキテクチャを示す（各クラスの振る舞いは付録 B 参照）。

また各製品、フィーチャ、クラスの関係は以下の通りである。

表 7.1: 製品・フィーチャ・クラスの対応関係

| 製品 | フィーチャ | クラス |
|---|---|---|
| 製品 1 | 時間計測機能<br>7 セグ LED 表示 | EventCtrl<br>SevenSegCtrl<br>Sw1Inh<br>TimeCountCych<br>SystemStatus<br>TimeData<br>SevenSegDataReg<br>EventFlag |
| 製品 2 | 時間計測機能<br>7 セグ LED 表示<br>スプリット計時機能<br>キャラクタ LCD 表示 | EventCtrl<br>SevenSegCtrl<br>SplitTimeListCtrl<br>LcdCtrl<br>Sw1Inh<br>Sw2Inh<br>TimeCountCych<br>MPL_SPLIT_TIME_LIST_CTRL<br>MPL_LCD_CTRL<br>SystemStatus<br>TimeData<br>SevenSegDataReg<br>LcdDataReg<br>MBX_SPLIT_TIME_LIST_CTRL<br>MBX_LCD_CTRL<br>SplitTimeList<br>SplitTimeIndex<br>EventFlag |

## 7.2 想定のモデル化

### 7.2.1 設計想定モデル

ストップウォッチ SPL は $\mu$ ITRON を開発の基盤としている。よって、想定モデルテンプレートとして図 4.11 を用いて、プロダクトラインアーキテクチャ上のシステム基盤に対する想定を設計想定モデルとして明示化する。ここでは、概念からメカニズムのインスタンス階層までの設計想定モデルを示す（図 C.1）

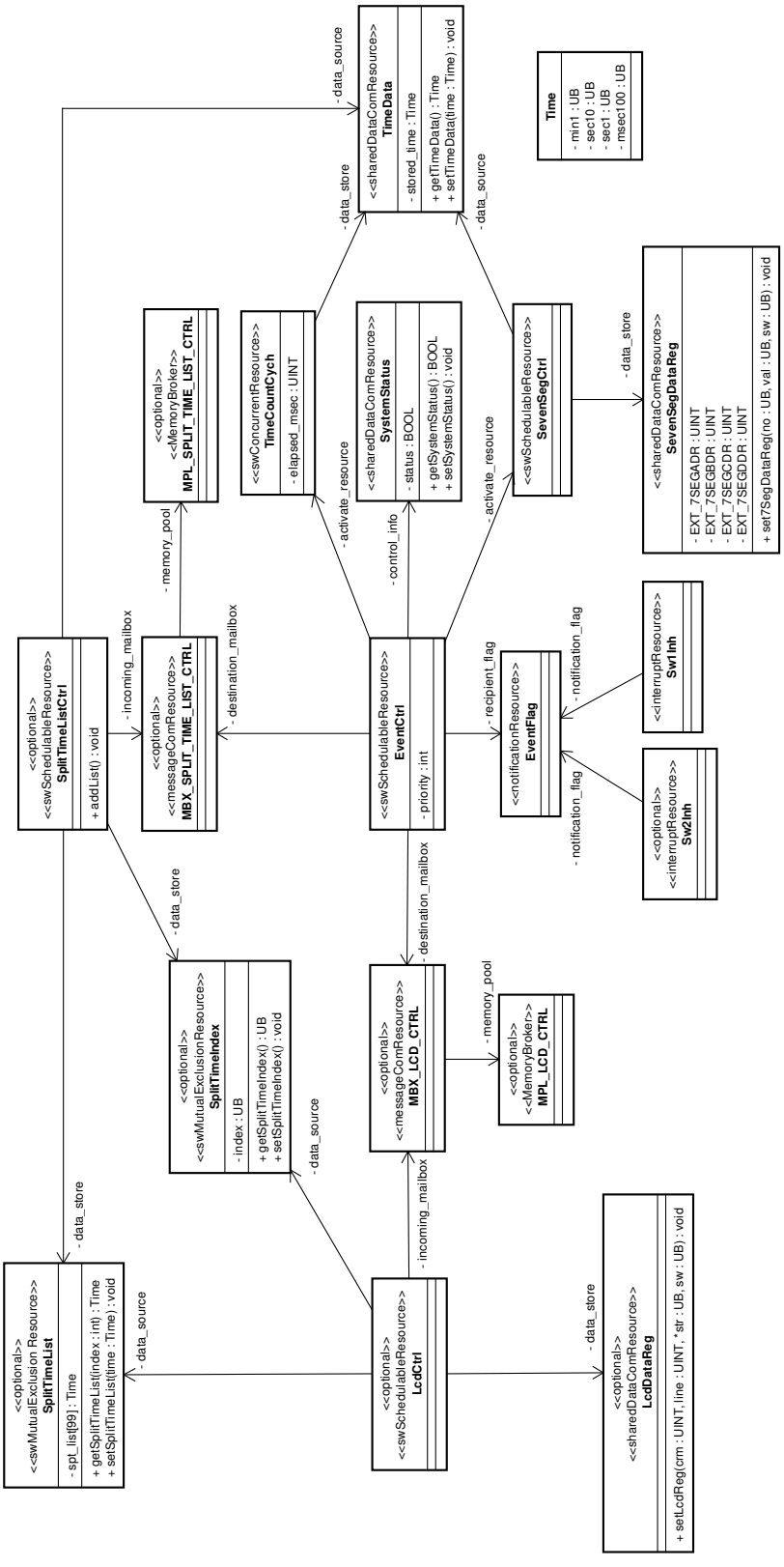


図 7.4: プロダクトラインアーキテクチャ

なお、各メカニズムのインスタンス階層以下の設計想定モデルは、付録 C にてメカニズムのインスタンスをルートとしたフィーチャモデルの記法に基づくグラフ構造で示す。

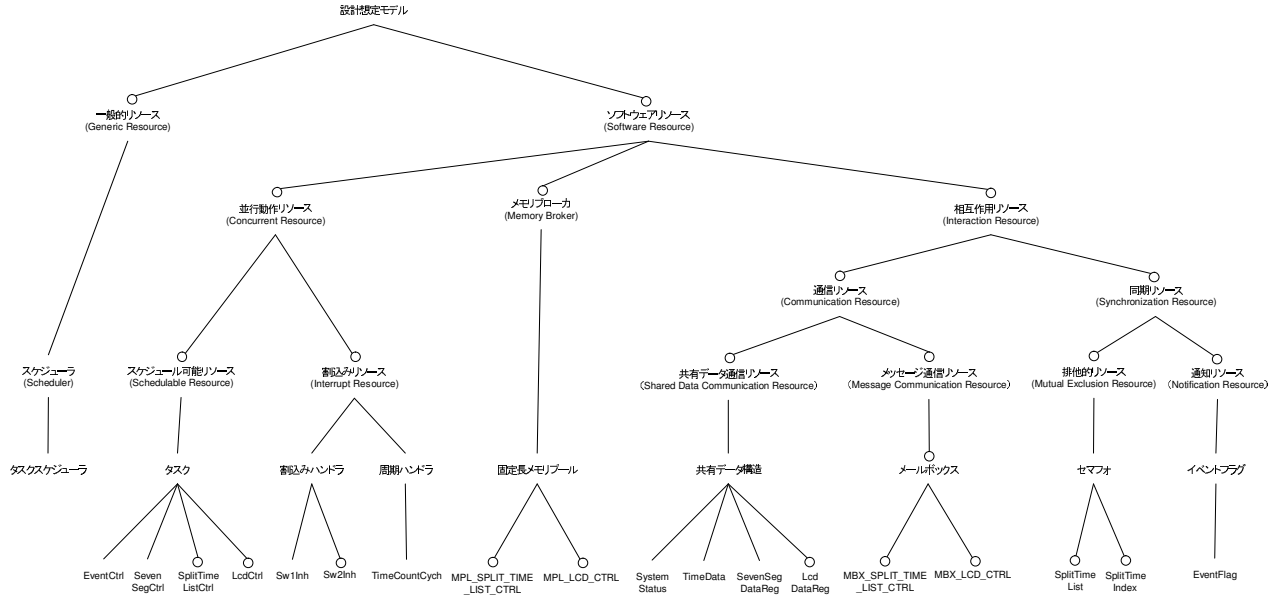


図 7.5: 設計想定モデル（メカニズムのインスタンス階層まで）

## 7.3 想定を含めた検証

製品 2 において、「SW2 が押されたら必ずスプリット計時が実行されて欲しい」という要求があったとする。ここでは、プロダクトラインアーキテクチャ設計がこの要求を満たしていることを確認する。

### 7.3.1 検証性質

このような要求を確認するために、以下の検証性質が満たされていることを検証する。

> `SplitTimeListCtrl` が `EventCtrl` から `SplitTimeList` の更新指示を受け、`SplitTimeListCtrl` が実行可能状態になったならば、いつか必ず `SplitTimeList` の更新を行う。

### 7.3.2 検証想定モデル

この検証性質に関係するのは、タスクと割り込みハンドラ、周期ハンドラ、排他的共有資源の振る舞いであると言える。よって、モデル検査を用いた検証において無関係であると考えられる以下の想定を設計想定モデルから絞り込み、検証想定モデルを構築する（付録 D にて設計想定モデルからの差分のみ掲載）。なお、括弧内には絞り込みの根拠を示した。

- Sw1Inh
  - vectorElements (振る舞いに無関係な設定のため)
  - routineConnect (振る舞いに無関係な設定のため)
- Sw2Inh
  - vectorElements (振る舞いに無関係な設定のため)
  - routineConnect (振る舞いに無関係な設定のため)
- TimeCountCych
  - periodElements (TimeCountCych のあらゆる実行タイミングを網羅的に検証するため、時間の概念は考慮しない)
- MPL_SPLIT_TIME_LIST_CTRL
  - createServices (メールボックスが利用可能な時点で生成されているものとし、抽象化した)
  - accessPolicy (振る舞いに無関係な設定のため)
  - mapServices (振る舞いに無関係な設定のため)
  - unMapServices (振る舞いに無関係な設定のため)
- MPL_LCD_CTRL
  - createServices (メールボックスが利用可能な時点で生成されているものとし、抽象化した)
  - accessPolicy (振る舞いに無関係な設定のため)
  - mapServices (振る舞いに無関係な設定のため)
  - unMapServices (振る舞いに無関係な設定のため)

### 7.3.3 検証性質を確認する仕組み

7.3.1 で明示した検証性質を確認するために、モデル検査器 SPIN において以下の LTL 式から生成される Never Claim を検証に用いる。

```
#define p (IntCtrl_flg == false)    /* 前提条件：割り込みが発生していない時に限り以降の
LTL 式を評価する */
#define c SplitTimeListCtrl@chosen  /* SplitTimeListCtrl が実行可能になった真になる */
#define d SplitTimeListCtrl@done    /* SplitTimeListCtrl による setSplitTimeList が完
了したら真になる */

   !(<>[]p -> [](c -> <>d))
```

上記の LTL 式は、「前提条件を満たしている時（＝割り込みが発生した場合を除き）、SplitTimeListCtrl が実行可能なったならば、いつか必ず SplitTimeListCtrl の実行が完了する」という意味を含んでいる。こ

