博士論文

モデル検査を用いた
フォールトアナリシス手法の提案

野口 秀人

主指導教員 青木 利晃

北陸先端科学技術大学院大学

情報科学研究科

平成 27 年 9 月

Abstract

A fault analysis process is discussed in this paper. Fault analysis is an activity to detect the fault that caused a failure, which has occurred during software testing. The objective of this work is to provide an effective fault analysis method especially for the "hard-to-reproduced" failure. In order to achieve this objective, a failure reproduction method using model extraction and model checking techniques is proposed in this paper. Moreover, a fault analysis process including the failure reproduction is also proposed. Assumed target of this work is so-called embedded software, which is embedded in the devices that control physical, electrical or electronic world outside the devices.

In late years, the purpose of embedded software is to add value of the products by using software, which realizes intellectual control depending on the situation, cooperation action of the plural devices or cooperation with the information service. Therefore, concurrency of the systems and complexity of embedded software in such systems increase rapidly. In addition, the behavior of world outside the systems, which embedded software controls through devices, is nondeterministic, and it is difficult to assume whole behavior of the systems beforehand.

One of the issues of such embedded software development is fault analysis. To detect the fault according to an observed failure, the developer generally tries to reproduce the failure by executing the system again under the same condition as that in the case of failure. However, the failure reproduction is sometimes quite difficult, since behavior of the target system is not constant because of above-mentioned concurrency and nondeterminism. This paper proposes an effective method of fault analysis for such a hard-to-reproduced failure.

Primarily, model checking technique is proposed to be applied to failure reproduction. Model checking is a powerful technique to decide whether the behavior model of the system models the predefined property, which is conventionally used during software development to find the unknown and unexpected behavior of the target system. One of the characteristics of this work is to use model checking technique for detecting behavior that reaches the observed failure that has occurred during testing.

Then, a model extraction method from source code is proposed for model checking against source code. One of the problems in practical use of model checking is a huge cost for constructing the behavior model of the target system. POM (Program-Oriented Modeling) framework that extracts a model from source code is proposed to solve this problem. The POM/MC tool that is a tool performing model extraction and model checking using the POM framework is also proposed. POM/MC enable its user to explicitly appoint the method for model extracting. This feature supports the failure reproduction in trial-and-error-manner experiments both to keep information necessary for fault analysis and to avoid state explosion in model checking.

Moreover, the fault analysis process is defined in formal manner. The fault analysis model that formalizes concepts of failure, fault and fault analysis is proposed. Then fault analysis is constructed from view of the hypothetico-deductive method, and an experimental fault analysis process based on hypothesis and predictions is formalized. This process is implemented by using POM/MC. Finally, feasibility of effective fault analysis by using the proposed method is shown through some case studies.

Keywords

fault analysis, model extraction, model checking, system testing, hypothetico-deductive method

要旨

本論文では，ソフトウェア開発において発生する故障（期待を逸脱する状態）に対し，その原因であるフォールトを特定する「フォールトアナリシス」の手法について論じる．特に，発生時と同様の条件下でソフトウェアを再実行しても再現が難しい故障に対して，その故障の原因を効率的に特定することが，本研究の目的である．本目的を達するため，本論文では，ソースコードからのモデル抽出技術とモデル検査技術を用いた故障再現手法，および同手法を用いたフォールトアナリシスのプロセスを提案する．

本研究では，組込みソフトウェアと呼ばれるソフトウェアを対象とする．組込みソフトウェアとは，物理現象や情報通信などを制御する装置に搭載されるソフトウェアをさす．近年，組込みソフトウェアは，制御の実現だけでなくソフトウェアによる付加価値の実現を目的に用いられる．例えば，状況に応じた知的な最適化制御や，ネットワークによる複数機器の協調動作，あるいは情報サービスとの連携などである．このため，ソフトウェアの並行動作が増えるなど，組込みソフトウェアの複雑さは急速に増大している．また，組込みソフトウェアが装置を介して制御する外界の振る舞いは非決定的であり，その振る舞いを事前に想定することは困難である．

このような背景の下，フォールトアナリシスが組込みソフトウェア開発の課題の１つとなっている．一般的に，フォールトを特定する際には，故障発生時のソフトウェアの振る舞いを理解するため，故障発生時と同一条件下でソフトウェアを再実行して故障を再現を試みる．しかし，上述の並行性や非決定性により対象システムの振る舞いは一定せず，故障発生時と同一条件下であっても，故障が再現しないことがある．本論文では，このような再現困難な故障に対するフォールトアナリシスの効率的な手法を提案する．

第一に，フォールトアナリシスへのモデル検査利用を提案する．モデル検査は，モデル化したシステムの振る舞いが，別途定義した性質を満たすか否かを検査する技術である．従来，モデル検査は，システムに期待される性質を満たすことを確認し，あるいは，性質を満たさない未知の故障を発見する目的で用いられてきた．本提案では，既に観測された故障に対して，故障に至る振る舞いを発見するためにモデル検査を用いる点が特徴である．

第二に，ソースコードからのモデル抽出技術を提案する．モデル検査の実用上の課題の１つに，システムの振る舞いモデルの作成にコストがかかることがある．この課題を解決するため，ソースコードからモデルを抽出する POM　(Program-Oriented Modeling) フレームワークを提案し，POM フレームワークを用いたソースコードからのモデル抽出と，モデル検査を行うツールである POM/MC を提案する．POM/MC の特徴は，ユーザがモデルの抽出方法を指定できる点にある．これにより，故障再現に必要な情報を残しつつ，モデル検査の課題の１つである状態爆発を回避するための試行錯誤的なモデル作成を実現し，モデル検査による故障再現を実現する．

第三に，フォールトアナリシスのプロセスを形式的に定義する．まず，故障やフォールト，フォールトアナリシスの概念を形式化したフォールトアナリシスモデルを提案する．次に，フォールトアナリシスを仮説演繹法の視点から実験と解釈し，仮説と予測に基づく実験的フォールトアナリシスプロセスを形式化する．さらに，実験的フォールトアナリシスプロセスを，POM/MC を用いて実現する．

最後に，いくつかのケーススタディを通して，提案手法により効率的なフォールトアナリシスが実現可能であることを示す．

目 次

第1章　はじめに

1.1　背景

1.1.1　組込みソフトウェア

筆者は，組込みソフトウェアを対象としたソフトウェア開発技術の研究に従事してきた．一般に，組込みソフトウェアとは，機械的，電気的もしくは電子的な制御を行うハードウェア装置に搭載されるソフトウェアのことをさす．組込みソフトウェアは装置を制御することで，装置にとっての外界である物理現象や情報通信などを間接的に制御する．従来，電気・電子的もしくは機械的に実現されていた装置の制御機能をソフトウェアに置き換えることで，部品数を減らし製造コストを低減することが，組込みソフトウェアを用いる理由の一つであった．製造コスト低減のため，装置に搭載される計算機の計算能力や記録能力などは限られることも多い．旧来の組込みソフトウェア開発では，限られた計算リソースを用いて，いかにソフトウェアを実現するかが課題の一つであった．また，装置および外界の応答時間に合わせた制御が必要であり，これを満たすリアルタイム性の実現も組込みソフトウェア開発の課題の一つである．近年では，組込みソフトウェアを導入する目的に，ソフトウェアよる付加価値の実現が加わっている．すなわち，状況に応じたより知的な制御の最適化や，ネットワークによる複数機器の協調動作，あるいは情報サービスとの連携などが，組込みソフトウェアに求められている．IoT (Internet of Things) の概念に示されるような，多くのデバイスがネットワークにつながりデータを流通させることで系をなすシステムや，Industie 4.0 で示されるような，状況をモニタリングし知的な判断を行い生産ラインを柔軟に変更するスマート工場や，CPS　(Cyber Physical Systems) と呼ばれる制御と情報処理が融合したようなシステムも提案され，ソフトウェアの役割は増大する一方である．

1.1.2　組込みソフトウェア開発の課題

1.1.1 節で示した組込みソフトウェアの役割の増大に伴い，組込みソフトウェア開発の困難さも増している．ここで，IPA/SEC によるソフトウェア産業の実態把握に関する調査の結果 [51] を示す．同調査では，国内外でソフトウェア開発を行う企業に対し，組込みソフトウェアおよびエンタープライズソフトウェアそれぞれの開発実態に関するアンケートを行った．図 1.1 は，組込みソフトウェア開発の課題についての回答結果であり，図 1.2 は，組込みソフトウェア開発における設計品質向上という課題に対する解決策についての回答結果である．各質問に対して該当する選択肢を 3 番目まで回答する複数回答方式をとっている．

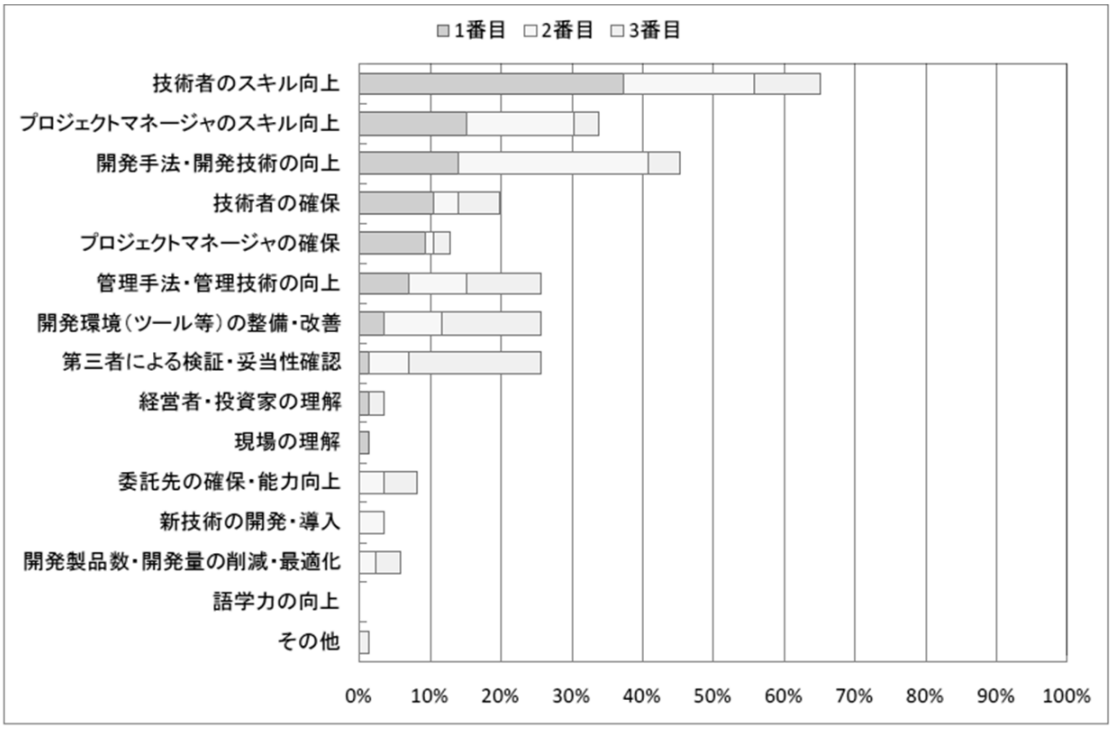

図 1.1: 組込み系ソフトウェア開発の課題 [51]

図 1.1 によれば，組込みソフトウェア開発で最も大きな課題は，設計品質の向上である．設計品質の向上を課題とする回答数は，開発コストや開発期間を課題とする回答を大きく上回っており，組込みソフトウェア開発は量の問題よりも質の問題に直面していると考えられる．さらに，図 1.2 によれば，設計品質を向上するために取り組むべき課題として，技術者のスキル向上についで，開発手法・開発技術の向上が必要であるとして挙げられており，組込みソフトウェア開発の質的課題に対して開発技術の向上が追いついていない状況を示唆する．

1.1.3 組込みソフトウェアの並行性と非決定性

組込みソフトウェア開発の難易度を上げる要因の一つとして，ソフトウェアの並行動作が挙げられる．図 1.3 に，組込みソフトウェアとそれを含むシステムの構成の概念図を示す．組込みソフトウェアは，装置に組み込まれた電子回路に搭載された CPU 上で動作する．装置は各種のセンサと制御装置（モータのようなアクチュエータや，電気の流れを制御する配電盤，情報の流れを制御する電子回路など）を有する．図では簡単のため単一のセンサと制御装置を記載しているが，実際には多数のセンサと制御装置を有するのが一般的である．組込みソフトウェアは，電子回路を介してセンサが観測した外界の状況を入力とし，状況に応じた制御目標を決定し，制御装置に対して制御目標に向けた制御指示を行う．制御装置は制御指示に従って外界に影響を与える．このとき，ある装置が外界に与えた影響は，外界を通して他の装置に影響を与える．また，装置はネットワークを介して他の装置と情報のやり取りをしながら協調動作を行う．このような多様かつ複雑な制御をリ

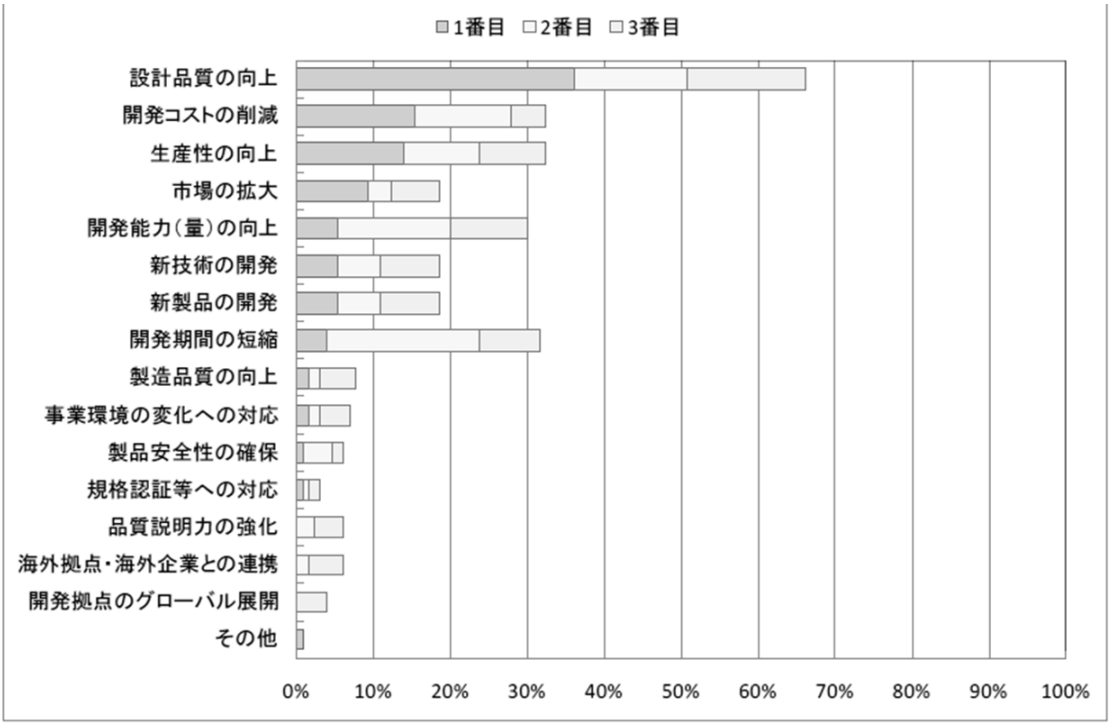

図 1.2: 組込み系ソフトウェアの品質向上の課題 [51]

アルタイムに行うために，ソフトウェアの並行化が進んでいる．ソフトウェアがマルチタスク/マルチプロセス化するほか，複数の CPU コアや CPU による処理の並行化，あるいは複数の制御装置からなる大規模システムでの装置間での並行処理も増えている．

ここで，図 1.4 に，製品として実際に開発されたシステムの例を示す．ただし，説明のために図は簡略化している．本システムは，複数のハードウェアモジュール (Entry, Module A, Module B, Module C) からなる搬送システムである．Entry モジュールから投入された物体をベルトコンベアによって運び，物体はハードウェアモジュール間を移動する．各モジュールは、それぞれ CPU を持っており、個々にソフトウェアが動作する．モジュール間は独自の通信用ネットワークで接続されている．各モジュールに組み込まれるソフトウェアは，センサを介してモジュールや物体の状況を把握し，ベルトコンベア，ゲート，物体に対する処理を制御する．

ハードウェアモジュールでは，搬送された各物体に対してある種の処理を施す．処理の内容は物体ごとに異なり，どのモジュールでどの処理を施すかは，それぞれの物体の投入前にオペレータが指示する．物体のモジュール間移動をセンサによって確認すると，物体への処理の指示情報もネットワークを介してモジュール間で受け渡される．各モジュールのソフトウェアは，指示情報に基づいて物体に施す処理を決定し，モジュールのハードウェアを制御して処理を施す．

本システムの動作は，ソフトウェアだけでは動作が決まらない様々な非決定的要素に影響を受ける．本システムで考えられるソフトウェア外部の非決定的要素の例を表 1.1 に示す．このような組込みソフトウェアにとってコントロールできない非決定的要素によって，ソフトウェアプログラムへ入力される値やその順序，タイミングなどが変化し，これらが

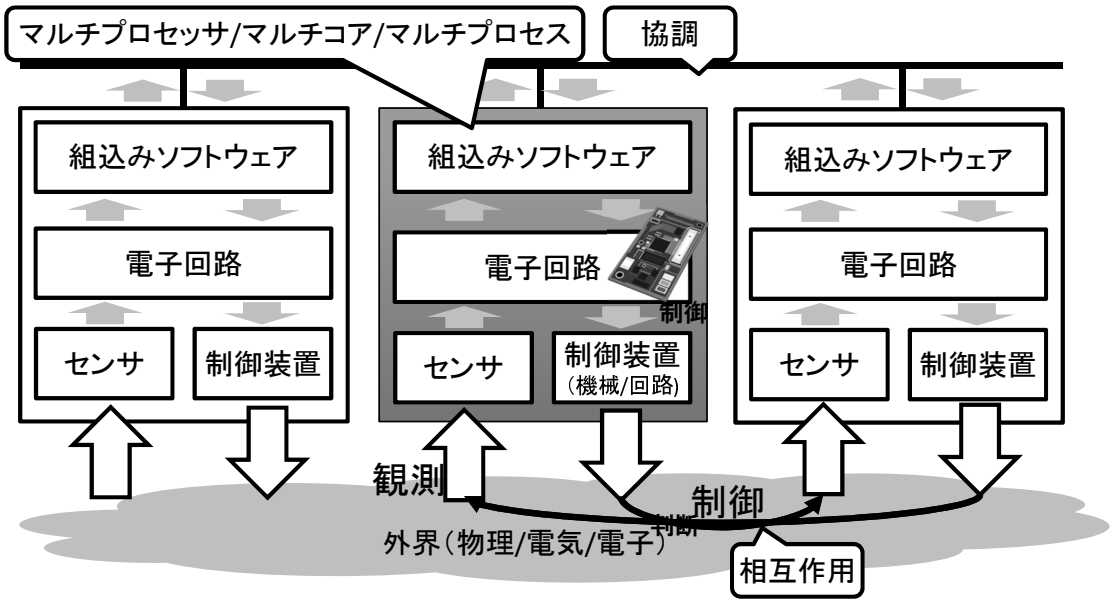

図 1.3: 組込みソフトウェアの並行性

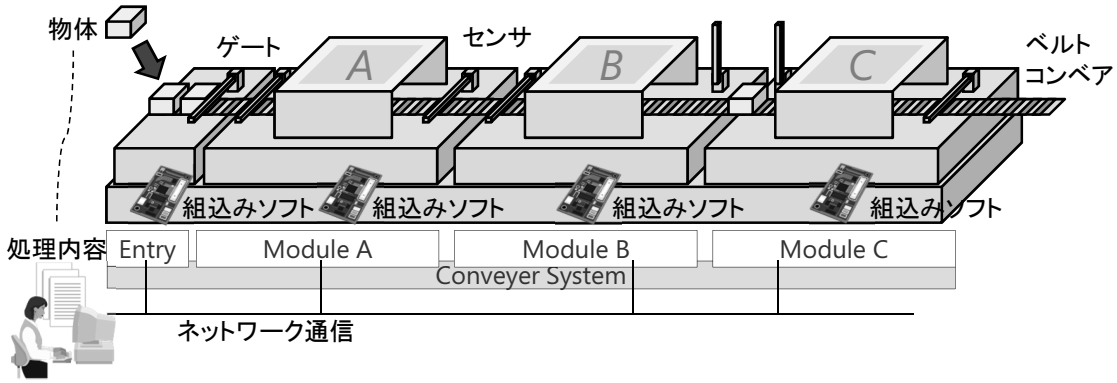

図 1.4: 組込みシステムの実例

表 1.1: システムの動作に影響を与える非決定的要素の例

要素	具体例
投入される物体	物体の投入タイミング
	物体間の距離
物体の移動	センサ感知タイミング
	ベルトコンベアの動作速度
	物体の移動タイミング
	物体の向き
ソフトウェア	プログラム実行タイミング
	プログラム実行順序
	通信タイミング・遅延

