博 士 論 文

モデル検査技術を活用した検証指向ソフトウェア設計手法の研究

指導教官　Defago Xavier 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

金井 勇人

2010 年 3 月

要 旨

本研究は，設計検証にモデル検査技術を適用する際に，その作業をより効率よく，かつ正確に行えるようにすることを目的とする．まず，モデル検査技術を適用するためには，検証性質の定義と設計に対応する検査モデルを構築する必要がある．この作業はソフトウェア技術者にとって容易でなく煩わしい作業である．これらの定義作業にはいずれも典型的な定石があることに注目し，それらをパターンとして提案する．次に，モデル検査では検査モデルに対して，検証性質に応じた修正を施すことが多い．こうした修正は横断的に行われるものが多く，検証を繁雑にするだけでなく，間違いを引き起こしやすくなる．修正を容易にするために、モデル検査技術での設計検証を目的としたアスペクト指向 UML モデリングを提案する．最後に，上記の手法を活用、体系化し、重要な性質の検証容易性を向上させることをねらった検証指向ソフトウェア設計手法を提案する。

目 次

第 1章

はじめに

近年，情報システムの社会基盤化，組込みコンピューティングの進展などに伴い，ソフトウェアの信頼性が大きな社会的関心事となっている．しかし，システムの多機能化，高性能化によって，ソフトウェアは大規模かつ複雑になってきており，従来の開発・検証手法で十分な信頼性を確保することが次第に難しくなっている．そうした中，経験則による手法だけでなく，形式的手法等，科学的手法の導入が期待されている．

形式的検証手法の１つにモデル検査技術 [3] がある．この検証技術は，検証対象を表現する有限状態モデルが，論理式で表現された性質を満たすかどうかを状態の網羅的な探索によって検証を行う技術のことである．モデル検査技術を UML 等で記述される設計モデルに適用することにより，ソフトウェアの信頼性を高めることが期待されている [8]. 例えば，高信頼性組込み用オブジェクト指向設計技術プロジェクト [4] において UML 設計検証にモデル検査技術を適用する研究がおこなわれ成果を挙げてきた．

本研究では，こうした UML 設計検証に対するモデル検査技術の適用を対象とする．特に，モデル検査に適した設計モデルのモデリング方法に注目する．

検証する性質は検査モデルの概念を利用して定義されるため，同様の性質を検証するにしても，検査モデルの構築方法によって性質の定義が単純にも複雑にもなり，その結果として検証の効果や効率が変わりうる．しかしながらモデル検査技術に精通していないソフトウェア技術者が，こうした事情を理解して検査モデルを構築することは困難である．

また一般にはひとつの検査モデルに対して複数の性質を検証するため，ひとつの検査モデルに対して検証性質に応じた修正を施すことも多い．こうした修正は横断的に行われるものが多く，検証を繁雑にするだけでなく，間違いを引き起こしやすくなる．さらに設計の過程においては，検査モデルを修正・拡張するたびにこうした複数の検証性質の検証をやりなおす場合も多く，その都度こうした修正を手作業で繰り返すことは現実的でない．

このように，検査モデルには通常の設計モデルとは異なった特性があり，通常のソフトウェア技術者が適切な検査モデルを効率的，効果的に構築することは一般に困難である．そこで本研究では，ソフトウェア技術者が UML 設計検証する際に，検証者が効率的かつ正確に検査モデルの構築，変更ができるように支援することを目的とする．

具体的には，検証性質に応じた適切な検証モデルの構築を支援するためにパターンの活用を提案する．検証性質などによって検査モデルの作成方法に典型的な定石があることに注目し，それらをパターンとして整理し，活用していくことで支援をするアプローチをとる．パターン化することで，それ自体を直接利用するだけでなく，ツールと連携するなど広く活用できる可能性が期待できる．関連研究として，論理式のパターン化の提案 [5] があるが，本研究の独創的な点は，ソフトウェア構造と検証性質を関連付けて，パターン化やツール支援をする点である．

また横断的な修正を支援するために，アスペクト指向モデリング技術の適用を提案する．アスペクト指向技術は横断的な関心事をカプセル化するための技術であり，モデリング技術にそれを適用することで，検査モデルに対して検証性質に応じた修正を効率的かつ間違いなく行うことが可能となる．

さらに本稿では，こうした技術を活用した検証指向の設計手法を提案する．検証を意図せずに作成したモデルに対して，モデル検査技術を適用することは，検証性質の定義などが困難となるなど，効果的ではない．したがって，そのソフトウェアにとって重要な性質を意識しながら，そうした重要な性質の検証が容易な設計モデル，ひいては検査モデルを作成することが重要となる．本研究では，上述したパターンやアスペクト指向技術を活用したモデル検査に適した設計モデリング手法について示す．

第 2章

問題

UML 設計をモデル検査技術により検証する際には，検査モデルを構築する必要があるが，検査モデルには通常の設計モデルとは異なる特性や，構築上の留意点があるため，それを十分に理解したモデリングが必要となる．しかしながら，モデル検査に精通していないソフトウェア技術者にとっては，こうしたモデル検査技術に依存した特性を理解することは敷居が高く，モデル検査技術の適用を妨げるひとつの要因となっている．

本研究の目的は，検査モデル構築上の困難さを軽減することで，ソフトウェア技術者による検査モデル構築を支援するとともに，それに基づいた検証指向の設計手法を提案することである．

検証性質は検査モデルに依存して定義されるため，その構築方法いかんによって，検証性質が容易にも複雑にもなり，その結果検証の効率や効果が大きく影響される．しかしながらモデル検査技術の専門家でないソフトウェア技術者にとって，検証性質に適した検査モデルを構築することは容易ではない．また，検証の特性として，同一の検査モデルに対して，様々な検証性質を定義し検証することが多いが，そうした場合検証性質に応じて検査モデルに対して修正を行う必要がある．しかしながらこうした修正は多くの場合横断的であり，繁雑かつ間違いを導入しやすい．

本研究では，こうした問題を含めて検証を容易にするための検査モデルと検証性質の特性として，検証容易性を定義する．本手法は検証容易性の一部に適応した検査モデルの構築，ならびに検査モデルに対する横断的な修正という問題を取

り上げ，それを改善する手法について提案する．以下，これらの問題について詳しく述べる．

2.1 検証性質に依存する検査モデル

検証性質は検査モデルに依存して定義される．そのため，設計しているシステムにおいて本質的な検証性質の検証を認識し，その検証を容易にする構造になるように設計することが重要である．例えば，ステートマシン図の場合では検証性質の定義において重要な状態を定義したり，クラス図の場合では検証性質の定義に関係する操作や属性を持たせたりすることなどが挙げられる．あるいはメッセージに関する検査をしたい場合には，メッセージの送信や受信がモデル上でとらえやすくなっていないと検証が困難となる．したがって，上記の場合，属性を検査するための，メッセージ送受信を検査するためのモデリングをする必要がある．ただ，これらのモデリングはある 1 つの性質のためのモデリングではなく，典型的な性質の分類のためのモデリングである．本研究では，この典型的な性質の分類を検証観点と呼ぶ．

つまり，検証観点を考慮して，モデリングすることが重要である．例として，CD オーディオシステムを考える．この CD オーディオシステムは，CD とラジオのモードを持っているものとする．モードを切り替える時に，LCD の表示を更新する．設計モデルのクラス図を図 2.1 に示す[1]．

[1] ここでは例題の説明に必要な部分のみを記述している．

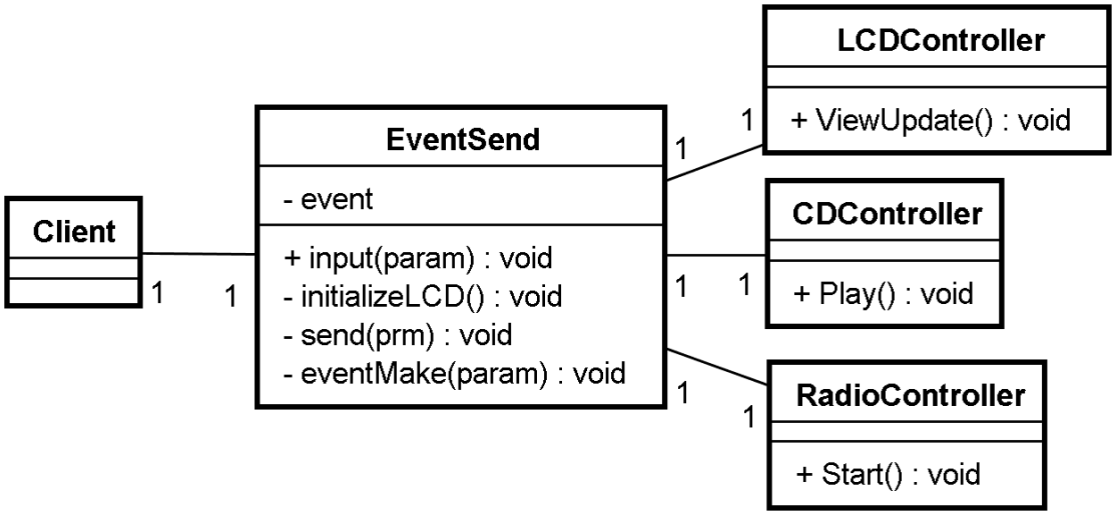


図 2.1: 設計モデルのクラス図

この例の振る舞いについて述べる．ユーザからのモード切り替えの入力キー操作を受け取った Clinet クラスは，EventSend クラスに input メソッドで入力キーを渡す．EventSend クラスは LCD を初期化後，モード切り替えのために音量操作をした後，モード切り替えのために CDController や RadioController に送るためのイベントを作り，各クラスにイベントを渡し，渡されたクラスはそのイベントに対応した処理をすることでモード切り替えが実現される．なお LCD は他のクラスからもアクセスされうるので，その操作の際にはセマフォを取らなければならないものとする．

図 2.2 に設計モデルの Client クラスのステートマシン図を示す．

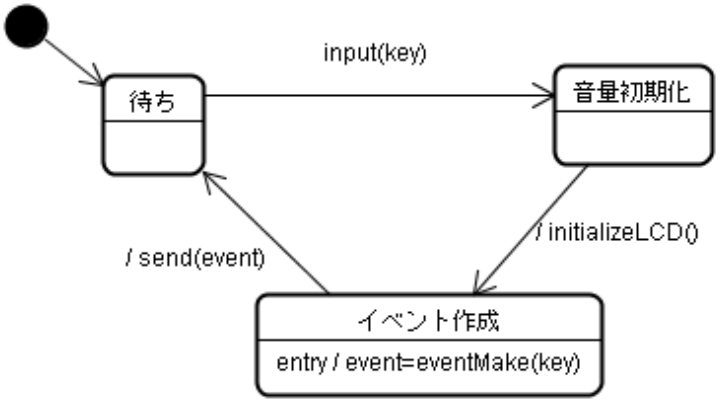


図 2.2: 設計モデルの EventSend クラスのステートマシン図

この設計モデルで下記のことの性質について検証したい場合を考える．

- メッセージの送信確認
- セマフォの取得確認

まず，メッセージの送信確認を検証したい場合を考える．この場合，メッセージ送信をしている部分に対して，送信できたことが確認できるステートマシン図でなければいけないが，上記のステート図ではそうした状態を捉えることが困難である．一方，図 2.2 のイベント作成状態をイベント送信中状態とイベント送信完了状態に分けて定義すると，そうした性質の定義・検証が容易となる．分割提議したものを図 2.3 に示す．send メソッドの中を詳細化したので，便宜的に変数 controller と sendingMake メソッドを追加した．このようにメッセージの送信確認に関する検証が重要であれば，こうしたモデルを構築する方がより適切であると考えられる．

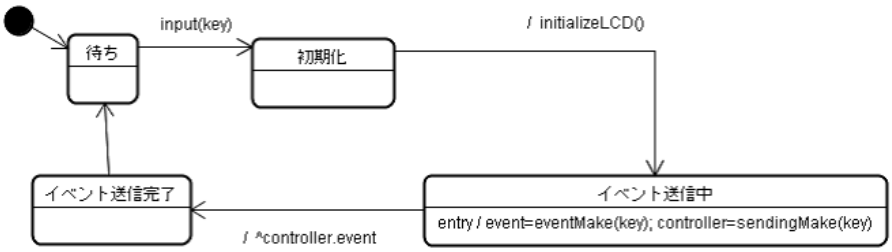


図 2.3: メッセージ送信確認の検査モデル

次に，セマフォの取得確認の検証をしたい場合を考える．この場合，セマフォを取得している部分において，取得できた状態が明示的に捉えられるモデルであることが望まれるが，図 2.2 ではそうしたモデルになっていない．初期化状態をセマフォ取得中状態とセマフォ取得完了状態に分け，図 2.4 に示すようなモデルにすると，セマフォの取得確認に関する性質の定義・検証が容易になる．なお，getSmf メソッドはセマフォを取得するメソッドで，OS などが提供する共通 API である．

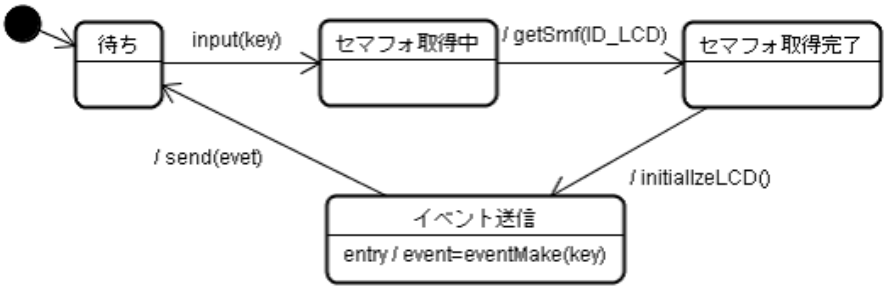


図 2.4: セマフォ取得確認の検査モデル

このように，検証する性質に応じて，それに適したモデルの構築方法が変わるため，重要な性質に関しては，当初よりそれを意識したモデルを構築することが望ましい．意識せずに作られたモデルに対して，検証時に手を加えることは繁雑であり，また間違いを導入しやすくなり，効果的な検証を阻害する．

2.2 検査モデルに対する横断的な変更

2.1 節で，設計モデルは検証観点を考慮して作成することによって，検証モデルに移行しやすくなると述べた．ただ，通常，１つの検査モデルに対して，１つの性質だけを検証することはない．複数の類似した性質や部分的にモデルを変更して様々な状況を想定し，検証することが多い．つまり，構造の骨格は決まっているが，類似性質や部分的な変更に対して検査モデルを変更する必要がある．検証における検査モデルに対する変更，追加の具体例は下記に挙げた例などが考えられる．

- 検証で利用する変数の追加

- アサーションの追加
- 部分的な変更 (例：メッセージ通信の同期，非同期)　など

そして，これらの変更の中には，モデルに対して横断的な変更を伴うものも多いため，検証性質に応じた横断的な変更を容易にすることが重要となる．下記に例題を用いて述べる．

排他的リソースを取得する場合を考える．排他的リソースがロックされている時，その排他的リソースを取得しようとしたタスクが，そのセマフォを開放するまで，待つ場合と待たない場合といった取得方法が考えられる．利用する OS が変わった場合やどのような利用の仕方をされても大丈夫なシステムを作りたい場合などは，この二つの取得方法に関してそれぞれ検証を行いたいことがあり得るが，そうした場合にはモデルの変更が必要となりうる．

図 2.5 は，複数のタスク A, B, .., F が Printer と Scanner というリソースをセマフォを利用しながら取得するシステムのクラス図である．またクラス A のステートマシン図を図 2.6 に示す．

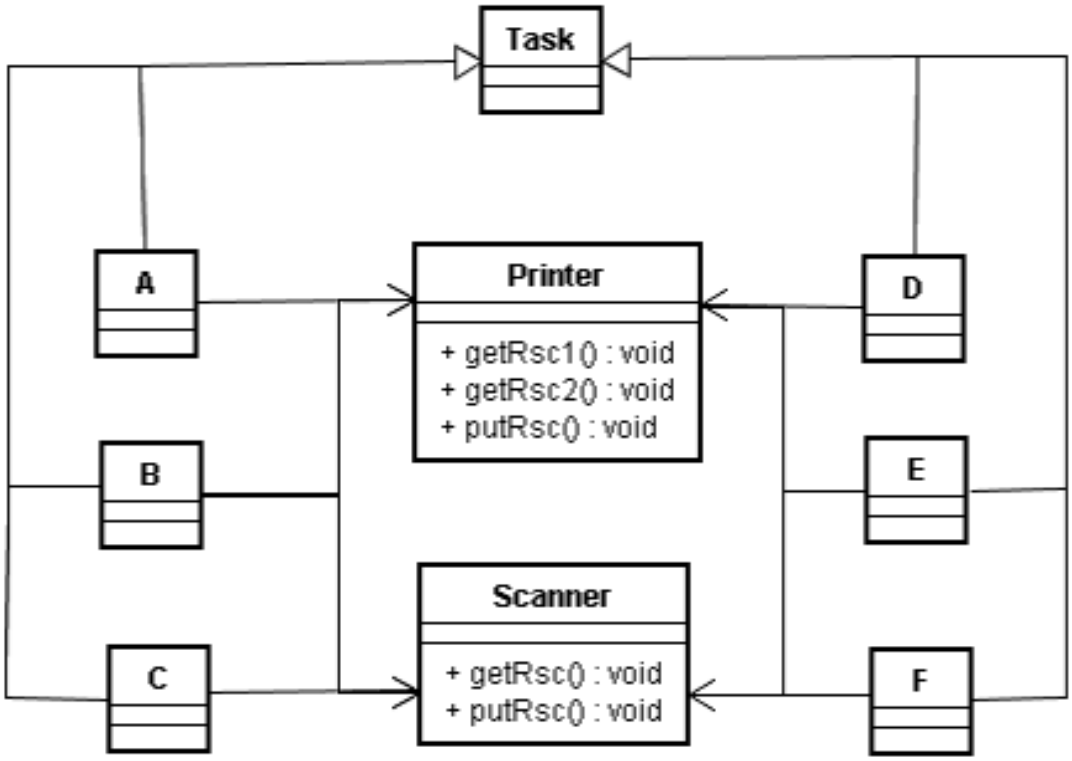


図 2.5: 例題:クラス図

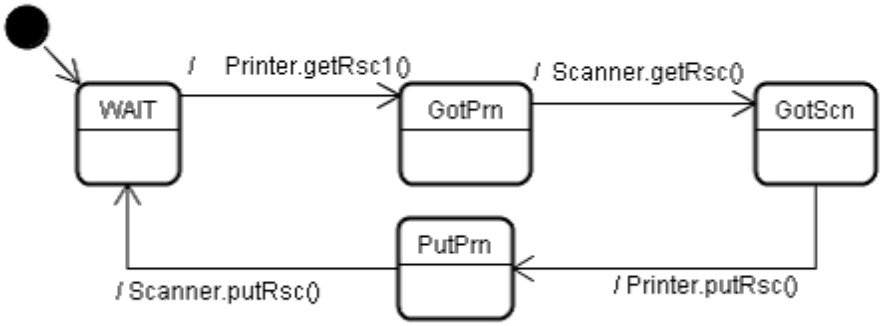


図 2.6: 例題：A のステートマシン図

この例で，セマフォの取得方法を代えて検証しようとすると，モデルに対して以下の変更が必要となる．

- クラス Task が状態 WAIT のときのみ，クラス Printer の操作 getRsc1() ではなく操作 getRsc2() を呼ぶように変更する．

この場合，A, B, ..F すべてのクラスに対して，上記の変更が必要となる．このように複数箇所のモデルの変更は，一般に複数個所に横断的に現れ，煩雑である．開発の過程では，設計と検証は交互に行われ徐々に完成度を高めていくことが多いが，そうした状況では設計を更新するたびに，モデルを変更しつつ両者の検証を行わなければならず，繁雑かつ間違いを引き起こしやすい．