こに、前提条件が付与されているのは、SplitTimeListCtrl が実行可能なった後に割り込みが無限回行われることにより、前提条件以下の式が満足されないためである。つまり、割り込みにより「[](c -¿ ¡¿d)」という評価式が正しく確認できないことを防ぐためである。

### 7.3.4 検査コード

モデル検査器 SPIN での検証を行うために、検証想定モデルに基づき検査コードを作成した（付録 E リスト E.1)。

### 7.3.5 モデル検査の結果

モデル検査器 SPIN を用いた検証の結果、以下のようなに不具合が検出された。

```
(Spin Version 4.3.0 -- 22 June 2007)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 912 byte, depth reached 3889, errors: 1
   17430 states, stored (19143 visited)
   30737 states, matched
   49880 transitions (= visited+matched)
   53852 atomic steps
hash conflicts: 224 (resolved)

Stats on memory usage (in Megabytes):
16.105  equivalent memory usage for states (stored*(State-vector + overhead))
16.241  actual memory usage for states (unsuccessful compression: 100.84%)
        State-vector as stored = 920 byte + 12 byte overhead
2.097   memory used for hash table (-w19)
32.000  memory used for DFS stack (-m1000000)
0.461   other (proc and chan stacks)
0.092   memory lost to fragmentation
50.891  total actual memory usage
```

### 7.3.6　不具合の概要

モデル検査器により検出された不具合の概要を、プロダクトラインアーキテクチャを簡略化したイメージ（図 7.6）を用いて示す。

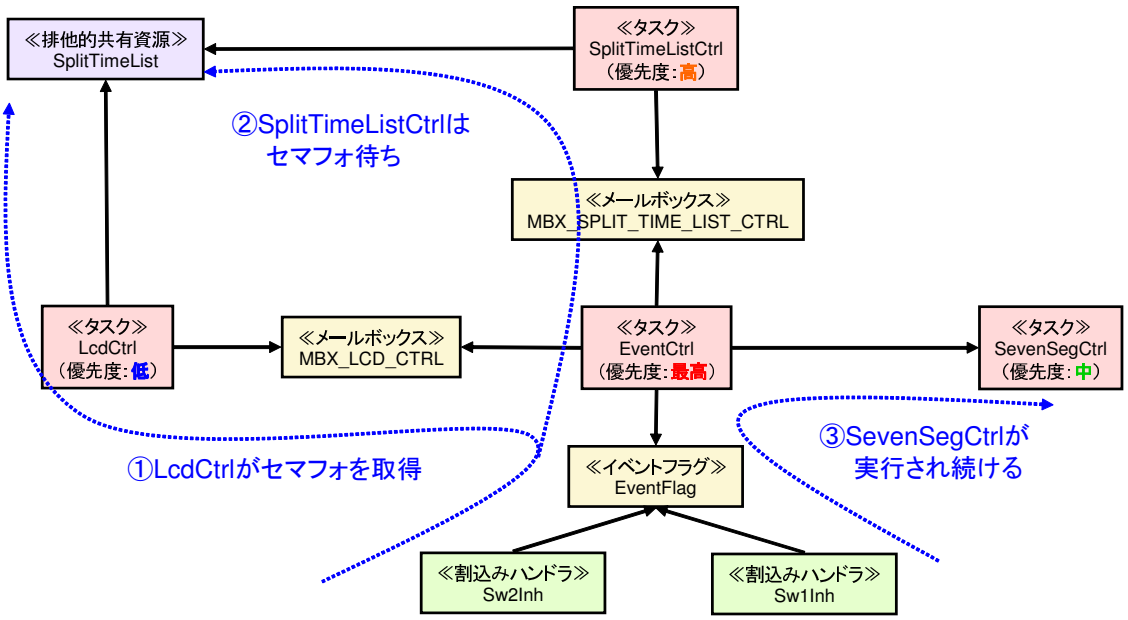


図 7.6: 不具合概要のイメージ

不具合の流れは以下の通りである。

1. Sw2Inh からの割込み処理を受け、EventCtrl が MBX_SPLIT_TIME_LIST_CTRL と MBX_LCD_CTRL 宛にメールを送信する。

2. LcdCtrl がメールを受信し、排他的共有資源である SplitTimeList にアクセスするためのセマフォを取得する。

3. このタイミングで、SplitTimeListCtrl がメールを受信し、SplitTimeListCtrl が実行可能となり、SplitTimeList にアクセスするためのセマフォを取得しようとするが、既に LcdCtrl が獲得済みのためセマフォの待ちキューに入る。

4. このタイミングで、SevenSegCtrl が EventCtrl より起床されると、LcdCtrl より高い優先度を持つ SevenSegCtrl に実行権が移り、永遠に LcdCtrl に実行権が移ることがなくなる。

つまり、セマフォを取得している LcdCtrl に実行権が回ってこないため、セマフォの解放が行われない。よって、セマフォ待ちをしている SplitTimeListCtrl が実行可能になることはないので、SplitTimeListCtrl による SplitTimeList の更新も行われない。

今回の検証で検出した不具合は、SevenSegCtrl の優先度と LcdCtrl の優先度に対する想定に誤りがあったことが直接的な原因となっている。つまり、優先度に対する想定の範囲の検討が不十分であったと言える。

### 7.3.7　優先度に対する想定の再検討

ここで、改めて各タスクの優先度に対する想定（OCL による制約）を表 7.2 にて確認する。

表 7.2: 不具合原因となったタスク優先度に対する想定

| タスク名 | 優先度に対する想定（OCL による制約） |
|---|---|
| EventCtrl | {inv: self.tskpri < SplitTimeListCtrl.tskpri<br>and self.tskpri < SevenSegCtrl.tskpri<br>and self.tskpri < LcdCtrl.tskpri} |
| SplitTimeListCtrl | {inv: self.tskpri < SevenSegCtrl.tskpri<br>and self.tskpri < LcdCtrl.tskpri} |
| SevenSegCtrl | {inv: self.tskpri < LcdCtrl.tskpri} |
| LcdCtrl | {inv: self.tskpri > SevenSegCtrl.tskpri} |

下線で示した部分が今回不具合の原因となったタスク優先度に対する想定であり、本来は最も低い優先度で実行されるべき SevenSegCtrl よりも、LcdCtrl の優先度の方が低くなるよう想定されていたことに、アーキテクチャ想定上の不整合があったといえる。

最後に改めて検討されたタスク優先度に対する想定の範囲を表 7.3 示す。

表 7.3: 再検討されたタスク優先度に対する想定

| タスク名 | 優先度に対する想定（OCL による制約） |
|---|---|
| EventCtrl | {inv: self.tskpri < SplitTimeListCtrl.tskpri<br>and self.tskpri < SevenSegCtrl.tskpri<br>and self.tskpri < LcdCtrl.tskpri} |
| SplitTimeListCtrl | {inv: self.tskpri < SevenSegCtrl.tskpri<br>and self.tskpri < LcdCtrl.tskpri} |
| SevenSegCtrl | {inv: self.tskpri > LcdCtrl.tskpri} |
| LcdCtrl | {inv: self.tskpri < SevenSegCtrl.tskpri} |

表 7.3 のタスク優先度に対する想定の範囲内で同様の検証を行った結果、以下のように 7.3.1 で示した検証性質に関して、不具合は検出されなかった。

```
(Spin Version 4.3.0 -- 22 June 2007)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 912 byte, depth reached 3889, errors: 0
  118773 states, stored (134949 visited)
  239552 states, matched
  374501 transitions (= visited+matched)
  409096 atomic steps
hash conflicts: 14624 (resolved)

Stats on memory usage (in Megabytes):
109.746 equivalent memory usage for states (stored*(State-vector + overhead))
109.041 actual memory usage for states (compression: 99.36%)
        State-vector as stored = 906 byte + 12 byte overhead
2.097   memory used for hash table (-w19)
32.000  memory used for DFS stack (-m1000000)
0.000   other (proc and chan stacks)
0.322   memory lost to fragmentation
143.460 total actual memory usage
```

## 7.4 例題を通じた手法の評価

$\mu$ ITRON を開発基盤としたソフトウェアプロダクトライン開発における、アーキテクチャ上の想定の明示化を想定モデルテンプレートに基づき行うとともに、モデル検査技術を用いてその想定の範囲内での検証を行った。その結果、従来暗黙的に扱われていた優先度の想定の範囲を明示化し、その想定の範囲に誤りが含まれていることをモデル検査による想定範囲内での網羅的な検査により検出した。

今回の例題に提案手法を適用することで、本研究のねらいとしていたアーキテクチャ上の想定の明示化ならびにその想定を含めた検証が行えたといえる。

# 第8章　議論

## 8.1　設計想定モデルを設計意図を表すドキュメントとして活用する

Trew らの経験からも明らかなように、コア資産の設計者とその利用者との間で互いのアーキテクチャ上の想定が一致していなければ、コア資産の再利用による生産性の向上は望めない。そのため、コア資産の設計段階で想定している利用のされ方を設計者の意図として明示化すべきであり、その利用者は再利用しようとしているコア資産の設計意図と開発しているシステムのそれが整合していることを確認する必要がある。

そこで、本稿で提案した設計想定モデルをコア資産設計者の意図を明示化するドキュメントとして活用できると考える。つまり、設計想定モデルは MARTE という標準に基づき構築されたテンプレートを基礎としたモデルであるため、コア資産設計者と利用者との間で誤解が生じにくいといえ、設計意図の整合を行う上で助けになると考える。

## 8.2　検証想定モデルを検証を行った範囲を表すドキュメントとして活用する

本稿では、ソフトウェアプロダクトライン開発を効率的に行うために、コア資産の想定するアーキテクチャ設計の範囲を明示し、それを検証する手法を提案した。しかし、あらゆるコア資産の利用のされ方を検証することは困難であり、現実的には検証を行う範囲を限定することが多い。よって、検証を行う際には実際に検証を行った範囲を、また再利用資産としてそれを利用する際にはその検証済みの範囲を明示的に意識することがアーキテクチャ不整合を防ぐ上で重要であると言える。

そこで、本稿で提案した検証想定モデルをコア資産の検証を行った想定の範囲を表すドキュメントとして活用できると考える。つまり、コア資産の検証を行う者は検証性質ごとに想定している範囲を検証想定モデルとして明示し、その想定範囲において検証性質が満たされていることが確認されていることを示す。また、コア資産を利用する者は検証想定モデルを参照することで、再利用先の製品において想定の範囲内で検証性質が満足されることを確認した上で設計にそのコア資産を組み込むことが可能になると考える。

## 8.3　ツールによる想定のモデル化支援

本稿で提案した手法では、想定をフィーチャモデルの表記法と OCL での制約を用いて表す。実規模の設計における想定をフィーチャモデルの表記法で表す場合、モデルの規模が大きくなり可用性や可読性が低下すると考えられる。

そこで、ツールにより想定のモデル化を支援することが有効であると考える。ツールにより、設計モデルと想定モデル上での要素の連携や想定モデルの階層化、検証性質と想定モデル関連付けなどを支援することで、手法を導入しやすくなるものと考えられる。