表 1.2: 並行バグの再現困難性 [18]

大変困難	19.2 %
困難	73.7 %
OK	20.9 %
比較的容易	5.8 %
容易	0.4 %

異なればソフトウェアの振る舞いが変化する．また，組込みソフトウェア自身も並行に動作する場合には，それらの動作順序やタイミングなどが非決定的に変化し，これによってソフトウェアの振る舞いが変化する．さらに，これらの非決定的な事象の発生確率は異なり，ある振る舞いは頻繁に発生するが，特定の振る舞いは滅多に発生しない，という現象が起こる．

1.1.4 フォールトアナリシス

1.1.3 節に示した組込みソフトウェアの非決定性に起因して困難となるソフトウェア開発上の活動の一つに，フォールトアナリシスがある．フォールトアナリシスとは，テスト中に発生した故障に対して，その原因であるフォールトを特定する行為である．ソフトウェアに故障が発見された場合，再度故障が発生しないようにソフトウェアプログラムの修正が求められる．ソフトウェアプログラムを修正するためには、故障を発生させた原因を特定する必要がある．一般に，製品開発において故障が発生した場合，故障時の振る舞いを理解し故障原因を特定するため，システムを再実行することで故障を再現しようとする試みがなされる．この試みを故障再現と呼ぶ．

先に示した搬送システムのように，ハードウェアやネットワークなどの外界やソフトウェア自体の振る舞いに非決定性が存在する場合，たとえ投入する物体や処理の指示が同じであっても，システムが同じ動作をするとは限らない．故障が再現できなければ故障発生時のシステムの振る舞いを把握できず，故障の原因を究明し修正することができない．実際のソフトウェア開発において，発生した故障に対する故障再現と，故障再現に基づく故障原因究明であるフォールトアナリシスに多くの時間を費やしている．

表 1.2 は,Godefroid らによる「並行バグ」（同文献におけるバグとは，本論文における故障に相当する）に関して，Microsoft 社内で実施した調査報告 [18] の一部である．この報告によれば，75%の回答者が並行バグの再現は大変困難もしくは困難であると回答している．また，表 1.3 は，同調査において示された並行バグのデバッグにかかる時間を示している．73%の回答者が，デバッグには 1 日以上かかると回答し，数ヶ月かかる場合もある．さらに，表 1.4 は並行バグの重大性を示しており，7 割を超える並行バグは重大なものとなる．

このように，並行実行するソフトウェアで発生する故障は，重大かつその修正に多大なコストがかかり，その一つの要因が故障の再現である．非決定性を持つソフトウェアの故障再現は，ソフトウェア開発の大きな課題の一つである．

表 1.3: 並行バグのデバッグに要する時間 [18]

数時間（1 日未満）	27.4 %
数日（1 週間未満）	63.4 %
数週間	8.3 %
数ヶ月	0.9 %

表 1.4: 並行バグの重大性 [18]

大変重大	25.8 %
	58.7 %
	13.5 %
重大でない	2.0 %

1.2 本論文で扱う問題

図 1.5 に，フォールトアナリシスの基本的な考え方と，その課題を示す．テスト中に故障が発生するとき，初期状態から故障に至る中間過程が観測可能ではないことも多い (図 1.5 (a))．システムによっては，動作ログやトレースを記録している場合もあるが，それらを記録する計算機リソース（計算時間や記録装置など）の制限により，記録は限定的である．特に，組込みシステムではハードウェアリソースが限られ記録装置も持たない場合が多く，動作過程が記録されていることは稀である．故障を再発させないようソフトウェアを改修するためには，故障発生時のソフトウェアの振る舞いを理解し，これを適切に修正する必要がある．そのため，従来，故障が発生した時と同様の条件下においてシステムもしくはソフトウェアを再実行し，故障を再現しようとしてきた．これを故障再現と呼ぶ．

しかし，故障発生時と同じ初期条件下でソフトウェアを再実行しても故障が発生せず，システムが正常終了してしまうことも多い．故障発生時と同様の条件下において故障が再現しない理由の一つとして，故障の原因となる事象の発生確率が低いことが挙げられる．ソフトウェアが有する並行性や非決定性のため，ソフトウェア実行の初期条件が同一であっても，ソフトウェアの振る舞いの可能性は複数存在し，振る舞いを決定する事象の発生確率は事象ごとに異なる．そもそも，発生確率の高い事象は早期のテストで発見され修正されるため，ソフトウェア開発において大きな問題にはならない．発生確率の低い事象を原因として発生する故障は，発見が遅れるために問題となり，発見後に原因究明のために実行を繰り返しても再現される確率も低いため，さらに大きな問題となる．もし再現ができたとしても，並行実行されるスレッドの振る舞いは複雑であり，故障の原因を特定することが困難な場合がある (図 1.5 (b))．

本研究が扱う問題は，以上で議論してきたような再実行では再現や原因特定が困難な故障に対して，効率的に故障の再現と原因特定を行うフォールトアナリシス手法を確立することである．

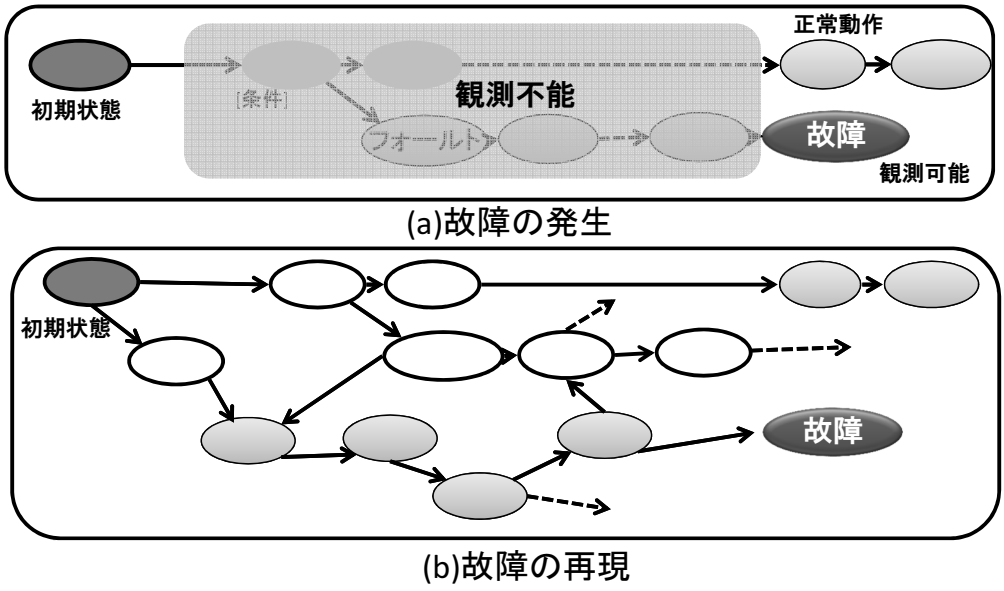

図 1.5: 故障の発生と再現

1.3 想定するフォールトアナリシスの業務フロー

図 1.6 に，本研究の前提条件として想定するフォールトアナリシスを行う業務フローを示す．まず，本研究で扱う故障は，ソフトウェア開発の工程のうち，製品開発部門によるシステムテストなどの後期のテスト，または品質保証部門による品質評価のためのテストにおいて発見された故障である．このとき，開発者とは異なるテスト者がテストを実施するものとする．テスト者は，テストに必要となる製品知識を有していることを想定する．

テスト者がソフトウェアテストにより故障を発見すると，発生した故障に関する報告は，製品開発を行う開発者に伝達される．開発者は製品について深い知見を有している．開発者は故障が再発しないようにソフトウェアを改修する責務を負っている．開発者はソフトウェア改修のために，フォールトアナリシスを行う．

さらに，開発者のほかに開発支援を行う技術者の存在を想定する．実務的には開発支援の専門部門や品質保証部門に置かれることが多いこの技術者は，開発者の支援を行う責務を負っている．開発支援者は，フォールトアナリシスに関して，フォールトアナリシス手法全体，モデル検査，後で述べるモデル変換ルールを作成する方法など，本研究で提案するフォールトアナリシス手法を実践するために要する知識を有するものと想定する．しかし，製品に関する知識，特に詳細なソフトウェアアーキテクチャなどの製品内部に関する知識を有することは想定しない．

フォールトアナリシスは，開発者から開発支援者への分析依頼に基づき，開発者と開発支援者の共同作業により実施することを想定する．その際，開発者は提案手法に対する基本的な理解と，モデル検査に関する基本的な知識を有しているものとする．開発者は，製品知識に基づいて，故障原因の推定や，製品内部の調査を行う．開発支援者は，提案手法に基づくツールの提供や，提案手法の実施に必要な振舞いモデルの抽出を行う．モデル検査の実行および，モデル検査結果の解釈は，開発者および開発支援者の協力をもって実施するものとする．

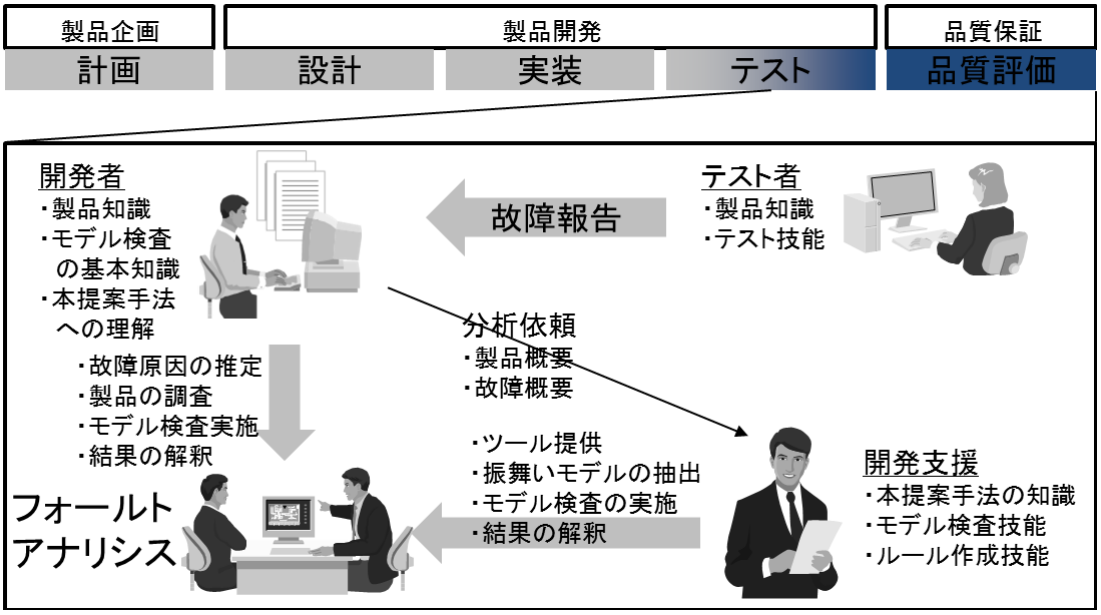

図 1.6: 想定するフォールトアナリシスの業務フロー

1.4 研究の目的

本研究の目的は，プログラム実行によって再現が困難な故障に関して，故障を再現し故障の原因であるフォールトを特定するフォールトアナリシスの効率的な手法を確立することである．特に，並行性や非決定性を持つ組込みソフトウェアに対する，モデル検査を用いた故障再現と，それに基づくフォールトアナリシスを実現する．前提として，前述のフォールトアナリシス体制を想定し，故障発生時のソフトウェアの状態がわかっており，ソースコードが存在するものとする．また，本論文では，ソースコードは C 言語で書かれているものとする．

提案するフォールトアナリシスの手法には，以下が含まれる：

- モデル検査を用いた故障再現の手法
- モデル検査を実現するためのツール
- 上記故障再現手法を用いたフォールトアナリシスの実施プロセス

また，本研究では提案するフォールトアナリシス手法について，以下の観点からソフトウェア開発実務での有用性を評価する．

- C 言語ソースコードに対してモデル検査が実施可能であること
- モデル検査によって故障再現が可能であること
- フォールトアナリシスのプロセスが，ソフトウェア開発において実践可能であること
- 従来手法に比べて，提案手法によるフォールトアナリシスが効率的であること

1.5 提案の骨子

本論文による提案の骨子は以下である．

第一に，モデル検査を用いたフォールトアナリシス手法を提案する．特に，フォールトアナリシスに必要な故障の再現にモデル検査を用いることで，プログラムの実行では再現が困難な故障の再現を可能とする．

第二に，モデル検査の対象となる振る舞いモデルをソースコードから抽出する手法を提案する．ソフトウェアに対するモデル検査を行うために，ソースコードから検査の対象となる振る舞いモデルを抽出する，Program-Oriented Modeling (POM) フレームワークを提案する．また，POM に基づいて振る舞いモデルを抽出し，モデル検査を実行する，ソースコードモデル検査ツール POM/MC を実現する．さらに，POM/MC を実現する上で，モデルを表現するためのメタモデルと，モデルの抽出を行う変換ルール集合を定義する．

第三に，POM/MC を用いたフォールトアナリシスプロセスを提案する．本プロセスの提案にあたり，フォールトアナリシスに関する概念を明確化する，フォールトアナリシスモデルを策定する．フォールトアナリシスモデルに従い，仮説演繹法に基づく実験的フォールトアナリシスプロセスを提案する．

1.6 本論文の構成

本論文の構成は以下である．第 1 章に，本論文の概要を示した．第 2 章に，本論文の背景となる既存技術を概説する．第 3 章に，本論文に示す研究のアプローチを示す．第 4 章に，本論文の提案の一つである POM/MC を提案する．第 5 章に，本論文のもう一つの提案である実験的フォールトアナリシス手法を提案する．第 6 章に，提案した手法を用いたケーススタディについて述べる．第 7 章に，提案手法を評価し，考察を述べる．第 8 章に，本論文に示す研究に関連する技術を示す．第 9 章に，本論文をまとめる．

第2章　背景技術と準備

2.1　ソフトウェアテストとフォールトアナリシス

2.1.1　故障とフォールト

本研究では，フォールトアナリシスの手法を扱う．関係する用語として，エラー (error)，フォールト (fault)，フェイラー (failure) がある．一般に，error の結果が failure，failure の結果が fault という因果関係で捉えられることが多い．しかい, それぞれの言葉の意味はそれらを定義する団体により若干の相違がある．

例えば，システム工学とソフトウェア工学に用いられる語彙を集めた ISO/IEC/IEEE std 24765:2010 [31] では，error，fault，failure という語を次のように定義している．error は広範的な概念である．fault はソフトウェアやハードウェアが内在する間違いであり，failure は fault により引き起こされる製品の機能遂行能力の喪失である．

error 1. fault を含むソフトウェアのような間違った結果を生む人間の行動，2. 間違ったステップやプロセスやデータ定義，3. 間違った結果，4. 計算された，観測された，または計測された値や条件と，正しい，指定された，もしくは理論的に正しい値や条件との間の相違

fault 1. ソフトウェアに含まれる error の出現，2. コンピュータプログラムに含まれる間違ったステップ，プロセスまたはデータ定義，3. ハードウェアデバイスやコンポーネントに含まれる defect

failure 1. 製品が要求された機能を達成する能力を失うこと，または明確に定義された制限内での動作から逸脱すること，2. システムまたはシステムコンポーネントが定義された限界内で要求された機能を達成できなくするイベント

また，ソフトウェア工学の語彙集である IEEE std 610.12-1990[3] では，次のように定義している．ISO/IEC/IEEE std 24765:2010 に比べると各用語の意味が限定されている．fault は，ハードウェアやソフトウェアが内在する間違いであり，failure は fault により引き起こされる製品の機能を実行できない状態としている．

error 計算された，観測された，または計測された値や条件と，正しい，指定された，もしくは理論的に正しい値や条件との間の相違

fault ハードウェアのデバイスまたはコンポーネントの defect．コンピュータプログラムの間違ったステップ，プロセス，またはデータ定義

failure システムやコンポーネントが，指定された要求性能通りに機能を実行することができない状態

ISO/IEC 2382-14[2] の翻訳を含む JIS X 0014[1] では，以下のように定義している．障害 (fault) を機能単位の能力の縮退または喪失，つまり故障 (failure) を引き起こす異常な状態としている．上記２つの用語集では，fault はソフトウェアに含まれる間違いであったが，ここでは実行時の状態と捉えている．故障 (failure) は，要求された機能を遂行する能力がなくなることとしている．

誤差・誤り (error) 計算，観測若しくは測定された値または状態と，真の，指定された若しくは理論的に正しい値または状態との間の相違

障害 (fault) 要求された機能を遂行する機能単位の能力の，縮退または喪失を引き起こす，異常な状態

故障 (failure) 要求された機能を遂行する，機能単位の能力がなくなること

本論文では，観測された故障とその原因の特定を目的としており，以下の議論では各用語の意味を次の意味で用いる．

エラー (error) コンピュータプログラムの間違ったステップ，プロセスまたはデータ定義

フォールト (fault) 故障を発生させる原因となる状態や条件

故障 (failure) ソフトウェアがユーザの期待を逸脱した状態

また，フォールトアナリシスとは，既に観測された故障について，その原因となったフォールトを特定する行為をさす．故障再現とは，フォールトアナリシスのために，故障発生時のソフトウェアの振る舞いを再現する行為をさす．

2.1.2 ソフトウェアテストとデバッグ

本研究で扱うフォールトアナリシスのソフトウェア開発上の位置づけを示すために，関連する概念であるソフトウェアテストとデバッグについて述べる．ソフトウェアテストは，ソフトウェアを実行することで，その動作が期待と一致するかどうかを確認する行為である（本論文では動的テストをソフトウェアテストと呼び，静的テストについては言及しない）．ソフトウェアテストに関する技術者資格制度である JSTQB のシラバス [11] によれば，ソフトウェアのテストの目的には，フォールトの摘出，品質レベルの確認，意思決定のための情報の獲得，フォールトの作りこみの防止がある．また，テストの活動には，ソフトウェアの実行のほか，計画，コントロール，テスト条件の選択，テストケースの設計と実行，実行結果のチェック，テスト完了基準の検証，テストプロセスやテスト対象システムに関する報告，テストのまとめや終了作業を含む．なお，JSTQB におけるフォールトとは，ソースコードやドキュメントの欠陥（本論文でのエラー）を意味する．本シラバスによれば，フォールトアナリシスは，テストの目的の一部を構成するものである．

一方，類似する概念にデバッグがある．前出の ISO/IEC/IEEE24765:2010[31] では，デバッグを 1. コンピュータプログラムのフォールトを発見し特定し修正すること 2. プログラム中のエラーを発見し，特定し，除去すること，と説明している．本研究で扱うフォールトアナリシスは，デバッグの一部を構成する行為としても捉えることができる．

なお，本論文では，一般に組み合わせテストやシステムテストと言われるテストレベルや品質保証で発見された故障を対象としており，それらのテストはブラックボックステストやグレーボックステストとして実施することが多い．これは，故障発生時のシステム内部の振る舞いが観測不能となる一因となっており，本研究でフォールトアナリシスのおける故障再現を課題の一つとして扱う背景ともなっている．