第 3章

アプローチ

本章では，第 2 章で述べた問題点に対する本研究でのアプローチを述べる．

2.1 節で述べた問題に対するアプローチを 3.1 節で，2.2 節で述べた問題に対するアプローチを 3.2 節で述べる．3.3 節で，前述した 2 つのアプローチを利用した検証指向設計手法の提案を述べる．

3.1　検証を考慮した設計パターン

ソフトウェア設計においては，特定のデザインパターン [9] に代表されるように，設計上の課題を解決するための設計モデルのパターンを提示するソフトウェアパターンの技術があり，効果を挙げている．我々は検証性質に応じた検査モデルの構築において，このパターン技術を適用するアプローチをとる．

形式検証分野へのパターンの適用例として代表的なものとしては，Dwyer らの提唱する検証パターン [5] がある．この検証パターンはよく使われる時相論理式の記述を性質ごとに分類しパターン化している．しかしながら Dwyer らのパターンは，検証性質と時相論理式のパターンであり，本研究で問題としている検査モデルに関するパターンを扱うものではない．本研究のパターンの特徴は特定の検証性質に対して，その性質の検証に適した検査モデルと，その検査モデル上でその性質を検証するための時相論理式とをあわせてパターン化している点である．このように検証性質とソフトウェア構造を合わせてパターン化することにより，どのような検査モデルを作成すれば，検証したい性質が検証できるのか明確に理解

することができる．

3.2 アスペクト指向技術を用いた変更手法

横断的な関心事を扱う技術としてアスペクト指向技術が提案されているが，本研究では，検査モデルに対する横断的な変更を効果的に扱うために，このアスペクト指向技術を適用するアプローチをとる．

アスペクト指向技術をUMLモデリングに適用する手法としては，例えばWEAVR[20]などが提案されている．本研究ではこうした既存のアスペクト指向モデリング技術をベースに，検査モデルの特徴を踏まえた検証のためのアスペクト指向モデリング手法と，その支援系を提案し，それに基づいた検証支援を検討する．

3.3 検証指向設計手法

設計検証は場当たり的に行っても，検証に必要な要素がモデル化できていないと容易に検証することは難しい．そのため，検証を考慮した体系だった設計手法が必要である．また，本手法は検証観点に関わる複数の検証性質群を設計のリファインに伴い，繰り返し検証するという流れをサポートする．上記のことを考慮した，設計手法を提案することでできるだけ検証をしやすくし，かつ繰り返しの検証を効率的に行うことが期待できる．

第 4章

モデル検査技術による設計検証

本章では，本研究が前提としている技術について簡単に紹介する．

モデル検査技術による検証ツールには様々なものがあるが [23] 本研究では，モデル検査ツールとしてSPIN を利用することを前提としている．まずSPIN[1][2] においてモデルの記述に利用される PROMELA と，性質の記述に利用される LTL(Linear Temporal Logic) について簡単に紹介する．またモデル検査技術を UML 設計検証に適用する方法にも多様なアプローチ [7] があるが本研究で想定する適用の枠組みについて述べる．

4.1 モデルと性質の記述

4.1.1 仕様記述言語 PROMELA

モデル検査ツール SPIN では，対象システムを仕様記述言語 PROMELA で記述する．ここでは対象システムを，チャネルを介してやりとりをする複数の並行動作プロセスとしてモデル化する．

下記は PROMELA におけるプロセス定義の例である．ここではプロセス P1 から P2 へ，チャネル ch を介してメッセージが受け渡されている．

```
chan ch [0] of {byte} ;
```

```
proctype P1(){
    byte cnt = 0 ;
L0:
    ch ! cnt ;
    cnt++ ;
    goto L0
}

proctype P2() {
    byte msg ;
L0:
    ch ? msg ;
    got L1
}
```

4.1.2 時相論理式 LTL

時相論理式とは，通常の論理式の構成要素である原始命題，論理積，論理和，論理否定の組み合わせに時間的な概念を持った時相論理演算子を加えたものである．SPIN では時相論理式の１つである LTL を性質の記述に利用する．

LTL で記述する時相論理演算子と意味を以下に示す．ここで p,q は任意の式である．

- U

  p U q ... q が真になるまで，ずっと p は真である．

- []

  [] p ... ずっと p は真である．

- <>

  <>p ... いつか p は真である．

- ->

  p ->q ... p が真ならば，q は真である．

下記は LTL 式の記述例であり,「常に，p が真ならばいつか q は真である」ということを意味している．

```
[](p -> <>q)
```

4.2 本研究で前提とする検証手順

設計モデルとして UML を利用した場合の検証手順は多くの方法が考えられる．本研究では，下記の検証手順を前提とする．また図 4.1 に検証手順を示す．図中の番号は下記の検証手順の番号と対応している．

1. モデル化

   UML 設計をモデル検査技術を用いて検証するアプローチは多様だが [6]，本稿では岸らの検証の枠組みを利用する [8][4].

   UML としては構造を表すクラス図と，振る舞いを表すステート図を用いる．クラス図中のクラスは，並行動作単位となるクラスと受動的なクラスとの 2 種類に分類され，前者に対してはステート図で振る舞いを定義する．チャネルはクラス間の関連（アクセスパス）として定義される．

   こうして表現されたモデルは，検証のために PROMELA に変換される．変換の基本的なアプローチは，以下のとおりである．

   - 並行動作単位となるクラスは process にマッピングされる．
   - 並行動作単位となるクラスの振る舞いは，基本的には通信によるメッセージの受信をイベントとするステート図から，対応する PROMELA コードに変換する．
   - アクセスパスとなる関連はチャネルにマッピングされる．

モデル化や変換の詳細は，既存研究と同一ではないが，上記の枠組みに基づいたモデル化を行う．こうしたモデル化は，例えば MARTE[25] や SPT[11] を使った際の動的意味の付与の仕方と整合するものであり，組込みソフトウェアのモデル化としてはひとつの自然なアプローチである．

2. 性質の記述

   システムが満たすべき性質，もしくは満たしてはならない性質を時相論理式 LTL を用いて記述する．この LTL は PROMELA で記述された対象システムの性質を記述するものであるため，その記述は PROMELA の記述に依存する．

3. 検証

   記述した性質が成立するか，もしくは成立しないかをモデル検査ツールを用いて検証する．エラーが検出された場合は判例を元に修正し，1〜3 を繰り返す．

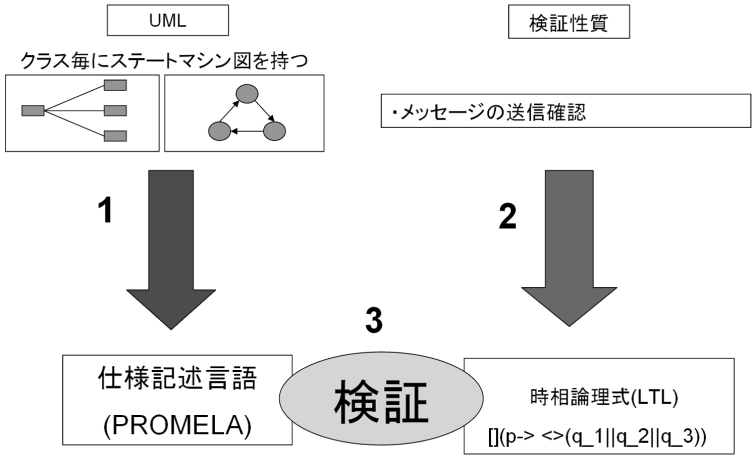


図 4.1: 本研究で前提とする検証手順

第 5 章

検証構造パターンの提案

5.1 はじめに

本章では 2.1 節で示した，検査性質に依存した検査モデル構築上の問題点に対する対策として，検証を考慮した設計パターン（検証構造パターンと呼ぶ）を提案する．

このパターンは，ソフトウェア構造，その構造に依存した性質，その性質を検証する際に利用できる時相論理式の組として定義されるが，このソフトウェア構造は，特定のソフトウェア構造のインスタンスを示すものではなく，その時相論理式が適用できるソフトウェア構造のファミリーを示すものである．パターンを提案するにあたっては，こうしたソフトウェア構造のファミリーを表現するための表記法が重要となるため，本研究ではその表記法についてもあわせて提案する．

5.2 検証構造パターンの提案

5.2.1 従来の代表的な検証パターン

ソフトウェア設計においてデザインパターン [9] を活用するように, ソフトウェア検証でもパターンを活用する提案がなされている．現在までに提唱された代表的な検証パターンとして，Dwyer らの検証パターン [5] がある．この検証パターンはよく使われる時相論理式の記述を性質ごとに分類しパターン化している．この

パターンの分類を図 5.1 に示す.

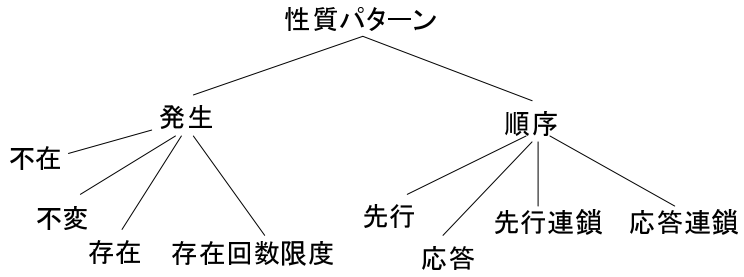


図 5.1: Dwyer らが提唱している検証パターンの分類

この検証パターンでは複数の時相論理式に対してのパターンを提示しており, SPIN で利用する LTL 式以外の時相論理式に対してもパターン化を行っているが, 以下に LTL 式のパターンの一例を示す.

- 不在...P はずっと偽である.

  []($!P$)

- 先行...P が真になる前に S が真になる.

  ([]$!P$) $\|$ ($!P$ U S)

Dwyer らの検証パターンは特定の対象モデルを想定せず，汎用的に利用できる論理式を提示している．そのためにある程度抽象度の高い性質に対応したパターンとなっており，特定のモデル固有の性質に関するパターン化は行われていない. 我々の研究のねらいは，ソフトウェア構造に応じた性質の検証を行うことであり, そうした目的においては，Dwyer らのパターンはやや汎用性すぎて，十分ではない．ソフトウェア検証のためのパターンでは，こうした対象モデルやそれに依存した性質の扱いが重要となると考えられる.

5.2.2 検証構造パターンの定義

本検証パターンの特徴はソフトウェア設計の特定の構造に対して，その構造において重要な性質をリストアップし，それぞれの性質を検証するためのパターン

化を行っている点である．このように構造と検証方法を合わせてパターン化することにより，実際のソフトウェア構造に関わる具体的な性質を検証するための検証方法とその検証ができる構造をパターン化できる．ソフトウェア技術者にとって利用しやすいパターンとなることが期待される．

本検証パターンは，以下の二つから構成される．

1. 設計モデルに多出する構造を検証しやすく抽象化した構造の記述
2. 上記の構造で確認が必要となる典型的な性質の検証方法の記述

検証パターンの記述項目

各検証パターンは，以下に示される項目によって構成される．

- 名前

  その検証パターンが対象とするソフトウェアの構造が簡潔に連想できる名前を示す．

- 検証目的

  検証で確認する性質群を示す．

- 構造

  - 静的構造

    UML のクラス図で表す．

  - 動的構造

    UML のステートマシン図で表す．

- 検証方法

  本検証パターンで定義している検証性質と検証方法を示す．

構造付き検証パターンの一覧

実際のソフトウェア構造を検討し，そこで多出する構造についてパターン化した．具体的には，企業より提供されたカーオーディオ・システムを分析して，そこで多出する構造や，必要な検証項目に注目し，整理してパターン化した．パターンの全ての内容は付録に添付する．

1. 2 オブジェクト間のメッセージ送受信

   Sender クラスから Reciever クラスへのメッセージの到達性を検証するパターン．

2. オブジェクト間連続メッセージ送受信

   Sender クラスから Seder_Receiver クラス，Receiver クラスへのメッセージ到達性の順番を検証するパターン．「2 オブジェクト間のメッセージ送受信」は，Sender クラスから Receiver クラスまでの間に介在するクラス群は検証対象ではないが，このパターンではそのクラス群も検証対象となる．

3. 2 オブジェクト間の属性値

   Informaion クラスの属性 data の値を検証する．例えば，ある特定の値になること，特定の値にはならないこと等を検証する．

5.2.3 メタパターンとしての活用

本研究では，メタパターンとしての利用も想定している．メタパターンとして利用することで，より具体的なパターンの作成を助けることができると考えている．例えば図 5.2 の構造を考える．

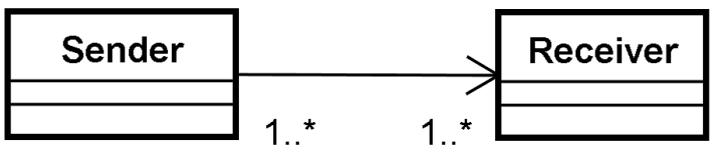


図 5.2: 検証構造パターンの構造の例

ここで次の検証性質を考える．

- Sender クラスのいずれかがメッセージを送信し，それを Receiver クラスのいずれかが受信する．

これに対応する LTL は次のように記述できる．

```
[]((send1 || send2 || ⋯ || sendN) -><>(receive1 || receive2 || ⋯ || receiveN))
```

接頭語 send の述語は，メッセージを送信したことを示しており，1, 2.., N の数字は複数のインスタンスの各々に対応した述語であることを示している．同様に接頭語 receive の述語は，メッセージを受信したことを示しており，1, 2.., N の数字は複数のインスタンスの各々に対応した述語であることを示している．

上記のようなメタパターンが定義されれば，これから派生して，例えば図 5.3, 5.4 に示すようなより限定的なパターンを定義することができる．

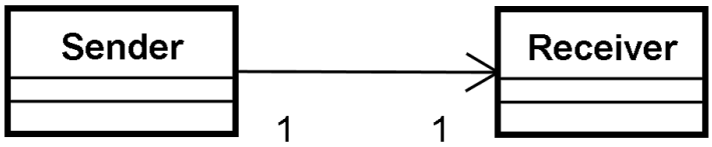


図 5.3: 1-1 の構造例

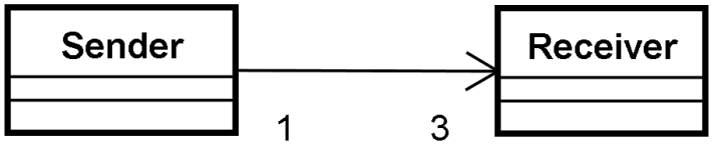


図 5.4: 1-3 の構造例

上述したメタパターンの LTL を限定することにより，図 5.3 の場合，以下のような LTL を導出することができる．

```
[](send1 -><>receive1)
```

同様に図 5.4 の場合，以下の LTL を導出できる．

```
[](send1 -><>(receive1 || receive2 || receive3))
```

このように図 5.2 で示すような検証構造パターンをメタパターンとして利用することにより，より限定的なパターンを派生することができることがわかる．これは，検証者が自分のよく扱うドメインに適したパターンを作成するのに役立つと考えられる．

5.3 パターン構造の表記方法

本節では，検証構造パターンにおける構造記述の表記法について提案する．構造記述は特定のソフトウェア構造を示すものではなく，その検証パターンを適用できるソフトウェア構造が満たすべき制約を示すことにより，構造のファミリーを表すものでなければならない．以下，そうしたソフトウェアの構造のファミリーを示すための表記方法を提案する．なお，表記には UML のクラス図，ステートマシン図を利用するが，これらの図の標準的なモデル要素に関しては説明を省略する．

5.3.1 制約

パターンを適用できるソフトウェア構造のファミリーを示すために，静的構造，動的構造の双方に対し，以下の観点から制約を課す．

- 静的構造
  - 必要なクラスと，そのクラスが持つべき属性と操作
  - 選択的なクラス（構造中に存在してもよいクラス）
  - クラス間の関連と多重度
- 動的構造
  - 必要な状態と遷移 (ガード，イベント，アクションも含む)
  - 選択的な状態，遷移と必要な状態と遷移の関係

5.3.2 表記

上述した制約を，以下の UML 表記法を用いて記述する．静的構造，動的構造両方に対してメタモデルを定義し，制約を与える．

以下，具体的な表記内容や，表記にあたって利用するステレオタイプ等の意味について説明する．

静的構造

(1) クラス

この検証構造パターンを利用するために必要なクラスを必須クラス，任意のクラスを選択クラスと呼ぶ．本表記法では，この必須クラスと選択クラスとの区別をするために，表 5.1 に示すステレオタイプを導入する．⟨⟨MandatoryClass⟩⟩ は必須クラスを表し，⟨⟨OptionalClass⟩⟩ は選択クラスを表す．

表 5.1: 静的構造のファミリーを表すためのステレオタイプ

ステレオタイプ	意味
⟨⟨MandatoryClass⟩⟩	必須クラス
⟨⟨OptionalClass⟩⟩	選択クラス

モデル検査には実行意味も与える必要があるため，これに関してもステレオタイプを付与する．実行意味のステレオタイプは OMG による組込み向け UML プロファイルである MARTE[25] を利用する．

本稿で利用するステレオタイプは，MARTE で定義されているもののうち表 5.2 に示すステレオタイプを利用する．⟨⟨SwSchedulableResource⟩⟩ は並行動作単位を表し，⟨⟨SwMutualExclusionResource⟩⟩ は排他機構を表す．

典型的には，これらのステレオタイプはそのクラスが以下を表す場合に使われる．

- ⟨⟨SwSchedulableResource⟩⟩

  タスク，active クラス

表 5.2: 動的意味を表すステレオタイプ

ステレオタイプ	意味
⟨⟨SwSchedulableResource⟩⟩	並行動作単位
⟨⟨SwMutualExclusionResource⟩⟩	排他機構

- ⟨⟨SwMutualExclusionResource⟩⟩

  セマフォ，Mutex

(2) 関連

関連の表記を図 5.5 に示す.

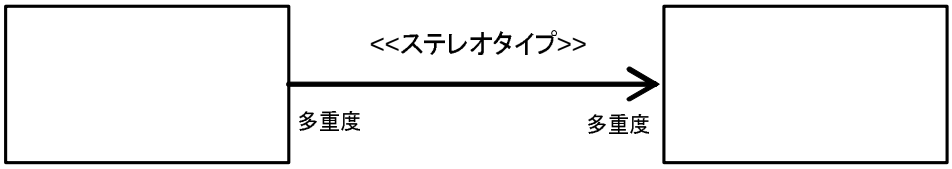


図 5.5: 関連の表記

関連のステレオタイプは 2 種類あり，重ねて利用する．1 つ目は通信方法を表す. ステレオタイプを表 5.3 に示す．⟨⟨SYNC⟩⟩ は同期通信を表し，⟨⟨ASYNC⟩⟩ は非同期通信を表す.

なお，同期通信か非同期通信かによって，検証性質に影響がある．例えば，メッセージの到達性の検証の場合，非同期通信では「送信後いつかは受信」という性質になるが，同期通信では,「送信と受信は同時に起きる」という性質となる．送信クラスから受信クラスへ矢印の関連とする．両端のクラスとも送受信する場合，矢印の表記はしない.