# 第9章　おわりに

## 9.1　まとめ

本稿ではソフトウェアプロダクトライン開発におけるコア資産の設計・検証ために、考慮すべきアーキテクチャ上の想定を想定モデルテンプレートとしてモデル化し、これに基づく設計時の想定の明示化ならびにその検証手法を提案した。

一般にアーキテクチャ上の想定には多様なものが含まれており、またそれらを網羅することは困難であるが、本稿で提案した想定モデルテンプレートは主にシステム基盤に対する想定をモデル化したものであると言える。また、想定モデルテンプレートに基づく設計想定モデルや検証想定モデルはシステム内部に対する想定としての情報を含んでいるといえる。

本研究では、特に組込みソフトウェア開発におけるリアルタイム OS の利用に関するアーキテクチャ上の想定を対象に検討を行った。$\mu$ ITRON をシステム基盤とする開発において基盤に対して考慮すべき想定を、MARTE におけるメカニズムの特徴付けに従いテンプレートとしてモデル化し、組込みシステムの製品系列開発における設計ならびに検証に適用しその評価を行った。

## 9.2　今後の課題

本稿ではシステム基盤に対する想定をテンプレート化する試みを行ったが、これは$\mu$ ITRON という限定的なシステム基盤に対してのみ有効なモデルであった。よって、OSEK[14] や T-Kernel[15] など他の RTOS の想定モデルテンプレートを提案すると共に、想定のモデル化支援ツールの作成も視野にいれテンプレート構築方法の更なる体系化を進めていきたいと考えている。

また、想定モデルテンプレートの活用方法についても、モデル検査技術による設計検証を事例に適用することで検討を加えたが、体系だった活用方法やその評価は今後の課題である。

## 9.3　謝辞

本研究を進めるにあたり、多大なるご指導を賜りました、岸知二特任教授、青木利晃特任准教授、片山卓也教授に感謝を申し上げます。

最後に、本研究を進める上で様々な議論や質問に快く応じてくださった、岸研究室、青木研究室、片山研究室、デファゴ研究室の皆様にこの場で感謝を申し上げたいと思います。ありがとうございました。

# 参考文献


[1] 岸知二, 野田夏子, 深澤良彰: ソフトウェアアーキテクチャ, ソフトウェアテクノロジーシリーズ, 共立出版, 2005.

[2] Trew, T.: Enabling the Smooth Integration of Core Assets: Defining and Packaging Architectural Rules for a Family of Embedded Products, In Proceedings of Software Product Line Conference 2005 (SPLC Europe), 2005.

[3] Garlan, D., Allen, R., and Ockerbloom, J.: Architectural Mismatch: Why Reuse Is So Hard, IEEE Software, Nov. 1995, pp.17-26.

[4] Klaus, P., Gunter, B., Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., 2005. 林幸一, 吉村健太郎, 今関剛 訳: ソフトウェアプロダクトラインエンジニアリング, エスアイビー・アクセス, 2009.

[5] 岸知二, 野田夏子: プロダクトライン開発のための想定モデリング, 情報処理学会 組込みシステム研究グループ, 2006.

[6] Kang, K.C., Lee, J., and Donohoe, P.: Feature-Oriented Product Line Engineering, IEEE Software, vol. 19, no. 4, pp. 58-65, July/Aug. 2002.

[7] 坂村健 監修, 高田広章 編: $\mu$ ITRON4.0 仕様 Ver.4.03.00, トロン協会.

[8] A UML Profile for MARTE, Beta 1, August 2007 OMG Adopted Specification, OMG Document #: ptc/07-08-04.

[9] Gomaa, H.: Designing Software Product Lines with UML - From Use Cases to Pattern-Based Software Architectures, Addison-Wesley, 2005.

[10] UML 2.0 OCL Specification, Final Adopted Specification, 2003, OMG Document #: pct/03-10-14.

[11] Holzmann, G.J.: The spin model checker: Primer and reference manual, Addison-Wesley Pub, 2003.

[12] Aoki, T.: Model Checking Multi-task Software on Real-time Operating Systems, International Symposium on Object-Oriented Real-Time Distributed Computing 2008, pp.551-555, 2008.

[13] Eriksson, M., Borstler, J., and Borg, K.: The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations, SPLC 2005, Springer-Verlag, 2005.

[14] OSEK/VDX Group: Operating System Specification 2.2.3, OSEK/VDX Group, 2005.

[15] T-Engine Forum: T-Kernel 仕様書 (Ver.1.00.00), 2005.

# 付 録A　テンプレート化したμITRON4.0仕様スタンダードプロファイルのメカニズム

1. タスク管理機能
    - タスクの生成（CRE_TSK）
    - タスクの起動（act_tsk）
    - 自タスクの終了（ext_act）
    - タスクの強制終了（ter_tsk）
2. タスク付属同期機能
    - 起床待ち（slp_tsk）
    - タスクの起床（wup_tsk）
    - 待ち状態の強制解除（rel_wai）
    - 強制待ち状態への移行（sus_tsk）
    - 強制待ち状態からの再開（rsm_tsk）
    - 自タスクの遅延（dly_tsk）
3. 同期・通信機能
    - セマフォ
        - セマフォの生成（CRE_SEM）
        - セマフォ資源の返却（sig_sem）
        - セマフォ資源の獲得（wai_sem）
    - イベントフラグ
        - イベントフラグの生成（CRE_FLG）
        - イベントフラグのセット（setr_flg）
        - イベントフラグのクリア（clr_flg）
        - イベントフラグ待ち（wai_flg）
    - データキュー
        - データキューの生成（CRE_DTQ）
        - データキューへの送信（snd_dtq）
        - データキューからの受信（rcv_dtq）
    - メールボックス

  - メールボックスの生成（CRE_MBX）
  - メールボックスへの送信（snd_mbx）
  - メールボックスからの受信（rcv_mbx）

4. メモリプール管理機能

   - 固定長メモリプール
     - 固定長メモリプールの生成（CRE_MPF）
     - 固定長メモリブロックの獲得（get_mpf）
     - 固定長メモリブロックの返却（rel_mpf）

5. 時間管理機能

   - 周期ハンドラ
   - 周期ハンドラの生成（CRE_CYC）
   - 周期ハンドラの動作開始（sta_cyc）
   - 周期ハンドラの動作停止（stp_cyc）