2.2 モデル駆動開発とモデル変換

2.2.1 モデル

モデルとは，対象物を特定の観点から捉えた本質を抽出したものである．モデルという言葉の意味は，分野や文脈によって様々な意味で用いられる．例えば，命題論理におけるモデルとは，与えられた命題論理式集合を充足する解釈，すなわち命題に対する真偽値割り当てを意味する．後述するモデル検査におけるモデルとは，定義された性質を充足する解釈であり，モデル検査とは，記述されたシステムの振る舞いが性質のモデルとなっているか否かを検査することである．また，モデル検査の対象となるシステムの振る舞いを記述した情報をモデルと呼ぶこともある．ソフトウェア開発におけるモデルとは，ソフトウェアの構造や振る舞いを記述したものを意味することが多い．この意味でのモデル記法は多く提案されており，代表的なものとして UML(Unified Modeling Language)[25] がある．

本論文では，ソフトウェアプログラムのソースコードから特定の観点で抽出される情報を，モデルと呼ぶこととする．また，モデル検査の対象となる，ソフトウェアの振る舞いを表現するモデルを，振る舞いモデルと呼ぶ．

2.2.2 モデル駆動開発

モデルを用いたソフトウェア開発手法の一つに，OMG(Object Management Group) が提唱するモデル駆動開発 MDA(Model-Driven Architecture)[37] がある．MDA は，システムを記述する複数のモデルに基づくソフトウェア開発のためのフレームワークであり，様々な抽象度のモデルを表現できることが特徴の一つである．図 2.1 は MDA におけるモデルの概念を示している．CIM(Computation-Independent Model) は対象システムの計算機による実現に依存しないモデルある．例えば，システムが対象とする業務のビジネスプロセスモデルが該当する．PIM(Platform-Independent Model) は，計算機の実行プラットフォームに依存しないモデルである．例えば，プログラム言語に依存せずソフトウェアの構造を示すクラス図である．PSM(Platform-Specific Model) は，実行プラットフォームに依存するモデルである．例えば，Java 言語に依存したパッケージ図やクラス図である．MDA では，これらのモデル間で段階的な変換を行い，モデルを詳細化することで，ソフトウェアへの要求から設計，実装までをモデルに基づいて実現する．

MDA を実現するため，モデルの記法を定義する技術と，モデル間の変換を定義する技術を OMG が提案している．しかし，それらの MDA 関連技術は，上述したような要求から実装への変換に利用が限定される技術ではない．例えば，MDA 関連技術を利用して，システムモデルからテストモデルを生成する Model-Based Testing の手法も提案されている

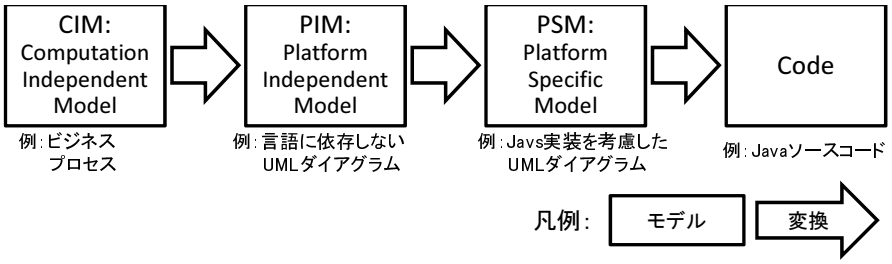

図 2.1: Model-Driven Architecture

表 2.1: MOF の 4 階層

階層	概説	例
M3	MOF そのものの自己定義	MOF2.0
M2	モデルの記法を定義するメタモデル	UML2.0
M1	メタモデルに基づいて記述されるモデル	UML ダイアグラム (クラス図など)
M0	モデルに基づいて実現される実体	システム

[39][12]．本研究でも，ソースコードからのモデル抽出を実現するために，これらの MDA 関連技術を利用する．

2.2.3 メタモデルとモデル変換

MDA を実現するための標準規格として，MOF(Meta-Object Faciliry)[23] が OMG によって提案されている．MOF は，表 2.1 のような 4 階層でしばしば説明される．M0 は，表現の対象となる実体であり，例えば動作するシステムそのものである．M1 は，システムを表現したモデルであり，例えば UML で記述されたモデルが該当する．M2 は，M1 のモデルの構造を定義するメタモデルであり，例えば UML の各ダイアグラムを定義するメタモデルである．M3 は，M2 のメタモデルの構造を定義するメタメタモデルである．MOF は M3 に該当し，メタモデルを定義するための構造を提供するとともに，MOF そのものが MOF による自己定義となっている．

モデルの実体は XML(eXtensible Markup Language) で表現される．XMI(XML Metadata Interchange)[22] は，XML を用いてメタデータ情報を交換するための規格である．XMI は，UML を扱うツール間でモデル情報をやり取りするために開発され，MOF のメタモデルに従うメタデータを扱うことができる．例えば，UML2.0 のメタモデルに基づいて記述するクラス図は，クラスとしての意味と図としての表現からなるが，XMI ではこれらのうち意味情報を扱う．

また，MOF に基づいたモデルを操作する技術として，QVT(Query/Views/Transformations)[21] がある．図 2.2 に，メタモデルとモデルの関係および QVT における変換の概念を示す．メタモデル MM_a とメタモデル MM_b は異なるメタモデルである．QVT による変換ルール R_{ab} は，2 つのメタモデル間のモデル要素の対応関係を定義するものである．メタモデル MM_a に基づくモデル M_a が存在するとき，モデル M_a に対して変換ルール R_{ab} を適用することで，メタモデル MM_b に基づくモデル M_b を得ることができる．

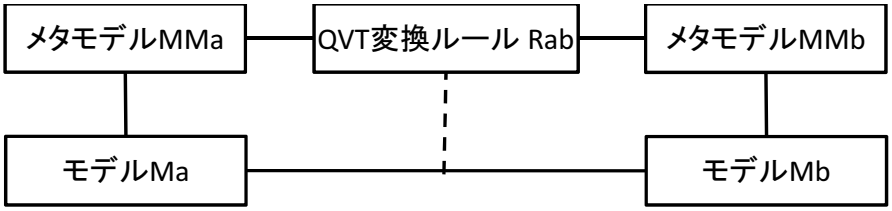

図 2.2: モデルとメタモデルおよび変換ルールの関係

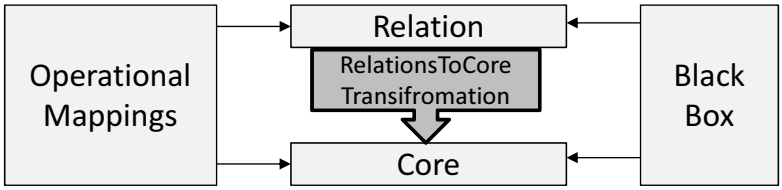

図 2.3: QVT の構成 [21]

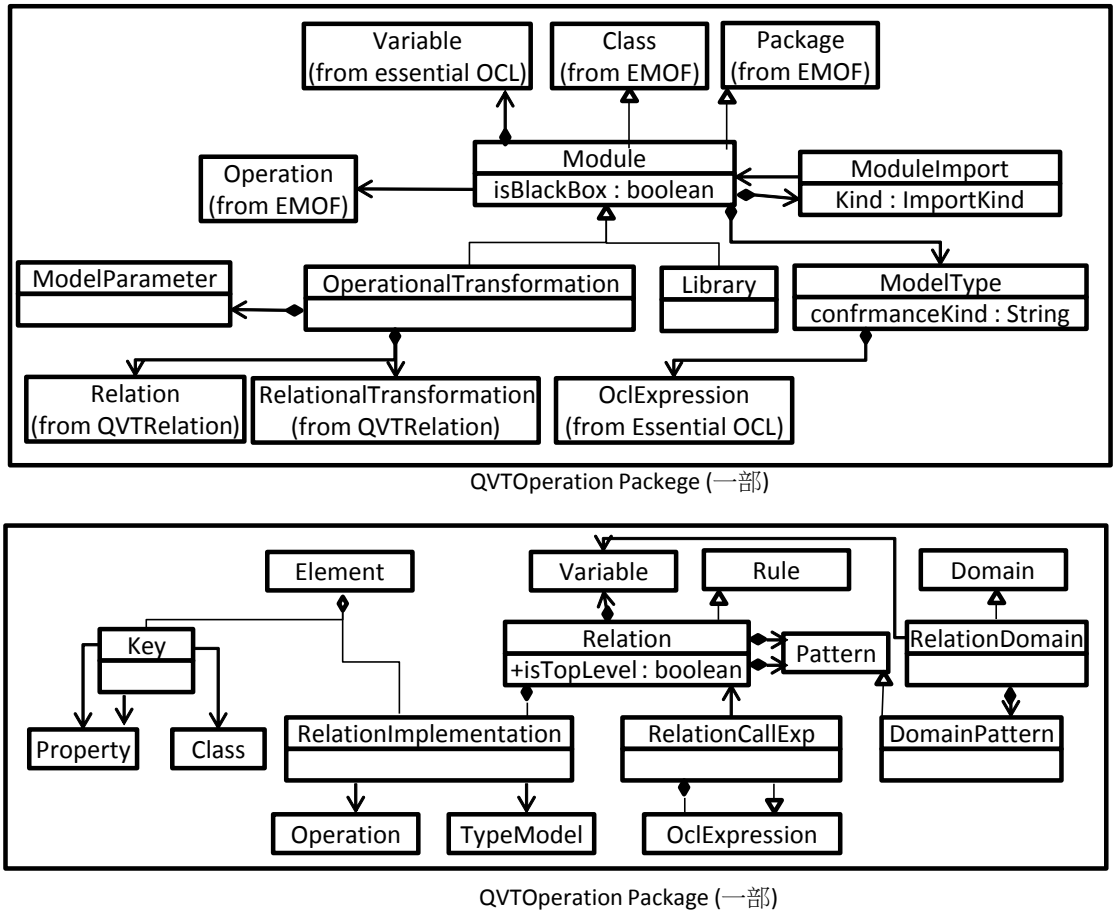

図 2.4: QVT のメタモデル ([21] より抜粋)

図 2.3 は，QVT の構成を示している．Core と Relations と呼ばれる基本的な構造を持つ．Core のメタモデルと言語は，EMOF と OCL の拡張を用いて定義され，メタモデル間の単純なパターンマッチとインスタンスの生成のみをサポートする．Relations は，より複雑なオブジェクトパターンマッチおよびオブジェクトテンプレートの生成をサポートする．Core と Relations の間では，対応関係が規定されており，Relations は Core によって表現可能である．Black box は，他の言語で表現された変換を呼び出す機構である．Operational Mappings は，命令型の言語であり，OCL の拡張となっている．副作用を持つ手続き型の記述により Core や Operations を用いた変換を定義することができる．

図 2.4 には，[21] より，QVT Operations に関わるメタモデルの一部を抜粋した．実際のメタモデルには，より多くの要素が含まれているので注意されたい．QVT/OperationalMapping のメタモデルが `QVTOperationalPackage` として定義されており，その中核はモデル変換を定義する `OperationalTransformation` である．また，`OperationalTransformation` は，QVT/Relation のメタモデルの要素である `RelationalTransformation` とメタモデル上の関連を持っており，QVT/OperationalMapping では，QVT/Relation を扱うことができる．また，QVT/OperationalMapping は，EMOF から Class や Package，Element といった構造を取り入れている．さらに，QVT/OperationalMapping および QVT/Relation は，OCL の拡張であり，変数や OCL の式を用いることができる．例えば，繰り返しを意味する `ForExp`(For Expression) や選択を意味する `SwithExp`(Switch Expression)， 制御の一つである `BreakExp`(Break Expression)，結果を返す `ReturnExp`(Return Expression) などである．QVT/OperationalMapping のこれらの文法を用いることで，メタモデルの要素間の変換を単純な対応関係でなく，手続き的な記述によりモデル間の変換を定義することができる．

また，本研究におけるツール実装では，Eclipse プロジェクトによる MOF に基づくモデリングフレームワークである EMF(Eclipse Modeling Framework) を構成する EMT(Eclipse Modeling Tool) と，QVT/operationalMapping の部分実装である QVTo (Operationl QVT) をツール基盤として用いる．また，XMI で表現されるターゲットモデルをテキストコードとして出力するため，MOF M2T[24] の実装である Acceleo を用いる．

2.3 モデル検査

モデル検査法とは，システムが満たすべき性質を定義した論理式 φ と，システムの動作を表現した振る舞いモデル M が与えられたとき，M が φ のモデルとなっているか否か，すなわち，$M \models \varphi$ であるか否かを決定する方法である．

2.3.1 SPIN

本研究で用いたモデル検査器 SPIN(Simple Promela INterpreter) は, ω オートマトン (Büchi オートマトン) を論理基盤としている [29]．SPIN は, 専用の仕様記述言語である Promela(後述) で記述した対象システムのモデル P と，時相論理 (Linear Temporal Logic; LTL) 式 f とを入力として受け付け,P を Büchi オートマトン $S(P)$ に変換する．また，P による f の充足可能性問題を次の問題として解釈する：$S(P)$ の ω-run σ に対して f が成

```
active proctype counter()
{
  do
    :: count++
    :: count--
    :: (count == 0) ->break
  od
}
```

図 2.5: Promela コードの例 (do)[30]

立するか否か，すなわち $\sigma \models f$ である．SPIN は P と f の両方を Büchi オートマトンに変換してこの問題を解いている．SPIN は，$S(P)$ が成り立たない ω-run を発見すると，これを反例として出力する．本論文では，$\overline{P, f}$ は，モデル P について f の反例を示すこととする．

また，SPIN では LTL 式のほかに，活性 (liveness) や Promela コード中に記述された表明などの検証も行える．本論文で提案する手法の実践でも，性質を定義するために LTL 式のほか表明も用いている．

2.3.2 Promela

Promela(Process/Protocol meta laguage) は，SPIN の入力となる仕様記述言語である．非同期的に並行動作する複数プロセスの振る舞いを手続き的に記述できる．文法の一部は C 言語に似ているが，特徴的な文法も持つ．後の議論で用いる文法について，例を用いて概説する．例は，Spin Home page で公開されている Promela 言語リファレンス [30] より引用する．

図 2.5 は，Promela の言語リファレンスから引用した Promela コードの例である．`proctype` がプロセス定義を示し，`counter` という名前のプロセスを定義している．`active` は，該当のプロセスが初期に起動するプロセスであることを示す．`do` は繰り返しを意味する．各`::` で始まる行は，オプショナルシーケンスと呼ばれ，ガード条件が満たされるオプショナルシーケンスのうちの一つが非決定的に選択されて実行される．図の例では，`count == 0` がガードであり，他のオプショナルシーケンスにはガードは定義されていない．`count == 0` が成立する場合には，3 つのオプショナルシーケンスのうちのいずれかが実行される．成立しない場合は，他の 2 つのオプショナルシーケンスのいずれかが非決定的に実行される．この例は，変数 `count` の値を非決定的に増加または減少させ，値が 0 になったときのみ非決定的に終了することになる．ここで，このプロセスの実行パターンは無限に存在する．このような非決定的な動作は他の手続き型言語ではあまり見られないが，仕様を記述する際には，振る舞いの可能性を網羅的に記述するために有用である．

図 2.6 は，プロセスの実行の例を示している．`init` は初期プロセスを意味する．`init` プロセスにおいて，`run` 演算子によってプロセス A を実行開始している．

また，プロセス間の通信はチャネルによって行われる．図 2.7 の例では，`chan` により `byte` 型の値を格納できる長さ 8 のチャネル `set` を定義している．`!`と`!!`が送信を表し，`?`と`??`が受信を表す．`!`と`!!`につづく引数が送信するメッセージであり，`?`と`??`につづく引数には変数名またはパターンを指定できる．`set?x` では，チャネル `set` の先頭のメッセージ

```
proctype A(byte state; short set)
{
  (state == 1) -> state = set
}

init {
    run A(1, 3)
}
```

図 2.6: Promela コードの例 (プロセス実行)[30]

```
chan set = [8] of { byte };
byte x;

set!!3; set!!5; set!!2;/* sorted send operations   */

set?x;                  /* get first element        */
if
:: set?                  /* copy first element       */
:: set??5               /* is there a 5 in the set? */
:: empty(set)
fi
active proctype counter()
{
  do
    :: count++
    :: count--
    :: (count == 0) ->break
  od
}
```

図 2.7: Promela コードの例 (メッセージ通信)[30]

表 2.2: SPIN で利用可能な時相記号

記号	名称	説明
`[]`	always	命題 `p` に対して，`[]p` は，常に `p` が成り立つことを意味する
`<>`	eventually	命題 `p` に対して，`<>p` は，将来のある時点で `p` が成り立つことを意味する
`U`	until	命題 `p` と `q` に対して，`p U q` は，将来のある時点で `q` が成り立つまで持続的に `p` が成り立ち続けることを意味する

を受信し変数 `x` に代入する．`set??5` では，チャネル `set` のどこかに `5` というメッセージがあれば，それを取り出す．チャネルの長さが 1 以上の場合には，送信側はチャネルにメッセージを格納すれば実行を進められるため，非同期通信を実現できる．チャネルの長さが 0 の場合には，送信側は受信されるのを待つため，同期通信を実現できる．なお，キューがフルになった場合に受信を待つか，メッセージを捨てて実行を先に進めるかは，SPIN の実行時オプションによって指定できる．

このように，Promela はプロセスの並行実行やプロセス間通信をその言語仕様の中に持っていることも特徴の一つである．そのため，Promela は，並行システムの動作仕様の記述と検証に用いられることが多い．

2.3.3 Linear Temporal Logic

Linear Temporal Logic (LTL; 線形時相論理) は，命題論理に時間に関する様相記号（時相記号）を加えた時相論理の一つである．すなわち，命題に対する真偽値割り当ての列について，その列が満たす性質を論理式で表現することができる．本論文では LTL については議論をしないため，意味論の厳密な定義は省略する．本研究で用いるモデル検査器である SPIN が採用している時相記号は，`[]`，`<>`，`U` の 3 つである．各時相記号の意味を表 2.3.3 に概説する．本論文では，SPIN で採用している時相記号に従って，LTL 式を以下のように定義する．

P を命題の集合とする．

- `p` $\in P$ であるとき，`p` は LTL 式である．
- `p`, `q` が LTL 式であるとき，$\neg$ `p`，`p` $\wedge$ `q`，`p` $\vee$ `q`，`[] p`，`<> p`，`p U q` は LTL 式である．

SPIN では，命題として Promela の式を定義することができる．例えば，Promela コード上の変数 `v` について，`0` にはならないという性質は，まず命題 `p` に対して “`#define p (v == 0)`” という定義を行った上で，LTL 式 “`! [] <> p`” を性質として表現できる．(SPIN の受理する LTL 式では，`!` は否定 ($\neg$) を意味する．また，SPIN は実際には never claim を入力とするため，検証したい性質の否定をとる必要がある．しかし，この否定はツールで自動的に行うことができ，また議論が複雑になるため，本論文では否定をとらない検証したい性質そのものを SPIN に入力する LTL 式を表現する．)

2.4 仮説演繹法

仮説演繹法とは，科学哲学の分野で論じられる古典的な確証モデルであり，Popper[41][42] や Hempel[26] らによって検討された．仮説演繹法では，観測された事実に基づき仮説を設定し，仮説から予測を導く．予測が成り立つか否かの実験を行い，その結果に基づき仮説の確からしさを確証する．すなわち，実験結果が予測と一致する場合には，予測を演繹的に導いた仮説の確からしさが高まったと蓋然的に考える．実験結果が予測と一致しない場合には，予測を演繹に導いた仮説が誤っていたと演繹的に考える．仮説演繹法のアルゴリズム的記述は次のようである [19].