表 5.3: 通信方法のステレオタイプ記述と意味

ステレオタイプ	意味
⟨⟨SYNC⟩⟩	同期通信
⟨⟨ASYNC⟩⟩	非同期通信

2 つ目はチャネルを表す．ステレオタイプを表 5.4 に示す．

共有関連 (shared_channel ステレオタイプの付いた関連) は複数のオブジェクトで共有のメッセージチャネルを利用することを意味する．非共有関連 (unshared_channel ステレオタイプの付いた関連) は各オブジェクトが個別のメッセージチャネルを利用することを意味する．

表 5.4: チャネルのステレオタイプ記述と意味

ステレオタイプ	意味
⟨⟨shared_channel⟩⟩	複数のオブジェクトで共有
⟨⟨unshared_channel⟩⟩	複数のオブジェクトで非共有

以上の表記方法を利用して記述した静的構造の例を図 5.6，5.7 に示す．図 5.6 は，SwSchedulableResource である Sender と Reciever という 2 つの必須クラスがあり，その間には任意個の SwSchedulableResource である Receiver_Sender クラスが存在しうるという構造上の制約を示している．Sender，Sender_Receiver，Receiver 間のやりとりは, 非同期通信または同期通信，チャネルは共有または非共有である．多重度により各クラスのオブジェクトが任意個であることを示している．また制約 or により，Sender と Sender_Receiverl，Sender と Receiver のどちらかの関連が無くてはいけないことを示している．制約 and は Sender と Sender_Receiver，Sender_Receiver と Receiver のどちらかの関連がある場合，どちらも関連がなければいけないことを示している．

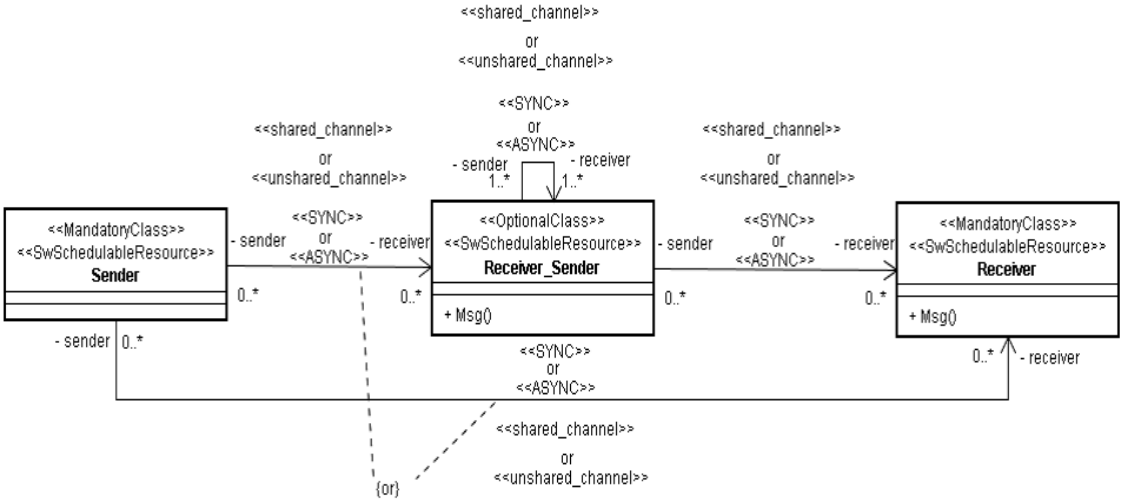


図 5.6: 静的構造の表記例 1

図 5.7 は，UpdateControl と Information という 2 つの必須クラスがあり，その間には任意のクラスが存在してはいけないという構造上の制約を示している．UpdateControl と Information 間のやりとりは，通信方法に依存しない．

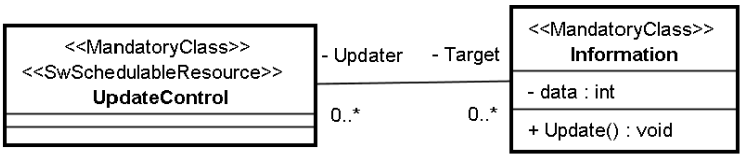


図 5.7: 静的構造の表記例 2

動的構造

動的構造の制約は，UML2.0 のステートマシン図を用いて行う．

(1) 状態

この検証パターンを利用するために必要な状態を必須状態，任意の状態を選択状態と呼ぶ．本表記法では，この必須状態と選択状態との区別をするために，表 5.5 に示すステレオタイプを導入する．⟨⟨MandatoryState⟩⟩ は必須状態を表し，⟨⟨OptionalState⟩⟩ は選択状態を表す．

表 5.5 で示したステレオタイプの詳細を下記に示す．

表 5.5: 状態へのステレオタイプ

stereotype	意味
⟨⟨MandatoryState⟩⟩	必須状態
⟨⟨OptionalState⟩⟩	選択状態

— MandatoryState

この検証パターンを利用するために必要な状態を必須状態と呼ぶ．必須状態には，その状態を示す状態名を記述する．

— OptionalState

任意の状態を選択状態と呼ぶ．必須状態との関係を示すために記述される．

(2) 遷移

この検証パターンを利用するために必要な遷移で ReceiveMessageEvent を持つ遷移，SendMessageAction を持つ遷移，選択的な遷移を区別するために，表 5.6 に示すステレオタイプを導入する．⟨⟨OptionalState⟩⟩ は選択的な遷移を表し，⟨⟨TwithRME⟩⟩ は ReceiveMessageEvent を持った遷移を表し，⟨⟨TwithSMA⟩⟩ は SendMessageAction を持った遷移を表す．

表 5.6: 遷移へのステレオタイプ

stereotype	意味
⟨⟨OptionalTransition⟩⟩	選択的な遷移
⟨⟨TwithRME⟩⟩	ReceiveMessageEvent を持った遷移
⟨⟨TwithSMA⟩⟩	SendMessageAction を持った遷移

表 5.6 で示したステレオタイプの詳細を下記に示す．

— OptionalTransition

遷移元パターン状態に対応したユーザーモデル状態と遷移先パターン状態に対応したユーザーモデル状態の間に任意のユーザーモデル状態，ユーザーモデル遷移が存在しても良い．

－ TwithRME

遷移元パターン状態に対応したユーザーモデル状態 A と遷移先パターン状態に対応したユーザーモデル状態 B の間にある遷移で，メッセージ受信のイベントである ReceiveMessageEvet が存在しなくてはいけない．

－ TwithSMA

次の 2 つのいずれかにメッセージ送信のアクションである SendMessageAction が存在しなくてはいけない．

- 遷移元パターン状態に対応したユーザーモデル状態の exit
- 遷移元パターン状態に対応したユーザーモデル状態 A と遷移先パターン状態に対応したユーザーモデル状態 B の間にあるユーザモデル遷移．

5.3.3 クラス図の変換

クラス単体の変換規則

検査モデルのクラス図は，モデル検査技術で検証するために PROMELA に変換される．以下に変換の方法を説明する．

図 5.8 はクラス単体の変換例である．メンバ属性は LTL 式で検証できるように，グローバル変数「クラス名__属性名」に変換する．クラス自体は次のようにステレオタイプに応じた変換をする．

- ⟨⟨SwSchedulableResource⟩⟩ → proctype クラス名
- ⟨⟨SwMutualExclusionResource⟩⟩ → proctype クラス名
- なし → proctype への変換無し

操作も次のようにステレオタイプに依存して変換される．

- ⟨⟨SwSchedulableResource⟩⟩ クラスの操作　→　mtype= {操作名}
- ステレオタイプなしのクラスの操作　→　インライン関数　クラス名_操作名 ()

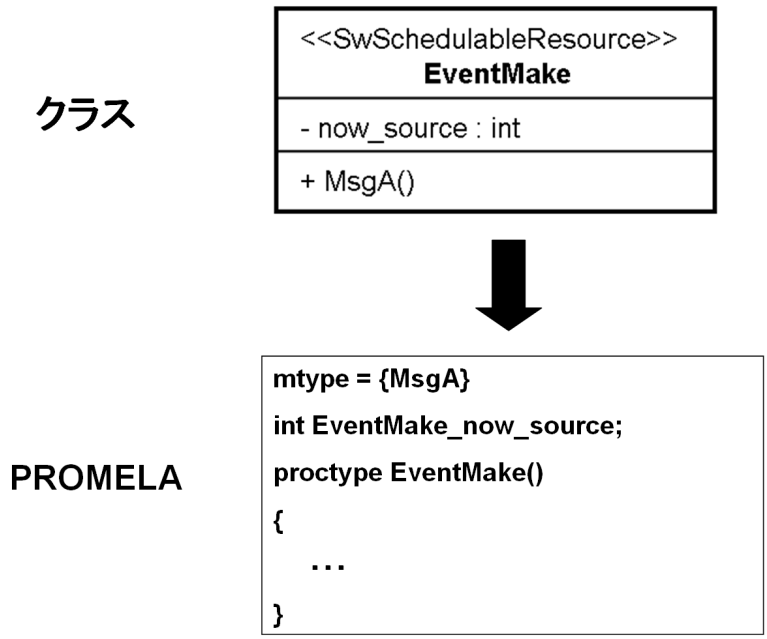


図 5.8: クラスから PROMELA への変換例

クラス関連の変換規則

双方向の場合２つのチャネルに変換する．以下のように非同期，同期，チャネルの共有，非共有はステレオタイプに依存し，変換される．

- ⟨⟨shared_channel⟩⟩⟨⟨SYNC⟩⟩ →　chan 送信クラス名_受信クラス名=[0] of {mtype，引数の型 1，･･･ }
- ⟨⟨shared_channel⟩⟩⟨⟨ASYNC⟩⟩　→　chan 送信クラス名_受信クラス名=[バッファ数] of {mtype，引数の型 1，･･･ }
- ⟨⟨unshared_channel⟩⟩⟨⟨SYNC⟩⟩　→　chan 送信クラス名_受信クラス名 [多重度数]=[0] of {mtype，引数の型 1，･･･ }

- ⟨⟨unshared_channel⟩⟩⟨⟨ASYNC⟩⟩ → chan 送信クラス名_受信クラス名 [多重度数]=[バッファ数] of {mtype，引数の型 1，… }

ステレオタイプ shared_channel は，複数のオブジェクトが共有できるチャネルとして定義される．また，ステレオタイプ unshared_channel は複数のオブジェクト固有のチャネルとして，オブジェクトの数だけ定義される．

図 5.9 に変換例を示す.

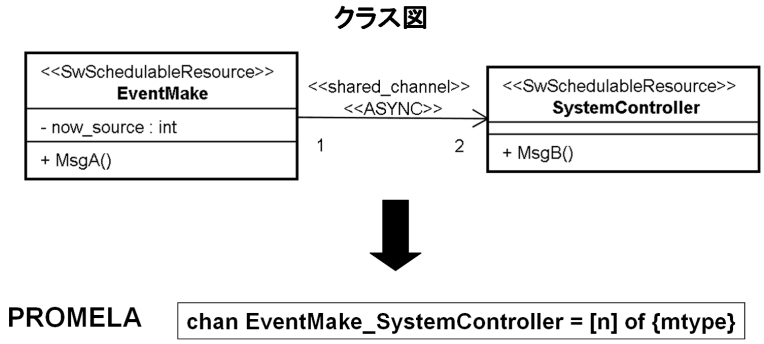


図 5.9: クラス図から PROMELA への変換例

5.3.4 ステートマシン図の変換

process に変換されるクラスに対しては，ステートマシン図が定義され，それは対応する process の本体部分にの PROMELA に変換される．

ステートマシン図で定義された状態遷移を実現するために，状態名に対応したラベルを設定し，イベント発生 (メッセージの受信等) に応じて対応するラベルにジャンプ (goto) するという形で実現する．

さらに，性質を表す時相論理式の中で，どの状態にいるかを示す変数を参照できると便利であるため，各状態に対応して「クラス名_状態名」という名称の bool 型グローバル変数を宣言する．

上記のラベルと goto 文で実現した状態遷移の処理ロジックの中で，ある状態に入ると対応する状態変数を true にし，状態から出ると false にする処理を含める．遷移は遷移元の状態中と解釈されるため，遷移完了直前に exit イベントが起き，その後，遷移元の状態変数が false にする．上記のような処理を行うことで，各状態

変数を参照することで状態中にいるかどうかを判断することができ，性質を表す論理式の記述に活用できる．

5.4 プロファイル

本検証パターンのための制約を UML プロファイルを利用して定義する．このプロファイルは，UML2.1.2 スーパーストラクチャー [16] を拡張しており，3 つのパッケージから構成される．5.4.1 節では静的構造，5.4.2 節では動的構造それぞれに適用するプロファイルについて述べる．

5.4.1 静的構造

最初のパッケージはクラス制約パッケージと呼ぶ．このパッケージはクラスに対する制約のステレオタイプを定義している．5.3.2 節の静的構造で説明したステレオタイプを定義している．UML での概要図を図 5.10 に示す．”Class”は UML2.1.2 に定義されている要素で，その要素に対して，”MandatoryClass”，”OptionalClass” というステレオタイプが付く．

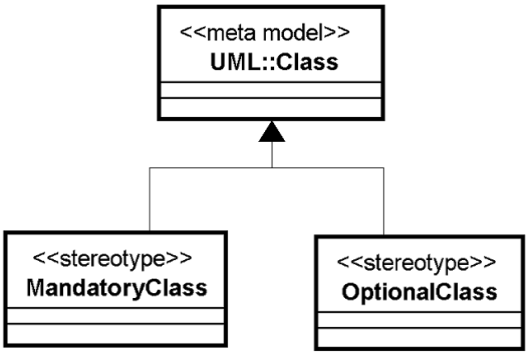


図 5.10: クラス制約パッケージ

5.4.2 動的構造

2 つ目のパッケージは状態制約パッケージと呼ぶ．このパッケージは状態に対する制約のステレオタイプを定義している．5.3.2 節の動的構造の (1) 状態で説明したステ

レオタイプを定義している．UMLでの概要図を図5.11に示す．”State”はUML2.1.2に定義されている要素で，その要素に対して，”MandatoryState”，”OptionalState”というステレオタイプが付く．

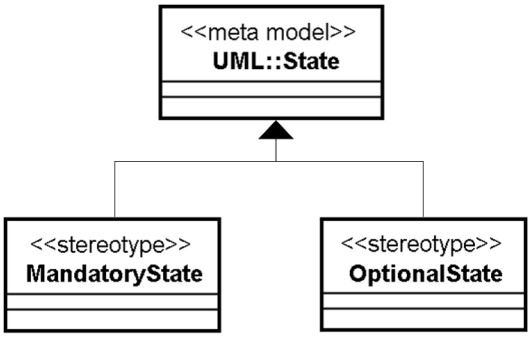


図 5.11: 状態制約パッケージ

3つ目のパッケージは，遷移制約パッケージと呼ぶ．このパッケージは遷移に対する制約のステレオタイプを定義している．5.3.2節の動的構造の(2)遷移で説明したステレオタイプを定義している．UMLでの概要図を図5.12に示す．”Transition”はUML2.1.2に定義されている要素で，その要素に対して，”MandatoryTransition”，”OptionalTransition”，”TwithRME”，”TwithSMA”というステレオタイプが付く．ただ，”MandatoryTrasition”は抽象的なステレオタイプなので，実際の設計モデル上では利用しない．

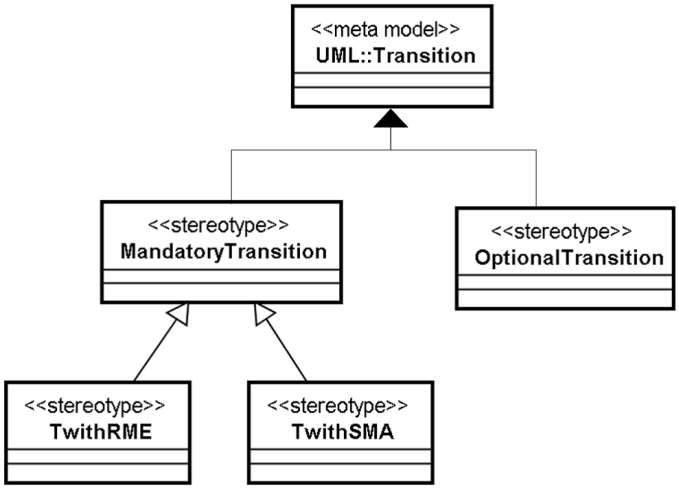


図 5.12: 遷移制約パッケージ

第 6章

アスペクト指向技術を利用した変更手法の提案

6.1 はじめに

本章では 2.2 節で指摘した横断的な変更に関わる問題への対応手法について提案する．2.2 節でも述べたように，検証では設計モデルに対して多くの変更を行う必要があり，多くの場合それは横断的である．そのため，我々はアスペクト指向技術を用いたアプローチをとった．本研究では UML での検査モデルに対する横断的な修正を扱うため，アスペクト指向言語ではなくアスペクト指向モデリングの技術を適用する．アスペクト指向モデリングの手法はいくつか提案されており，それらのひとつに WEAVR[20] がある．WEAVR はモデルの記述に SDL を用いているが，本研究ではモデル検査技術を適用するために有限状態モデルへのマッピングをするため，ステートマシン図を利用している点に特徴がある．

6.2 アスペクトを利用した変更手法のねらい

2.1 節で指摘したように，ひとつの検査モデルに対して検証目的に応じた修正を加えたい場合がある．前述の例でいえば，排他的リソースの取得の際にリソースがロックされている場合，そのセマフォを取得しようとしたタスクが，そのセマフォを開放するまで，待つ場合と待たない場合といったメカニズムが考えられ，そ

れらの両者に関して検証を行いたい場合，取得の箇所の修正が必要となるが，それらが複数個所に横断的に表れる場合がある．

こうした修正を効果的かつ確実に行うために，アスペクト指向技術の適用が有効であると考えられる．アスペクトは横断的な関心事をカプセル化する手段であり，リソース取得の例でいえば，取得方法を切り替える際に，それに対応したアスペクトを用意して検査モデルにウィーブすることで，複数個所の修正が行われ効果的である．

なお，こうしたメカニズムの切り替えなど，特定のドメインの検証を対象とすると，よく行われる横断的な変更をリストアップできると考えられる．こうしたよく使われるアスペクトを本研究ではアスペクトパターンと呼ぶ．

6.3 アスペクト指向 UML 検査モデル

6.3.1 概要

提案するアスペクト指向 UML 検査モデルは，AspectJ[26] などと同様にジョインポイントモデルに基づいている．以下，用語について簡単に説明する．

- ジョインポイント

  挿入文を挿入させることが可能な箇所

- ポイントカット

  ジョインポイントの集合から挿入させる箇所を抽出するための条件式

- アドバイス

  ジョインポイントに対する挿入文の位置関係

  AspectJでは，around(ジョインポイントと入れ替え), after(ジョインポイントの後に挿入), befor(ジョインポイントの前に挿入) などの位置指定ができる．

- 挿入文

  ジョインポイントに挿入される文

本提案では，こうしたジョインポイントモデルを，UML のステートマシン図に対して適用する．ステートマシン図への適用の例を，図 6.1 を用いて説明する．

図 6.1: アスペクトの例題

①がウィーブ前のモデルである．ジョインポイントがメソッド呼び出しと変数の代入である場合，例題上の”A.func1()”と”A.var =1”がジョインポイントになる．AspectJ の表記法に基づきポイントカットを，”call(A.func1())”と記述した場合，本例題では”A.func1()”が該当することになる．アドバイスが”around”，挿入文が”A.func2()”の場合，②に示すように”A.func1()”と”A.func2()”が入れ替わる．

提案するアスペクト指向 UML 検査モデルは，上記の例のようにステートマシン図に対してジョインポイントモデルを適用したものとなっている．これは本モデルはモデル検査技術を用いた検証のための検査モデルの記述を行うものであるため，ステートマシン図にアスペクト指向技術を適用することが便利であるからである．モデルの記法は，後述するように Stein ら [14] の提案をベースにし，本モデル向けに一部修正を行っている．6.3.2 節以降に，本アスペクト指向 UML 検査モデルの内容について説明する．