6. 割込み管理機能

   - 割込みハンドラの定義（DEF_INH）

# 付 録B　ステートマシン図（ストップウォッチ SPL）

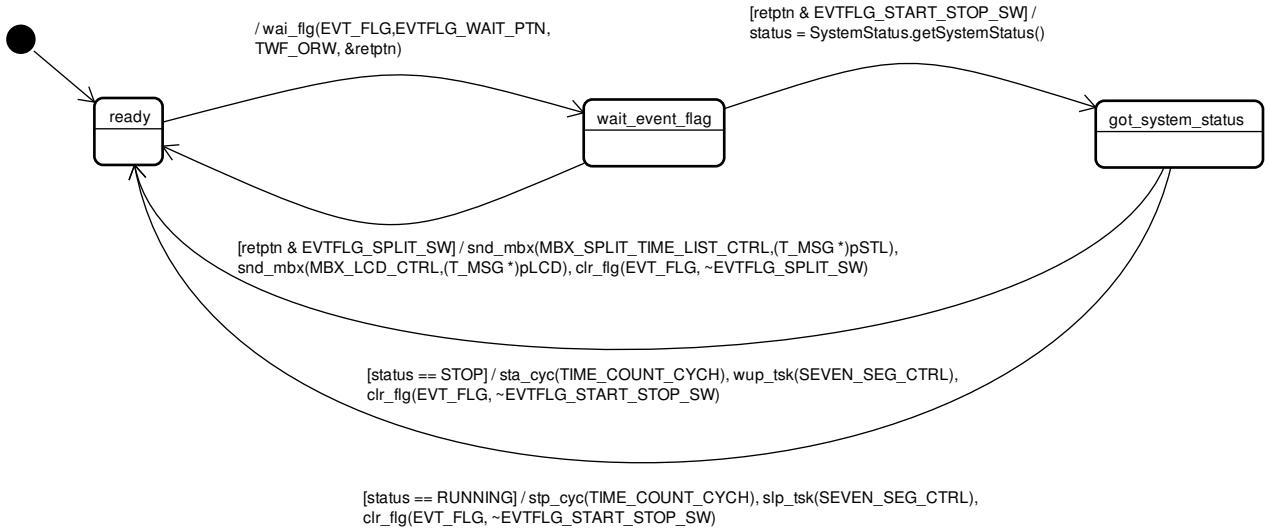


図 B.1: EventCtrl

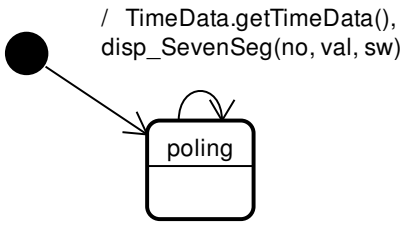


図 B.2: SevenSegCtrl

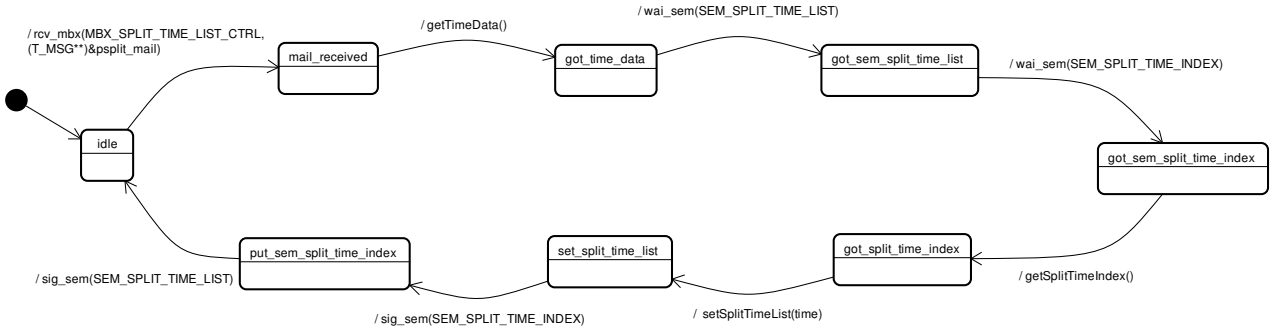


図 B.3: SplitTimeListCtrl

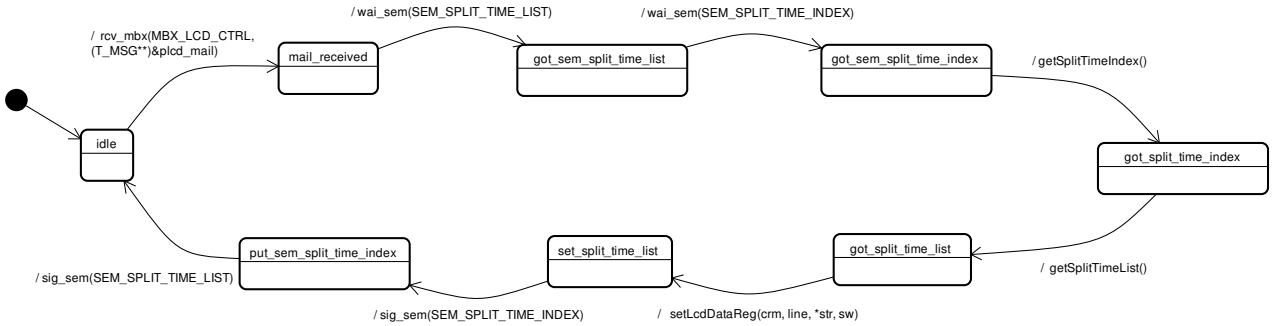


図 B.4: LcdCtrl

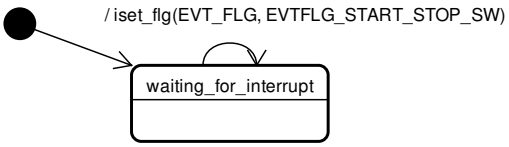


図 B.5: Sw1Inh

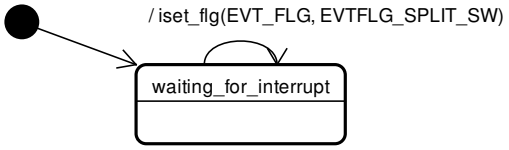


図 B.6: Sw2Inh

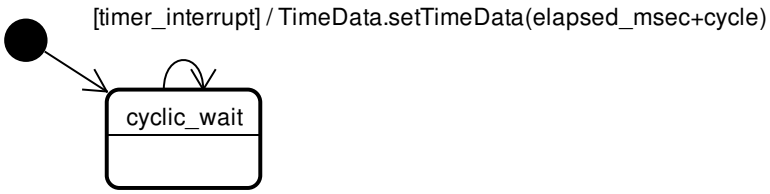


図 B.7: TimeCountCych

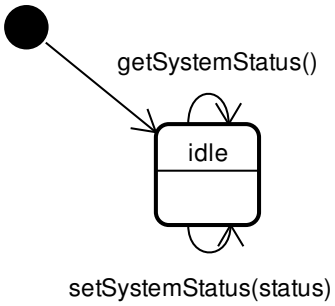


図 B.8: SystemStatus

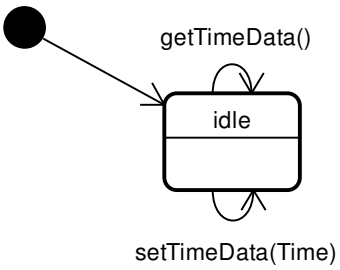


図 B.9: TimeData

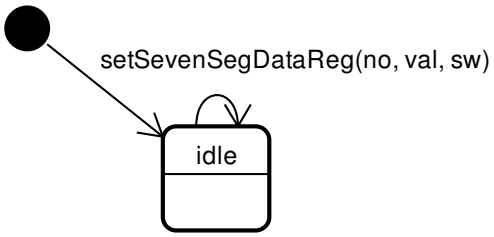


図 B.10: SevenSegDataReg

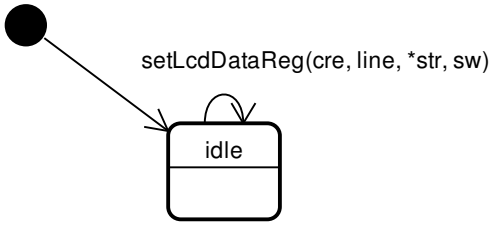


図 B.11: LcdDataReg