1. 観測する．
2. 観測を説明する仮説を立てる．
3. 仮説から観測可能な予測を導く．
4. 予測が正しいかどうかを確認する．予測が正しければステップ 3 に戻る．
   予測が間違っていれば仮説を却下し，ステップ 2 へ戻る．

ここで注意すべきは，ステップ 2 は帰納推論であり，ステップ 3 は演繹推論である点である．帰納推論は，個別の事実から一般的な法則を見いだそうとする推論であり，演繹推論は，一般的な前提から個別的な結論を導く推論である．演繹推論は必然的に正しいと言えるが，帰納推論は蓋然的に正しいに過ぎない．演繹推論が科学的に正しいことは論をまたないが，帰納推論を科学的にどう扱うかは科学哲学の古典的な問題である．なぜならば，科学の基本となる観察や実験は個別の事象であり，そこから一般的な法則を見いだすことは帰納推論に他ならないにも関わらず，帰納推論は推論の正しさが担保されないためである．

仮説演繹法は，観測した個別事象に基づく仮説の設定を認めた上で，仮説からの演繹で具体的な予測を立て，予測の正しさを実験や観測で確認する．予測が正しければ，それを導いた仮説の確からしさが増したと考え，次の予想の確認を行う．予測が正しくなければ，それを演繹的に導いた仮説は演繹的に却下される．これを仮説が十分に確からしいと認められるまで繰り返す．このことにより，仮説自体は帰納推論であるが，その確からしさを確証する (もしくは棄却する) 手順は演繹推論として扱うことができるのが特徴である．

第3章　研究のアプローチ

本研究で提案するフォールトアナリシス手法は，大きく次の 3 点の提案からなる．

1. 故障再現にモデル検査を用いる提案

2. モデル検査のための振る舞いモデルをソースコードから抽出する提案

3. モデル検査によるフォールトアナリシスのプロセスを形式化する提案

本章では，1〜3 について以下に概要を示す．2 については 4 章で，3 については 5 章で，それぞれ詳細を示す．

3.1　モデル検査による故障再現

本論文では，ソフトウェアの実行では再現が困難な故障に対して，その再現のためにモデル検査を用いることを提案する．図 3.1 に，モデル検査による故障再現の基本概念を示す．図 3.1(a) は，図 1.5(a) と同じである．図 1.5(b) で見たように，故障発生時と同じ初期条件で対象ソフトウェアを動作させても，並行性や非決定性により遷移しうる状態は複数存在し，故障に至る確率が低い場合が多いため，故障の再現が困難である．

そこで，図 3.1(b) のように，モデル検査を用いた故障再現の方法を考える．まず，観測された故障状態の否定を性質 φ として定義する．すなわち，φ を満たすときには故障は発生しておらず，φ が満たされないときに故障が発生したとみなす．また，ソフトウェアの振る舞いを振る舞いモデル M として定義する．モデル検査によって $M \vDash \varphi$ であること，つまり故障は発生しないことを検査する．振る舞いモデル M が故障を含むソフトウェアの振る舞いを表現しているならば，φ に反する状態が存在することになり，モデル検査は失敗して反例が得られる．この反例は，φ が満たされない状態に至る振る舞い，すなわち故障が発生する振る舞いを示している．本論文では，この反例を得ることを，故障の再現とみなす．初期状態からのソフトウェア実行と異なり，モデル検査では故障の発生確率に影響を受けることなく，故障状態を含む反例を得ることができる．

従来，筆者らは以上の考えに基づき，モデル検査を用いた故障再現を製品開発に適用してきた．図 3.2 は，実施した手順を示している．テストで故障が発生した場合，故障を特徴づける条件の否定を LTL 式として表現する．また，製品の仕様書およびソースコードを参照し，ソフトウェアの振る舞いを Promela コードとしてモデリングする．このように作成した LTL 式と Promela コードに対して SPIN を用いてモデル検査を実施する．振る舞いモデルが故障を含めば，反例として故障に至る実行パスが得られる．さらに，故障に対する修正を行ったのち，修正後の振る舞いを表現した振る舞いモデルを作成し，振る舞いモデルにおいて故障が発生しないことを確認する修正確認にもモデル検査を用いた．

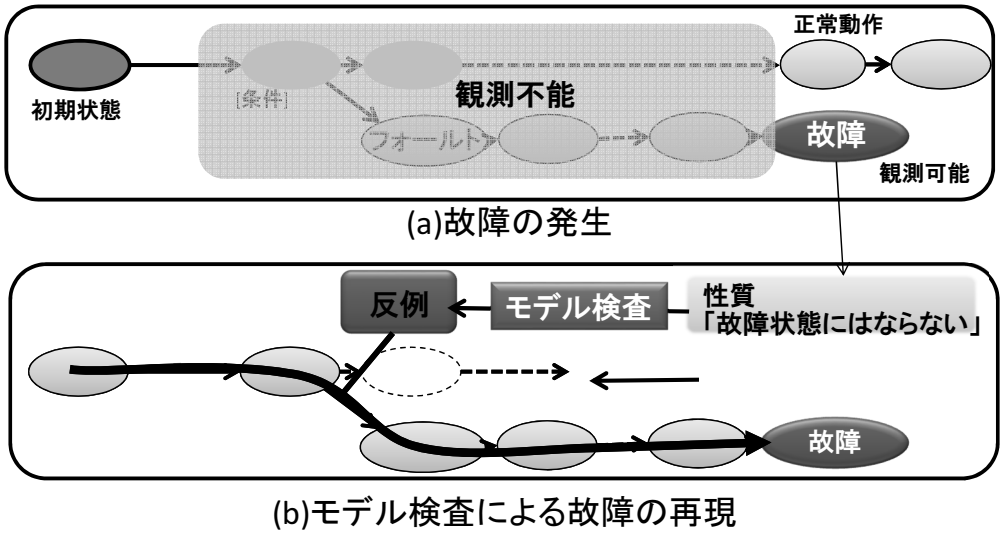

図 3.1: モデル検査による故障再現

表 3.1: 故障再現のためのモデル作成でのトレードオフ

情報の保持	情報の捨象
C 言語と Promela の意味論の対応づけ	状態爆発を回避するための振る舞いの縮退
故障を再現する振る舞いの維持	反例の理解性を上げるための絞り込み

このようなモデル検査による故障再現を実施する上で考慮すべき点がある．元のソースコードの情報の保持と捨象のトレードオフ (表 3.1) である．振る舞いモデルのとりうる状態数が多い場合，モデル検査を実行するために大量の時間やメモリ量を必要とし，現実的な計算機リソース上でモデル検査を実施できない場合がある．このモデル検査にかかる問題は，一般的に状態爆発と呼ばれる．状態爆発を回避するため，振る舞いモデルを改変し情報の捨象が行われる．しかし，情報を捨象しすぎると，故障状態を含まない振る舞いモデルとなってしまい，故障が再現されない．

そのため，従来のモデル検査を用いた故障再現では，適切に情報を捨象した振る舞いモデルを作成する試行錯誤を要した．図 3.3 は，試行錯誤の概念を示している．(a) では，ソースコードの多くの部分を振る舞いモデルで表現したため，状態数が多く状態爆発が発生する．そのために，振る舞いモデルを作りなおす必要がある．(b) では，ソースコードの一部を振る舞いモデルで表現し状態爆発を回避したが，故障状態を含まない振る舞いモデルのため，故障が再現されない．(c) では，故障状態を含むが，故障に至る状態遷移が元のソースコードには存在しない実行パスであり，この反例はいわゆる偽反例である．(d) では，故障状態を含む反例を得られたが，反例が複雑なためにフォールトを特定できない．(e) において適切に故障を再現しフォールトを特定できる．

このような振る舞いモデルの試行錯誤的な作成は自動的に行うことはできず，モデル検査による故障再現には 1〜10 週間を要した．以上のような実務における経験に基づき，モデル検査を用いた故障再現を実用化するためには，振る舞いモデル作成の試行錯誤をサポートする技術が必要であると判断した．

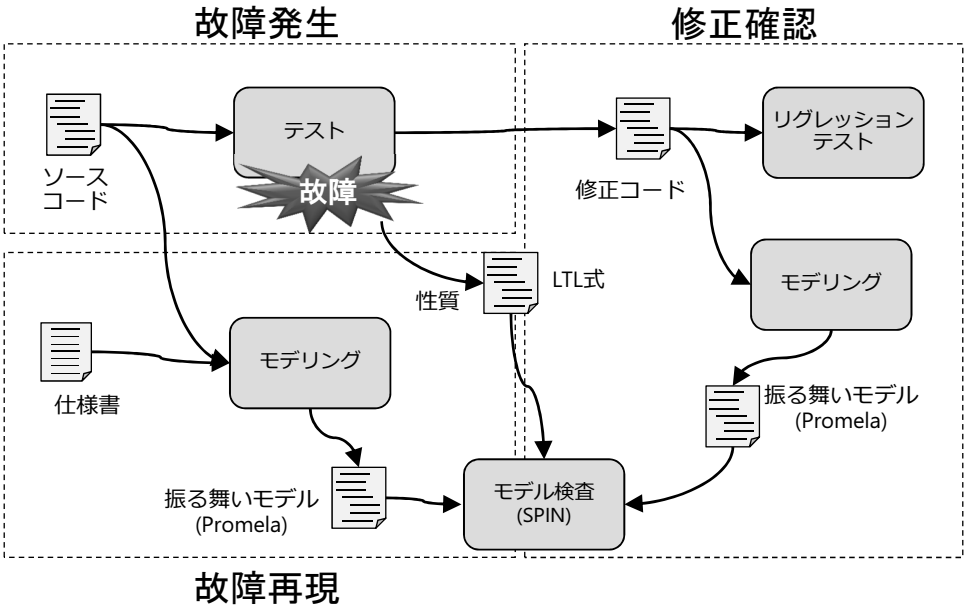

図 3.2: モデル検査による故障再現手順

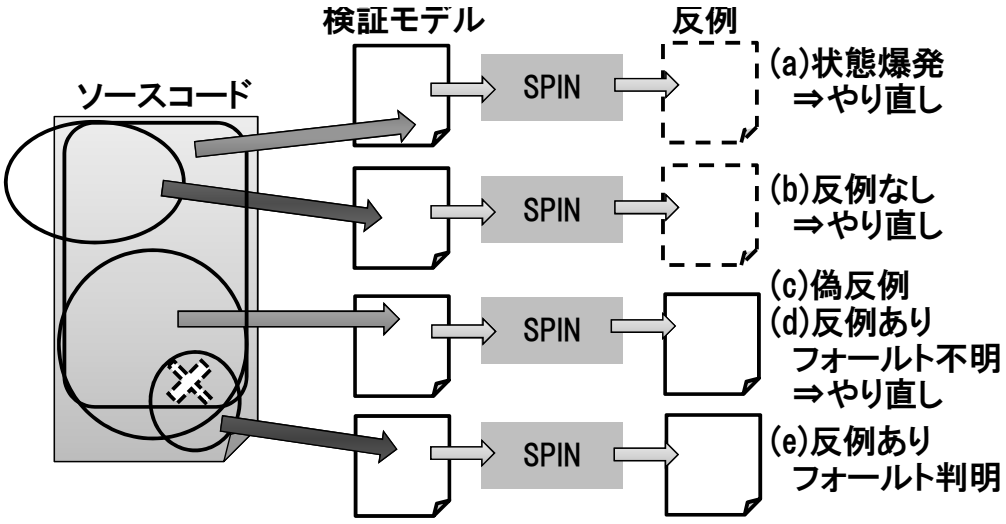

図 3.3: モデル検査による故障再現手順

3.2 ソースコードからのモデル抽出

3.2.1 ソースコードからの振る舞いモデル抽出の要件

3.1 節で述べたように，モデル検査による故障再現の実務経験から，モデル検査のための振る舞いモデル作成の支援が必要であると判断した．本章で，ソースコードからの振る舞いモデル抽出のアプローチを提案する．以降では，ソースコードは C 言語で書かれており，振る舞いモデルは Promela で表現することを想定する．

振る舞いモデルの抽出では，以下の点を考慮する．まず，ソースコードと振る舞いモデルの言語間の意味論の変換を行う．一般に，ソフトウェアプログラムに用いる言語と，モデル検査の振る舞いモデルに用いる言語では意味論が異なる．そのため，意味論の異なる言語間の意味論の変換（マッピング）が必要となる．また，実行環境に関わる変換も必要である．例えば，C 言語で並行プログラムを実現する方法の一つに Pthreads ライブラリがある．Pthreads ライブラリ自体が C 言語で書かれているため，それを Promela 言語に変換する手法も考えられるが，状態数が増加し状態爆発を引き起こす原因となる．しかし，Promela 言語には独自のプロセス単位での並行化機構を持ち，それを用いて Pthreads ライブラリの動きを模擬することで状態数を削減することができる．このように，ソースコードの記述に用いられている言語の構成要素だけでなく，ライブラリなどの実行環境も適切に振る舞いモデルの言語要素に変換することで，現実的なモデル検査を実現できる．OS(Operating System) やミドルウェアなどの扱いも同様である．

3.1 節で述べたように，故障再現のための振る舞いモデルの作成では，情報の捨象と保持のバランスを取るための試行錯誤が行われる．つまり，ソースコードを一対一に振る舞いモデルに翻訳するのではなく，故障再現を行う開発者が情報の取捨をコントロールできる必要がある．また，その際には，意図的にソースコードとは異なる振る舞いを検証モデルで表現するような，破壊的な変更も行われる．

3.2.2 Program-Oriented Modeling (POM) コンセプトフレームワーク

このような意味論の変換と，意図的な情報の取捨選択を行うため，ユーザが抽象化方法を指定可能なモデル抽出のアプローチを提案する．これを，Program-Oriented Modeling (POM) と名づける．

POM は，ソフトウェアのソースコードから，ユーザが指定する観点でモデルを抽出するコンセプトフレームワークである．2.2.1 で述べたように，モデルとは対象物を特定の観点から捉えた本質を抽出したものである．つまり，対象物そのものではなく，それを用いる観点から必要な情報のみを取り出して構造化することに意味がある．POM は，ソースコードを対象物として，ソフトウェア開発における様々な観点に向けたモデルを抽出することを指向している．ソフトウェア開発における観点とは，例えば本論文で用いるようなプログラムの並行実行の検証のためのプログラムの振舞いや，ソフトウェア理解のためのプログラムの構造などがある．ソフトウェア開発において望まれる観点は様々であるため，開発者が必要とする観点，すなわちプログラムからモデルとして抽出する情報を指定できることが望ましい．この抽出するモデルの指定には，どのような情報をどのような形式で抽出するか，という点を指定できる必要がある．つまり，プログラムから情報の抽出と，

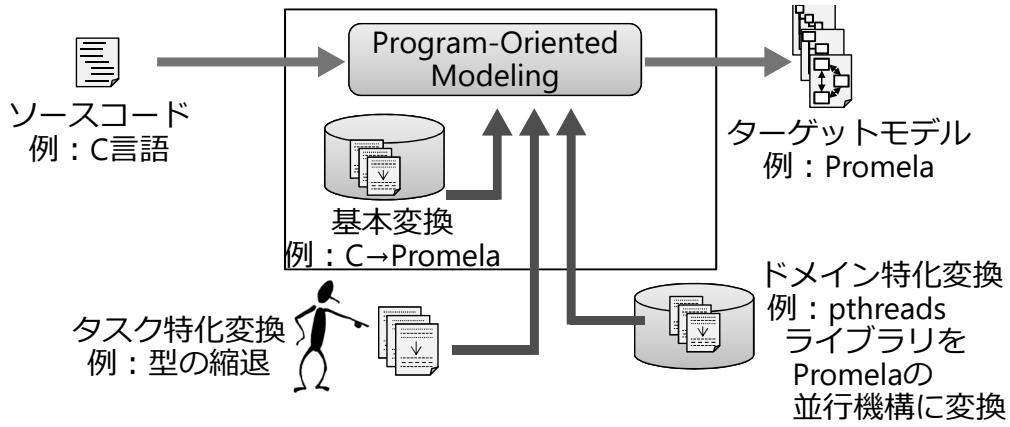

図 3.4: POM の基本概念

ソースコードからモデルへの構造の変換の 2 点である．POM コンセプトフレームワークは，これらについてユーザが定義する仕組みを持つ．

図 3.4 に，POM の基本概念を示す．POM への入力はソースコードであり，POM が抽出するモデルをターゲットモデルと呼ぶこととする．ソースコードからターゲットモデルを抽出する方法は，次の 3 種類の変換によってユーザが定義する．

基本変換 ソースコードとターゲットモデルとの間の変換 (マッピング) を行う．例えば，C 言語と Promela との間の意味論を対応づける．

ドメイン特化変換 対象システムのドメインに特化した変換を行う．例えば，Pthreads ライブラリの利用を，Promela の並行実行機能に対応づける．

タスク特化変換 ターゲットモデルを用いて行うタスクに特化した変換を行う．例えば，モデル検査における状態爆発の回避のために，変数の型をより小さな型に置き換えるような変換をユーザが意図的に指示する．

これらのうち，基本変換はソースコードとターゲットモデルを記述する言語によって決定する．タスク特化変換は，ユーザの意図によって選択される．ドメイン特化変換は，ドメインによって決定するが，具体的な変換方法はユーザの意図によって選択される部分もある．POM では，これらの変換を分離し，また，各変換を変換ルールとして定義することで，変換方法の置き換えを実現する．これらにより，意味論の変換と意図的な情報の取捨選択を行う，構成可能なモデル抽出を実現する．

POM が実現するものが構成可能なモデル抽出であることからわかるように，ユーザが抽出したい情報については，それを抽出する変換ルールを与える必要がある．つまり，POM は言語間の自動変換を指向するものではなく，ユーザが指定する変換ルールに基づく半自動抽出である点に注意されたい．変換ルールが指定されれば，それに基づくモデル抽出は自動的に行われる．逆に言えば，入力ソースコードが持つ情報のうち，変換ルールで抽出されない情報は捨てられる．また，変換の正しさは変換ルールに依存する．

3.2.3 POM/MC: POM for Model Checking

また，POM コンセプトフレームワークに基づいて，C 言語で記述されたソースコードから，Promela コードで書かれた振る舞いモデルを抽出するツールである POM/MC (POM for Model Checking) を提案する．また，POM/MC では，抽出した振る舞いモデルに対して SPIN によるモデル検査を実施し，その結果である反例を可視化する機能を持つ．POM/MC により，ユーザが変換ルール集合を指定し，それに基づく振る舞いモデルを抽出し，モデル検査を実施し，その結果を解析するという，故障再現のためのモデル検査の一連の流れを実現する．

なお，POM/MC が Promela モデルを抽出点において，C 言語を Promela 言語に変換することは，POM/MC の本質ではない．本研究で扱うフォールトアナリシス，特に並行性を持つプログラムの故障再現という観点に対して，並行プロセスの振る舞いを表現するモデルを抽出することがモデル抽出の意味である．ターゲットモデルの記述言語が Promela であることは，モデルを検査する際に SPIN を用いるための要請である．また，Promela コードの意味が C 言語コードと同一かという点において，POM フレームワークおよび POM/MC はツールそれを保証する仕組みは持たない．むしろ，ユーザが指定する変換ルールに従って，C 言語コードとは厳密には意味が異なるが，その並行動作を表現するモデルを抽出することに注意されたい．

3.3 実験的フォールトアナリシスプロセス

ついで，フォールトアナリシスのプロセス (手順) を定義する．これまでに述べたように，フォールトアナリシスは試行錯誤を伴う．POM/MC を用いる際には，振る舞いモデルを抽出するための変換ルール集合を選択しなければならない．その手順を明確にすることで，モデル検査を用いたフォールトアナリシスを開発者らが実践できるようにする．

本論文では，フォールトアナリシスプロセスを定義するため，フォールトアナリシスの特徴づけを行い，フォールトアナリシスモデルを定義する．次に，試行錯誤の手順を形式化するため，仮説演繹法の枠組みを用いて，上記フォールトアナリシスモデルを拡張する．さらに，拡張したフォールトアナリシスモデルに POM/MC を適用したプロセスを提案する．本論文では，仮説演繹法に基づく試行錯誤を実験と捉え，提案するフォールトアナリシスのプロセスを，実験的フォールトアナリシスプロセスと呼ぶこととする．

モデル検査による故障再現は，発生確率に関する問題を回避する一方で，その結果の解析は困難である．これに，仮説を立てた上での実験と解析を組み合わせることで，フォールトアナリシスが容易となることが期待できる．また，条件を絞り込んだ状態で実験を行うことは，モデル検査の実用化の大きな課題である状態爆発の回避も期待できる．