6.3.2 アスペクトの記述

本研究では Stein ら [14] により提案された UML 上でアスペクト記法を部分的に利用する．この記述は AspectJ で定義されている内容を UML 上で表現できるようにしている．この記法では構造に関するアスペクトと振る舞いに関するアスペクトとの二種類が提供されているが，本提案ではこのうち振る舞いに関する記述，具体的にはポイントカットとアドバイスの記法を利用する．一方，構造に関する記述，具体的にはイントロダクションの記述は利用しない．イントロダクションとは UML のテンプレートを利用して，継承先や属性などの構造の変更をしている．

また，Stein らは挿入文をシーケンス図で記述するように定義している．本研究ではシーケンス図ではなく，ポイントカットに属性”sentence”を追加し，この属性に挿入したい任意の文を代入する．

表記法を表 6.1 に，記述例を図 6.2 に示す．

表 6.1: アスペクトの表記法

要素	表記法
Aspect	<< *aspect* >> アスペクト名
PointcutPart	<< *pointcut* >>pointcut_任意のポイントカット名 (クラス名 仮引数) {base = プリミティブポイントカット (引数)}
AdvicePart	<< *advice* >>advice_id 任意の番号 アドバイス種類 (仮引数) {base = 任意のポイントカット名 (仮引数) ;sentence = 任意の文 }

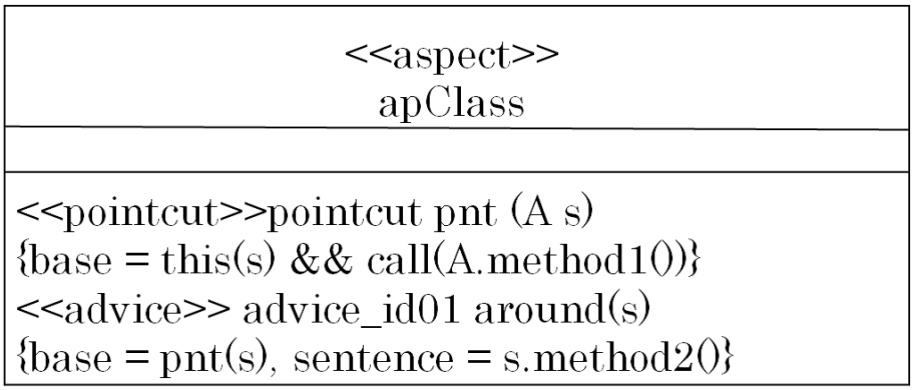


図 6.2: アスペクトの記述例

Aspect はクラス記号を用いて記述する．アスペクト名の上に，ステレオタイプ aspect を記述する．図 6.2 は，apClass というアスペクトの記述例である．アスペクトを表すクラス記号中には，ポイントカットとアドバイスが記述される．

- ポイントカット

ステレオタイプ pointcut の宣言に続けて”pointcut_任意のポイントカット名 (クラス名 仮引数)”と記述する．”pointcut”は接頭語である．記述例では，”pointcut_pnt (A s)”と記述している．

- ユーザーポイントカットの記述

  ユーザーが定義する任意のポイントカットを意味する．記述方法は，{} 内に”{base = プリミティブポイントカット (引数)}”と記述する．base は stein らによって拡張された，ステレオタイプ pointcut のメソッドが持つ，ポイントカットの式を代入する特別な属性であり，プリミティブポイントカットを || と&&でつないで記述する．記述例では，”{base = this(s) && (call(A.mehtod1()))}”の部分がこれに相当する．

- アドバイス

  ステレオタイプ advice の宣言に続けて”advice_id 任意の番号 アドバイスの種類 (仮引数)”と記述する．”advice”は接頭語である．記述例では，”advice_id01 around(s)”の部分がこれに相当する．

  - 対応するポイントカット

    {} 内に”base = ポイントカット名 (引数)”と記述する．base は stein らによって拡張された，ステレオタイプ advice のメソッドが持つ，ポイントカット名を代入する特別な属性である．記述例では，”base = pnt(s)”の部分がこれに相当する．

  - 挿入文

    {} 内に”sentence = 任意の文”と記述する．記述例では，CallOperation-Action の場合であり，”s.method2()”と記述している．複数の文を記述する場合は，; でつなげて記述する． 例えば，”s.method2();x++;”と記述する．

6.3.3 検査モデルへのアスペクトの利用例

本稿が提案しているアスペクトを利用した変更例を図 6.3 を用いて説明する．図 6.3 の左の図が変更前の検査モデルである．変更前の検査モデルは，リアルタイ

ム OS の仕様である μ ITORN[28] に準拠した OS を利用することを想定している．メッセージ通信を行っており，メッセージバッファを利用してメッセージを送る検査モデルである．

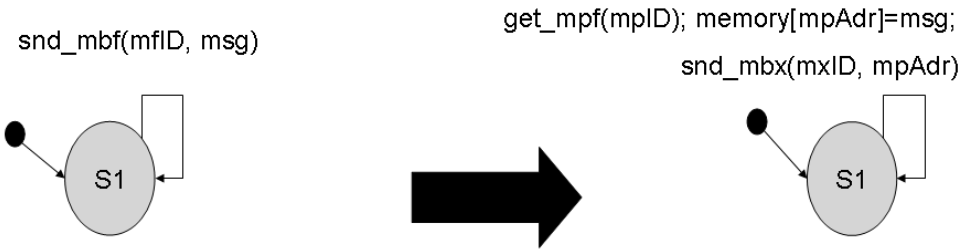


図 6.3: 検査モデルへのアスペクト利用例

この検査モデルの処理がメッセージバッファからメールボックスを利用するように変更することを考える．このために具体的に次の変更が必要である．

- サービスコールの変更

  メモリバッファを利用してメッセージ送信を行うサービスコール snd_mbf からメールボックスを利用してメッセージ送信を行う snd_mbx に変更する．

- メモリブロック取得の追加

  snd_mbf を利用する場合，自動的にメッセージを格納するメモリブロックを取得するが，snd_mbx を利用する場合，ユーザーがメッセージを格納するメモリブロックを取得する必要がある．そのため，メモリブロック取得の処理を追加する．

上記の変更するためのアスペクト記述を図 6.4 に示す．

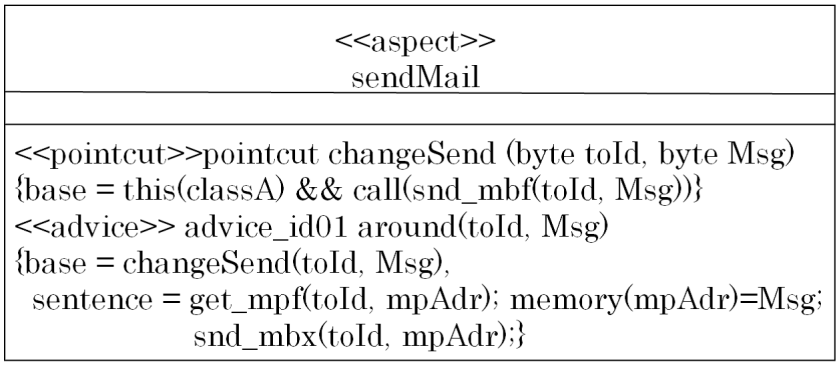


図 6.4: 通信メカニズムを変更するアスペクトの例

図 6.4 のアスペクトを図 6.3 の左のステートマシン図にウィーブすることにより，図 6.3 の右のステートマシン図に修正することができる．

なお図中のサービスコール，変数は μ ITRON の仕様書をベースに検証用にカスタマイズしたものである．サービスコール snd_mbf，get_mpf，snd_mbx はそれぞれ，メッセージバッファへの送信，メモリプールからのメモリブロックの取得，メールボックスへの送信である．引数 mfID，msg，mpID，mpAdr，mxID はそれぞれ，メッセージバッファの番号，メッセージ，メモリプールの番号，get_mpf で取得したメモリブロックのアドレス，メールボックスの番号である．変数 memory は，メモリを抽象化させた変数配列である．添え字を番地と考える．

6.3.4 アスペクトパターンの活用

機能の変更に使われるポイントカットやアドバイスには類似のものがたびたび使われるので，それをパターン化することでアスペクトの適用をさらに容易にすることができる．そのため，本研究ではパターン化するためのしくみを提案する．

本研究では定義したアスペクトを継承し，定義をオーバーライドすることによって既存のアスペクトをパターンとして流用できるようにする．下記の部分を継承して利用することができる．

- ポイントカットの条件式

  ポイントカットの属性 base の値

- 対応するポイントカット

  アドバイスの属性 base の値

- 挿入文

  アドバイスの属性 sentence の値

なお，パターンとして定義されるアスペクトは一部を未定義のままにしておくこともできる．この場合，必ず継承して利用することが必要である．

例としてを図 6.5 に排他的リソース取得方法の変更パターンを示す.

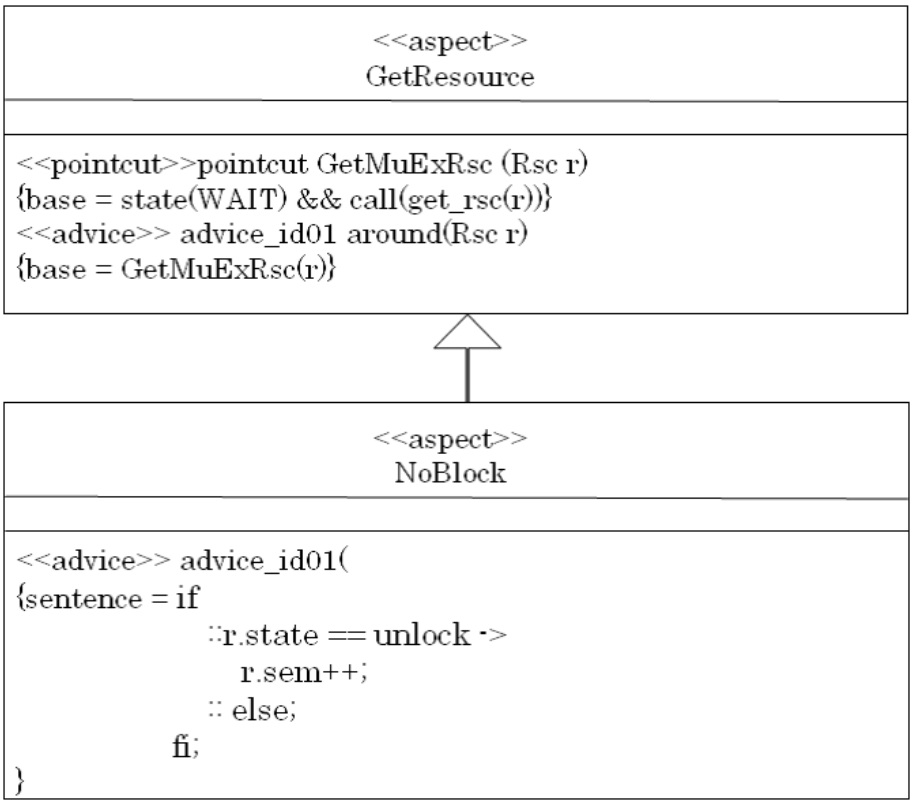


図 6.5: 排他的リソース取得方法の変更パターン

この例題の場合，オーバーライド可能な部分は，ポイントカットの属性 base の値である”state(WAIT) && call(get_rsc(r))”, アドバイスの属性 base の値である”GetMuExRsc(r)”になる．オーバーライドが必ず必要なのは，定義されていないアドバイスの属性”setence”である．

6.3.5 メタモデル

本節では提案するアスペクト指向モデルのメタモデルについて説明する．図 6.6 はアスペクトの定義部分である．

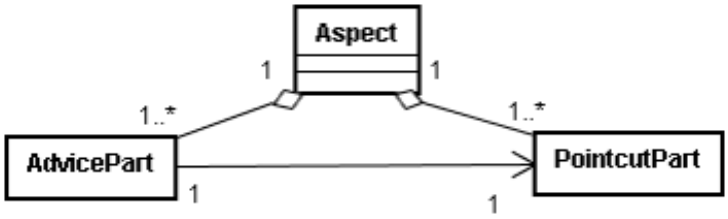


図 6.6: アスペクトの定義

アスペクトは，アドバイス部分とポイントカット部分によって，構成される．それぞれ，AdvicePart，PointcutPart と呼ぶ．AdvicePart は，適応する PointcutPart と対応付けられる．

図 6.7 は PointcatPart の定義である．

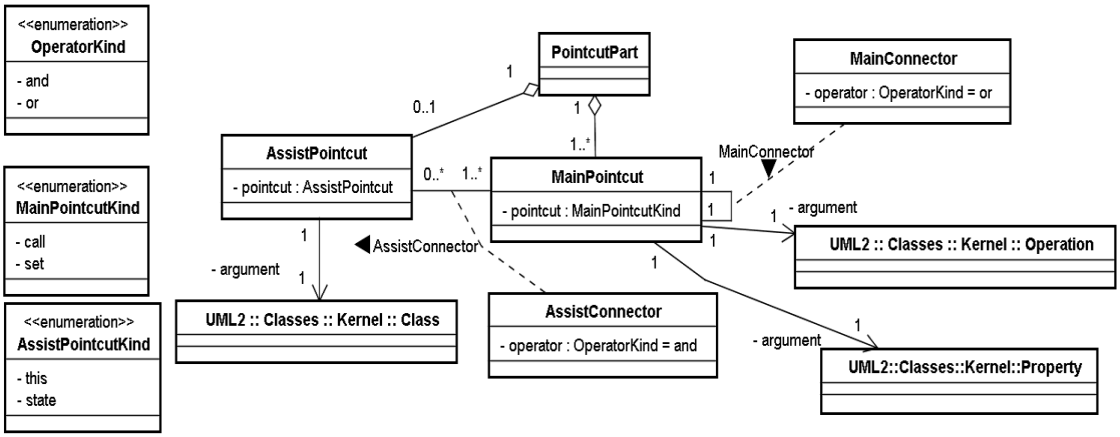


図 6.7: ポイントカットの定義

PointcutPart は，１つまたは複数の MainPointcut を持つ．ポイントカットには，事前に用意されているプリミティブポイントカットとプリミティブポイントカットを組み合わせた名前付ポイントカットがある．MainPointcut は，MainPointcutKind で定義されたプリミティブポイントカットに対応する．MainPointcutKind では，Operation の呼び出しをジョインポイントとするプリミティブポイントカット call と

Property への代入をジョインポイントとするプリミティブポイントカット set を定義している．Operation は，UML2.1.2 で定義されている要素で，本研究ではメソッドを指す．同様に，Property は UML2.1.2 で定義されている要素で，本研究では変数を指す．MainPointcut の引数として，Operation と Property を取る．つまり，本研究としてのジョインポイントは，Operation を呼び出すアクションか Property に代入するアクションであることを意味する．PointcutPart が MainPointcut を複数持つ場合，MainPointcut 間は or の関係になる．

PointcutPart は，複数の AssistPointcut を持つことが可能である．AssistPointcut は，AssistPointcutKind で定義されたプリミティブポイントカットに対応する．AssistPointcutKind では，プリミティブポイントカット this，state を定義している．this は，アクションを実行している Class のジョインポイントに限定するプリミティブポイントカットである．Class は，UML2.1.2 で定義されている．state は，クラスが特定の状態である時のジョインポイントに限定するプリミティブポイントカットである．AssistPointcut は，MainPointcut で限定したジョインポイントをさらに限定するポイントカットである．必ず，MainPointcut と一緒に利用される．AssistPointcut の引数として，Class を取る．MainPointcut と AssistPointcut 間は and の関係になる．

図 6.8 は AdvicePart の定義である．

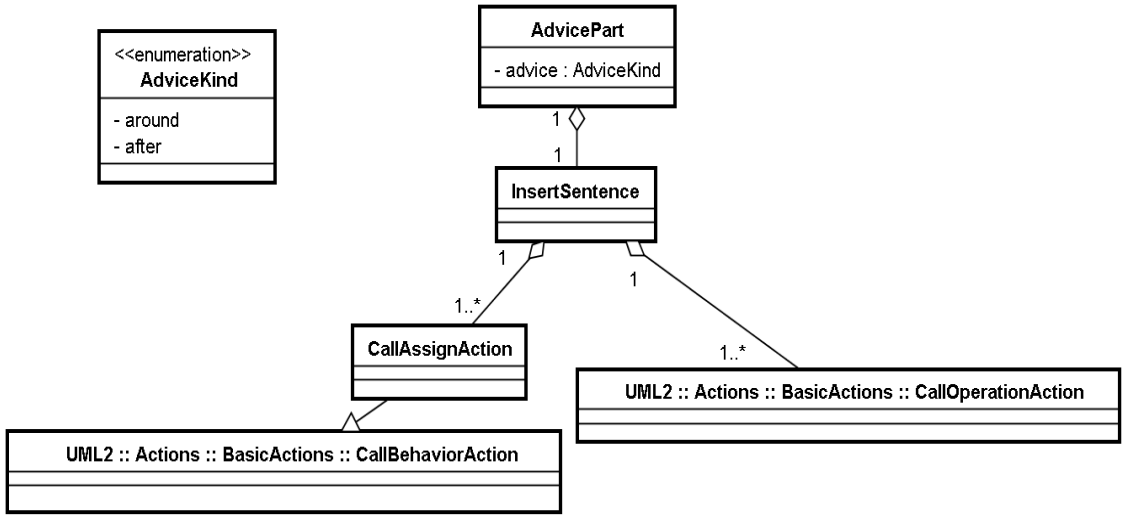


図 6.8: アドバイス部分の定義

AdvicePart は，１つのアドバイスを持ち，AdviceKind で定義されたアドバイス種類に対応する．AdviceKind では，ジョインポイントと挿入文をそっくり入れ替えるアドバイス種類 around，ジョインポイントの後に挿入文を追加する after を定義している．また，AdvicePart は１つの InsertSentence を持つ．InsertSentence は，ジョインポイントに挿入される文を意味する．InsertSentence は，CallassignAction と UML2.1.2 で定義されている CallOperationAction から構成される．CallAssign-Action は，代入文を実行するアクション，CallOperationAction は，Operation を実行するアクションである．

6.4 検証支援環境

6.4.1 全体像

提案したアスペクト指向 UML 検査モデルを用いた検証手順を図 6.9 に示す．なおここでアスペクト PROMELA は，PROMELA をアスペクト指向拡張した言語である (6.4.2 節参照).