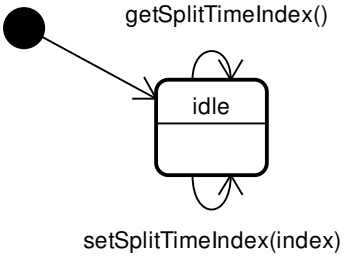


図 B.12: SplitTimeList

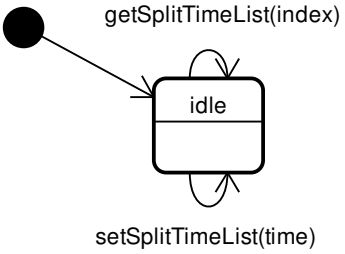


図 B.13: SplitTimeIndex

# 付 録C　設計想定モデル(ストップウォッチSPL)

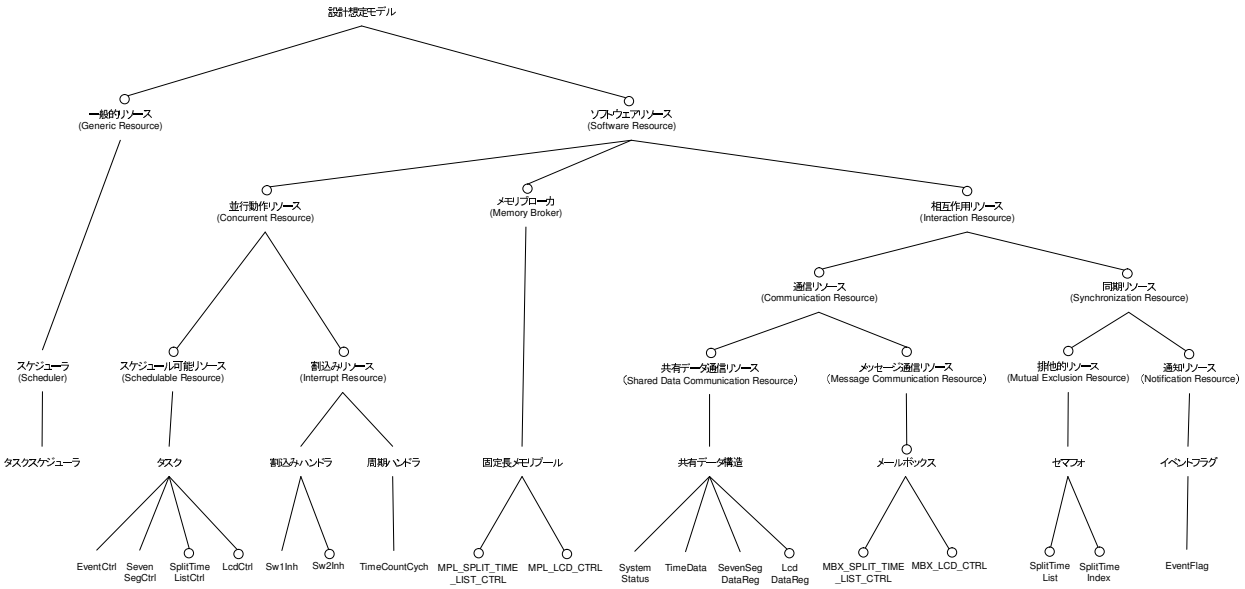


図 C.1: 設計想定モデル（メカニズムのインスタンス階層まで）

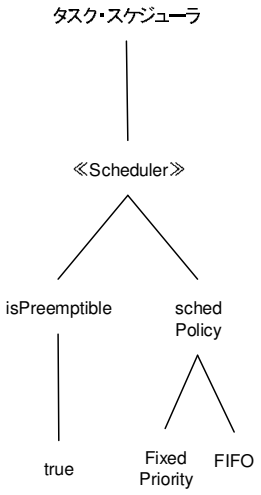


図 C.2: タスクスケジューラ

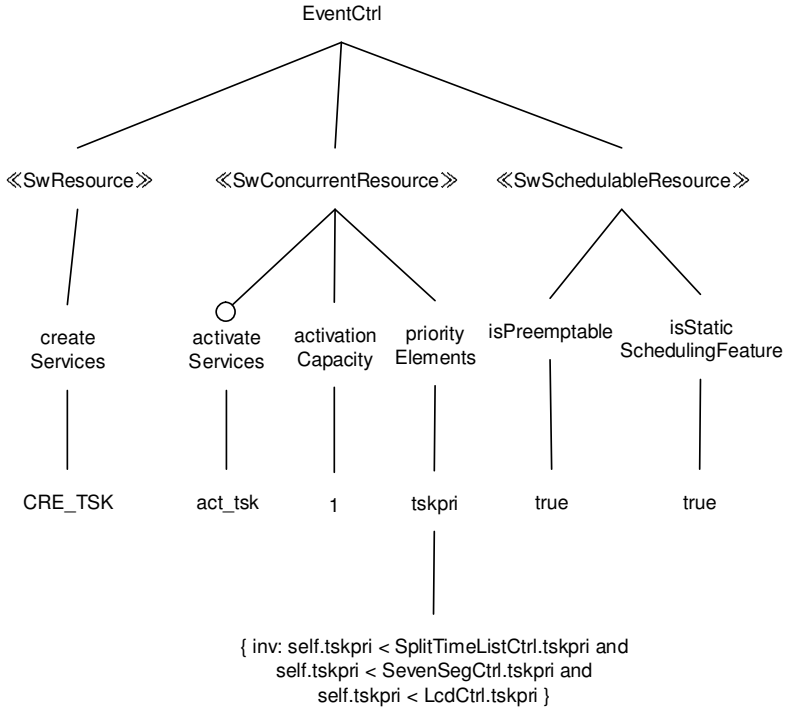


図 C.3: EventCtrl

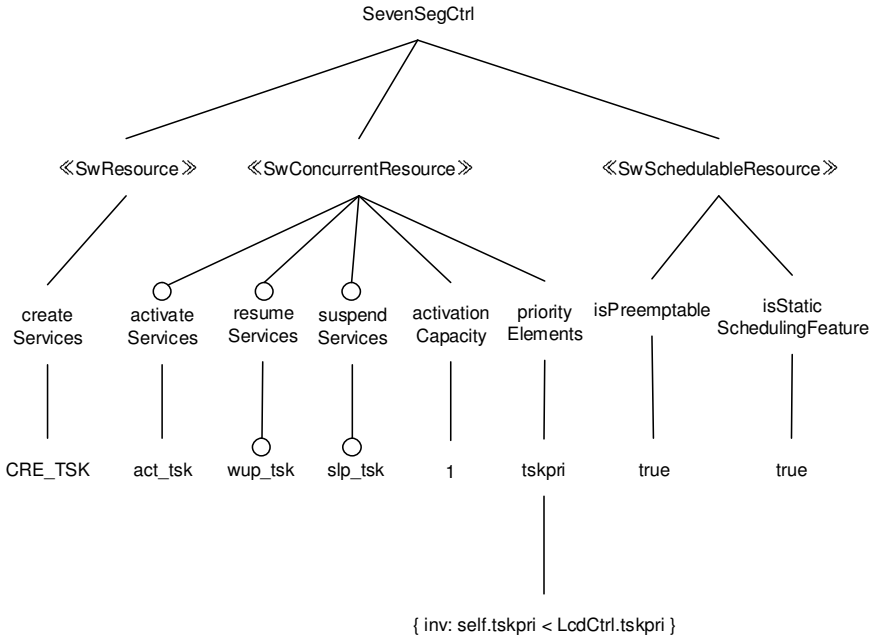


図 C.4: SevenSegCtrl

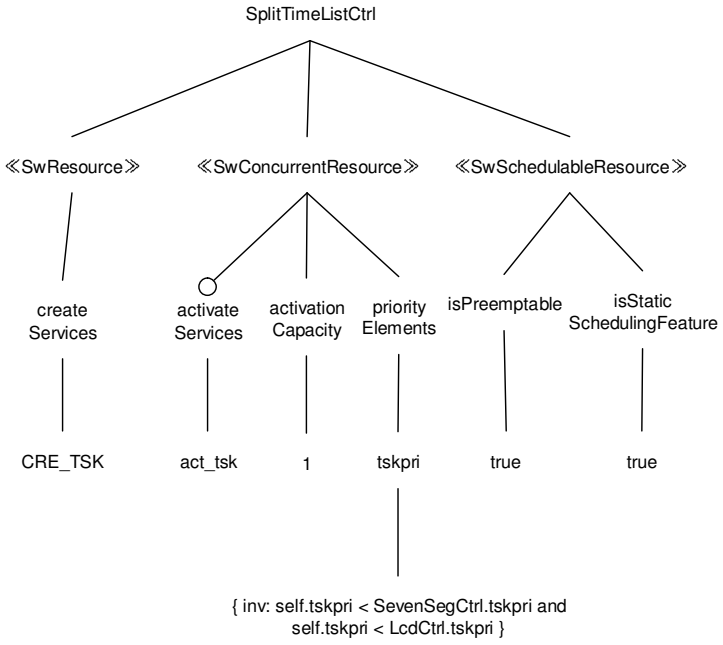


図 C.5: SplitTimeListCtrl

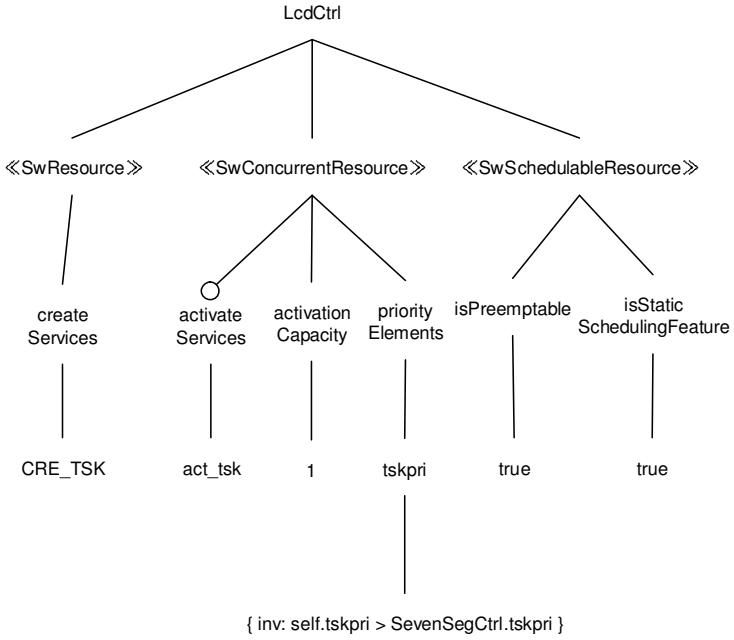


図 C.6: LcdCtrl

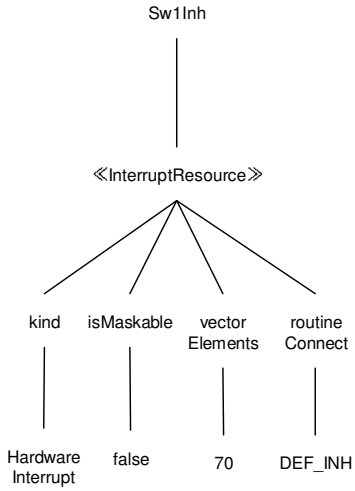


図 C.7: Sw1Inh

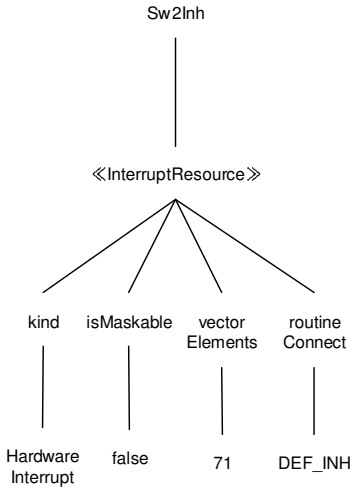


図 C.8: Sw2Inh

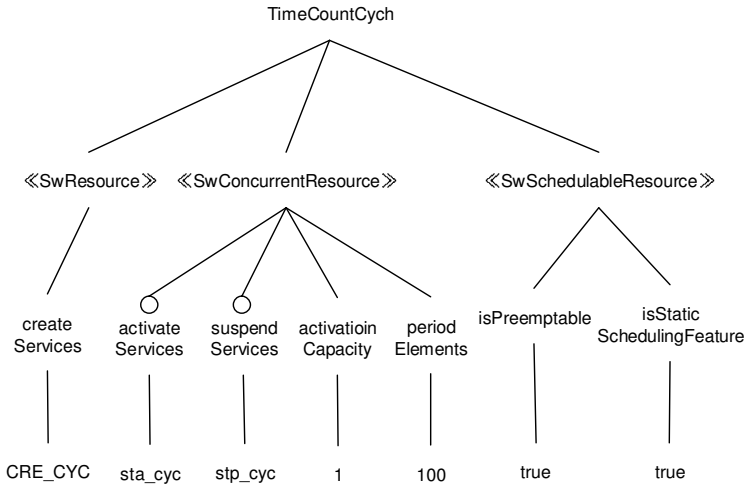


図 C.9: TimeCountCych

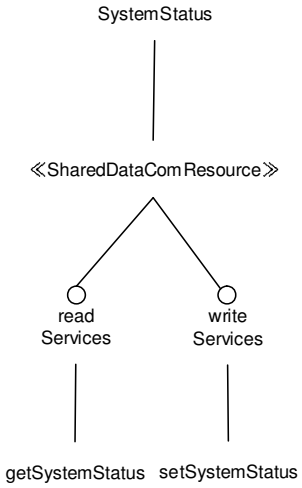


図 C.10: SystemStatus