3.4 例題

先に示した運搬装置は実製品であり，本論文で示す提案の説明に用いるには大規模かつ複雑で理解が困難である．以降では説明用の例題として，queue_BUG.c と呼ばれるサンプルプログラム（以降，キュープログラムと呼ぶ）を用いる．キュープログラムは，ソー

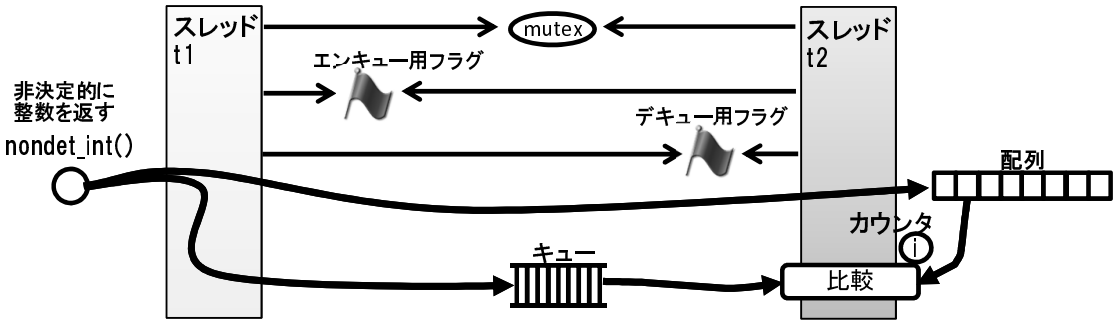

図 3.5: キュープログラムの構成図

```
void *t1(void *arg)
{
   int valu   e, i;

   pthread_mutex_lock(&m);
   value = nondet_int();
   if (enqueue(&queue,value)) {
      goto ERROR;
   }

   stored_elements[0]=value;
   if (empty(&queue)) {
      goto ERROR;
   }

   pthread_mutex_unlock(&m);

   for(i=0; i<(SIZE    - 1); i++)
   {
      pthread_mutex_lock(&m);
      if (enqueue_flag)
      {
         value = nondet_int();
         enqueue(&queue,value);
         stored_elements[i+1]=value;
         enqueue_flag=FALSE;
         dequeue_flag=TRUE;
      }
      pthread_mutex_unlock(&m);
   }

   return NULL;

   ERROR:
      ;
}
```

```
void *t2(void *arg)
{
   int i;

   for(i=0; i<SIZE; i++)
   {
      pthread_mutex_lock(&m);
      if (dequeue_flag)
      {
         if
(!dequeue(&queue)==stored_elements[i]) {
            goto ERROR;
            ERROR:
               ;
         }
         dequeue_flag=FALSE;
         enqueue_flag=TRUE;
      }
      pthread_mutex_unlock(&m);
   }

   return NULL;
}
```

図 3.6: キュープログラムのコード例（部分）

スコードモデル検査ツールのコンペティションである SV-COMP [8] のベンチマークプログラムの一つとして公開されている．キュープログラムの概要構成図を図 3.5 に示す．また，キュープログラムのコードの一部を図 3.6 に示す．キュープログラムのコード全体は，付録 A.1 に掲載する．

キュープログラムは，1 つのキュー (FIFO) と 2 つのスレッド (*t1* および *t2*) からなる．スレッドは，Pthreads ライブラリを用いて実装されている．スレッド *t1* は，関数 `nondet_int` を用いて，非決定的に整数値を決定し，これをキューに格納する．同時に，スレッド *t1* は，同じ値を配列 `stored_elements` に記録する．スレッド *t2* は，キューから順に整数値を取り出す．2 つのスレッド *t1* と *t2* との間では，キュー操作について mutex により排他制御がなされ，2 つのフラグ `enqueue_flag` と `dequeue_flag` によりキュー操作順序が制御される．スレッド *t2* は，キューから整数値を取り出すとき，配列 `stored_elements` と比較して，キューに格納された整数値が順番に取り出されたことを確認する．プログラムには，いくつかのエラー検出コードが組み込まれている．例えば，スレッド *t2* において，キューに格納された値と配列に記録された値に相違が発見された場合，キュープログラムはエラーを表示して動作を停止する．実際，そのように停止する故障があることが知られている．

第4章　POM/MC: ソースコードからのモデル抽出とモデル検査

本章では，3.2 節で示したソースコードからユーザが意図するモデルを抽出する POM コンセプトフレームワークと，POM に基づいてモデル抽出とモデル検査を実現する技術である POM/MC の詳細を示す.

4.1　POM:Program-Oriented Modeling

4.1.1　POM コンセプトフレームワーク

POM(Program-Oriented Modeling) は，本研究が提案する，プログラムのソースコードからプログラムのモデルを抽出するアプローチである．図 4.1 に，POM アプローチを実現するための，コンセプトフレームワークを示す．POM コンセプトフレームワークとは，ソースコードからターゲットモデルを抽出し，これをターゲットコードとして出力するフレームワークである．POM コンセプトフレームワークは，構文解析，中間モデル，ターゲットコード生成からなる．入力はソースコード，出力はターゲットコードである．ソースコードを構文解析した結果は中間モデルの初期モデルとなる．ユーザが定義するモデル変換集合に基づき段階的に中間モデルを変換する．最終的な中間モデルは，テンプレートに従ってターゲットコードとして出力される.

POM の有する中間モデルには次の 3 種類からなる：

実装モデル パーサーによるソースコード解析によって得られるプログラムの構造を表現する.

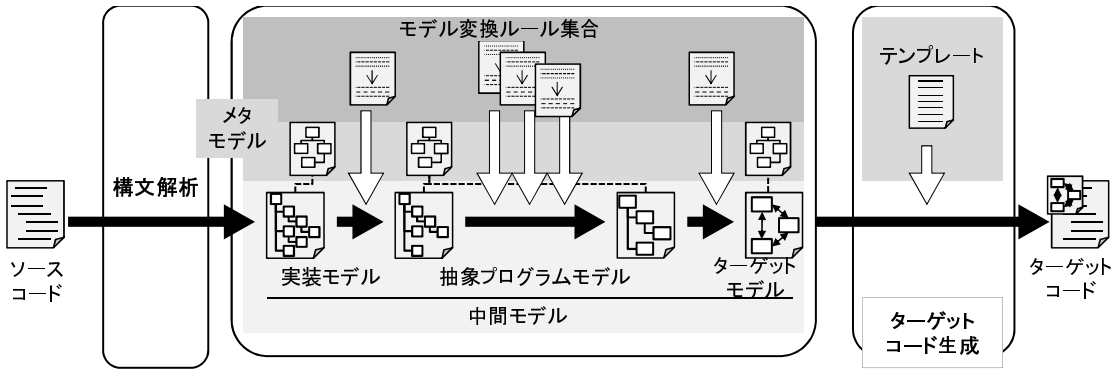

図 4.1: POM コンセプトフレームワーク

抽象プログラムモデル 実装モデルとターゲットモデルとをつなぐ役割を果たすモデルであり，例えばC言語のような特定のプログラミング言語に依存しない抽象表現を持つ．また，抽象化の対象となるモデルである．

ターゲットモデル 出力となるターゲットコードの内部表現であり，ターゲットコード生成器によりモデルからテキスト表現へと変換される．

各モデルの構造と意味を定義するため，それぞれに対応したメタモデルを定義する．すなわち，実装メタモデル，抽象プログラムメタモデル，ターゲットメタモデルである．

また，3.2.2節に示した3種の変換を実現するため，変換ルール集合を定義する．基本変換は，実装モデルと抽象プログラムモデルの間の変換ルールと，抽象プログラムモデルとターゲットモデルの間の変換ルールにより実現する．タスク特化変換とドメイン特化変換は，抽象プログラム間での変換ルールで実現する．

ここで，POMコンセプトフレームワークを，実装モデル，抽象プログラムモデル，ターゲットモデルの3種の中間モデルにより構成した理由を示す．その理由は，(1) 言語非依存化，(2) 言語変換とモデル抽象化の分離の2点である．それぞれについて説明する．

まず，(1) 言語非依存化について述べる．本論文においてPOMフレームワークを，モデル検査を実施するためにC言語ソースコードからPromelaモデルを抽出する目的に用いた．しかし，POMフレームワークは，これに特化したフレームワークではない．ソフトウェアのソースコードから，ユーザの目的に合った任意のモデルを抽出することを目的とした包括的なフレームワークである．したがって，入力となるソースコードを記述するプログラム言語はC言語に限らず，Javaなどの他の言語であることも想定している．同様に，出力となるターゲットコードを記述する言語はPromelaに限らず，UMLなどの他の言語であることも想定している．POMコンセプトフレームワークでは，実装モデルが入力言語に，ターゲットモデルはターゲットコードの言語に依存する．実装メタモデルをC言語ではない他の言語のメタモデルに置き換えることで，その言語を入力言語として扱うことができる．その際には，入力言語の構文解析器と，実装モデルから抽象プログラムモデルへの基本変換を，入力言語に合わせて作成すればよい．同様に，ターゲットメタモデルを他の言語のメタモデルに置き換えることで，その言語を出力言語として扱うことができる．その際には，抽象プログラムモデルからターゲットモデルへの基本変換と，ターゲットコード生成のためのテンプレートを出力言語に合わせて作成すればよい．一方，抽象プログラムモデルは，特定の言語に依存しない抽象表現を持ち，入出力言語への依存は少ない．したがって，入力言語に関わる変更では実装メタモデルとその周辺だけを考慮すればよく，出力言語に関わる変更ではターゲットメタモデルとその周辺だけを考慮すればよい．入力言語と出力言語の組み合わせを変更できることが，本フレームワークの利点の一つである．

また，システムによっては，複数のプログラム言語で書かれたソースコード群から構成される場合がある．その場合には，複数のプログラム言語用の実装モデルおよび構文解析器と基本変換を作成することで，言語の異なるソースコード群を一つの抽象プログラムモデル上で統合的に扱うことも意図している．

次に，(2) 言語変換とモデル抽象化の分離について述べる．たとえば，C言語からPromelaモデルを抽出する場合，C言語とPromela言語という異なる言語間での文法および意味論の変換と，Promela言語にどのような情報を抽出するかというモデルの抽象化との2つの

観点を考慮する必要がある．POM フレームワークでは，これらを独立して表現することが可能である．言語間の変換については，上記 (1) 言語非依存化で述べたように，実装メタモデルとターゲットメタモデル，および，基本変換によって実現できる．モデル抽象化については，抽象プログラムモデル上での変換で実現される．これにより，モデル抽象化の定義においては，言語を意識することなく抽象化の意味のみに集中して検討することができる．ただし，後述する POM/MC ツールの実装においては，入力言語として C 言語，出力言語として Promela 言語のみを実現している．

4.1.2 POM コンセプトフレームワークの定式化

図 4.1 に示した POM コンセプトフレームワークを定式化する．
まず，メタモデルとモデルの関係を定義する．

定義 1 (メタモデルとモデル) mm をメタモデルとする．m が mm に基づいて定義されたモデルであることを，$m : mm$ と表記とする．

例 1 (メタモデルとモデル) C 言語メタモデルを mm_C とし，mm_C に基づいた C 言語モデルを m とすると，$m : mm_C$ である．

次に，モデル変換に関する表記を定義する．

定義 2 (モデル変換) mm_a と mm_b をメタモデルであるとする．モデル変換 $r_{a\to b}$ が mm_a から mm_b へのマッピングを定義する変換であることを $r_{a\to b} : mm_a \to mm_b$ と表記する．また，モデル $m_a : mm_a$ を変換 $r_{a\to b} : mm_a \to mm_b$ で変換した結果を，$r_{a\to b}(m_a)$ と表記する．このとき，$r_{a\to b}(m_a) : m_b$ である．

例 2 mm_C を C 言語メタモデル，mm_{abs} を抽象プログラムメタモデルであるとする．C 言語メタモデルから抽象プログラムメタモデルへのマッピング $r_{C\to abs}$ は，$r_{C\to abs} : mm_C \to mm_{abs}$ である．また，C 言語モデル $m_c : mm_C$ を $r_{C\to abs}$ で変換した結果 m は，$m = r_{C\to abs}(m_c)$ であり，$m : mm_{abs}$，すなわち m は抽象プログラムモデルとなる．

以上に定義したメタモデル，モデル，モデル変換を用いて POM コンセプトフレームワークを定義する．なお，ソースコードパーサーとそれによるソースコードから実装モデルへの変換はソースコードのプログラム言語に．ターゲットコード生成器およびテンプレートとそれらによるターゲットモデルからターゲットコードへの変換はターゲットコードのモデル言語に，それぞれ依存して決定する．そのため，ソースコードと実装モデル，ターゲットモデルとターゲットコードは内容的には同一視できる．以下では，POM コンセプトフレームワークを特徴づけるモデル抽出を，実装モデルからターゲットモデルへの変換で定義することとする．

定義 3 (POM 変換) 言語 L_i に対するメタモデル mm_i，抽象プログラムメタモデル mm_a，言語 L_t に対するメタモデル mm_t，変換 $r_i : mm_i \to mm_a$, 変換 $r_a : mm_a \to mm_a$, 変換 $r_t : mm_a \to mm_t$ が与えられたとき，6 項組 $Pom = \langle mm_i, mm_a, mm_t, r_i, r_a, r_t \rangle$ は POM フレームワークである．また，Pom はモデル $m_i : mm_i$ から $r_t(r_a(r_i(m_i))) : mm_t$ への変換であり，この変換結果を $Pom_{L_i\to L_t}(r_a, m_i)$ と表記する．

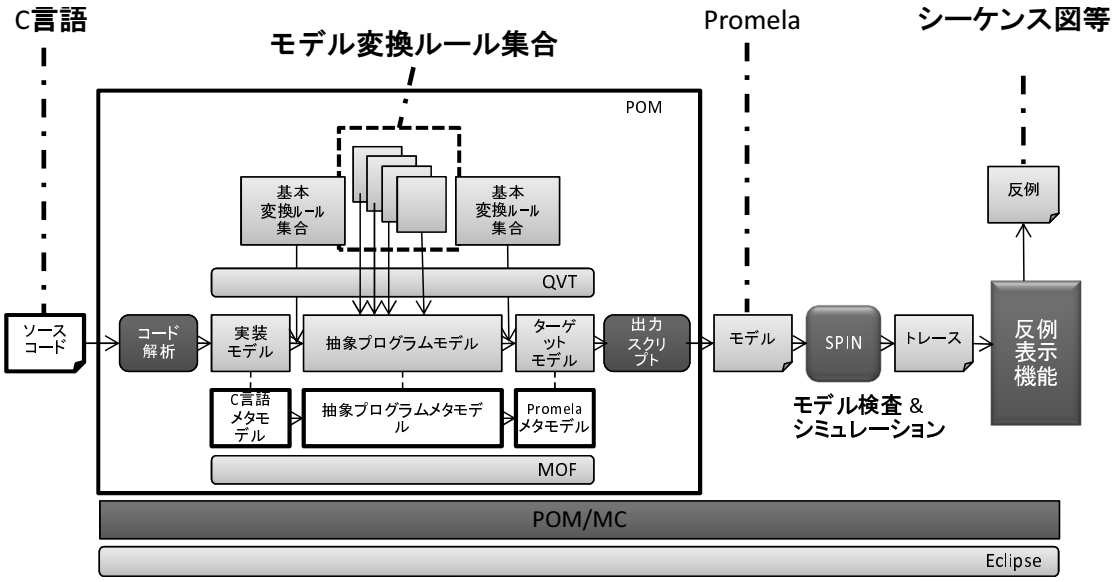

図 4.2: POM/MC の概略構成

例 3 *C* 言語に対する実装メタモデルを mm_C，抽象プログラムメタモデル mm_a，*Promela* 言語に対するターゲットメタモデル $mm_{Promela}$ とする．また，実装モデルから抽象プログラムへの変換 $r_C : mm_C \rightarrow mm_a$，変換 $r_a : mm_a \rightarrow mm_a$，変換 $r_{Promela} : mm_a \rightarrow mm_{Promela}$ が与えられたとする．このとき，$Pom = \langle mm_C, mm_a, mm_{Promela}, r_C, r_a, r_{Promela} \rangle$ は，*C* 言語プログラムから *Promela* モデルへの変換を行う．

また，$m_t = Pom_{C \rightarrow Promela}(r_a, m_s)$ は，抽出される *Promela* モデルを意味し，$m_t : mm_{Promela}$ を満たす．つまり，m_t は変換 r_a によって抽象化された *Promela* モデルである．

4.2 POM/MC: POM に基づくモデル検査ツール

POM/MC(POM for Model Checking) は，POM コンセプトフレームワークを実現し，ソースコードからのモデル抽出と，抽出したモデルに対するモデル検査を実現するツールである．本研究では，入力言語は C 言語，抽出する振る舞いモデルは Promela，モデル検査器は SPIN を用いる POM/MC を開発した．

図 4.2 に POM/MC の構成の概略を示す．まず，POM の実現として，MOF に基づく 3 つのメタモデルを定義した．実装メタモデルとしての C 言語メタモデル，抽象プログラムメタモデル，ターゲットメタモデルとしての Promela メタモデルである．

次に，QVTo を用いた変換ルール集合を策定した．変換ルール集合には，C 言語メタモデルと抽象プログラムメタモデルとの間，抽象プログラムメタモデル間，抽象プログラムモデルと Promela モデル間の 3 種類が存在する．POM/MC では，変換ルール集合の中から，適用する変換ルール集合をユーザが選択できる．

モデル検査には SPIN を用いる．POM/MC は，SPIN の実行と，その結果を処理する機能を有する．具体的には，モデルコードを入力として SPIN を実行し，反例トレースを得る．反例トレースに用いて SPIN によりガイデッドシミュレーションを行い，反例の実

行ログを得る．この実行ログを Eclipse 上にシーケンス図として表示する．

モデル検査に SPIN，モデルの表現に Promela を採用したのは，本研究で扱うフォールトアナリシス，特に並行性を持つプログラムの故障再現において，並行プログラムの振舞いを表現するモデルを得るためである．したがって，POM/MC の目的は任意の C 言語プログラムを等価の意味を持つ Promela コードに変換することではなく，対象となる C 言語プログラムについて，並行性を持つプログラムの故障再現という観点において，必要となる情報を持つモデルを抽出することである点に注意されたい．

4.2.1 POM/MC の形式化

POM コンセプトフレームワークについで，POM/MC を定義する．POM/MC は，C 言語ソースコードを入力とし，Promela モデルを抽出した上で，Promela モデルと別途与えられた性質について SPIN を用いて検査する．これによりモデル検査の結果と反例が得られる．

定義 4 *C*言語ソースコード sc に対して，sc の実装モデル $m(sc) : mm_C$，性質 φ と，$Pom = \langle mm_C, mm_a, mm_{Promela}, r_C, r_a, r_{Promela} \rangle$ が与えられたとき，*POM/MC*は，$m_t = Pom_{C \to Promela}(r_a, m_{sc})$ を生成し，$m_t \models \varphi$ でなければ，反例 $\overline{m_t, \varphi}$ を出力する．$m_t \models \varphi$ であれば反例 $\overline{m_t, \varphi}$ は空である．また，$pommc(sc, r_a, \varphi)$ は，反例 $\overline{Pom_{C \to Promela}(r_a, m_{sc}), \varphi}$ を示す．

例 4 $pommc(sc, r, \varphi)$ とは，ソースコード sc に対して変換 r を用いて抽象化した検証モデルについて，性質 φ に対するモデル検査を行い，得られる反例を意味する．

4.3 POM/MC のメタモデル

POM/MC を構成するメタモデルとして，C 言語メタモデル，抽象プログラムメタモデル，Promela メタモデルを作成した．各メタモデルの記述には，2.2.3 節に示した MOF を用いた．図 4.3 に，各メタモデルを示す．ただし，説明上，実際のメタモデルに対して要素ならびに要素名を簡略化している点に注意されたい．各メタモデルは，抽象意味グラフ (Abstract Semantic Graph; ASG)[10] を表現している．ASG とは，抽象構文木 (AST;Abstract Syntax Tree) に参照情報を付加したものである．