(1) 本モデルを用いて記述された検査モデルを Promala に変換する．ここではアクティブなクラスは Promela のプロセスに，アクセスパスとなる関連は Promela のチャネルに，ステートマシン図は対応する Promela の内部コードに変換される．変換方法は岸らのツールの規則 [24][15] とほぼ同様である．

(2) 検査モデルに対して修正が必要な場合は，その修正を行うアスペクトを定義する．定義に際してはアスペクトパターンを継承して特殊化して定義してもよいし，新規にアスペクトを定義してもよい．記述されたモデルは 6.4.2 節で述べる変換規則に基づいてアスペクト PROMELA に変換されえる．

(3) (1) と (2) で変換したものをアスペクト PROMELA のウィーバーに入力する．

(4) アスペクト PROMELA のウィーバーによって，自動でウィービングされた PROMELA が生成される．

本検証支援環境は，この手順のうち (1) と (2) の部分を支援する．(1) は今回は手動で行ったが，岸らのツール [24][15] とほぼ同様の技術でツール化することが可能である．

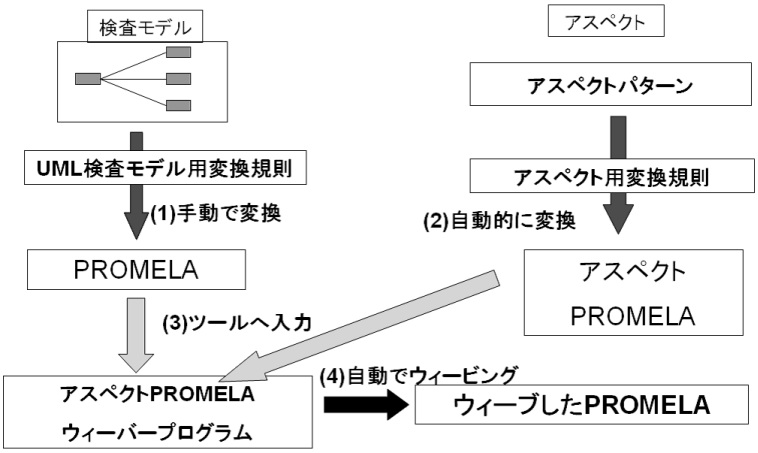


図 6.9: アプローチの全体像

6.4.2 アスペクト PROMELA への変換

アスペクト PROMELA

本支援環境は，アスペクト指向化された PROMELA 言語を内部的に用いて実現されている．以下，その概略を説明する．アスペクト PROMELA は，大野ら [13] により提案されたアスペクト指向 PROMELA 言語である．特徴としては，AspectJ と同様 joinpoint モデルを採用している．以下に記述例を示す．

```
around :
allStmnt(proctype("A")) && chan("ch","!","")
{# ch!msg1 #}
```

1 行目がアドバイス，2，3 行目が pointcut，4 行目が挿入文になる．この例の意味は，”proctype A 内のチャネル ch を使ったメッセージ送信全てを，ch!msg1 に置き換える”となる．

変換規則

プリミティブポイントカットは同様の意味を表すアスペクト PROMELA の記述に変換される．以下にその対応を示す.

- call
  - label(”_クラス名_メソッド名”)
- set
  - label(”_クラス名_変数名”)
- this
  - allStmnt(proctype(”クラス名”))
- state
  - allStmnt(label(”クラス名”))

アスペクトの各定義とアスペクト PROMELA への対応関係を述べる．アドバイス種類は，1 行目の先頭，”:”をはさんで，ポイントカットが生成される．2 行目からは”{##}”の中に挿入文が生成される.

アスペクトの UML 表記からアスペクト PROMELA への変換例を示す．変換前のアスペクトの UML 表記を図 6.10 に示す.

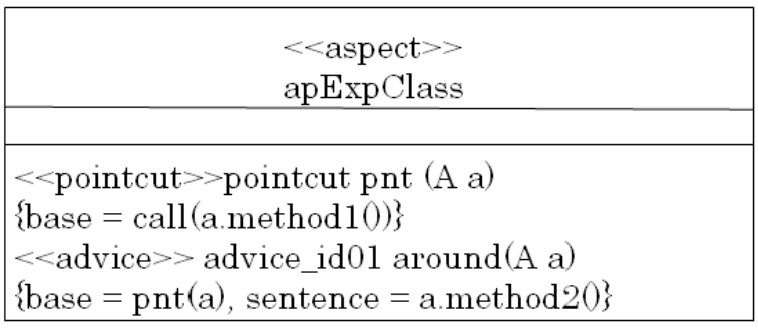


図 6.10: アスペクト UML 表記の記述例

変換後のアスペクト PROMELA の記述例を下記に示す.

```
around : label("_A_method1")
{#  A.method2() #}
```

6.4.3 ツール上の制約

この節では，6.4.1 節で述べた (2) で利用するツールへの記述方法について述べる．本支援環境では，利用した UML エディタの制約上，6.3.2 節での記法を一部以下のように変更している．

6.3.2 節で述べたアスペクト記述との相違について述べる．Aspect，AdvicePart，PointcutPart の各ステレオタイプはタグ値として記述する．他の記述は 6.3.2 節で述べた記述と同じである．

図 6.10 の例の場合，アスペクトは図 6.11 のように記述する．

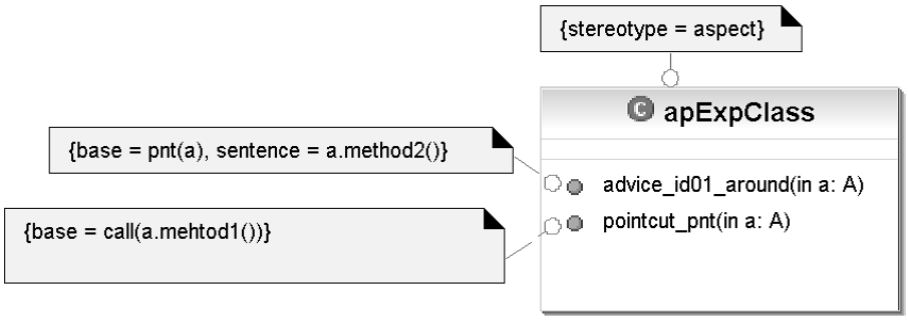


図 6.11: 検証支援ツールでの記述例

6.4.4 継承の支援

本ツールで支援する継承の種類は 2 つある．1 つ目はアスペクトクラスの継承，2 つ目はウィーブ対象になるクラスの継承である．1 つ目のアスペクトクラスの継承を支援している理由は，ポイントカット，アドバイス，挿入文の再利用をよりしやすくなるメリットがあるためである．本研究で仕組みを提案するアスペクトパターンは，アスペクトクラスの継承を利用することを前提としている．2 つ目のウィーブ対象になるクラスの継承を支援している理由は，本研究の意図として

ポイントカットに定義されたクラスのサブクラスもそのポイントカットに適用する必要があるためである．AspectPromela はサブクラスまで適用しないため，本ツールを利用することでサブクラスまで適用できるようにしている．

アスペクトの継承

本研究で提案しているアスペクトパターンは継承を用いて，利用することを想定している．本ツールでパターンを利用した例を図 6.12 に示す．この例の場合，アスペクトクラス BlockGet はアスペクトクラス GetResource からアドバイスの種類である”around”，ポイントカットの式”state(WAIT) && call(get_rsc(r)))”を引き継ぐ．そして，アスペクトクラス BlockGet は，挿入文である”getBlcRsc(r)”を追記している．

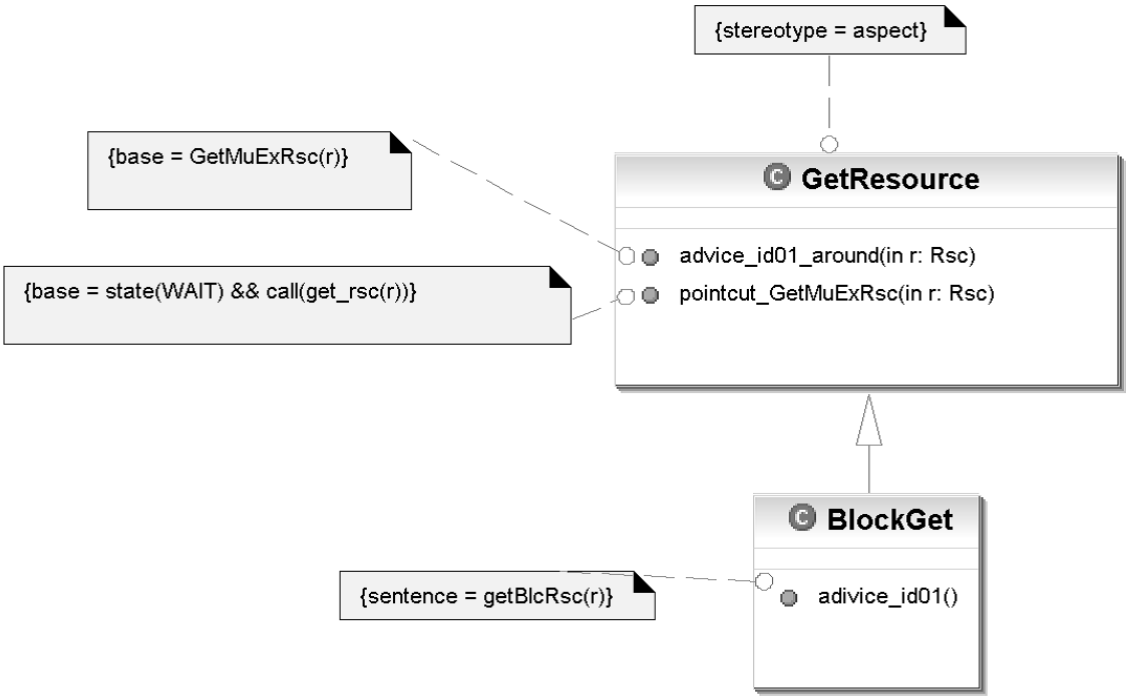


図 6.12: 検証支援ツールでのアスペクトパターンの利用例

クラスの継承

UML 上でスーパークラスがポイントカットの対象となった場合，その修正はサブクラスにも及ぶが，アスペクト PROMELA は継承の機能をもたないため，ツー

ルでそれを補う必要がある．具体的には，スーパークラスに対するポイントカットが定義された場合，そのサブクラスに対するポイントカットを生成する支援をする必要がある．例を図 6.13 に示す．この場合，クラス A はクラス B のスーパークラスなので，A をポイントカットの対象とした場合，B も必然的にポイントカットの対象となる．そのため，２つアスペクト PROMELA の記述が必要になる．

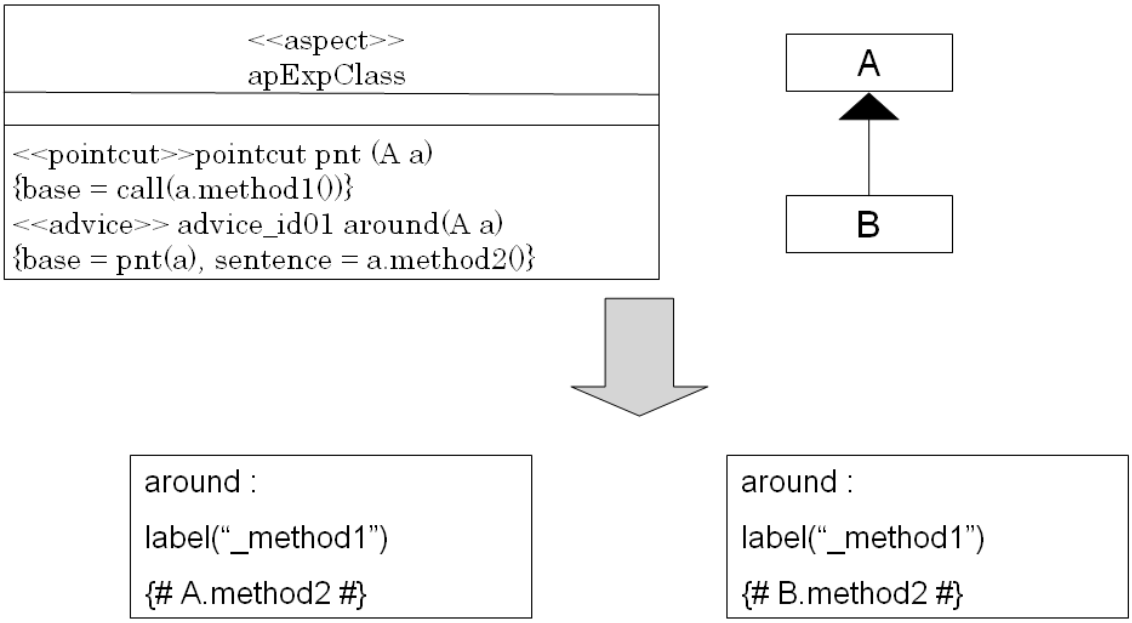


図 6.13: サブクラスに対応する例

第 7章

検証指向設計手法

3.3 節でも触れたが，設計検証は場当たり的に行っても，検証に必要な要素がモデル化できていないと容易に検証することは難しい．そのため，検証を考慮した体系だった設計手法が必要である．この章では，本稿で述べている 2 つのアプローチを利用した検証を考慮した設計手法を提案する．まず，7.1 節で検証容易性について説明する．検証容易性とは検証のしやすさのことである．次に，7.2 節では検証指向設計手法の内容について述べる．この節では，各手順での作業内容だけでなく，7.1 節で述べた検証容易性に対して，どのように有効化を述べる．

7.1 UML 設計検証における検証容易性

Bach はテストのしやすさの特性をテスト容易性として定義している [27]．本研究ではモデル検査のしやすさを検討するにあたり，Bach のテスト容易性を参考にして，UML 設計に対するモデル検査のしやすさの特性を検証容易性として定義した．本検証容易性のうち (1)〜(4) は検査モデル（クラス図・ステートマシン図）の性質であり，(5) は検証性質記述の性質である．

(1) 制御容易性

変更可能な個所が局所化されている性質である．

一般に検証時には複数の性質を検証するが，性質に応じて検証モデルを変更することがある．

こうした変更が局所化されていれば性質に応じた変更が容易になり，検証も容易になる．岸らも変更を繰り返し検証することの必要性を提示している [24]．UML においては，具体的に　以下のような性質として表される．

- クラス図
  - 変更する変数が局所化されている．
  - 変更するメソッドが局所化されている．
  - 変更するメッセージが局所化されている．
- ステートマシン図
  - 変更する状態が局所化されている．
  - 変更するイベントが局所化されている．
  - 変更するアクションが局所化されている．
  - 変更するガードが局所化されている．

(2) 観測容易性

検証に必要な情報を観測しやすい性質である．

ある性質を記述する時に，あるアクションが実行されたことやある状態になったことを利用して記述する．こうした，アクションや状態などを容易に観測できれば検証も容易になる．Gallardo らも観測可能にする設計方法の有効性を提示している [29]．UML においては，具体的に　以下のような性質として表される．

- クラス図
  - 検証対象の変数が定義されている．
- ステートマシン図
  - アクションの完了を観測できる状態を定義している．
  - イベントの完了を観測できる状態を定義している．

(3) 可用性

円滑に検証をしやすい性質である．

検証と関係ないのない事によって，検証が阻害されないようになっているかを示している．Holzmann は検証性質に関係ない要素をできるだけ抽象化することが必要であることを提示している [2]．

（a）検証が途中で止まることはない．

ある検証をしている時，その検証の本質的でない処理で止まってしまった場合，きちんと検証ができない．したがって，この検証の本質的な部分以外は正常に動作することが円滑な検証をもたらす．UML においては，具体的に以下のような性質としてあらわされる．

- クラス図
  - ∗ 途中で止まるメソッドはない．
- ステートマシン図
  - ∗ 起こりうる全てのイベントが遷移に対応している．
  - ∗ 起こりうる全てのガードが遷移に対応している．

（b）検証性質に関係のない要素は定義されていない．

この検証には関係ない変数や状態などがあると，検証は複雑になり，検査モデルや検証性質の作成を複雑にさせる．したがって，検証に関係のない要素を記述しなければ，検証をより容易に行うことができる．UML においては，具体的に　以下のような性質としてあらわされる．

- クラス図
  - ∗ 検証性質に関与していない変数はない．
  - ∗ 検証性質に関与していないメソッドはない．
- ステートマシン図
  - ∗ 検証性質に関係のない状態は無い．
  - ∗ 検証性質に関係のないアクションはない．
  - ∗ 検証性質に関係のないイベントはない．

* 検証性質に関係のないガードはない.

(4) 安定性

変更が少ない性質である.

変更が少ないことによって，検証がしやすくなることを示している．Aygün らはモデルを変更しても、検証性質を変更しないことの有効性を提示している [17].

（a）モデルの変更によって，検証記述の変更はない.

モデルを変更する度に LTL やアサーションなどの検証記述を変更しなければならない場合，円滑な検証するのは難しい．モデル検査の場合，部分的に変更して検証を繰り返すことはよくあるためである．したがって，モデル変更によって，検証記述の変更がなければ，検証をより容易に行うことができる．UML においては，具体的に　以下のような性質としてあらわされる.

- クラス図
  - * LTL で利用されている変数は変更されない.
  - * アサーションで利用されている変数は変更されない
- ステートマシン図
  - * LTL に利用している状態は，変更されない.
  - * アサーションに利用している状態は，変更されない.

（b）アクターを再利用して，検証ができる.

同じような入力列に対する検証を行いたいことがよくある．アクターは，検証において再利用しやすい要素と言える．したがって，きちんとアクターを再利用できるようにモデル化していると検証をより容易に行うことができる．UML においては，具体的に　以下のような性質としてあらわされる.

- クラス図
  - * メソッドの名前，引数，返り値を変更しない.
- ステートマシン図

* アクターから受信するメッセージを変更しない.
* アクターに送信するメッセージを変更しない.

(5) 簡潔性

LTL を簡潔に作成できる.

複雑な LTL を必要とせず，簡潔に記述することができる．LTL では，一見単純な性質でも式が複雑になることがあり，非常に難解である．したがって，体系的により簡潔に記述することが，検証をより容易に行うことができる．Dwyer らもパターンの利用による簡潔性の有効性を提示している [5]．具体的には以下のような性質としてあらわされる.

- LTL
  * 簡潔な LTL で記述できる.

7.2 検証指向設計手法

7.2.1 概要

本手法における設計の手順をフローチャートで図 7.1 に示す.