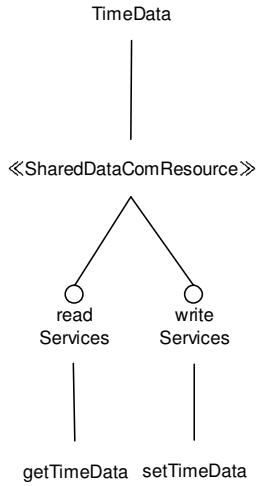


図 C.11: TimeData

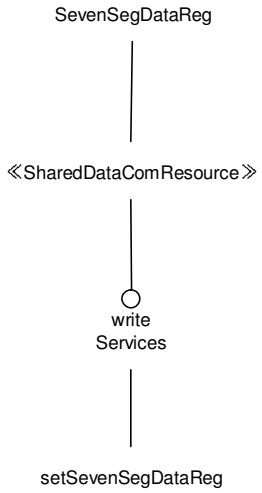


図 C.12: SevenSegDataReg

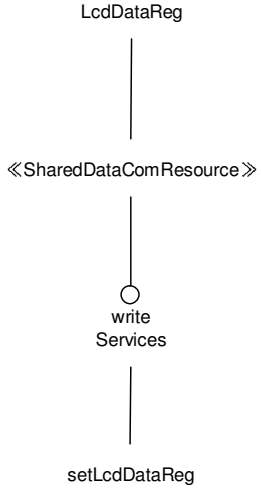


図 C.13: LcdDataReg

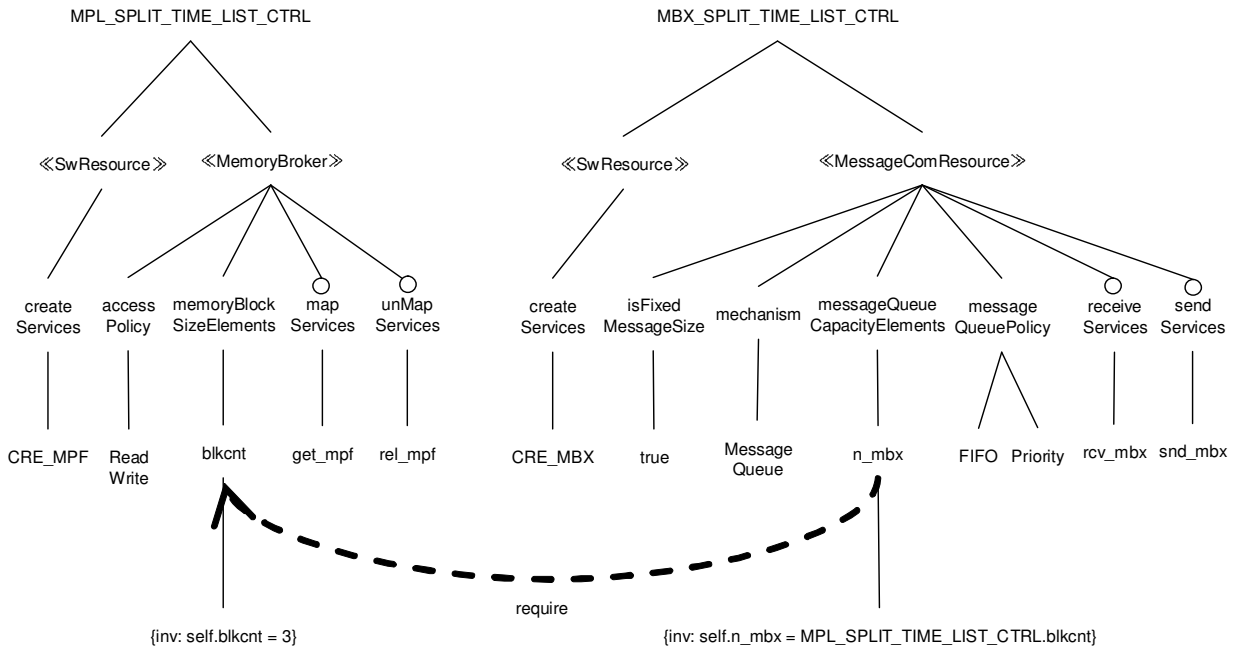


図 C.14: MPL_SPLIT_TIME_LIST_CTRL と MBX_SPLIT_TIME_LIST_CTRL

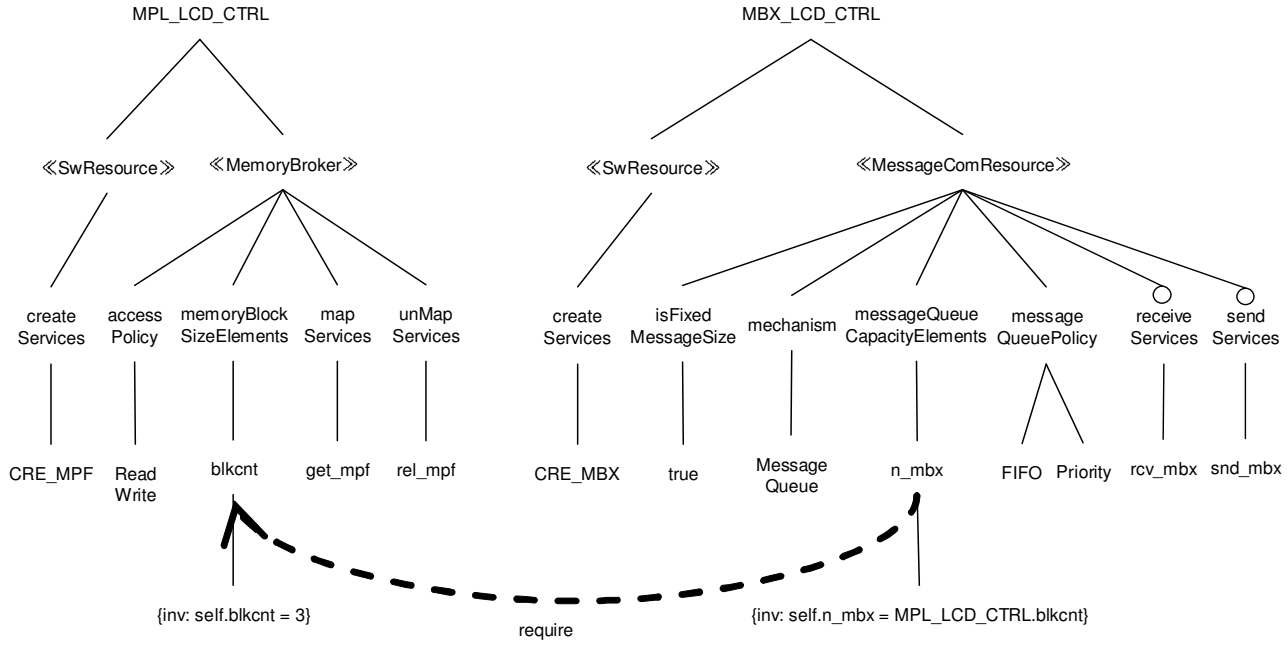


図 C.15: MPL_LCD_CTRL と MBX_LCD_CTRL

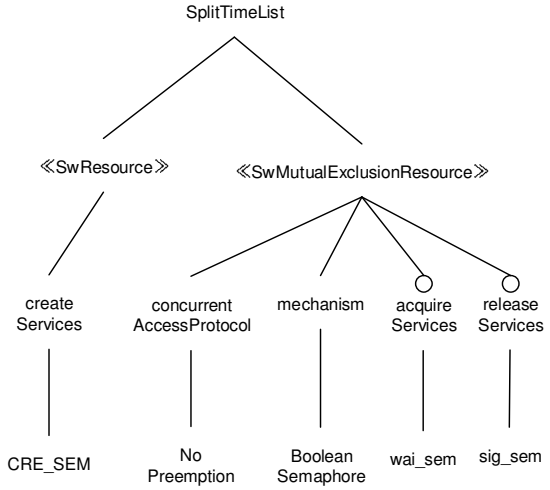


図 C.16: SplitTimeList

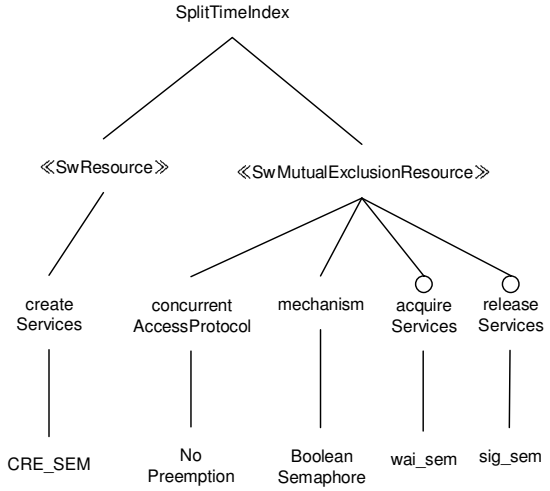


図 C.17: SplitTimeIndex

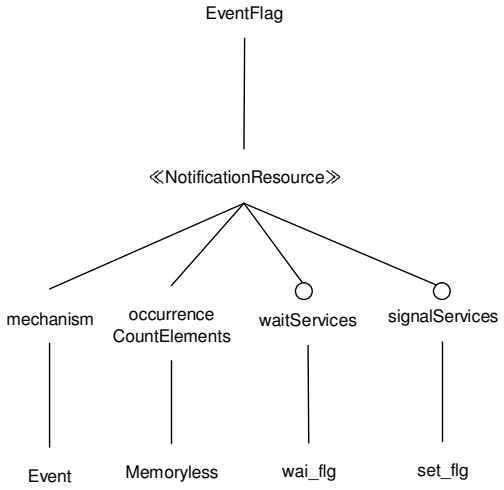


図 C.18: EventFlag

# 付 録D　検証想定モデル(ストップウォッチSPL)

- タスクスケジューラ
- EventCtrl
- SevenSegCtrl
- SplitTimeListCtrl
- LcdCtrl
- SystemStatus
- TimeData
- SevenSegDataReg
- LcdDataReg
- SplitTimeList
- SplitTimeIndex
- EventFlag