4.3.1 C 言語メタモデル

定義した C 言語メタモデルを概説する．C 言語メタモデルは，C 言語の ASG を表現している．まず，抽象構文木に関係する定義を示す．宣言を意味する `Decl` (Declaration) クラスが存在する．`Decl` クラスのサブクラスとして，変数宣言を意味する `VariableDecl` (Variable Declaration) クラスと，関数宣言を意味する `FunctionDecl` (Function Declaration) がある．関数引数を意味する `ParameterDecl` (Parameter Declaration) クラスは，`VariableDecl` クラスのサブクラスである．また，文を意味する `Stmt` (Statement) クラスが存在し，複合文を意味する `CompoundStmt` (Compound Statement) クラス，式宣言を意味する `ExpStmt` (Expression Statement)，For 文を意味する `ForStmt` (For

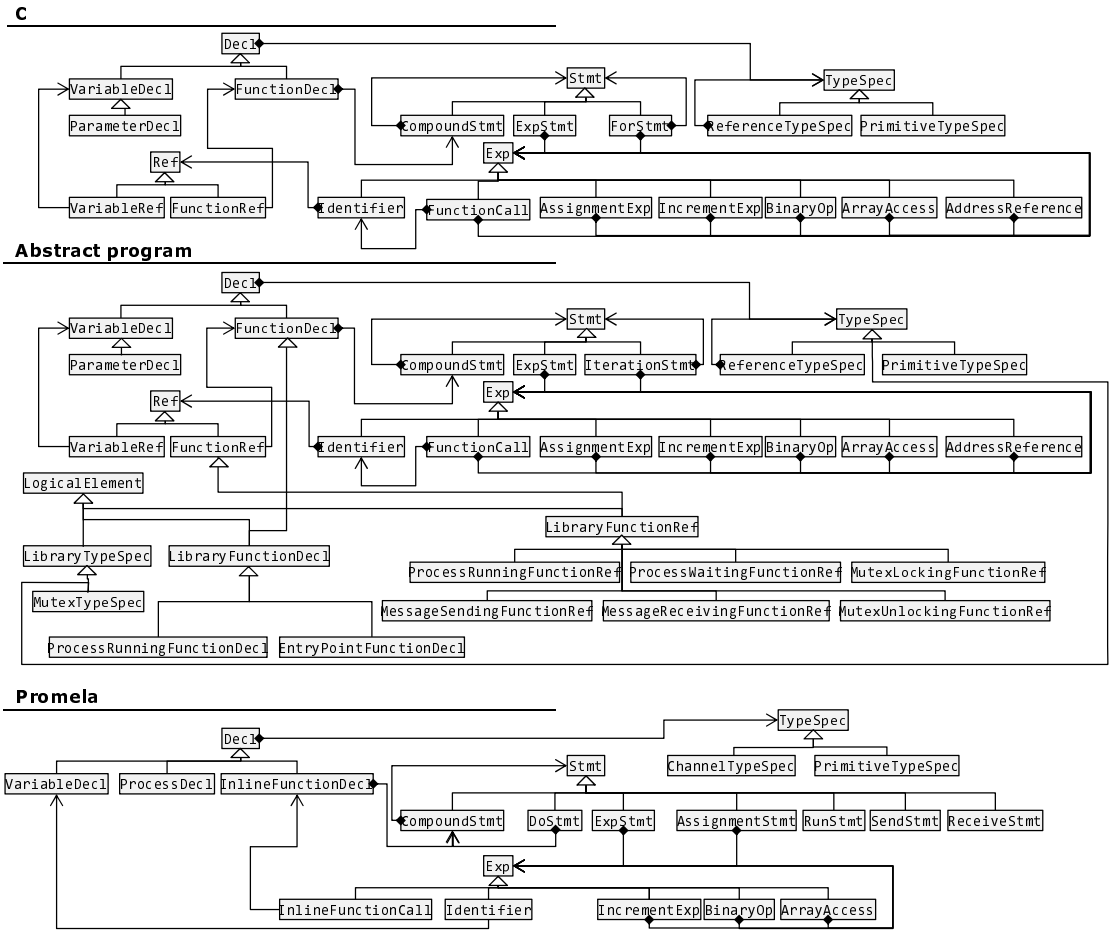

図 4.3: POM/MC のメタモデル (上:C 言語; 中央:抽象プログラム; 下: Promela; それぞれ一部)

Statement) クラスは，Stmt のサブクラスである．複合文 FunctionDecl は，関数宣言 FunctionDecl を構成する．複合文 FunctionDecl は，文 Stmt からなる．For 文 ForStmt は，文 Stmt と，式 Exp からなる．式宣言 ExpStmt は，式 Exp からなる．Exp クラスのサブクラスとして，識別子 Identifier，関数コール FunctionCall (Function Call)，代入式 AssignmentExp (Assignment Expression)，インクリメント演算子 IncrementExp (increment Expression)，二項演算子 BinaryOp (Binary Operation)，配列アクセス ArrayAccess (Array Access)，アドレス参照 AddressReference (Address Reference) がある．これらは，Exp を構成する．また，型指定子 TypeSpec (Type Specifier) は，宣言 Decl を構成する．TypeSpec のサブクラスとして，参照型指定子 ReferceTypeSepc (Reference Type Specifier) と，プリミティブ型指定子 PrimitivyTypeSpec (Primitive Type Specifier) があり，参照型指定子は別の型指定子から構成される複合型である．このようなメタモデルにより C 言語の構文木をモデルとして表現することができる．

次に参照情報について説明する．メタモデル上では，参照情報は Ref (reference) クラスによって表現している．変数参照 VariableRef (Variable Reference) と，関数参照 FunctionRef (Function reference) が Ref のサブクラスとして存在する．Ref が Identifier と関連を持ち，VariableRef が VariableDecl と，FunctionRef が FunctionDecl とそれぞれ関連を持つ．これにより，識別子 Identifier(のサブクラス) から宣言 Decl（のサブクラス）への参照が表現できる．

4.3.2 抽象プログラムメタモデル

抽象プログラムメタモデルは，C言語のメタモデルをベースとして，一般的な手続き型言語を表現できるように設計した．AGS の表現については，ほぼ C 言語メタモデルと同じであるが，一部相違がある．C 言語メタモデルでは ForStmt クラスが存在するが，抽象プログラムメタモデルではそれに代わるものとして IterationStmt (Iteration Statement) が存在する．これは，for 文や while 文など繰り返しを意味する構文を統一的に扱うことを意図している．

また，抽象プログラムメタモデルの特徴の一つとして，“論理要素”がある．論理要素は，抽象プログラムメタモデルでは，LogicalElement (Logical Element) で表現している．論理要素とは，ソースコードレベルの言語要素よりも抽象度の高い役割を示すものである．図 4.3 ではライブラリ，特にプロセスに関する論理要素を定義している．ライブラリで提供される型や関数も，ソースコード上では他の型や関数と同じであるが，特定の意味を持つライブラリを用いるという意味を与えることで，Promela によるモデル化において，特別に扱うことができる．LogicalElement のサブクラスとして，ライブラリの提供する型を意味する LibraryTypeSpec (Library Type Specifier)，ライブラリの提供する型の宣言を意味する LibraryFunctionDecl (Library Function Declaration)，ライブラリの指定する関数の参照を意味する LibraryFunctionRef (Library Function Reference) を定義している．LibraryTypeSpec のサブクラスとして Mutex 型を意味する MutexTypeSpec (Mutex Type Specifier) を定義し，Mutex 型について他の型とは異なる特別な意味を与えている．LibraryFuctionDecl と LibraryFunctionRef についても，それらのサブクラスとしてライブラリの提供する関数について，特別な意味を持たせている．

4.3.3 Promela メタモデル

Promela メタモデルでは，Promela の抽象構文木を表現している．Promela の構文は C 言語と共通する要素もあるため，C 言語メタモデルと Promela メタモデルは類似しているが，異なる部分もある．例えば，C 言語では for 文を意味する `ForStm` が，Promela では do 文 `DoStmt`(Do Statement) となっている．また，C 言語では代入式 `AssignmentExp` は文 `Exp` を構成要素として持つことができるため代入式の中に代入式を入れることができる $(a = (b = c))$ が，類似する Promela の代入文 `AssignmentStmt` は (Assignment Statement) は文を要素に持つことができないため，代入の代入はできない．また，Promela の持つプロセス機構である `RunStmt` (Run Statemenet)，メッセージ送受信を行う `SendStmt` (Send Statement) と `RecieveStmt` (Receive Statement) をメタモデルに含む．

4.4 POM/MC の変換ルール集合

4.4.1 モデル抽出の考え方

C 言語コードから Promela コードへの変換は，ユーザが与える変換ルール集合によって行われる．従って，C 言語コードを同じ意味を持つ Promela コードへ自動変換するものではない．変換ルール集合によっては，意図を持って C 言語コードとは意味の異なる Promela コードに変換する場合がある．たとえば，ポインタによる参照を，参照先の実体へのアクセスに置き換えるような変換ルールがある．これは，参照先が静的に解決する場合には Promela 言語によりポインタ参照と同様に振舞うが，そうでない場合は Promela コードは不適切な振舞いを示す．

図 4.4 に抽象化を伴うモデル抽出の考え方を示す．POM/MC ではモデルは XML で表現されるが，ここでは理解をしやすくするため，疑似コードで示す．モデル変換は，基本的には語彙の変換である．例えば，C 言語の for ブロックを Promela の do ブロックに変換する (図 4.4(a))．あるいは，`pthread_create` 関数の呼び出しは，Promela の `run` ステートメントに変換する．この変換は，言語に汎用的ではなく，ドメインに依存するものであり，ユーザによって定義される．また，変換は多段階に行われる．例えば，`pthread_create` から `run` への変換は次の 2 ステップで行う (図 4.4(b))．まず，Pthreads ライブラリ用のモデル変換ルールが，`pthread_create` 関数の呼び出しを “process creation statement” として認識する，認識された関数呼び出しは，抽象プログラムモデルにおける “process creation” というロールを持つ関数コールを表現する要素に変換される．ついで，`pthread_create` 関数の引数として指定されている関数の定義を，“process” を意味する要素に変換する．最後に，“process creation” を意味する抽象要素を，ターゲットメタモデルの `run` ステートメントに変換する．

4.4.2 モデル変換ルール集合の構成

C 言語から Promela モデルへの変換は，内部モデルに対して，ユーザが構成したモデル変換ルール集合を適用することで実現する．この変換プロセスは次の 3 ステップで行われる．

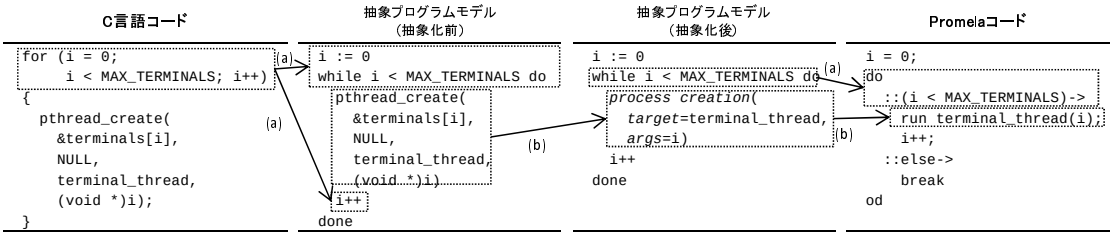

図 4.4: モデル抽象化の概念

C → 抽象プログラムモデル

まず，C 言語メタモデルにより定義される実装プログラムモデルから，抽象プログラムメタモデルにより定義される抽象プログラムへと変換する．この変換は，C 言語要素から抽象プログラム要素への一対一対応が主である．

付録 B.1 に，実装モデルから抽象プログラムモデルへの変換ルールの例として，C 言語の for 文を抽象プログラムモデルのイテレーションに変換するルールの記述を示した．

抽象化 (抽象プログラムモデル → 抽象プログラムモデル)

次に，抽象プログラムモデル上での抽象化に伴う変換を行う．この変換は，次のようなフェーズを含む．

1. 文法の平準化: 他の抽象化に先駆けて，文法の違いを平準化する．例えば，C 言語では代入のネストが可能だが，他の言語では可能ではない場合がある．そこで，代入のネストを複数の代入に分解するような変換を行う．例えば，`a=(b=c)` を `b=c;a=b` へと変換する．

2. 環境マッピング: ソフトウェアの外部環境，例えばハードウェア，とのインタフェースとなるライブラリ関数や変数，あるいは特定の機能を持つユーティリティを，スタブ関数や論理要素にマッピングする．

3. 縮退: 不要な処理を削除したり，データ構造を抽象化することで，フェイラーに関係すると思われる範囲にモデルを絞る．

4. 意味論の適合: Promela メタモデルに対応する要素がない抽象プログラムメタモデル上の要素を，Promela メタモデル上の要素で表現する．例えば，C 言語における多次元配列は Promela ではサポートされないため，一次元配列と構造体を用いて表現する．

付録 B.3 に，抽象プログラムモデル上での抽象化に関わる変換ルールの例として，Pthreads ライブラリの利用を，抽象プログラムモデルの論理要素に変換するルールの記述を示した．

抽象プログラムモデル → Promela

抽象プログラムメタモデルからPromelaメタモデルへの変換を行う．付録B.2に，文に関する変換の一部を示した．特に，抽象プログラムモデル上でのイテレーションを，Promelaモデルの `do` に変換するルールの記述を抜粋した．

4.4.3 モデル変換ルール集合の作成

モデル変換ルールの記述には，2.2.3に示したQVToを用いた．QVToは命令型言語の一種であり，メタモデル間の変換を手続き的に記述することができる．

図4.5に，図4.1における“Pthreads”ルールの記述の一部を例として示す．モデル変換ルール `convertToPthreadFunctionReference` は，Pthreadsライブラリに関係する関数呼び出し識別し，抽象プログラムメタモデルで定義された関連する要素に変換し，関数宣言を対応する要素に変換するものである．本モデル変換ルールの仮引数は，関数呼び出し `FunctionCall` クラスの `function` オブジェクトである．`FunctionCall` は，抽象プログラムメタモデルで定義された要素の一つであり，関数呼び出しを意味する要素であった．QVToでは `switch` 文を用いることができ，`function.getName()` で得られる文字列，すなわち，呼び出される関数の名前による分岐を定義している．例えば，関数名が `pthread_create` である場合には，この関数呼び出しを，プロセスの実行を意味する抽象プログラムメタモデルの論理要素の一つである `ProcessRunnningFunctionReference` オブジェクトに変換する．ここでは記載していないが，`pthread_create` 関数呼び出しの引数については，別途 `ProcessRunnningFunctionReference` オブジェクトの引数への変換を行っている．

QVToでは手続き的な記述が可能であるため，上記に示した例よりも，複雑な変換アルゴリズムを定義することもできる．例えば，表4.1に示した“型宣言の最適化”ルールは，抽象プログラムモデルをたどり定数を判別した上で，値の代入関係を解析し変数の取りうる値の最大値および最小値を見積もることで，変数の値域に関して探索範囲を縮退するような変換を行う．

```
mapping Reference::convertToPthreadFunctionReference(function: FunctionCall): Reference {
  init {
    switch {
      case (function.getName() = 'pthread_mutex_lock') {
         ...
      }
      ...
      case (function.getName() = 'pthread_create') {
        result := object ProcessRunningFunctionReference {
                          declaration := self.oclAsType(FunctionReference).declaration;
                        }
      }
      case (function.getName() = 'pthread_join') {
        ...
      }
       ...
      else {
        result := self;
      }
    }
  }
}
```

図 4.5: QVTo によるモデル変換ルール記述の例

表 4.1 に作成した抽象化に関わるモデル変換ルール集合の例を示す．ルールの種類の列には，図 3.4 で示した，基本変換，ドメイン特化変換，タスク特化変換との対応を示した．抽象化の種類の列には，ルールを意味論の変換と抽象化に分類した．名称の列には，ルールによる変換内容を示す名称を説明用に与えた．概説列に，各ルールの変換内容を概説した．

表 4.2 に，本研究で作成した変換ルール集合により，C 言語のコードがどのように Promela コードに変換されるかについて代表的なものを示す．ただし，変換ルール集合により変換が定義されるものである．POM/MC の変換能力がこれに限定されるものではなく，また，異なる変換方針に基づく変換ルール集合を定義することもできる．POM/MC の構文解析は C 言語コンパイラフレームワークである LLVM[34] の構文解析機能を利用しており，任意の C 言語プログラムを解析して実装モデルに ASG を格納できる．実装モデルを抽象プログラムモデルに変換する際に，変換ルール集合に定義された情報のみがプログラムモデルに格納され，さらに抽象プログラムの情報が抽象化される．その結果が表 4.2 であり，Promela に変換可能な C 言語プログラムの要素と変換方針は変換ルール集合により変更可能である．

表 4.1: モデル変換ルール集合の例

ルールの種類	抽象化の種類	名称	概説
基本	意味論の変換	式の展開	代入や関数呼び出しなど C 言語で入れ子になった式を展開する．(例) $a = (b = c) \rightarrow b = c; a = b;$
		多次元配列の分解	C 言語における多次元配列を一次元の配列をフィールドに持つ構造体に分解する．
		ポインタのエイリアス解析	ポインタ参照の変化を解析し，ポインタによる参照先データ参照を実体の参照に置き換える．(例) $int\ i, *j; j = \&i; *j = 5; \rightarrow int\ i; i = 5;$
ドメイン特化		Pthreads	POSIX スレッドライブラリ (Pthreads) の提供するスレッド生成や排他制御の機構を，Promela のプロセスやガード機構に置き換える．
		状態機械ドライバ	このルールの引数で指定された関数を周期的に呼ばれる構造に置き換え，状態遷移設計を再現する．
タスク特化		エントリポイント	このルールの引数で指定された関数を Promela の初期プロセス (init) に指定する
		スタブ	指定された関数をスタブで置き換える．以下の 4 種類のスタブをサポートしている．(1) 機能を持たないスタブ (2) 定数を返すスタブ (3) 指定された値域内のランダムな値を返すスタブ (4) 別途指定されたモデルコードを呼ぶスタブ
		表明	指定されたソースコード中のラベルへのジャンプを表明に置き換える
	抽象化	型宣言の最適化	整数型で宣言された変数と関数について，代入関係から最大値と最小値を決定し，その範囲を表現するのにより小さい型で十分な場合には，より小さい型に変更する．
		未使用宣言の削除	ドライバから利用されている宣言を再帰的に求め，使われていない宣言を削除する．

表 4.2: POM/MC における代表的な C 言語から Promela への変換

C 言語要素	Promela への変換方針
関数	inline 関数
関数引数	inline 関数引数
関数返値	inline 関数引数
main 関数	init プロセス (main 以外のスタートポイントも指定可能)
変数宣言	変数宣言 (Promela がサポートする型に限る)
構造体	構造データ型
配列	配列
多次元配列	構造データ型の配列
if	if
for	do
while	if
ポインタ参照	参照先の実体の参照 (静的に参照解決できる場合に限る)
代入	代入
入れ子の代入	複数文での代入

```
[template expressionToString(exp: Expression)]
[if (exp.oclIsUndefined()) ] >>>Undefined Expression<<<
[elseif (exp = null) ] >>>Null Expression<<<
[elseif (exp.oclIsInvalid()) ] >>>Invalid Expression<<<
[elseif (exp.oclIsKindOf(RunExpression))]run [exp.oclAsType(RunExpression).process.name /]
([expressionsToString(exp.oclAsType(RunExpression).arguments) /])
[elseif (exp.oclIsKindOf(Identifier))]
[if (exp.oclAsType(Identifier).declaration<>null and not exp.oclAsType(Identifier).
declaration.oclIsInvalid() )][exp.oclAsType(Identifier).declaration.name /]
[else]>>>Missing declaration(Identifier)<<<[/if]
[elseif (exp.oclIsKindOf(FieldAccess))] …(省略)…
[elseif (exp.oclIsKindOf(Constant))] …(省略)…
[elseif (exp.oclIsKindOf(ReservedVariable))] …(省略)…
[elseif (exp.oclIsKindOf(ArrayAccess))] …(省略)…
 …(省略)…
[/if]
[/template]
```