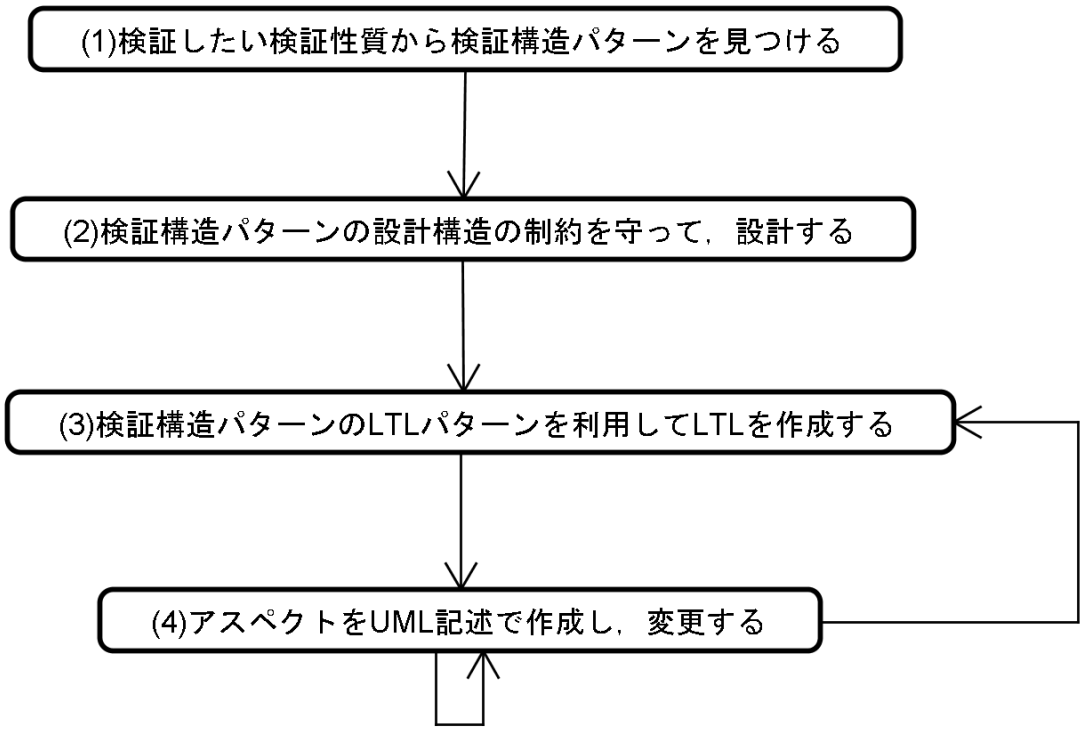


図 7.1: 設計手順

図 7.1 を参照しながら，本手法の概要を説明する．本手法では，(1) から (4) を順に行う．番号 (1) から (4) は図中の番号と対応する．

「(1) 検証したい検証性質から検証構造パターンを見つける」では，検証者が検証したい性質に合致するパターンを検証構造パターンから見つける．すなわち，パターンの検証目的に，検証者が検証したい性質が含まれているパターンを，検証構造パターンのカタログから見つける．問題の構造がパターンの構造と合致しているかどうかを確認する．

「(2) 検証構造パターンの設計構造の制約を守って，設計する」では，まず制約を守って静的な構造についてクラス図を作成する．制約を守ってステートマシン図が必要なクラスがあれば，それに対してステートマシン図も作成する．

「(3) 検証構造パターンの LTL パターンを利用して LTL を作成する」で LTL を作成する．検証構造パターンに定義されている LTL パターンを利用する．述語の置き換えの部分をユーザー毎に変更して利用する．置き換えは，(2) で作成した状

態毎に生成される状態変数で置き換える．また，式自体はクラスの多重度などに合わせて，パターンで定義されている規則にしたがって記述する．

「(4) アスペクトを UML 記述で作成し，変更する」でアスペクトパターンを利用する場合は，再利用できるアドバイスやポイントカットの部分を見つけて，適用するアスペクトパターンを見つける．特殊化したいアドバイスやポイントカットをオーバーライドして特殊化する．アスペクトパターンを利用しない場合は，最初からアスペクトを UML 記述で作成する．

この手順の具体的な適用方法は設計対象や開発の状況によって異なるが，ひとつの典型的な適用方法を以降の節で説明する．

7.2.2 検証性質の発見

この節では,「(1) 検証したい検証性質から検証構造パターンを見つける」の内容について述べる．

検証者が検証したい性質に合致するパターンを検証構造パターンから見つける．どのように見つけて行くか述べる．この手順に進む前に要求仕様書ができている前提とする．

まず，検証性質から適するパターンを見つける．パターンの中で，検証性質を自然言語で直感的に記述しているのでそれを理解し，最終的に適用するパターンを見つける．次に，クラスの役割に注目する．パターンに定義してあるクラス図からは，その属性やメソッドを理解する．また，ステートマシン図からは，各クラスの振る舞いを理解し，各クラスの役割を理解する．作成しようとしている設計モデルに対応できるクラスがあるか確認する．次に，クラスの動的意味に注目する．本パターンで定義している動的意味は前述した通り 2 つある，並行動作単位と排他的リソースである．それぞれ，ステレオタイプ $<< SwSchedulableResource >>$, $<< SwMutualExclusionResource >>$ で表現されている．作成しようとしている設計モデルに対応できるクラスがあるか確認する．次に，必須クラスと選択クラスに注目する．それぞれ，ステレオタイプ $<< MandatoryClass >>$, $<< OptionalClass >>$ で表現されている．作成しようとしている設計モデルに対応できるクラスがあるか確認する．

そして，次に関連に注目する．先ほどクラスの役割に注目して，パターンのクラスと対応付けしたユーザーの設計モデルのクラスが，パターンと同じクラス間の関連を持っているか確認する．次にその関連の役割に注目する．メッセージ送信に関する関連では，本パターンは2つのことに関して区別している．1つ目は，通信方法である．同期，非同期はそれぞれステレオタイプ $<< SYNC >>$，$<< ASYNC >>$ で表現されている．作成しようとしている設計モデルに対応できる関連があるか確認する．2つ目は，チャネルである．同期，非同期はそれぞれステレオタイプ $<< SYNC >>$，$<< ASYNC >>$ で表現されている．作成しようとしている設計モデルに対応できる関連があるか確認する．次に多重度に注目する．パターンのクラスと役割を対応させているユーザーのクラスの多重度を比較する．パターンのクラスが想定している多重度の範囲であるか確認する．

7.2.3 制約に基づく設計方法

この節は,「(2) 検証構造パターンの設計構造の制約を守って，設計する」の内容について述べる．どのように設計して行くか述べる．

まず，クラス図を作成する．必ず，パターンのクラス図に定義されたメソッドと属性を持ったクラスを作成する．これによって，パターンのクラスとユーザーが作成するクラスで対応関係が決定する．次に，その対応関係に基づいて，パターンと同じ関連を持つように，関連を作成する．その時，メッセージ通信に関する関連の場合，通信方法に注意して作成する．ステレオタイプ $<< SYNC >>$ の時は同期，$<< ASYNC >>$ の時は非同期である．次に，多重度を決定する．また，パターンと設計モデルの対応関係に基づいて，パターンで定義されている多重度の範囲を超えないように決定する．

次に，ステートマシン図を作成する．パターンとの対応関係に基づいてステートマシン図の必要なクラスを決定する．次に，そのクラスに対してステートマシン図を作成していく．パターンのステートマシン図に記述されている必須状態に対応する状態を作成する．ステレオタイプ $<< MandatoryState >>$ のついている状態である．次に，任意状態と遷移を作成する．パターンで定義されている，必須状態と任意状態の関係を守って遷移を作成する．ステレオタイプ $<< OptionalState >>$

がついている状態が任意状態である．この時注意するのは，必須遷移と選択遷移である．必須遷移はステレオタイプ $<< TwithRME >>$，$<< TwithSMA >>$ の付いた遷移である．$<< TwithRME >>$ は，遷移にイベント RecieveMessageEvent が，$<< TwithSMA >>$ は，遷移にアクション SendMessageAction が必要である．選択遷移はステレオタイプ $<< OptionalTransition >>$ の付いた遷移である．パターンで必須状態間または，必須状態と選択状態の間に必須遷移がある場合，必ず制約を守って必須遷移を作成する．

この制約を守ることで，(3) で作成する LTL で利用する状態を作成する．それにより，述語で定義されることが容易に観測でき，検証容易性の観測容易性に貢献している．

7.2.4 検証性質の作成

この節では,「(3) 検証構造パターンの LTL パターンを利用して LTL を作成する」の内容について述べる．どのように LTL を作成するか述べる．

まず，述語と状態変数を対応させる．状態変数は，先ほどステートマシン図で作成した状態１つ１つに対応しているものである．”#define ”を用いて，述語と状態変数を対応させる．次に，その述語の数に合わせて，LTL 式自体を作成していく．述語の数に合わせて，LTL を作成する方法はパターンに定義しているので，それにしたがって LTL を作成していく．

上記のようにあらかじめ簡潔に定義されているパターンの LTL を利用することで複雑な式にならず，簡潔な式を作成できるようになる．これにより，検証容易性の簡潔性に貢献している．

7.2.5 モデルの変更方法

この節では「(4) アスペクトを UML 記述で作成し，変更する」 の内容について述べる．どのようにアスペクトを作成するか述べる．

最初に，ポイントカットを作成する．まず，変更したいメソッド呼び出しまたは，変数への代入を決める．そして，メソッド呼び出し，変数へ代入はそれぞれ，プリミティブポイントカット call()，set() を用いて，作成する．また，ポイント

カットをもっと絞りたい場合は，プリミティブポイントカット this()，state() を用いる．this() は対象のクラスを限定する．state() は対象の状態を限定する．必ず，call() または set() に対して，&&で接続して利用する．

次にアドバイスを作成する．アドバイス種類は around か after を選択する．ジョインポイントと入れ替えたければ around，後に挿入したければ after を選択する．次に挿入文を作成する．

アスペクトパターンを利用する場合は，再利用できるアドバイスやポイントカットの部分を見つけて，適用するアスペクトパターンを見つける．特殊化したいアドバイスやポイントカットをオーバーライドして特殊化する．

アスペクトを用いることで，修正箇所が局所化されるので，検証容易性の制御容易性に貢献している．

第 8章

評価

評価の目的は，7.1 節で述べた検証容易性を用いて，本手法を利用した場合と利用しなかった場合でどちらが，より検証容易であるか評価することである．

評価は事例に適用して行う．具体的な評価方法について 8.1 節で，評価に利用する事例を 8.2 節で述べる．また，8.3 節で評価結果について述べる．

8.1　各特性毎の評価目的と方法

8.1.1　制御容易性の評価

評価目的

制御容易性の評価を行う．具体的には，モデルを変更するにあたって，追加，変更する箇所の数を比べて，どちらが局所的になっているか評価する．

評価方法

検査モデルに対して，次の変更を行う．

- 呼び出すメソッドの変更
- アサーションの挿入

例えば，図 8.4 の車内ライティングシステムの場合を考える．事例の詳細については，8.2 節で後述する．メソッド SPL をメソッド SPL2 に変更することを考える．例として，図 8.1 に示す BSAVER クラスのステートマシン図で考える．

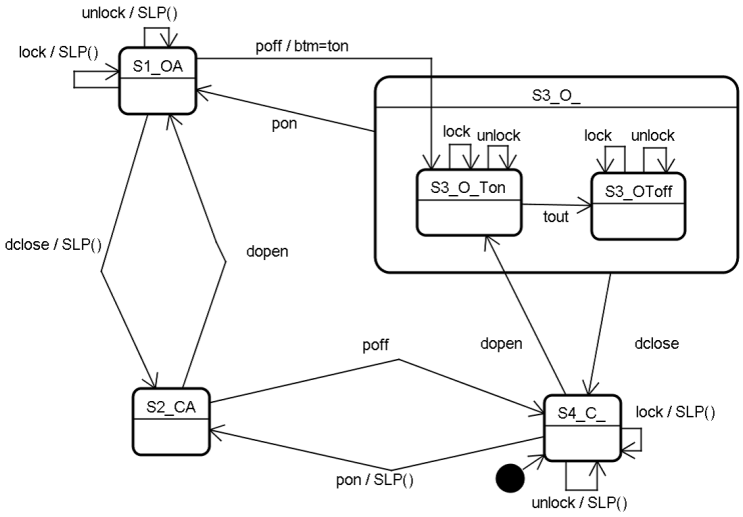


図 8.1: BSAVER クラスのステートマシン図

変更後は図 8.2 のようになる．

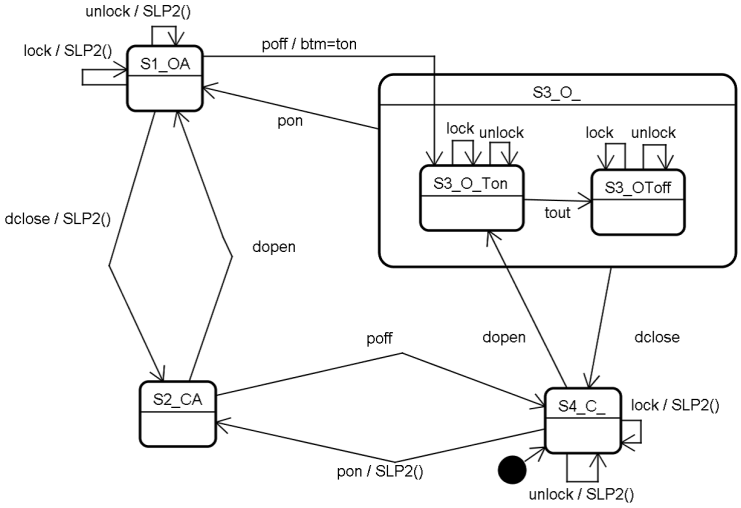


図 8.2: 変更後

これを本稿で提案しているアスペクトを利用した変更手法を利用すると，図 8.3 のように記述できる．

図 8.3: アスペクの記述

以上のように，この場合だと手動で変更すると 6 箇所の変更が必要だが，アスペクトを用いると１つのアスペクトを記述するだけで対応ができる．本手法を利用しない場合とした場合の記述箇所の数を比べ，評価する．

8.1.2 観測容易性の評価

評価目的

観測容易性の評価を行う．具体的には，LTL の述語の定義に参照される変数，ラベル，演算子の数を比較し，どちらが単純に定義できているか評価する．

評価方法

モデル化の仕方によって，事象の観測する方法が違う．検証のことを考慮せずモデル化すると複雑な観測方法になる場合がある．

例えば，以下の述語は同じ事象を意味しているとする．

```
#define p (x>=10) && (x<=20) && C1@L1


#define p state1
```

以上のように，この場合だと上の述語は，変数2つ，演算子4つ，ラベル1つであるのに対し，下の述語は変数が1つであり，より下の述語のほうが単純に定義ができていると言える．このような，述語の定義の複雑さの違いを評価する．

8.1.3 簡潔性の評価

評価目的

簡潔性の評価を行う．具体的には，LTL 自体の項と演算子の数を比較し，どちらが簡潔に定義できているか評価する．

評価方法

次に，簡潔性ではLTL式自体の複雑さを評価する．例えば，以下のようなLTLを考える．

①

```
(((!q && !r && !s && !t && !u) U p)
     && ((!r && !s && !t && !u ) U q)
     && ((!s && !t && !u ) U r)
     && ((!t && !u ) U s)
     && (( !u ) U t)  && <>u
```

②

```
(!q U p) && (!r U q)
     && (!s U r) && (!t U s) && (!u U t)  && <>u
```

①と②は同じ意味のLTLである．しかし，①の項は21個，演算子は36個，②の項は10個，演算子は16個であり，②のほうがより簡潔であると言える．このような，LTLが記述しやすいように，より簡潔にわかりやすく記述できているかを評価する．

8.2 事例

1つ目の事例は企業から提供いただいた自動車の室内灯を制御するライティングシステムである．このシステムではドアセンサ (DSENSOR) と電源センサ (PSENSOR) をセンサ監視機能 (SFIX) が定期的に監視して状況の変化を把握し，それに基づき，ドア開閉に基づく点灯判断 (DCTRL)，電源保護からの点灯判断 (BSAVER)，消灯遅延判断 (TCTRL) がそれぞれライトの制御判断をし，最終的に点灯制御機能 (LCTRL) がそれらを総合してライトの制御を行う．図 8.4 にこのシステムのクラス図を示す．

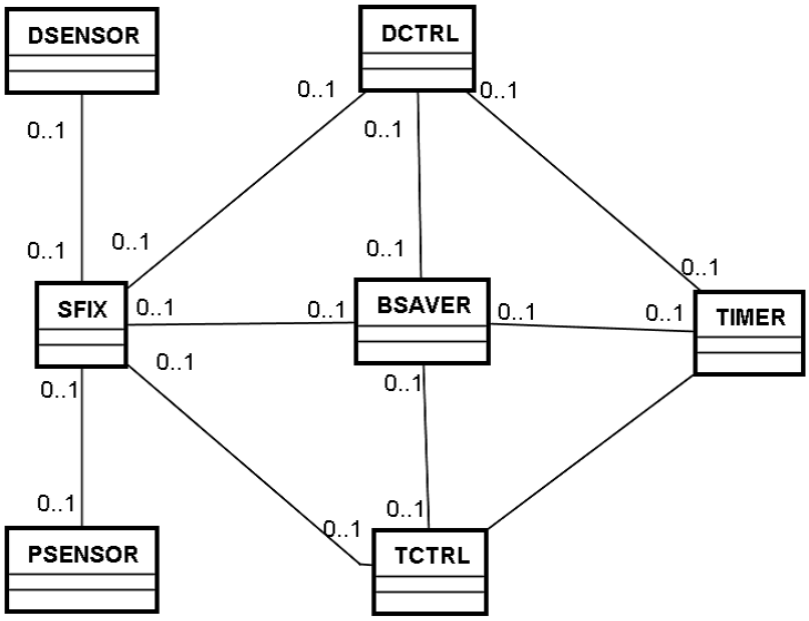


図 8.4: 車内灯ライティングシステム

2つ目の事例は企業から提供いただいた自動車のオーディオを制御するシステムである．このシステムではボタン入力の入力があると割り込み (Interupt) が発生し，イベント制御 (ApEventMakeCtrl) に通知する．その通知は非同期で行われ正常に遅れなかった場合，タイマー (Timer) を利用して再通知される．イベント制御からその通知を元にイベントが作られ，ソース切り替え制御 (ApSourceChangeCtrl)，メニュー制御 (ApMenuCtrl)，CD 制御 (ApCDCtrl)，音量制御 (ApVolumeCtrl) に通知される．その後各制御が関係クラスにイベントを通知する．CD 制御は CD 再生制御 (ApCdCtrl1)，CD リピート制御 (ApCdCtrl2) に，音量制御は電子音量制御

(EvolMecha) に通知する．また，CD 再生制御と CD リピート制御は CD メカ制御 (CdMecha) に，CD メカ制御と電子音量制御は電源制御 (IcPwr) にそれぞれ通知する．このシステムの検証には，青木らが提案している RTOS をエミュレートした SPIN 用ライブラリ [30] を利用する．図 8.5 にこのシステムのクラス図を示す.

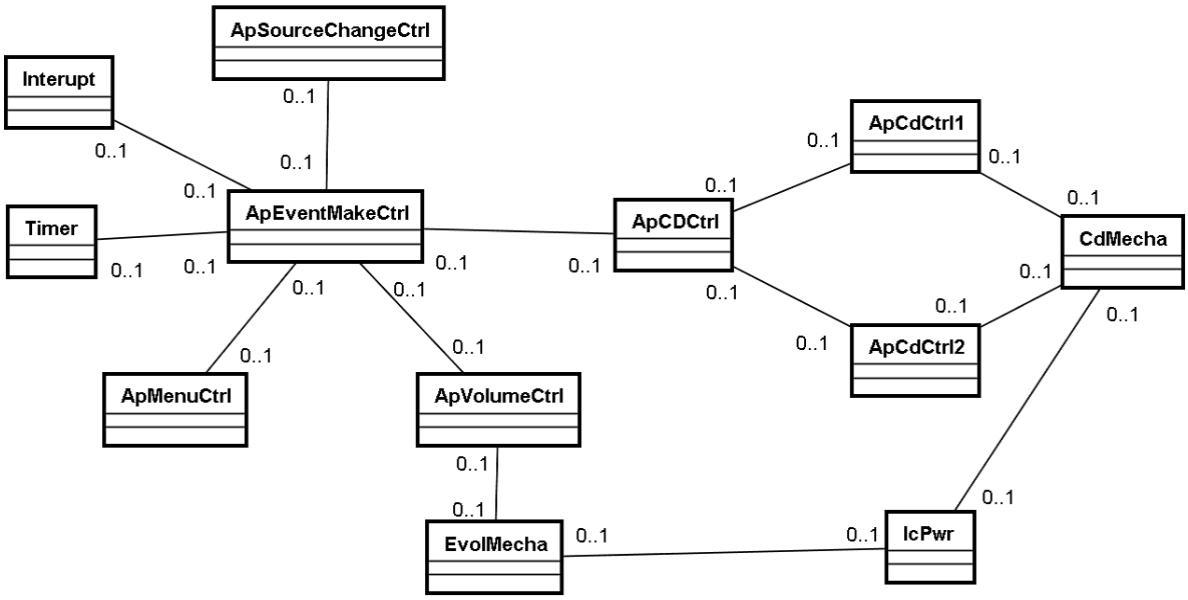


図 8.5: カーオーディオシステム

8.3 評価結果

8.3.1 制御容易性の評価結果

以下に評価の結果を示す.

- ライティングシステム

  1. 呼び出すメソッドの変更

     メソッド SPL をメソッド SPL2 に変更

     - 事例のまま　33 箇所
     - 本手法　1 箇所

2. アサーションの挿入

   ライトのオン，オフを管理する変数を検査するアサーションを挿入

   - 事例のまま　28 箇所
   - 本手法　8 箇所

- カーオーディオ

1. 呼び出すメソッドの変更

   メソッド TimCtrl_RequestOneshotTimer を削除

   - 事例のまま　8 箇所
   - 本手法　1 箇所

2. アサーションの挿入

   ミュートのオン，オフを管理する変数を検査するアサーションを挿入

   - 事例のまま　6 箇所
   - 本手法　2 箇所

2 つの事例とも，アスペクトを利用すると変更箇所がより少なくなっていることがわかる．これにより，より変更箇所が局所化しているので，本手法を利用することで制御容易性がより良くなったと言える．

8.3.2　観測容易性の評価結果

以下に評価結果を示す.