以上のメカニズムのインスタンスの検証想定モデルは、設計想定モデルと想定の範囲に変更がないので省略した。

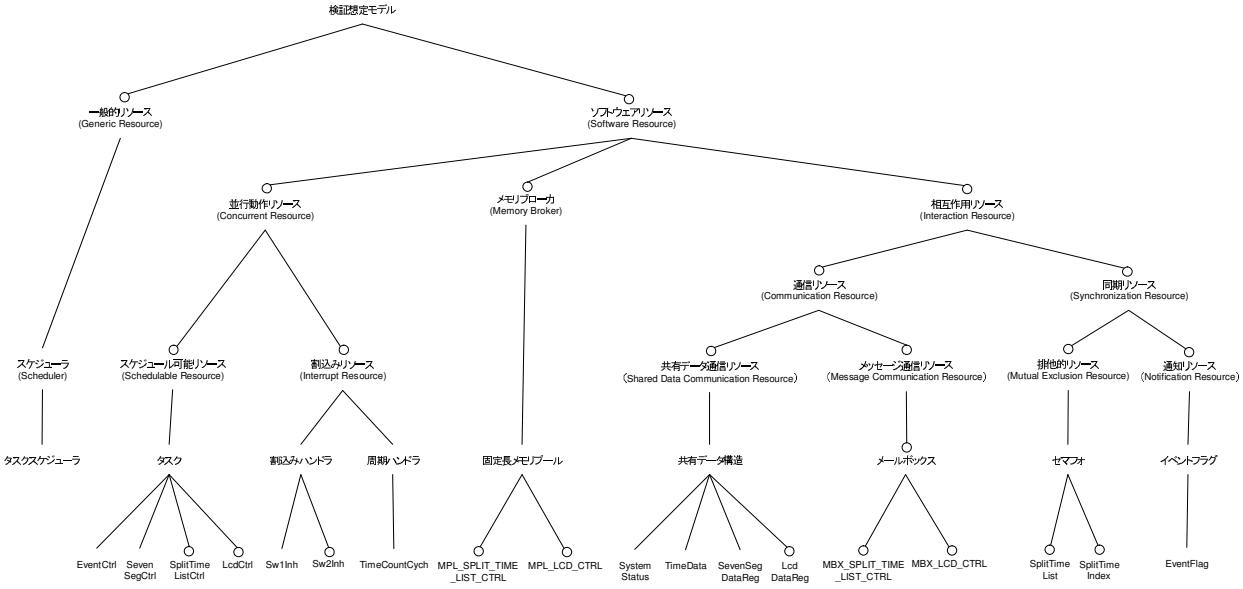


図 D.1: 検証想定モデル（メカニズムのインスタンス階層まで）

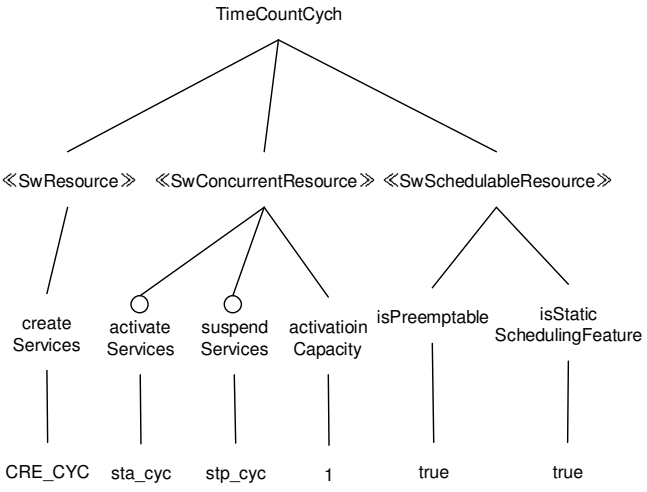


図 D.2: TimeCountCych

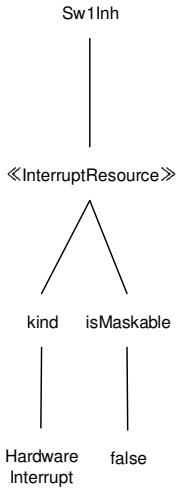


図 D.3: Sw1Inh

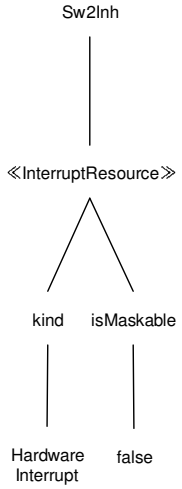


図 D.4: Sw2Inh

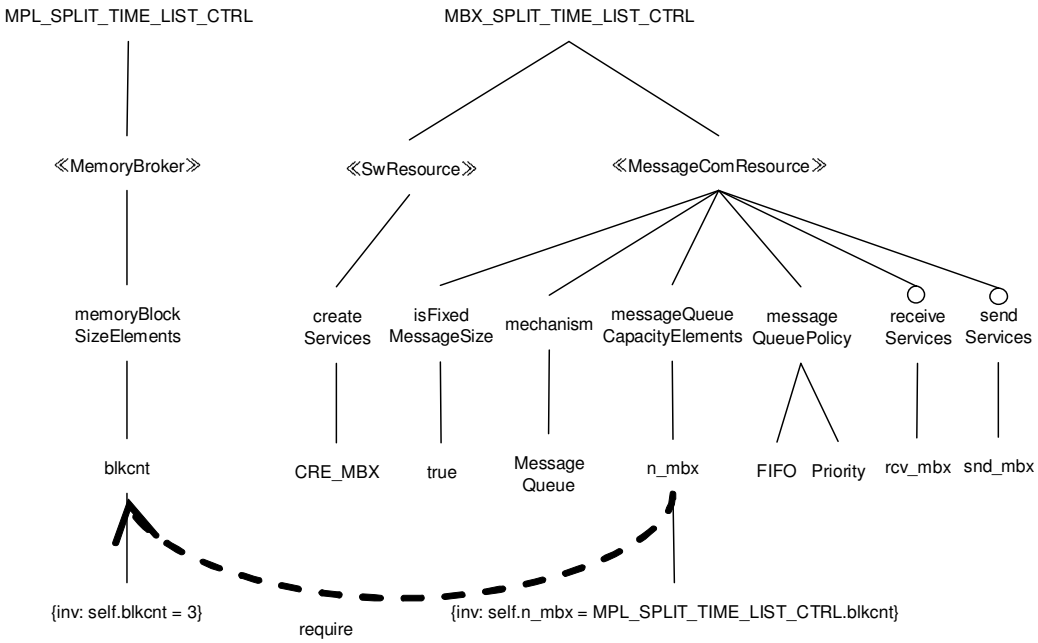


図 D.5: MPL_SPLIT_TIME_LIST_CTRL と MBX_SPLIT_TIME_LIST_CTRL

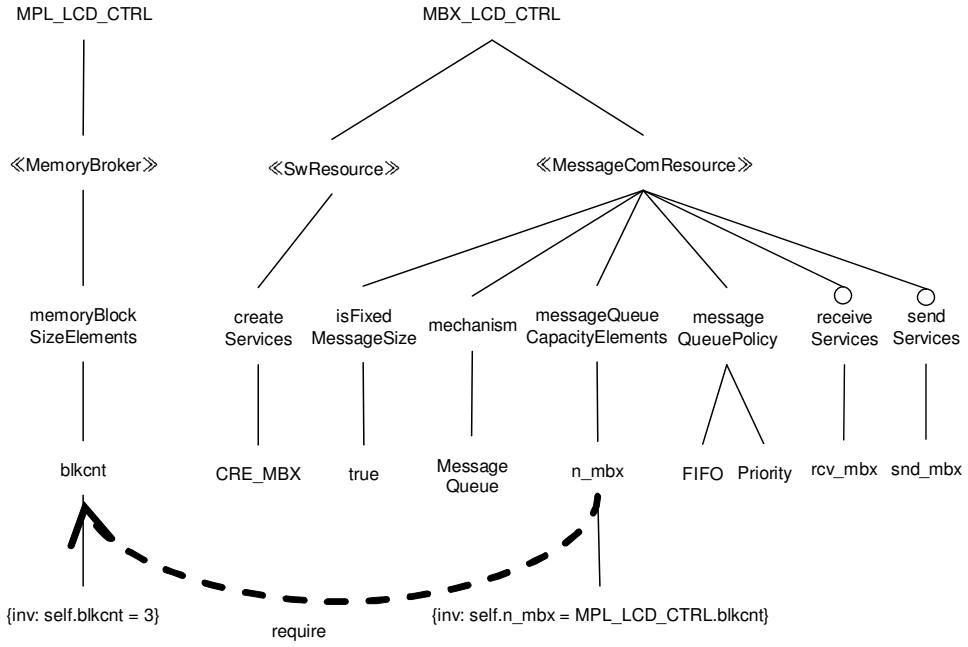


図 D.6: MPL_LCD_CTRL と MBX_LCD_CTRL

# 付 録E　検査コード (ストップウォッチSPL)

List E.1: 検査コード (ストップウォッチ SPL)

```
1  #include "rtoslib.spin"
2
3  #define true  1
4  #define false 0
5
6  /* 各タスクのPID  */
7  #define EVENT_CTRL              1
8  #define SEVEN_SEG_CTRL          2
9  #define SPLIT_TIME_LIST_CTRL    3
10 #define LCD_CTRL                4
11
12 /* イベントフラグのID */
13 #define EVTFLG_SW 0
14
15 /* イベントフラグのパターン */
16 #define FLGPTN_SW1 1 /* 0b0000001 */
17 #define FLGPTN_SW2 2 /* 0b0000010 */
18
19 /* 待ちビットパターン */
20 #define WAIPTN_EVTFLG_SW 3 /* 0b00000011 */
21
22 /* メールボックスのID */
23 #define MBX_SPLIT_TIME_LIST_CTRL 0
24 #define MBX_LCD_CTRL             1
25
26 /* メッセージ */
27 #define MSG_RECORD_SPLIT_TIME 0
28 #define MSG_UPDATE_LCD        0
29
30 /* 固定長メモリプールで獲得可能なメモリブロック数 */
31 #define MPL_SPLIT_TIME_LIST_CTRL_blkcnt 3
32 #define MPL_LCD_CTRL_blkcnt             3
33
34 /* セマフォのID */
35 #define SEM_SPLIT_TIME_LIST   0
36 #define SEM_SPLIT_TIME_INDEX  1
37
38 /* セマフォの初期値 */
39 #define N_SEM_SPLIT_TIME_LIST   1
40 #define N_SEM_SPLIT_TIME_INDEX  1
41
42 /* システムの状態 */
43 #define SYS_STOP    0
44 #define SYS_RUNNING 1
45
46 /* 周期ハンドラの状態 */
47 #define CYCH_STOP   0
48 #define CYCH_START  1
49
50 /***************************/
51 /* タスク優先度に関する制約 */
52 /***************************/
53
```

```
54 /* 優先度逆転が発生する制約 */
55 #define EVENT_CTRL_TSKPRI_CONSTRAINT ( \
56   (EventCtrl_tskpri < SevenSegCtrl_tskpri) && \
57   (EventCtrl_tskpri < SplitTimeListCtrl_tskpri) && \
58   (EventCtrl_tskpri < LcdCtrl_tskpri) )
59 #define SEVEN_SEG_CTRL_TSKPRI_CONSTRAINT ( \
60   (SevenSegCtrl_tskpri < LcdCtrl_tskpri) )
61 #define SPLIT_TIME_LIST_CTRL_TSKPRI_CONSTRAINT ( \
62   (SplitTimeListCtrl_tskpri < SevenSegCtrl_tskpri) && \
63   (SplitTimeListCtrl_tskpri < LcdCtrl_tskpri) )
64 #define LCD_CTRL_TSKPRI_CONSTRAINT ( \
65   (LcdCtrl_tskpri > SevenSegCtrl_tskpri) )
66
67 /* 優先度逆転が発生しない制約 */
68 /* #define EVENT_CTRL_TSKPRI_CONSTRAINT ( \ */
69 /*   (EventCtrl_tskpri < SevenSegCtrl_tskpri) && \ */
70 /*   (EventCtrl_tskpri < SplitTimeListCtrl_tskpri) && \ */
71 /*   (EventCtrl_tskpri < LcdCtrl_tskpri) ) */
72 /* #define SEVEN_SEG_CTRL_TSKPRI_CONSTRAINT ( \ */
73 /*   (SevenSegCtrl_tskpri >= LcdCtrl_tskpri) ) */
74 /* #define SPLIT_TIME_LIST_CTRL_TSKPRI_CONSTRAINT ( \ */
75 /*   (SplitTimeListCtrl_tskpri < SevenSegCtrl_tskpri) && \ */
76 /*   (SplitTimeListCtrl_tskpri <= LcdCtrl_tskpri) ) */
77 /* #define LCD_CTRL_TSKPRI_CONSTRAINT ( \ */
78 /*   (LcdCtrl_tskpri <= SevenSegCtrl_tskpri) ) */
79
80 /* 割込み制御フラグ */
81 bit IntCtrl_flg = false;
82
83 /* 周期ハンドラ::TimeCountCych */
84 bit TimeCountCych_status = CYCH_STOP;
85
86 inline TimeCountCych_sta_cyc() {
87   printf("CALL%d:TimeCountCych_sta_cyc()\n", _pid);
88   TimeCountCych_status = CYCH_START;
89 }
90
91 inline TimeCountCych_stp_cyc() {
92   printf("CALL%d:TimeCountCych_stp_cyc()\n", _pid);
93   TimeCountCych_status = CYCH_STOP;
94 }
95
96 /* 共有データ構造::SystemStatus */
97 bit SystemStatus_status = SYS_STOP;
98
99 inline SystemStatus_setSystemStatus(new_status) {
100   printf("CALL%d:SystemStatus_setSystemStatus()\n", _pid);
101   SystemStatus_status = new_status;
102 }
103
104 inline SystemStatus_getSystemStatus(ret_status) {
105   printf("CALL%d:SystemStatus_getSystemStatus()\n", _pid);
106   ret_status = SystemStatus_status;
107 }
108
109 /* 共有データ構造::TimeData */
110 inline TimeData_setTimeData() {
111   printf("CALL%d:TimeData_setTimeData()\n", _pid);
112 }
113
114 inline TimeData_getTimeData() {
115   printf("CALL%d:TimeData_getTimeData()\n", _pid);
116 }
117
118 /* 共有データ構造::SevenSegDataReg */
119 inline SevenSegDataReg_setSevenSegDataReg() {
```

```
120     printf("CALL%d:SevenSegDataReg_setSevenSegDataReg()\n", _pid);
121   }
122
123   /* 共有データ構造::LcdDataReg */
124   inline LcdDataReg_setLcdDataReg() {
125     printf("CALL%d:LcdDataReg_setLcdDataReg()\n", _pid);
126   }
127
128   /* 排他的リソース::SplitTimeList */
129   inline SplitTimeList_setSplitTimeList() {
130     printf("CALL%d:SplitTimeList_setSplitTimeList()\n", _pid);
131   }
132
133   inline SplitTimeList_getSplitTimeList() {
134     printf("CALL%d:SplitTimeList_getSplitTimeList()\n", _pid);
135   }
136
137   /* 排他的リソース::SplitTimeIndex */
138   inline SplitTimeIndex_setSplitTimeIndex() {
139     printf("CALL%d:SplitTimeIndex_setSplitTimeIndex()\n", _pid);
140   }
141
142   inline SplitTimeIndex_getSplitTimeIndex() {
143     printf("CALL%d:SplitTimeIndex_getSplitTimeIndex()\n", _pid);
144   }
145
146
147   /* タスク::EventCtrl */
148   proctype EventCtrl() provided (turn == EVENT_CTRL) {
149     byte ret_status = SYS_STOP;
150     byte cnt_snd_mbx = 0;
151     byte MBX_SPLIT_TIME_LIST_CTRL_n_mbxq = MPL_SPLIT_TIME_LIST_CTRL_blkcnt;
152     byte MBX_LCD_CTRL_n_mbxq = MPL_LCD_CTRL_blkcnt;
153
154   idle:
155     atomic {
156       wai_flg(EVENT_CTRL, EVTFLG_SW, WAIPTN_EVTFLG_SW, TWF_ORW);
157       goto wait_event_flag;
158     }
159   wait_event_flag:
160     atomic {
161       if
162         /* SW1(計測/停止の切替)が押された場合 */
163         ::(evtflg[EVTFLG_SW].flgptn & FLGPTN_SW1) ->
164           SystemStatus_getSystemStatus(ret_status);
165   got_system_status:
166           if
167             /* 停止中の場合 */
168             ::(ret_status == SYS_STOP) ->
169               TimeCountCych_sta_cyc();
170               wup_tsk(SEVEN_SEG_CTRL);
171               SystemStatus_setSystemStatus(SYS_RUNNING);
172               clr_flg(EVTFLG_SW, ~FLGPTN_SW1);
173               goto idle;
174             /* 計測中の場合 */
175             ::(ret_status == SYS_RUNNING) ->
176               TimeCountCych_stp_cyc();
177               slp_tsk(SEVEN_SEG_CTRL);
178               SystemStatus_setSystemStatus(SYS_STOP);
179               clr_flg(EVTFLG_SW, ~FLGPTN_SW1);
180               goto idle;
181           fi;
182           /* SW2(スプリット記録)が押された場合 //OPTIONAL */
183         ::(evtflg[EVTFLG_SW].flgptn & FLGPTN_SW2) ->
184           if
185             ::(cnt_snd_mbx < MBX_SPLIT_TIME_LIST_CTRL_n_mbxq &&
```

```
186                 cnt_snd_mbx < MBX_LCD_CTRL_n_mbxq) ->
187                 snd_mbx(MBX_SPLIT_TIME_LIST_CTRL, MSG_RECORD_SPLIT_TIME, _pid);
188                 snd_mbx(MBX_LCD_CTRL, MSG_UPDATE_LCD, _pid);
189                 clr_flg(EVTFLG_SW, ~FLGPTN_SW2);
190                 cnt_snd_mbx++;
191                 goto idle;
192             ::else ->
193                 goto idle;
194           fi;
195     fi;
196   }
197 }
198
199 /* タスク::SevenSegCtrl */
200 proctype SevenSegCtrl() provided (turn == SEVEN_SEG_CTRL) {
201 poling:
202   atomic {
203     TimeData_getTimeData();
204     goto got_time_data;
205   }
206 got_time_data:
207   atomic {
208     SevenSegDataReg_setSevenSegDataReg();
209     goto poling;
210   }
211 }
212
213 /* タスク::SplitTimeListCtrl //OPTIONAL */
214 proctype SplitTimeListCtrl() provided (turn == SPLIT_TIME_LIST_CTRL) {