図 4.6: M2T テンプレートの例

4.4.4 テンプレートの作成

ターゲットコードは，モデルからコードを生成する M2T[24] テンプレートの実装の一つである Acceleo[1] を用いて生成される．図 4.6 に作成したテンプレートの一部を示し，ターゲットモデルからターゲットコードの生成について説明する．図に示したのは，Promela メタモデル上の `Expression`(文) 要素に対する，Promela コードを生成するためのテンプレートである．[と] で囲まれた部分が M2T の構造を示し，それ以外の部分が生成するコードとなっている．テンプレート名が `expressionToString` であり，仮引数として `Expression` クラスの `ext` をとる．M2T では，`if`，`elseif`，`else` といった制御構造が利用できる．2 行目から 4 行目までは，モデルが不正であった場合のエラー処理である．5 行目に，プロセス生成に関するコードの生成を定義している．すなわち，`exp` が `RunExpression` クラスのオブジェクトであるならば，文字列 “run” を出力したのち，`exp` のプロセス名を示す文字列を出力し，“(” に引き続き `exp` の引数を出力したのち，“)” を出力するという定義がなされている．以上により，Promela における `run` 文によるプロセス生成のためのコードが生成される．他の文に関しても同様である．このように，Promela メタモデルの要素に対して，対応して生成するコードを定義する M2T テンプレートを作成した．

4.5 POM/MC ツールの実装

POM/MC ツールでは，C 言語ソースコードの解析器として，LLVM/Clang[2] を用いた．LLVM/Clang の持つ内部データを取り出し，実装モデルの ASG を構成している．モデル変換プラットフォームとして，Eclipse Modeling Tools(EMT)[20] を用いた．EMT は，Eclipse Modeling Framework (EMF)[44] に基づいたモデリング環境を構築するための Eclipse 拡張の 1 つである．各メタモデルは，Ecore モデルにより定義され，メタモデルを

[1] `http://www.acceleo.org`, 2012-06-01
[2] `http://llvm.org`, 2012-06-01

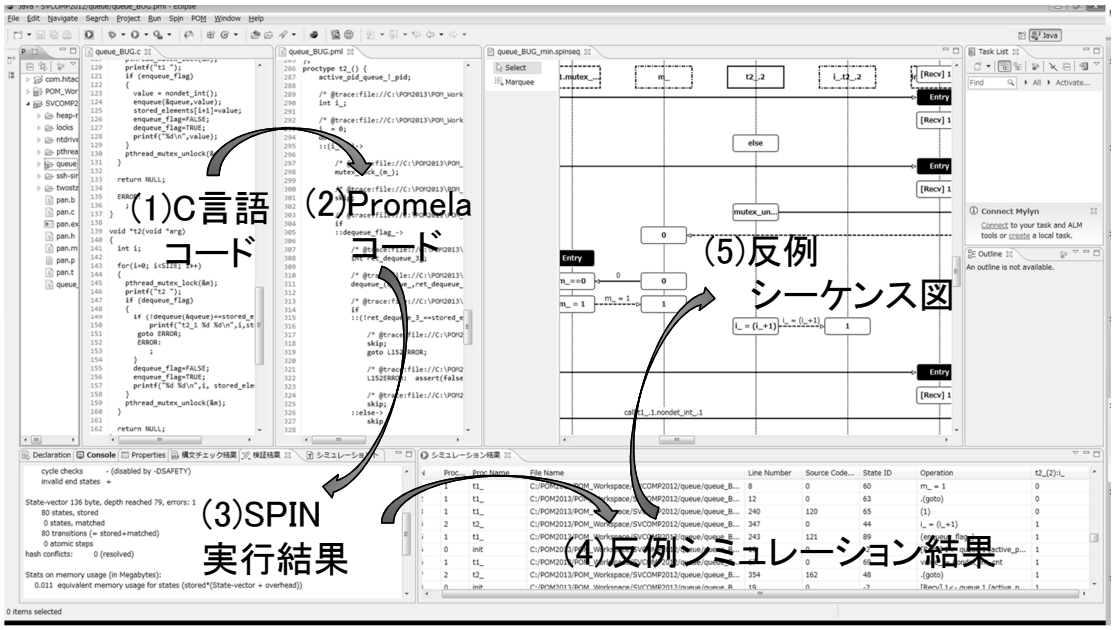

図 4.7: POM/MC ツールの画面例

用いたモデル変換ルールは，Operational QVT (QVTo) プラグイン [3] を用いて記述される．QVTo は，MOF QVT/OperationalMapping の実装の一つである．

図 4.7 に，実装した POM/MC ツールの画面例を示す．(1) に対象となる C 言語ソースコードを表示している．(1) からモデル抽出によって得られた Promela コードを (2) に表示している．(2) に対して SPIN を用いてモデル検査を実施した結果を (3) に表示している．(3) によって得られた反例トレースをシミュレーションした結果を (4) に表示している．(4) では，変数などによって表示をフィルタリングすることができる．(4) を元に作成したシーケンス図を (5) に表示している．(5) では，特定のアクターの表示を省略するなど，反例理解を容易にする機能を有する．POM/MC ツールは，これらの一連の実行をサポートし，モデル検査による故障再現を実現する．

4.6 POM/MC の評価実験

POM/MC を用いて C 言語ソースコードから Promela モデルを抽出しモデル検査を実施できることを確認するため，いくつかのサンプルプログラムについて評価実験を行った．表 4.3 に，各サンプルプログラムに対して，モデル抽出とモデル検査を実施した結果を示す．分類に SVCOMP とあるプログラムは，キュープログラムも含め，SVCOMP のベンチマークとして公開されているソースコードである．分類にオリジナルとあるプログラムは，筆者らが評価用に作成したプログラムである．いずれも，Pthreads ライブラリを用いて，複数のスレッドが並行して動作するプログラムである．以下では，本論文の例題とし

[3] `http://www.eclipse.org/m2m/`, 2012-06-01

表 4.3: ケーススタディのまとめ

分類	プログラム名	C (LOC)	Promela (LOC)	シンボル数
SVCOMP	test_lock_14.BUG.c	163	353	32
	queue_BUG.c	157	215	40
	twostage_3_BUG.c	111	190	30
	fib_bench_BUG.c	31	97	18
オリジナル	哲学者問題	104	203	175
	座席予約システム	495	599	

て用いたキュープログラムと，より実システムに類似した構造を持つ座席予約システムを用いた評価実験の結果について述べる．

4.6.1 キュープログラム

付録 A.1 にキュープログラムの C 言語ソースコードを，付録 A.2 に POM/MC によって抽出した Promela コードを掲載した．また，付録 B.1 には，C 言語モデルから抽象プログラムモデルへの変換を定義する変換ルールの例を，付録 B.3 には，抽象プログラムモデル上での変換を定義する変換ルールの例を，付録 B.2 には，抽象プログラムモデルから Promela モデルへの変換の例を掲載した．このような変換ルールを作成することで，C 言語から Promela への変換と，モデル検査が可能であることを確認した．

4.6.2 座席予約システム

キュープログラムより現実のシステムに近い例として，簡単な座席予約システムを模擬する C 言語プログラムを作成した．これに対して，POM/MC を用いてモデル抽出とモデル検査を行った．

座席予約システムの概説

このプログラムは，座席の予約を管理する一つのサーバーと，予約を行う複数のクライアントからなる．クライアントからサーバーへは，指定した席の空き状況を問い合わせる “checking availability”，指定した席の予約をリクエストする “reservation”，指定した席の予約確認を行う “confirmation” メッセージを送る．サーバーは，各メッセージに対して，空き状況の有無，予約可否，予約状況を返す．このプログラムには，ダブルブッキングを起こす問題が含まれている．

複数のクライアントから同時に “checking availability” メッセージを受けた場合，すべてのクライアントに予約可能と返してしまうためである．

このプログラムでは，クライアント-サーバーシステムの並行動作を実現するため，Pthreads ライブラリを用い，サーバーとクライアントは各々がスレッドとして動作する．

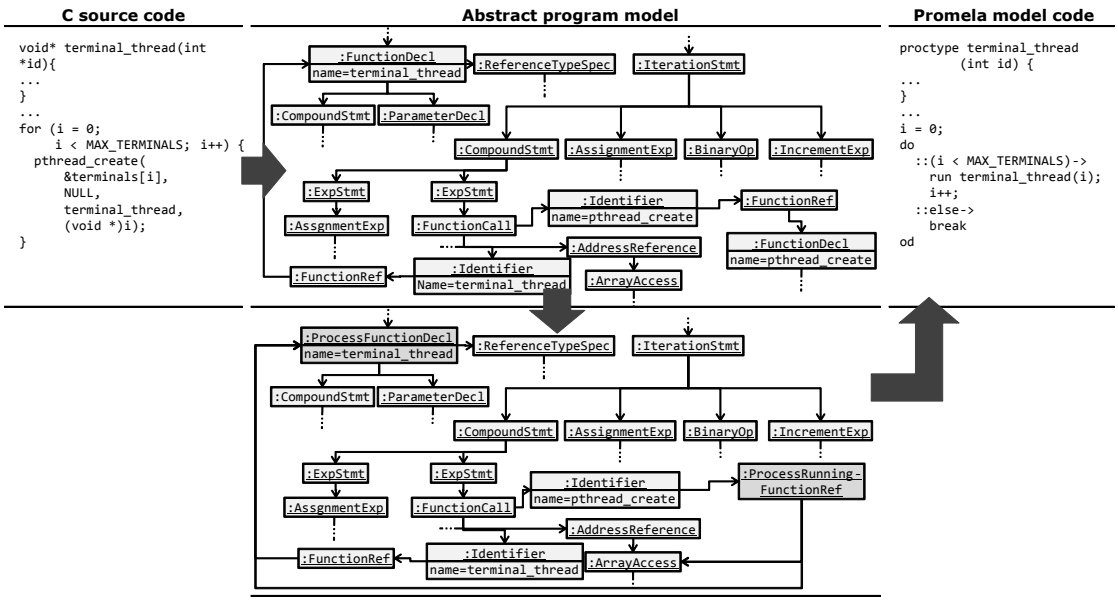

図 4.8: Pthreads ライブラリ利用の抽象化

モデル変換ルールによる変換の例

本節では，座席予約システムのケースで用いたモデル変換の例として，Pthreads ライブラリが提供する関数と型の利用を，Promela 言語の並行プロセスで実現するため変換について解説する．また，スレッド間の通信の変換についても概説する．

図 4.8 に，モデル変換の手順を示す．元となる C 言語ソースコードと対応する Promela コード，それらを橋渡しする抽象プログラムモデルを示している．C 言語モデルと，Promela モデルは，各コードと対応しているため，ここでは省略する．

4.4.1 節で述べたように，C 言語の `pthread_create` 関数は，Promela の `run` 文に対応する．また，C 言語の関数宣言は，Promela の proctype 宣言に対応するものである．

`pthread_creat` 関数に対応する `FunctionCall` オブジェクトを参照している `FunctionRef` オブジェクトを，`ProcessRunnigFunctionRef` オブジェクトに置き換えている．抽象プログラムモデルから Promela モデルへの変換により，`ProcessRunnigFunctionRef` オブジェクトは Promela の run 文の要素に対応づく．これにより，`pthread_creat` 関数は，run 文に変換されることになる．

また，`ProcessRunnigFunctionRef` オブジェクトが参照している関数は，`ProcessFunctioDecl` オブジェクトに変換される．`ProcessFunctioDecl` オブジェクトは，Promela の proctype 宣言に対応づく．これにより，`pthread_creat` 関数の引数であるスレッド関数の宣言は，Promela の `Proctype` 宣言に変換されることになる．このようにして，`pthread_creat` 関数によるスレッド生成が，run 文によるプロセス生成に変換できる．

次に，通信の変換について説明する．座席予約システムの実装では，クライアントとサーバー間の通信を行うメッセージ関数として，送信を行う `chan_send` 関数，受信を行う `chan_recieve` 関数，初期化を行う `chan_init` 関数を作成した．また，メッセージキュー

を実現する構造体 struct 型である chan_s を定義した．

例えば，chan_send 関数の呼び出しを示す FunctionCall オブジェクトを参照する FunctionRef オブジェクトは，論理要素である MessageSenfingFunctionRef オブジェクトに対応づける．抽象プログラムモデル上の論理要素と Promela の言語要素の対応を定義することで，MessageSenfingFunctionRef オブジェクトを “!” 演算子による送信文に対応づける．これにより，chan_send 関数の呼び出しは，“!” 演算子による送信文に変換できる．受信を行う chan_recieve 関数についても，他のメッセージ関数も同様である．

これらの言語間の意味論の変換を行うモデル変換ルール集合のほか，状態爆発を防ぐことを目的に以下のルールを定義した．

- 座席数を示す定数の値を変更する
- 一括して同じ処理を繰り返すループ（例えば，座席予約情報の初期化）をループのない処理に変更する

結果

準備したモデル変換ルールを用いることで，Promela モデルを抽出することができた．抽出された Promela モデルは約 700 行であった．また，モデル検査によって，ダブルブッキング問題を再現できた．

第5章　実験的フォールトアナリシスプロセス

本章では，フォールトアナリシスのプロセスを提案する．4 章で故障再現のためのモデル抽出とモデル検査を実現するツールである POM/MC を示した．これにより，フォールトアナリシスの一部である故障再現を支援できるようになったが，ソフトウェア開発においてフォールトアナリシスを実践するためには，フォールトアナリシス全体の手順を明らかにする必要がある．そのため，本章ではフォールトアナリシスモデルを形式化する．また，フォールトアナリシスプロセスに POM/MC を組み込んだプロセスも提案する．本提案プロセスは，仮説演繹法に基づいた形式化が特徴であり，故障再現を実験と捉え，仮説の下での実験を繰り返すことでフォールトを絞り込んでいく．そこで，本論文では提案プロセスを実験的フォールトアナリシスプロセスと呼ぶ．

5.1　フォールトアナリシスの形式化

本節では，フォールトアナリシスプロセスの検討に先立ち，フォールトアナリシスとは何であるかについて検討し形式的に定義する．

5.1.1　故障再現に関する考察

2.1.1 節で述べたように，フォールトアナリシスとは既に観測された故障について，その原因となるフォールトを特定する行為をさす．故障とは，ソフトウェアがユーザの期待を逸脱した状態であり，フォールトとは故障を発生させる原因となる状態や条件である．

ここで注意すべきは，故障再現とは，故障発生時と全く同一の振る舞いを再現するものではないことである．図 5.1 に示したように，ソフトウェアは外部から与えられる様々な条件により分岐の選択を繰り返し，その結果到達する状態の一つが故障として観測される．最終的に故障が観測されれば，どのような過程を経たとしても，故障は再現されたと考えるのが妥当である．むしろ，当初の故障発生時よりも，フォールトの解析が容易なより単純な実行過程を経るのは，フォールトアナリシスの文脈では望ましいことである．また，故障そのものについても，全く同一の状態を再現する必要はない．例えば，ある特定の変数 v_1 の値が期待と異なる故障があった場合，再現する故障においては変数 v_1 の値のみを再現すればよく，再現時の他の変数 v_2 の値が当初の故障の値と異なっていても，故障は再現されたと考えられる．さらに，状況によっては「変数 v_1 の値が期待と異なる」という状況を再現できればよく，再現時の変数 v_1 の値ですら当初の故障発生時と異なってよい場合もある．そして，何が当初の故障状態と同一であれば故障を再現したと考えてよいのかは，故障の内容や状況によって異なる．

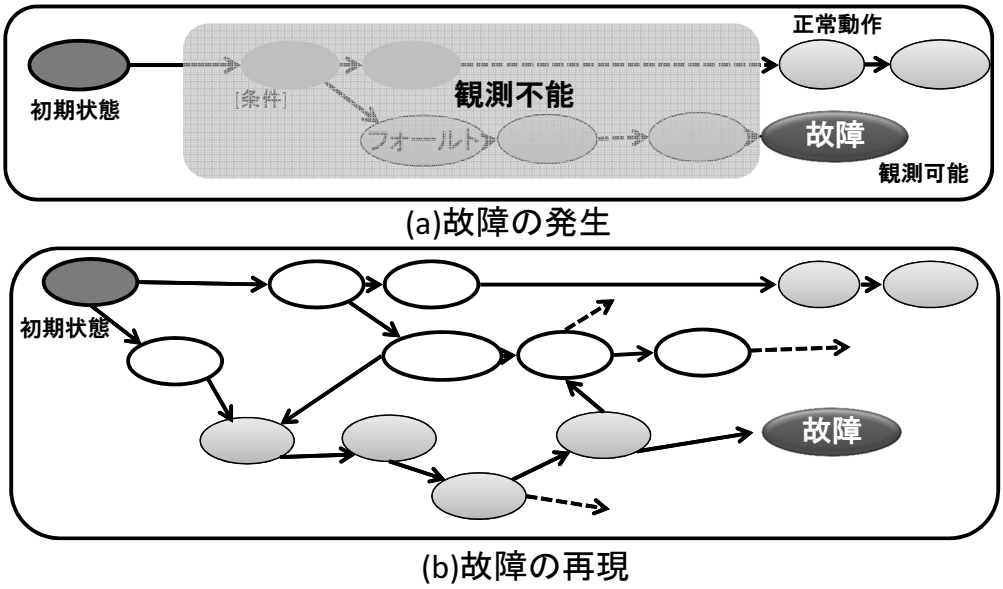

図 5.1: 故障の観測と再現 (再掲)

また，対象となるソフトウェア自体も故障発生時と全く同一である必要はない．再現困難な故障の再現が本論文で扱う問題であるが，従来のソフトウェア実行による故障再現においても，無作為にソフトウェアの実行を繰り返しているわけではない．開発者らは，故障の状態と彼らの持つプログラムやデバッグに関する知見に基づき，故障の原因について仮説を立てる．プログラム実行時には，仮定された原因が成立する環境を作成しプログラムを実行し，同様の故障が発生することを確認する．あるいは，原因が成立しない環境を作成しプログラムを実行することで，同様の故障が発生しないことを確認する．環境の作成には，ソフトウェアの改変も含まれる．このような意図的な改変は，故障の発生確率を向上するだけでなく，結果の解析性も上げる．すなわち，不作為な実行による結果を解析する際には，様々な条件が成立している可能性とその故障との関連性を想定しなければならないが，仮定の下で改変されたソフトウェアは，特定の条件が成立している（もしくは成立していない）ことが明確であり，その条件と故障との関連性を検討しやすい．

このように，当初の故障が観測された時点とは異なるソフトウェアで故障とは異なる状態について扱う故障再現とフォールトアナリシスを議論するためには，故障とは何か，故障を再現するとは何か，フォールトとは何か，フォールトを特定するとは何かを明確にしなければ，議論が成立しない．

5.1.2 フォールトアナリシスモデル

本節では，フォールトアナリシスプロセスの検討に先立ち，フォールトアナリシスを明確にするため，フォールトアナリシスモデルを提案する．まず，故障，フォールト，故障再現およびフォールトアナリシスの概念を定義する．

2.1.1 で述べたように，本論文では，故障とフォールトを対象ソフトウェアの状態で表現する．ソフトウェアの状態とは，ソフトウェアプログラム実行中の瞬間的なスナップショッ

トとし，変数への値の割り当てによって定まるものとする．以下，簡単のため，プログラムを状態と遷移の組からなる状態遷移系として扱うことにする．

定義 5 (プログラムと状態) S を状態の集合とし，$T \subseteq S \times S$ となる T を遷移の集合とするとき，(S,T) をプログラムとする．

例 5 (プログラムと状態) 例えば，ソフトウェアプログラムでは，プログラムカウンタとメモリに格納された値により，プログラムの状態を定義できる．プログラムカウンタの取りうる値の集合を Pc，メモリの番地の集合を Add，メモリが取りうる値の集合を V とするとき，状態集合 S はこれらの組み合わせ，すなわち $S = Pc \times Add \times V$ のように定義することができる．また，ソースコードとは，状態をどのように変化させるかを規定したものであり，遷移 T を定義したものであると考えることができる．例題として挙げたキュープログラムにおいては，考慮すべきメモリとして，キュー，配列，カウンタ，$mutex$ がある．これらの値の組み合わせと，プログラムカウンタの値で状態を定義することができる．