- ライティングシステム

  検証性質:メッセージ送受信の応答性

  - 事例のまま

```
#define p sch?[unlock] && DSENSOR_LOCK_OPEN
#define q BSAVER:m == unlock && BSAVER_S3_O_Ton
```

```
[](p -> <>q)
```

– 本手法

```
#define p DSENSOR_LOCK_OPEN_SentUnlock
#define q BSAVER_S3_O_Ton_ReceivedUnlock

[](p -> <>q)
```

- カーオーディオ

  検証性質:メッセージ送受信の応答性

  – 事例のまま

```
#define sent
(mbx[0].index[_i]
==ApEvMkCtrl_chSendRetrayButtonOperation[ApEvMkCtrl_chIndex])
&& Timer_SentMsg
#define received
(mbx_index[0]
== ApEvMkCtrl_chSendRetrayButtonOperation[ApEvMkCtrl_chIndex])
&& Timer_ReceivedMsg

[](p -> <>q)
```

– 本手法

```
#define p Timer_SentNoticeEvent
#define q Cd_ReceivedNoticeEvent
```

```
[](p -> <>q)
```

ライティングシステムは，送信をチャネルの中身を参照することで，定義している．それだけでは，どのクラスが送信したのかわからないので，状態変数を一緒に参照する必要がある．つまり，変数の数が，1つの定義で3つ，演算子2つである．カーオーディオも同様で，送信をinline関数のライブラリーの変数を参照することで，定義している．それだけでは，どのクラスが送信したのかわからないので，状態変数をいっしょに参照する必要がある．参照された変数などは，ライティングシステムと同様である．2つの事例とも本手法の場合，1つの変数で述語が定義できているので，より観測容易だと言える．

8.3.3 簡潔性の評価結果

以下に評価結果を示す.

- ライティングシステム

  検証性質:メッセージ送受信の応答性

  - 事例のまま

```
#define p sch?[unlock]
#define q DSENSOR_LOCK_OPEN
#define r BSAVER:m == unlock
#define s BSAVER_S3_O_Ton

[]((p && q) -> <>(r && s))
```

  - 本手法

```
#define p DSENSOR_LOCK_OPEN_SentUnlock
```

```
#define q BSAVER_S3_O_Ton_ReceivedUnlock

[](p -> <>q)
```

- カーオーディオ

  検証性質:メッセージ送受信の応答性

  – 事例のまま

```
#define p
(mbx[0].index[_i]
==ApEvMkCtrl_chSendRetrayButtonOperation[ApEvMkCtrl_chIndex])
#define q Timer_SentMsg
#define r
(mbx_index[0]
== ApEvMkCtrl_chSendRetrayButtonOperation[ApEvMkCtrl_chIndex])
#define s Timer_ReceivedMsg



[]((p && q) -> <>(r && s))
```

  – 本手法

```
#define p Timer_SentNoticeEvent
#define q Cd_ReceivedNoticeEvent

[](p -> <>q)
```

本手法を利用しない場合だと，複雑に記述してしまう可能がある．本手法を利用することで，簡潔に記述することができる．

第 9章

議論と関連研究

本稿では，モデル検査技術による UML 設計検証を行う際の検査モデルの作成に関し，検証構造パターンと検査モデルへのアスペクト指向技術の適用による支援手法ならびに，それに基づく検証指向の設計に関して提案を行った．以下，本研究に関する議論と，関連研究について触れる．

9.1 議論

9.1.1 検証構造パターン

本提案の検証パターンの特徴は，検証性質，対象システムの構造，時相論理式を組にしてパターン化している点であり，Dwyer らが性質と時相論理式の組をパターン化している点とアプローチが大きく異なる．これは，検証性質は対象システムに依存して定義されるため，構造とその構造上で一般に確認が求められる性質とをペアにする方が，よりその構造に立ち入った性質をパターン化でき，特にソフトウェアの設計検証などにおいてはより有用性が高まると考えたからである．実際，Dwyer らのパターンが応答性などのかなり抽象度の高い性質に関わるパターンを定義しているのに対し，我々のパターンはメッセージや属性に関わる性質など，よりソフトウェア特有の性質を扱えるようになっている．

9.1.2 検証構造パターンで扱う性質の分析

本研究では，検証性質を，それが対象とするソフトウェア構造の一般性の観点から，図 9.1 に示すように 3 つに分類して捉えている．

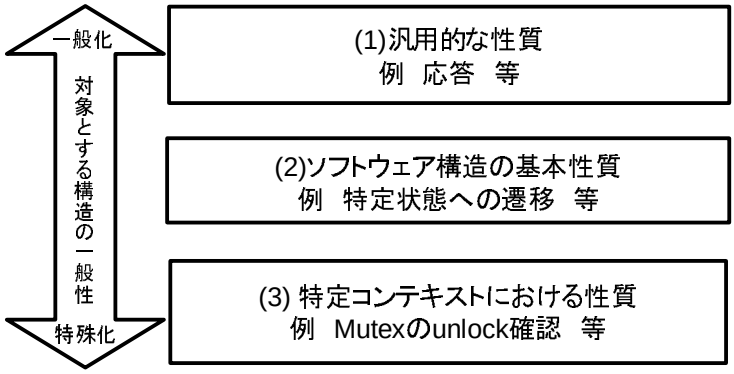


図 9.1: 本研究で定義する性質体系

ここで，(1) がより一般的であり，(2)，(3) となるにつれ，性質が特殊化される．図 9.1 での各分類についてそれぞれ説明する．

(1) 汎用的な性質

検証対象に関しての制約をもたない一般的な性質．5.2.1 節で紹介した Dwyer らによる検証パターンが対象にしている性質はこの分類にあたる．

(2) ソフトウェア構造の基本性質

ソフトウェア設計上での基本的な構造に関する性質．例えば，メッセージの送受信や状態遷移などに関する性質はこの分類にあたる．

(3) 特定コンテキストにおける性質

ソフトウェア設計上の特定の問題解決のための構造に関する性質．例えば，Douglass による Real-Time Design Patterns(RTDP)[12] でパターン化されている特定構造などに関わる性質はこの分類にあたる．

(2) の性質は，(3) より汎用的な構造に関わる性質である．したがって，(2) の性質を検証する手法は (3) の性質を検証するための土台になると考えられる．本研究で提案した検証構造パターンは，(2) に相当する性質をパターン化している．

9.1.3　検証構造パターンによる検証容易性

8 章において，検証容易性の観測容易性と簡潔性について評価を行った．

まず，観測容易性は，述語の定義が 2 つの結果とも本手法のほうが簡潔になり，わかりやすく表現できるようになっている．ただ，カーオーディオの事例で利用されている RTOS をエミュレートした inline 関数でメッセージを送信 (snd_mbx) する時，ある条件で，メッセージの送信完了を正確に観測できない．それは，メッセージ送信したタスクより優先度の高いタスクがメッセージ受信 (rcv_mbx) でメッセージ待ち状態になっている時である．この時，メッセージ送信したタスクは次の状態 (メッセージ送信完了状態) に遷移する前に，メッセージ受信したタスクに切り替わるので，メッセージ送信が完了したにも関わらず，観測できない．この場合，メッセージを受信するタスクが送信タスクより優先度が低い，または同じである場合，問題なく観測できる．

次に，簡潔性については，本手法を利用しない場合だと，複雑に記述してしまう可能があるが本手法を利用することで，簡潔に記述することができる．直感的に記述すると冗長になってしまう検証性質に，特に有効であると言える．

なお現時点ではパターンから LTL の導出は手続きとはいえ，自動的な手順としては定義されていない．構造との適合から LTL の導出までのより厳密な手順としての定義は今後の課題である．

9.1.4　メタパターンとしての利用

本提案のパターンは，メタパターンとして利用できる．すなわちこのパターンをベースに，より限定的かつ具体的なパターンを導出することができる．ソフトウェア技術者にとっての有用性・実用性という観点から，この抽象度の高いパターンからどう具体的なパターンを作るか，体系化することも重要だと考えられる．どのような具体的なパターンをカタログ化することが有用かについては，今後検討が必要である．

9.1.5　アスペクトモデリング技術の適用

アスペクト指向技術は，プログラミング言語に対しては AspectJ を中心に多くの適用提案があり，また実務でも利用が広がっている．またモデリングに対してアスペクト指向技術を適用するという提案も複数あるが，これらはいずれも設計やコード生成などを対象としたものである．本稿では，アスペクト指向技術を検査モデルの構築に適用した点に新規性がある．

一般にひとつの検査モデルに対しては複数の性質の検証が行われるが，検証を行う性質によって，検査モデルに対して assertion の付加や，その検証のために状態を捉える変数の追加などを行うことが多い．そうした修正の多くは横断的なものであるため，アスペクト指向技術の適用を提案した．実際の開発の過程においては，設計が修正・拡張されるたびに，過去に行った検証を含めて再検証を繰り返すことも多く，そのたびに検査モデルを直接修正することは煩雑であり，かつ間違いが導入されやすい．アスペクト指向技術の適用は，こうした検証モデルや検証作業の特徴に適合していると考える．

本研究では，8 章において，検証容易性の制御容易性について評価を行った．本手法を利用することで，より変更箇所を局所化することができた．アサーションの挿入や共通メソッドの呼び出しという典型的な横断的な性質のものに，実用性があることがわかった．

なお，本研究では，アスペクトパターンの形式の定義のみを行い，具体的なパターンは提示しなかった．その理由は，ポイントカットやアドバイスはドメインに依存するものだと考えたからである．ただし上記の検証パターンのカタログ化の議論同様，実用面からはアスペクトパターンの整備は重要であると考えている．

9.2　関連研究

9.2.1　検証パターン

5.2.1 節でも述べたように Dwyre ら [5] は，性質とそれを表現する時相論理式のパターン化を行っている．本研究との違いは，検証対象のモデルの構造をパターンに含めていない点である．このため，定義される性質は構造に依存しない抽象

度の高いものとなり，ある意味汎用性は高いが，特定の構造に依存したより特殊な性質のパターン化には不向きである．本研究はソフトウェア構造（設計）の検証を行うという目的に焦点をあてており，対象としたい性質はソフトウェア構造に依存したものが多い．そうした意味で，本研究の目的からは，我々のアプローチが妥当であると考えている．

また Aygün ら [17] も時相論理式に対してパターン化を行っている．このパターンは，Receiver-Controller-Actor 構造に基づき，同期検証に注目したものとなっている．本研究との違いは，検証項目が１つに絞られている点である．本研究では，主にメッセージ通信に注目しているので，同期だけでなく，メッセージ通信に関わることが検証できる．

矢野ら [18] のパターンは，状態遷移表に基づいて，その状態遷移表のアクションや分岐に注目し，どのアクションが実行されたかやどの分岐を通ったかを検査できるパターンを提案している．本研究は，タスクなどの並行動作単位のやりとりに注目しているが，矢野らのパターンは１つの並行動作単位またはシステム全体の処理の流れに注目している点に違いがある．さらに，パターンではないが，LTL の記述支援としては，LTL の可読性を支援する研究も提案されている．小池ら [19] は習得困難な LTL をソフトウェア開発現場に導入するために，LTL 検査式の図示記法を考案している．こうした研究はソフトウェア技術者にとって，性質や論理式の理解を容易にすることが期待されるため，本研究のパターンの成果などとの相乗が期待できると考える．

9.2.2 検証支援ツール

検証を支援するツールという観点から，関連する研究について触れる．

Lilius らは vUML[7] を提案している．これは，UML クラス図から構造に関する情報をステートマシン図から振る舞いに関する情報を得て，仕様記述言語 PROMELA に変換し，SPIN によるモデル検査を行うものである．例えばライブロックなどの 7 性質に対し自動的に検証し，反例があればシーケンス図として表示する機能を持っている．この vUML では，よく使われるライブロックなどの検証についてはパターン化し，自動的に検証することができるようになっている．しかしながら

本研究のように，より多様な性質のパターン化の支援は行っていない．

Schäfer らは HUGO[10] を提案している．これも同様にステートマシン図から振る舞いに関する情報を得て，仕様記述言語 PROMELA に変換し，SPIN によるモデル検査を行う．性質は，シーケンス図で与える．本研究と同じように，メッセージ通信に注目し，そのイベントの順序を検証性質している．しかしながら，本研究が扱っている属性に関する性質などには対応していない．

9.2.3 アスペクト指向モデリング

アスペクト切り替え手法の実装方法である，設計モデル上のアスペクト手法の関連研究について述べる．

Cottenier らは WEAVR[20] を提案している．これは，SDL 上で設計モデルとアスペクトを記述し，SDL 上でウィーブを行う手法である．本研究の違いは，ウィーブされる対象を UML のステートマシン図としている点である．

Gray らは AODM(Aspect-Oriented Domain Modeling)[21] を提案している．これは，ドメイン専用言語に対するアスペクトを扱うための手法である．アスペクトを記述するのに ECL(Embedded Constraint Language) という言語を導入している．ECL は OCL を拡張すると共に QVT の考えを取り入れた言語である．属性や関連などのモデル要素を追加するなどの機能を持つ．また鵜林らは AspectM[22] を提案している．これは，MDA に利用するために提案された手法である．UML のクラスに対する操作など 6 つのジョインポイントを持つ．UML 上で，アスペクトをウィービングし，モデル変換する．これらの AODM や AspectM はポイントカットに状態を扱えない．一方，本研究では状態をポイントカットで扱うことができる点に特徴がある．

第 10章

おわりに

本研究では，モデル検査技術により UML 設計検証を行う際の検査モデルの構築に注目し，モデル検査技術に精通していないソフトウェア技術者がより容易かつ確実に検証を行うための手法について提案した．

どのような検査モデルを構築するかによって，検証性質の定義が容易にも複雑にもなるため，本研究では性質，構造，時相論理式という組でパターンを定義する手法を提案した．前述したように今回の評価では，検証容易性の制御容易性，観測容易性については，有用性が言えたが，簡潔性については言えなかった．今後，評価に利用した検証性質とパターン化している性質の見直しをする必要がある．どのような性質が適当か，今後の課題として考えて行きたい．

一方検査モデルに対して横断的な修正が行われうる問題に対して，アスペクト指向モデリングの技術を適用する手法を提案した．ひとつの検査モデルに対して様々な性質の検証を繰り返し行うことは検証のひとつの特徴であり，アスペクト指向技術の適用は，そうした性質の検証のたびに，検査モデルに対して行わなければならない横断的な修正を，より効率的，安全に行うことを可能とするものである．なお，アスペクトのパターンの形式を提案したが，具他的にパターン化しなかったが，検証構造パターンのように有用ないくつかのパターンを見つけ，パターン化していきたい．

さらに本研究では上記に基づいた検証指向の設計手法についても検討した．この手法は我々の過去の検証経験を踏まえた，ひとつのひな型的な手順を与えるものである．

また本提案を支援するツールも開発し，提案の有効性の確認や実証に利用した．一度 LTL を定義しても，設計が変更されると，その LTL を修正しなければならない場合もあり，そうした作業の支援を含め，今後よりツール機能の拡張やリファインを行っていきたい．

謝辞

本研究を行なうに当たり, 終始御指導を賜わった岸知二客員教授に深謝致します.

また, 日頃から有益な御助言をいただき, 多面に渡って励ましていただいた Defago Xavier 准教授，青木利晃准教授，矢竹健朗特任助教に感謝致します.

最後に, 本論文をまとめるに当たって御協力いただいた岸・Defago 研究室ならびに青木研究室の諸兄に厚く御礼申し上げます.

参考文献

[1] http://spinroot.com/spin

[2] G.J,Holzmann. : The SPIN Model Checker.
Addison-Wsley 2004.

[3] E.Clarke,O.Grumberg,and D.Peled : Model Checking, MIT 1999.

[4] 岸知二，青木利晃，中島震，野田夏子，片山卓也：プロジェクト紹介：高信頼組込み用オブジェクト指向設計技術，情報処理学会ソフトウェア工学研究会，SE146-7, pp41-46, 2004.

[5] Matthew B. Dwyer, George S. Avrunin and James C.: Patterns in Property Specifications for Finite-state Verification, the 21st International Conference on Software Engineering, May, 1999

[6] Timm Schafer, et. al: Model Checking UML State Machines and Collaborations,Workshop on Software Model Checking,2001.

[7] Lilius,J. and Paltor,I. P.: vUML: a Tool for Verifying UML Models,TUCS Technical Report No.272, 1999.

[8] Kishi, T., et. al.: Project Report: High Reliable Object-Oriented Embedded Software Design, The 2nd IEEE Workshop on Software Technology for Embeddedand Ubiquitous Conputing Systems (WSTFEUS'04),2004.

[9] E.Gamma,R.Helm,R.Johnson and J.Vlissides : Design Patterns Elements of Reusable Object-Oriented Software, ADDISON-WESLEY 1995.

[10] Schäfer T., Knapp A. and Merz, S., Model Checking UML State Machines and Collaborations, Electronic Notes in Theoretical Computer Science 55(3),2001.

[11] OMG : UML Profile for Schedulability, Performance, and Time Specification version1.1

[12] Douglass, Bruce Powel. : Real-Time Design Patterns. Addison-Wsley 2003.

[13] 大野真一朗，岸知二：モデル検査のためのアスペクト指向でのモデル記述支援環境，情報処理学会ソフトウェア工学研究会，SE159-6, pp41-48, 2008.

[14] Stein, D., et. al. :“ A UML-based Aspect- Oriented Design Notation For AspectJ ”. Proceedings of the 1st International Conference on AOSD. Enschede, the Netherlands April 2002. PG106-112.

[15] N.Noda and T.Kishi : Design Verification Tool for Product Line Development, the 11th International Software Product Line Conference, Demonstrations, 2007

[16] OMG : Unified Modeling Language, Superstructure, V2.1.2

[17] R. S. Aygün and A. Zhang : ”SynchRuler” A Rule-Based Flexible Synchronization Model with Model Checking, IEEE Transactions on Knowledge and Data Engineering, Vol.17, No.12, pp.1706-1720, 2005

[18] 矢野恭平，小池隆：状態遷移表のモデル検査における LTL 検証パターン，情報処理学会ソフトウェア工学研究会，SE163, pp303-310, 2009.

[19] 小池憲史，吉田聡，大崎人士：LTL モデル検査のための図示記法，第 14 回ソフトウェア工学の基礎ワークショップ (FOSE2007)　予稿集

[20] Cottenier, T., van den Berg, A., Elrad, T.: Motorola WEAVR: Model Weaving in a Large Industrial Context. In: Aspect-Oriented Software Development, Vancouver, Canada (2007)

[21] Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale, A, and Natarajan, B.: An Approach for Supporting Aspect-Oriented Domain Modeling, In Proceedings of International Conference on Generative Programming and Component Engineering (GPCE 2003), pp.151-168, 2003.

[22] 鵜林尚靖, 佐野慎治, 前野雄作, 村上聡, 片峯恵一, 橋本正明, 玉井哲雄, ”アスペクト指向に基づく拡張可能な MDA モデルコンパイラ, ”組込みソフトウェアシンポジウム 2004 論文集, pp.104-107, Oct.2004.

[23] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, “ NuSMV 2: An OpenSource Tool for Symbolic Model Checking, ” Proceeding of International Conference on Computer-Aided Verification (CAV 2002), pp. 27-31, Copenhagen, Denmark, July, 2002.