215 idle:
216   rcv_mbx(MBX_SPLIT_TIME_LIST_CTRL, _pid);
217 mail_received:
218   if
219     ::(mbx_index[MBX_SPLIT_TIME_LIST_CTRL] == MSG_RECORD_SPLIT_TIME) ->
220 chosen:
221       TimeData_getTimeData();
222 got_time_data:
223       wai_sem(SEM_SPLIT_TIME_LIST, _pid);
224 got_sem_split_time_list:
225       wai_sem(SEM_SPLIT_TIME_INDEX, _pid);
226 got_sem_split_time_index:
227       SplitTimeIndex_getSplitTimeIndex();
228 got_split_time_index:
229       SplitTimeList_setSplitTimeList();
230 set_split_time_list:
231       SplitTimeIndex_setSplitTimeIndex();
232 done:
233 set_split_time_index:
234       sig_sem(SEM_SPLIT_TIME_INDEX);
235 put_sem_split_time_index:
236       sig_sem(SEM_SPLIT_TIME_LIST);
237       goto idle;
238   fi;
239 }
240
241 /* タスク::LcdCtrl //OPTIONAL */
242 proctype LcdCtrl() provided (turn == LCD_CTRL) {
243 idle:
244   rcv_mbx(MBX_LCD_CTRL, _pid);
245 mail_received:
246   if
247     ::(mbx_index[MBX_LCD_CTRL] == MSG_UPDATE_LCD) ->
248       wai_sem(SEM_SPLIT_TIME_LIST, _pid);
249 got_sem_split_time_list:
250       wai_sem(SEM_SPLIT_TIME_INDEX, _pid);
251 got_sem_split_time_index:
```

```
252         SplitTimeIndex_getSplitTimeIndex();
253 got_split_time_index:
254         SplitTimeList_getSplitTimeList();
255 got_split_time_list:
256         LcdDataReg_setLcdDataReg();
257 set_lcd_data_reg:
258         sig_sem(SEM_SPLIT_TIME_INDEX);
259 put_sem_split_time_index:
260         sig_sem(SEM_SPLIT_TIME_LIST);
261         goto idle;
262   fi;
263 }
264
265 /* 割込み管理プロセス */
266 proctype IntCtrl() {
267 idle:
268   atomic {
269     if
270       ::(IntCtrl_flg == false) ->
271         IntCtrl_flg = true;
272       ::(IntCtrl_flg == true) ->
273         IntCtrl_flg = false;
274     fi;
275     /* printf("INT: IntCtrl_flg = %d\n", IntCtrl_flg); */
276     goto idle;
277   }
278 }
279
280 /* 割込みリソース::Sw1Inh */
281 proctype Sw1Inh() {
282 waiting_for_interrupt:
283   atomic {
284     if
285       ::(IntCtrl_flg == true) ->
286         set_flg(EVTFLG_SW, FLGPTN_SW1);
287         goto waiting_for_interrupt;
288     fi;
289   }
290 }
291
292 /* 割込みリソース::Sw2Inh //OPTIONAL */
293 proctype Sw2Inh() {
294 waiting_for_interrupt:
295   atomic {
296     if
297       ::(IntCtrl_flg == true) ->
298         set_flg(EVTFLG_SW, FLGPTN_SW2);
299         goto waiting_for_interrupt;
300     fi;
301   }
302 }
303
304 /* 割込みリソース::TimeCountCych */
305 proctype TimeCountCych() {
306 cyclic_wait:
307   atomic {
308     if
309       ::((TimeCountCych_status == CYCH_START)&&(IntCtrl_flg == true)) ->
310         TimeData_setTimeData();
311         goto cyclic_wait;
312     fi;
313   }
314 }
315
316 init {
317   /* タスク優先度 */
```

```
318   int EventCtrl_tskpri = -1;
319   int SevenSegCtrl_tskpri = -1;
320   int SplitTimeListCtrl_tskpri = -1;        /* //OPTIONAL */
321   int LcdCtrl_tskpri = -1;                  /* //OPTIONAL */
322
323   /* タスク生成フラグ */
324   bit cre_tsk_flg = false; /* true:生成条件成立 false:生成条件不成立  */
325
326   atomic {
327     /* RTOSライブラリ初期化 */
328     ini();
329
330     /***********************/
331     /* タスク優先度の決定  */
332     /***********************/
333     /* 暫定的なタスクの優先度をSPINの非決定性を用いて代入 */
334     if
335       ::EventCtrl_tskpri = 0;
336       ::EventCtrl_tskpri = 1;
337       ::EventCtrl_tskpri = 2;
338       ::EventCtrl_tskpri = 3;
339     fi;
340     if
341       ::SevenSegCtrl_tskpri = 0;
342       ::SevenSegCtrl_tskpri = 1;
343       ::SevenSegCtrl_tskpri = 2;
344       ::SevenSegCtrl_tskpri = 3;
345     fi;
346     if
347       ::SplitTimeListCtrl_tskpri = 0;
348       ::SplitTimeListCtrl_tskpri = 1;
349       ::SplitTimeListCtrl_tskpri = 2;
350       ::SplitTimeListCtrl_tskpri = 3;
351     fi;
352     if
353       ::LcdCtrl_tskpri = 0;
354       ::LcdCtrl_tskpri = 1;
355       ::LcdCtrl_tskpri = 2;
356       ::LcdCtrl_tskpri = 3;
357     fi;
358
359     /* 暫定的な優先度が各タスク優先度の制約を満たしているか確認*/
360     if
361       ::EVENT_CTRL_TSKPRI_CONSTRAINT
362       ::else -> EventCtrl_tskpri = -1;
363     fi;
364     if
365       ::SEVEN_SEG_CTRL_TSKPRI_CONSTRAINT
366       ::else -> SevenSegCtrl_tskpri = -1;
367     fi;
368     if
369       ::SPLIT_TIME_LIST_CTRL_TSKPRI_CONSTRAINT
370       ::else -> SplitTimeListCtrl_tskpri = -1;
371     fi;
372     if
373       ::LCD_CTRL_TSKPRI_CONSTRAINT
374       ::else -> LcdCtrl_tskpri = -1;
375     fi;
376
377     /* 全てのタスク優先度が制約を満たした場合のみタスク生成フラグを立てる*/
378     if
379       ::( (EventCtrl_tskpri != -1) &&
380           (SevenSegCtrl_tskpri != -1) &&
381           (SplitTimeListCtrl_tskpri != -1) &&
382           (LcdCtrl_tskpri != -1) ) ->
383         cre_tsk_flg = true;
```

```
384     fi;
385
386     /* 制約を満たした場合のみタスクを生成・起動 */
387     if
388       ::cre_tsk_flg ->
389         /* タスク作成 */
390         cre_tsk(EventCtrl_tskpri, EVENT_CTRL);
391         cre_tsk(SevenSegCtrl_tskpri, SEVEN_SEG_CTRL);
392         cre_tsk(SplitTimeListCtrl_tskpri, SPLIT_TIME_LIST_CTRL); /* //OPTIONAL */
393         cre_tsk(LcdCtrl_tskpri, LCD_CTRL); /* //OPTIONAL */
394
395         /* イベントフラグ作成 */
396         cre_flg(EVTFLG_SW, 0); /* 0b00000000 */
397
398         /* メールボックス作成 //OPTIONAL */
399         cre_mbx(MBX_SPLIT_TIME_LIST_CTRL);
400         cre_mbx(MBX_LCD_CTRL);
401
402         /* セマフォ作成 //OPTIONAL */
403         cre_sem(SEM_SPLIT_TIME_LIST, N_SEM_SPLIT_TIME_LIST);
404         cre_sem(SEM_SPLIT_TIME_INDEX, N_SEM_SPLIT_TIME_INDEX);
405
406         /* タスク起動 */
407         act_tsk(EVENT_CTRL);
408         act_tsk(SEVEN_SEG_CTRL);
409         act_tsk(SPLIT_TIME_LIST_CTRL);
410         act_tsk(LCD_CTRL);
411
412         /* スリープ状態(強制待ち状態)へ移行 */
413         slp_tsk(SEVEN_SEG_CTRL);
414
415         /* プロセス実行 */
416         run EventCtrl();
417         run SevenSegCtrl();
418         run SplitTimeListCtrl(); /* //OPTIONAL */
419         run LcdCtrl(); /* //OPTIONAL */
420         run IntCtrl();
421         run Sw1Inh();
422         run Sw2Inh(); /* //OPTIONAL */
423         run TimeCountCych();
424     fi;
425   }
426 }
427
428 #define p (IntCtrl_flg == false)      /* preconditon */
429 #define c SplitTimeListCtrl@chosen   /* chosen */
430 #define d SplitTimeListCtrl@done     /* done */
431
432 /*****************************/
433 /* 検証性質：優先度逆転      */
434 /* !(<>[]p -> [](c -> <>d)) */
435 /*****************************/
436
437 /***************************************
438  (いつか)pがtrueになり,
439   cがtrueになったならば,
440   いつかdがtrueになる(ことが常に起こる)
441
442   =前提条件を満たしている時,
443   SplitTimeListCtrlが実行可能なったならば,
444   いつかSplitTimeListCtrlの実行が完了する
445  ***************************************/
446
447 never {
448 T0_init:
449         if
```

```
450             :: (! ((d)) && (c) && (p)) -> goto accept_S44
451             :: (! ((d)) && (c)) -> goto T0_S23
452             :: ((p)) -> goto T0_S48
453             :: (1) -> goto T0_init
454             fi;
455     accept_S44:
456             if
457             :: (! ((d)) && (p)) -> goto accept_S44
458             fi;
459     T0_S23:
460             if
461             :: (! ((d)) && (p)) -> goto accept_S44
462             :: (! ((d))) -> goto T0_S23
463             fi;
464     T0_S48:
465             if
466             :: (! ((d)) && (c) && (p)) -> goto accept_S44
467             :: ((p)) -> goto T0_S48
468             fi;
469     }
```