[24] 岸知二，野田夏子：組込みソフトウェアのための UML 設計検証支援環境，組込みシステムシンポジウム，pp50-pp57，2006

[25] OMG : A UML Profile for MARTE, Beta1,2007

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In Proc. ECOOP 2001, LNCS 2072, pp. 327. 355, 2001.

[27] Bach, James, Heuristics of Software Testability, www.satisfice.com/tools/testable.pdf, 2003.

[28] トロン協会：μ ITRON4.0 仕様,Ver.4.02.00,2004

[29] M. M. Gallardo, P. Merino and E. Pimentel.“ Debugging UML Designs with Model Checking ”, in Journal of Object Technology, vol. 1, no.2, 2002, pp. 101-117.

[30] 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, pp.56-63, 2005

付録

- 名前

  ２オブジェクト間のメッセージ送受信検証パターン

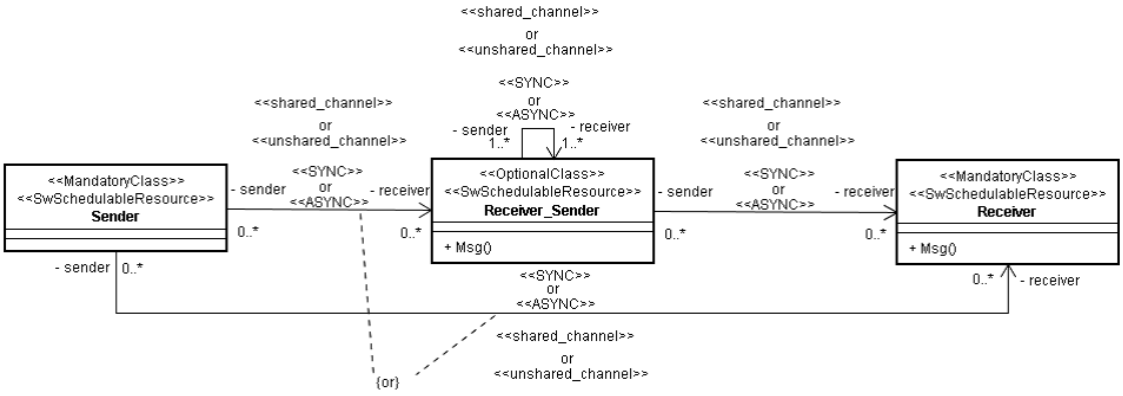
- 検証目的

  ２オブジェクト間のメッセージ送受信に関する以下のことを満たすか検証する.

  – １つのメッセージの特定
  – 複数のメッセージの特定
  – 時間差のある複数のメッセージの特定
  – 特定メッセージの送信または受信の否定

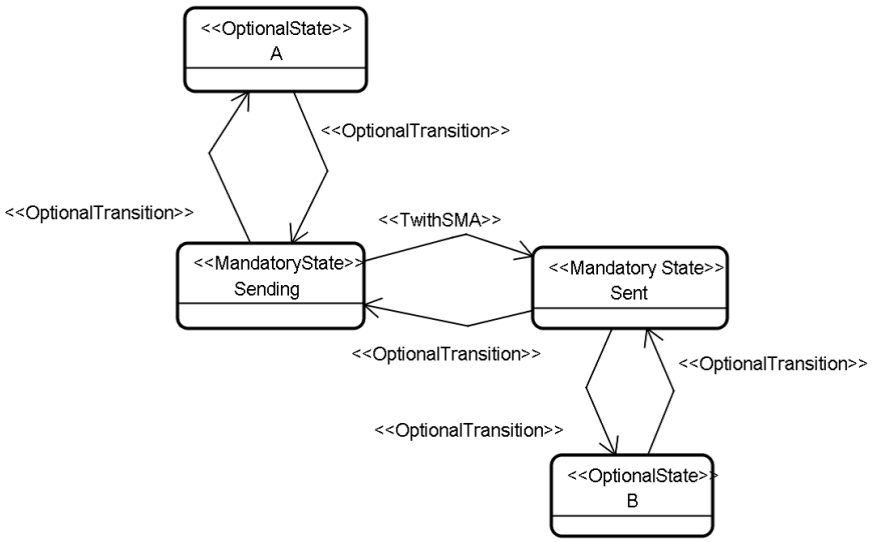
- 構造

― 静的構造

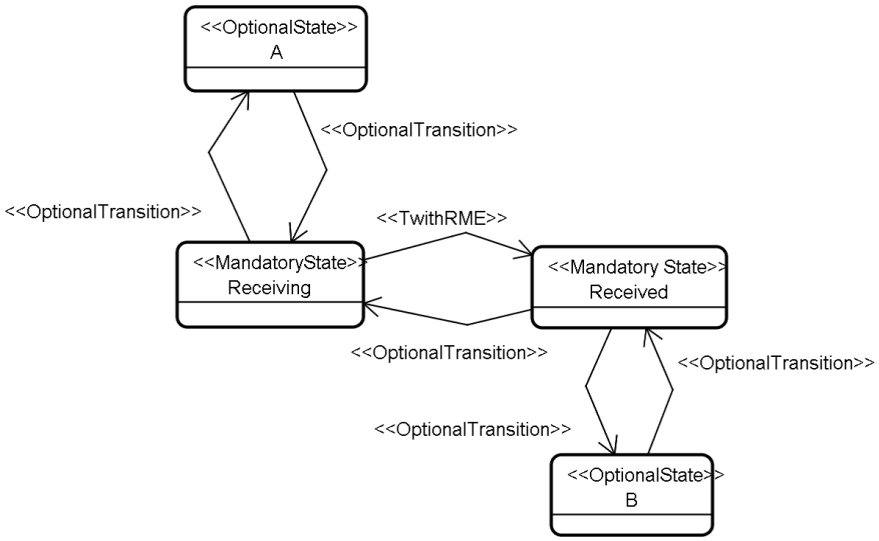


― 動的構造

・Sender



・Receiver



- 性質と検証方法

1．1つのメッセージの特定

－ 性質

特定した1つのメッセージを含んだメッセージの送信と受信が行われる.

– 検証方法

以下の LTL 式で検証する.

```
[](Sending -> <> Receiving)
```

なおここで述語は以下を表す.

・Sending... 特定の Send オブジェクトが特定のメッセージを送信した Sending 状態

・Receiving... 特定の Receiver オブジェクトが特定のメッセージを受信した Receivig 状態

2．複数のメッセージの特定

– 性質

特定した複数のメッセージを含んだメッセージの送信と受信が行われる.

– 検証方法

以下の LTL 式で検証する.

```
[]((Sending1 || Sending2 ||
      ··· || SendingN)
     -> <>(Receiving1 || Receiving2 ||
      ··· || ReceivingN))
```

なおここで述語は以下を表す.

・Sending1... 特定の Sender オブジェクト 1 が特定のメッセージ 1 を送信した Sending 状態

・Sending2... 特定の Sender オブジェクト 2 が特定のメッセージ 2 を送信した Sending 状態

…

・SendingN... 特定の Sender オブジェクト N が特定のメッセージ N を送信した Sending 状態

・Receiving1... 特定の Receiver オブジェクト 1 が特定のメッセージ 1 を受信した Receivig 状態

・Receiving2... 特定の Receiver オブジェクト 2 が特定のメッセージ 2 を受信した Receivig 状態

…

・ReceivingN... 特定の Receiver オブジェクト N が特定のメッセージ N を受信した Receivig 状態

3．時間差のある複数のメッセージの特定

― 性質

いつか特定した複数のメッセージの送信と受信が行われる．

― 検証方法

以下の LTL 式で検証する．

```
!([]((<>Sending1 &&  <>Sending2 && ⋯ && <>SendingN)
-> <>Receiving1 && <>Receiving2 && ⋯ && <>ReceivingN)))
```

なおここで述語は以下を表す.

・Sending1... 特定の Sender オブジェクト 1 が特定のメッセージ 1 を送信した Sending 状態

・Sending2... 特定の Sender オブジェクト 2 が特定のメッセージ 2 を送信した Sending 状態

…

・SendingN... 特定の Sender オブジェクト N が特定のメッセージ N を送信した Sending 状態

・Receiving1... 特定の Receiver オブジェクト 1 が特定のメッセージ 1 を受信した Receivig 状態

・Receiving2... 特定の Receiver オブジェクト 2 が特定のメッセージ 2 を受信した Receivig 状態

…

・ReceivingN... 特定の Receiver オブジェクト N が特定のメッセージ N を受信した Receivig 状態

４．特定メッセージの送信または受信の否定

― 性質

特定した複数のメッセージの受信が行われない.

― 検証方法

以下の LTL 式で検証する.

```
[](Sending -> <> Receiving)
```

なおここで述語は以下を表す.

・Sending... 特定の Sender オブジェクトが特定のメッセージを送信した Sending 状態

・Receiving... 特定の Receiver オブジェクトが特定のメッセージを受信した Receivig 状態

- 名前

  オブジェクト間連続メッセージ送受信検証パターン

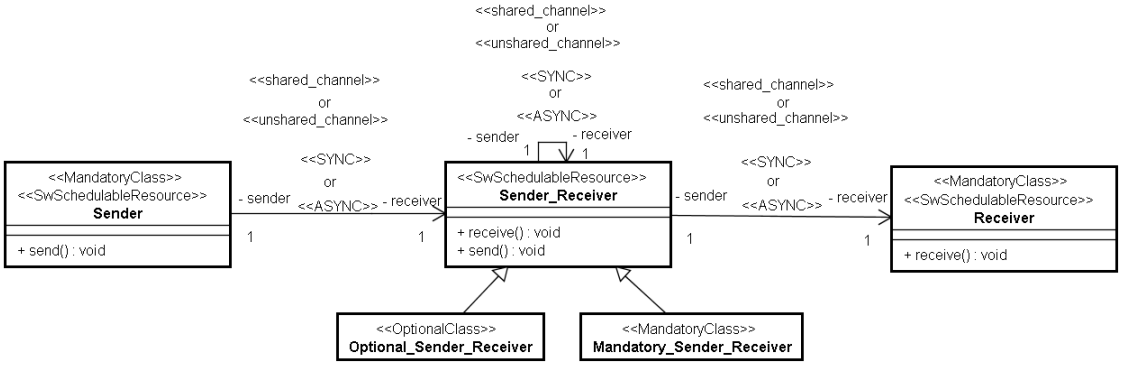
- 検証目的

  ２オブジェクト間のメッセージ送受信に関する以下のことを満たすか検証する.

  – １つのメッセージの特定

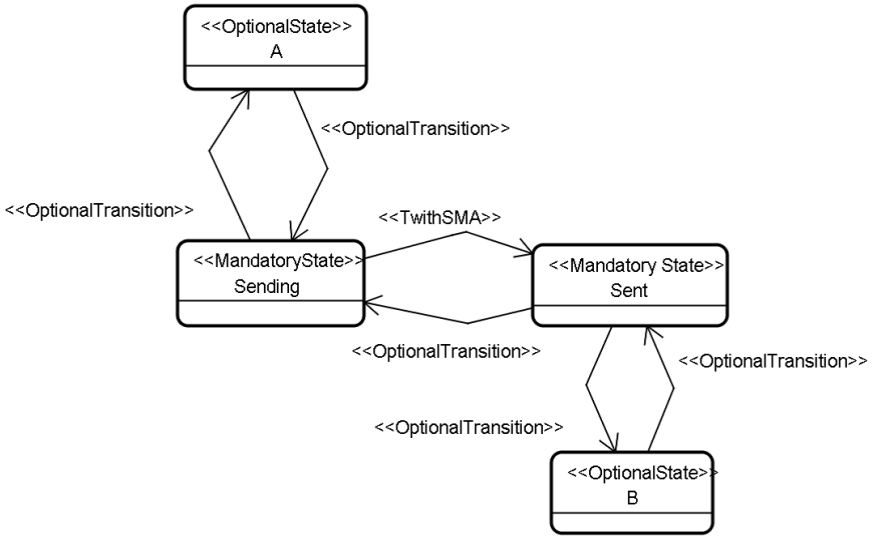
- 構造

— 静的構造

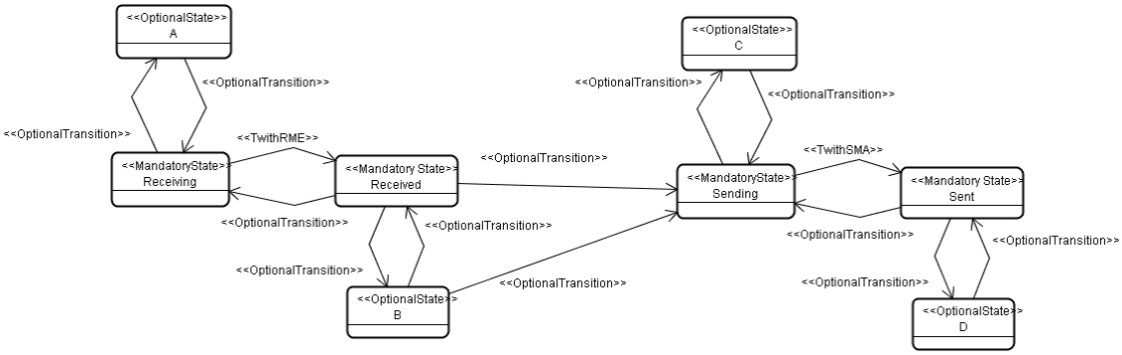


— 動的構造

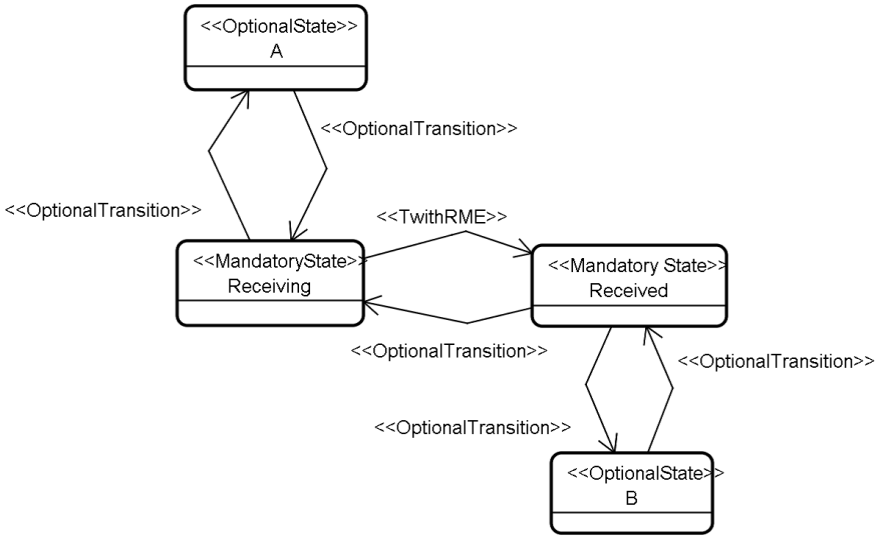
- Sender



- Sender_Receiver



・Receiver



- 性質と検証方法

1．1つのメッセージの特定

－ 性質

特定した1つのメッセージを含んだメッセージの送信と受信が行われる.

－ 検証方法

以下の LTL 式で検証する.

```
[](((!SR_Receing_1 && !SR_Sending_1 ||
    ··· || ! R_Receiving) U S_Sending)
  &&((!SR_Sending_1 || ··· || ! R_Receiving) U SR_Receing_1)
  &&
   ···
  &&(!SR_Receing_N U R_Receiving))
```

なおここで述語は以下を表す.

・S_Sending...Sender オブジェクトが特定のメッセージを送信した状態

・SR_Receiving1...Sender_Receiver オブジェクト 1 がメッセージを受信した状態

・SR_Receiving2...Sender_Receiver オブジェクト 2 がメッセージを受信した状態

・・・

・SR_ReceivingN...Sender_Receiver オブジェクト N がメッセージを受信した状態

・SR_Sending1...Sender_Receiver オブジェクト 1 がメッセージを送信した状態

・SR_Sending2...Sender_Receiver オブジェクト 2 がメッセージを送信した状態

・・・

・SR_SendingN...Sender_Receiver オブジェクト N がメッセージを送信した状態

・R_Receiving...Receiver オブジェクト 1 がメッセージを受信した状態

- 名前

  ２つのオブジェクト間の属性値検証パターン

- 検証目的

  ２つのオブジェクト間の属性値の関係を検証する。

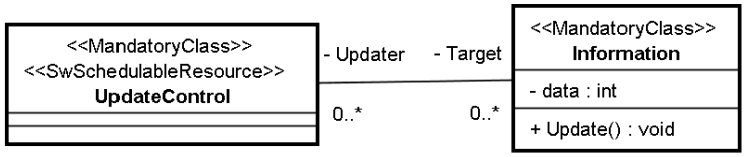
  – 1 属性に対する特定値の検出

  – 複数属性に対する特定値の検出

  – 複数の属性に対する厳密な特定値の検出

- 構造

— 静的構造



— 動的構造

・Update_Control

なし

・Information

なし

• 性質と検証方法

1．1属性に対する特定値の検出

— 性質

特定したオブジェクトの1属性が特定の値または以上，以下，超える，未満になる．

— 検証方法

以下の LTL 式で検証する．

```
<> SpecificValue
```

なおここで述語は以下を表す．

・SpecificValue... 特定の Information オブジェクトが特定の値または以上，以下，超える，未満になった

述語には，=,<,>,<=,=> を使った任意の式が入る．

2．複数属性に対する特定値の検出

－ 性質

特定した複数オブジェクトの複数属性が特定の値または以上，以下，超える，未満になる．

－ 検証方法

以下の LTL 式で検証する．

```
<>(Receiving1 || ··· || ReceivingN)))
```

なおここで述語は以下を表す．

・SpecificValue1... 特定の Information1 オブジェクトが特定の値または以上，以下，超える，未満になった

・・・

・SpecificValueN... 特定の InformationN オブジェクトが特定の値または以上，以下，超える，未満になった

述語には，=,<,>,<=,=> を使った任意の式が入る．

3．複数の属性に対するの厳密な特定値の検出

－ 性質

特定した複数オブジェクトの複数属性が特定の値または以上，以下，超える，未満になる．

– 検証方法

以下の LTL 式で検証する.

```
!([](( SpecificValue1 &&  ···  && !SpecificValueN) ||
      ··· || (SpecificValue1  &&  ···  && SpecificValueN))))
```

なおここで述語は以下を表す.

・SpecificValue1... 特定の Information1 オブジェクトが特定の値または以上, 以下, 超える, 未満になった

・・・

・SpecificValueN... 特定の InformationN オブジェクトが特定の値または以上, 以下, 超える, 未満になった

述語には, =,<,>,<=,=> を使った任意の式が入る.

本研究に関する発表論文

[1] 金井勇人, 岸 知二: UML 設計モデル検査技術のための検証パターンの提案, 情報処理学会　ソフトウェア工学研究会, SE152-3, pp17-24, 2006.

[2] 金井勇人, 岸 知二: UML 設計に対するモデル検査のための検証パターンの提案と評価, 情報処理学会 ソフトウェアエンジニアリングシンポジウム, pp185-192, 2006.

[3] 金井勇人, 岸 知二: UML 設計に対するモデル検査のための検証パターン，情報処理学会　全国大会, 2007.

[4] 岸 知二, 金井勇人: 組込みソフトウェア設計検証へのモデル検査技術の適用と考察, IPA SEC journal, No.12, pp10-19, 2008.

[5] 金井勇人, 岸 知二: モデル検査のためのアスペクト指向メカニズム切り替え手法の提案，情報処理学会　ソフトウェア工学研究会, SE161-7, pp49-56, 2008.(CS 領域奨励賞)

[6] 金井勇人, 岸 知二: ソフトウェア設計に対するモデル検査のための検証パターン, 情報処理学会論文誌, Vol.49 No.10, pp3493-3507, 2008.