なお，ここでプログラムカウンタと述べたのは，ソフトウェアプログラムのどこを実行しているのかを意味しており，実際には CPU 上のプログラムカウンタでなく，ソースコード上の位置や，ソースコードを解析した結果得られる抽象構文木上の位置であっても構わない．

フォールトアナリシスを行うためには，故障に至る状態の遷移を考える必要がある．ここで，プログラム $p = (S,T)$ において，状態 s_1 から状態 s_2 への遷移可能であることを明示的に表現するため，これを状態間の到達可能性と呼び，$s_1 \triangleright_p s_2$ と表すこととする．

定義 6 (到達可能性) $p = (S,T)$ をプログラムとする．s_1, $s_2 \in S$ について，関係 $s_1 \triangleright_p s_2$ が成立するのは，$(s_1, s_2) \in T$ が成り立つ，もしくは，$s_1 \triangleright_p s$ かつ $s \triangleright_p s_2$ となる s が存在するとき，かつそのときに限る．

例 6 (到達可能性) キュープログラムにおいて，スレッド t_1 が関数 `non_det_int` を呼び出して非決定的な値を取得した状態を s_1 とし，スレッド t_2 がキューから当該値を取り出した状態 s_2 とすると，キュープログラムでは $s_1 \triangleright s_2$ であることが期待される．

テスト等で故障として観測された状態を「故障状態」と呼ぶこととする．

定義 7 (故障状態) (S,T) をプログラム， s_0 を $s_0 \in S$ となる状態とする．故障が状態 s_0 で観測されたとき，s_0 を故障状態と呼ぶ．

例 7 (故障状態) キュープログラムにおいて，キューから取り出した値と，配列から取り出した値が一致しない状態 s_0 とき，その状態は故障状態である．

故障の原因となったフォールトを示す状態を「フォールト状態」と呼ぶこととする．

定義 8 (フォールト状態) プログラム $p = (S,T)$ に対して，$s_0 \in S$ が故障状態であるとき，状態 $s_1 \in S$ が $s_1 \triangleright_p s_0$ かつ s_1 が故障の原因を示すならば，s_1 はフォールト状態である．

例 8 (フォールト状態) キュープログラムにおいて，キューから取り出した値と，配列から取り出した値が一致しない故障状態 s_0 があるとする．s_0 以前の状態として，故障を発生するきっかけとなった状態，例えば取り出す配列のインデックスがキューのインデックスがずれた状態 s_1 がある場合，$s_1 \triangleleft s_0$ かつ s_1 は s_0 の原因であるので，s_1 はフォールト状態である．なお，故障状態 s_0 に対して，その原因となる状態は唯一とは限らず，フォールト状態は複数存在する場合があることに注意が必要である．たとえば，状態 $s_2 \triangleright_p s_1$ において配列のインデックスをインクリメントしたとすると，s_2 を故障の原因と捉えフォールト状態と考えることができる．

5.1.1 節で述べた議論より，故障再現においては，観測された故障と全く同一の故障を再現するわけではない．換言すると，何をもって故障とみなすかという，故障を特徴づける条件を定めることが故障再現に必要である．したがって，ここでは，状態を故障とみなす条件という概念を導入し，これを「故障条件」と呼ぶこととする．

定義 9 (条件の集合) $p = (S, T)$ をプログラム，$s \in S$ を状態，C を条件の集合とする．関係式 $C \Vdash_p s$ は，プログラム p の状態 s において，C のすべての要素 c が成立することを示す．

例 9 (条件の集合) キュープログラムを p,「キューから取り出した値と，配列から取り出した値が一致しない」という条件を c，c を要素として含む集合を C とするとき，キューから取り出した値と，配列から取り出した値が一致しない状態の一つ s について，$C \Vdash_p s$ と表記する．なお，この条件に合致する状態は唯一ではなく，集合 $S' = \{s | C \Vdash_p s\}$ を考えるとき，その要素は複数存在することに注意されたい．

ついで，故障を特徴づける故障状態について定義する．

定義 10 (故障条件) 観測された故障 fr について $C \Vdash_p fr$ が成立するとき，C は故障条件である．

例 10 (故障条件) キュープログラム p において,「*5* 番目の整数値について，キューから取り出した値が *10*，配列に格納された値が *15*」という状態 fr が観測されたとする．これは故障状態にある．しかし，キューと配列で値が異なることが，この故障の本質であり，何番目であるか，あるいは，値が何であるかは，この故障を特徴づけるものではない．条件 c「キューから取り出した値と，配列から取り出した値が一致しない」が故障を特徴づける条件である．このとき，c からなる集合 C は $C \Vdash_p fr$ を満たし，故障条件である．

故障再現とは，故障条件が成立する状態を発見する行為である．発見された状態を再現故障と呼ぶ．

定義 11 (再現故障) $p = (S, T)$ をプログラム，$fr \in S$ を観測された故障状態とする．$C \Vdash_p fr$ が成立する Cr について，$Cr \Vdash_p fr'$ となる状態 fr' を見つける行為を故障再現と呼ぶ．状態 fr' を再現故障と呼ぶ．

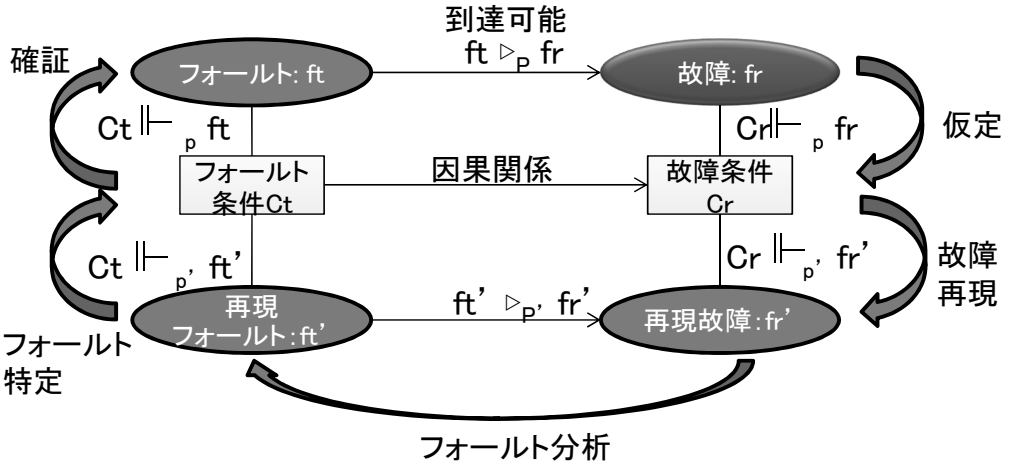

図 5.2: フォールトアナリシスモデル

例 11 (再現故障) 例 *10* において，条件 c「キューから取り出した値と，配列から取り出した値が一致しない」からなる集合 C に対して,「*3* 番目のキューから取り出した値が *2*, 配列から取り出した値が *4*」という状態 fr' を発見したとする．ここで $C \Vdash_p fr$ が成り立ち，fr' は再現故障である．

再現故障 fr' を得られた後，故障の原因となるフォールトを探す．再現故障に対するフォールトを再現フォールトと呼ぶこととする．

定義 12 (再現フォールト) $p = (S, T)$ をプログラム，$fr' \in S$ を再現故障とする．$ft' \triangleright_p fr'$ となる状態 ft' が fr' の原因であるとき，ft' を再現フォールトと呼ぶ．

例 12 (再現フォールト) 例 *11* において,「*3* 番目のキューから取り出した値が *2*, 配列から取り出した値が *4*」という状態 fr' に対して,「カウンタ i が *2* の時にキューからの取り出しないにも関わらず，カウンタをインクリメントする」状態 ft' があったとする．$ft' \triangleright_p fr'$ であり，ft' により fr' が引き起こされると考えられるため，ft' は再現フォールトである．

ここで，当初の故障が発生した際には，再現フォールト状態が発生したわけではないことに注意をする必要がある．故障と同様に，フォールトを特徴づける条件という概念を導入し，これをフォールト条件と呼ぶこととする．

定義 13 (フォールト条件) $p = (S, T)$ をプログラム，$fr' \in S$ を再現故障，Ct を条件の集合とする．$Ct \Vdash_p fr'$ が成立するとき，Ct をフォールト条件と呼ぶ．

例 13 (フォールト条件) 上記の例で,「カウンタ i が *2* の時にキューからの取り出しないにも関わらず，カウンタをインクリメントする」状態 ft' であるが，カウンタの値が他の値でも同様に故障を発生すると考えられ,「キューからの取り出しがないにも関わらず，カウンタをインクリメントする」という条件を含む集合 Ct がフォールト条件となる．

図 5.2 に，故障，フォールト，再現故障，再現フォールトの関係を示す．故障発生時と，故障再現時との間を，故障条件およびフォールト条件が橋渡ししているのが特徴である．

さらに，フォールトアナリシスのフローを矢印で表現した．これを，フォールトアナリシスモデルとして提案する．

フォールトアナリシスモデルに従って，フォールトアナリシスの手順を説明する．まず，プログラム p において，故障 fr が観測される．次に，$Cr \Vdash_p fr$ を満たす故障条件 Cr を仮定する．次に，$Cr \Vdash_{p'} fr'$ となる再現故障 fr' を探索する．これが故障再現である．ここで，p ではなく，p' となっている理由は，3.3 節に述べた通りである．すなわち，故障再現においてはプログラム p そのものでなく，故障の再現性や解析結果の解析性を考慮し，意図的に改変したプログラムを用いることがある．この改変の結果を p' で示している．次に，再現故障 fr' の原因となった $ft' \triangleright_{p'} fr'$ を満たす再現フォールトを探す．これをフォールト解析と呼ぶ．次に，再現フォールト ft' を特徴づける $Ct \Vdash_{p'} ft'$ を満たす条件 Ct を設定する．これによりフォールトが特定されたとみなすこととする．最後に，フォールト条件 Ct を満たし，$ft \triangleright_p fr$ となるフォールト状態 ft が，観測された故障発生時に存在していた可能性を確認する．これが確認できると，フォールト条件が確証される．ここで，フォールト状態 ft は必ずしも観測可能とは限らないため，確認できるのは状態そのものでなく，状態の存在の可能性の有無である．

5.2 仮説演繹法に基づく実験的フォールトアナリシスプロセス

5.2.1 フォールトアナリシスと仮説演繹法

5.1 節では，フォールトアナリシスの構造を示すフォールトアナリシスモデルを形式化した．本節では，仮説演繹法を導入し，フォールトアナリシスを手順化したフォールトアナリシスプロセスについて論じる．ここでは，フォールトアナリシスのプロセス定義に仮説演繹法を用いる意味を論じる．

本論文では，フォールトとは故障の原因となる状態であると定義した．すなわち，フォールトと故障の間には因果関係があり，前節での形式化においても，そのように定義した．しかし，因果関係は確定的に存在し一意に決定するものではない．プログラムの状態の変化は連鎖的であり，状態間の因果関係も連鎖的である．したがって，故障に対して唯一の状態をその原因として限定することは本質的に不可能である．例として次のような状況を仮定する．複数のプログラムによるリソース競合でシステムが停止した故障があったとする．プログラム上では，システム停止の原因は並行プロセスのデッドロックであった．デッドロックの原因は，プロセス間の排他処理の漏れであった．排他処理の漏れの原因は，リソースに対する読み出しと書き込みの違いにあった．このような状況において，システムが停止した故障の原因を，唯一に決定することはできない．いずれも故障の原因ではあり，どれを原因として捉えるかは，システムの設計思想やこのプログラムの修正可能性などによって異なる．

デイビッド・ヒュームは，事象 A が事象 B である必要十分条件として，(1)A と B が隣接している，(2)A が B に時間的に先行している，(3) A と B の間に必然的な結びつきがある，と定義した．その上で，必然的な結びつきは客観的に存在するのではなく，認知によって形成されると主張した．上記に示したように，ソフトウェアのフォールトと故障の関係も同様であり，(1)(2) を満たした状態間において因果関係は蓋然的に存在すると考えられる．同様に，故障条件も客観的に存在するものではない．故障について最初に観測さ

れた現象と，故障の本質が異なる場合は多々あり，何をもって故障とみなすかは，必ずしも確定的でない．

故障条件や，故障に対するフォールトが確定的でない点について，故障が本質的にそのようなものであると本研究では捉えており，それを定式化したのが先に示したフォールトアナリシスモデルである．すなわち，フォールトアナリシスのプロセスの検討においては，故障やフォールトに関する蓋然性について論じる必要がある．蓋然性に関してフォールトの確からしさを高めて確信を持つことがフォールトアナリシスであり，そのような確信を科学的に扱う手法として仮説演繹法に基づいた実験的フォールトアナリシスプロセスを提案する．

5.2.2 拡張フォールトアナリシスモデル

前節で提案したフォールトアナリシスモデルに対して，仮説演繹法に基づいたフォールトアナリシスモデルの拡張を行う．図 5.3 は，図 5.2 を拡張したフォールトアナリシスモデルである．図 5.3 の上部は観測された故障を示し，下部は故障再現を示している．仮説演繹法における仮説と予測はそれらをつなぐ役割を果たしている．

観測された故障 fr に対して，故障条件 Cr を仮定する．次に，Cr に対して，その原因に関する仮説 h を設定する．ここで行われるのは，帰納推論である．次に，仮説 h からの演繹的帰結として予測 pd が導かれる．予測は複数存在しうる．ここでは，予測は事前条件 pre と事後条件 $post$ の組であるとする．すなわち予測は，pre が成立する状況化でプログラム p を実行すると $post$ が観測される，という形をとる．pro と故障条件 Cr とが一致する場合もあれば，そうでない場合もある．予測が故障を再現するものである場合には，$post$ と Ct とは一致する．予測が故障条件の一部に関する言及である場合もあり，その場合は $post$ は Ct から導かれる条件になる．予測が故障条件を満たさない場合もある．これは，ある条件下では，故障は発生しないという予測の場合である．このような予測は，仮定の確からしさを補強する役割を果たす．

図 5.3 下部の故障再現およびフォールト分析は，予測ごとに行う．これを実験と呼ぶ．まず，予測の事前条件 pre を満たすよう，プログラム実行環境を整える．必要に応じてプログラム p を pre が成立するように改変し，プログラム p' を作成する．次に，予測の条件 $post$ を故障再現における故障条件として採用する．pre が成立する環境化で $post$ が成り立つ状態 fr' を観測できたとき，故障が再現されたとみなす．以降，フォールト条件 Ct' を特定するまでは，拡張前のフォールトアナリシスモデルと同様である．

次に，各予測に対する実験で得られたフォールト条件 Ct' を勘案し，故障時のフォールト条件 Ct を決定する．$Ct \Vdash_p ft$ となるフォールと状態 ft が存在する可能性が確認されたら，仮説は確証され，そのもっともらしさが増す．仮説が確証された場合には，さらにフォールト条件を絞り込むため，仮説を精緻化する．もし，故障が再現しない，あるいは適切なフォールト条件がないなど，実験に失敗した場合は，仮説が棄却される．仮説が棄却された場合には，新たな仮説を設定する．以上をフォールトが確信できるまで繰り返す．

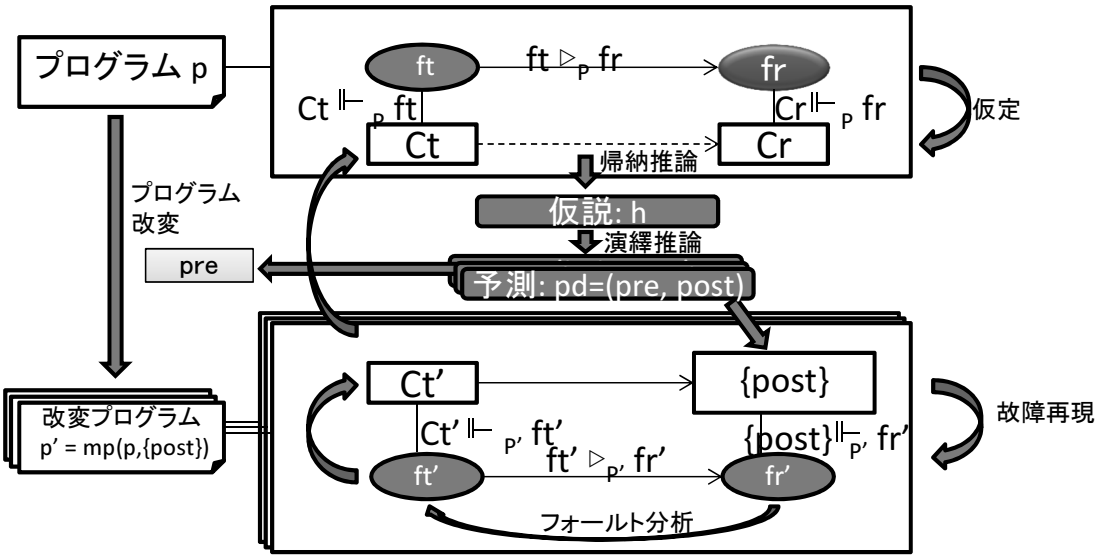

図 5.3: 仮説演繹に基づく拡張フォールトアナリシスモデル

5.2.3 実験的フォールトアナリシスプロセス

図 5.3 に示した仮説演繹法に基づく拡張フォールトアナリシスモデルに従って，実験的なフォールトアナリシスプロセスを定義する．仮説演繹法に従って，大きく 3 つのパートに分割して考える．仮説の設定と予測を行う推定フェーズ，予測に基づいた故障再現を行う故障再現フェーズ，故障再現の結果から仮説の確からしさとフォールトを確証する確認フェーズである．

推定フェーズ

本フェーズでは，故障レポートや開発者らの知識に基づいて，故障の原因に関する仮説を設定する．ついで，複数の予測を仮説から導く．各予測は，2 つの条件の組とする．対象ソフトウェアがどのような環境下で実行されるかを示す事前条件と，その条件下で実行されたプログラムが至る状態を示す事後条件である．

定義 14 (仮説) 仮説は，観測された故障から帰納推論によって得られる故障の原因に関する推測の結果である．

例 14 (仮説) キュープログラムで設定できる仮説の一つは，故障の原因がキューのオーバーフローである，というものである．あるいは，スレッドの並行動作が原因であるという仮説もありうる．他には，排他処理の失敗，配列の領域外アクセス，非決定的な特定の値など，様々な原因を仮説として設定できる．

定義 15 (予測) p をプログラム，h を仮説とする．$(pre,\ \ post)$ を条件の組とし，事前条件 pre の下でプログラムを実行した結果，事後条件 $post$ が成立することを意味するものとする．記法 $h \vdash_p pd$ は，プログラム p に関する知識の下で h から pd が導かれることを示す．このとき，pd を予測と呼ぶ．

例 15 (予測) p をキュープログラム，$\hbar$ を「故障の原因がキューのオーバーフローである」とする．これに対して，予測 pd_1 = (キューのサイズが $1, 2$ つめの値で故障が発生する) が予測の一つとして考えられる $(\hbar \vdash_p pd_1$．あるいは，予測 pd_2 = (格納する数字の数がキューより小さい, 故障は発生しない) も導出可能な予測の一つである $(\hbar \vdash_p pd_2)$.

故障再現フェーズ

仮説 $\hbar$ から予測 $(pre, post)$ が導かれたとする．対象プログラム p を事前条件 pre を満たすように改変する．故障再現では，事後条件 $post$ を満たす p' の状態を探す．このとき，$\{post\} \Vdash_{p'} fr'$ が成り立つ.

定義 16 (改変プログラム) $p = (S, T)$ をプログラムとする．条件集合 C が与えられたとき，$C \Vdash_{p'} s'$ となる $s' \in S'$ を状態として持つよう p を改変してできるプログラム $p' = (S', T')$ を C による p の改変プログラムと呼び，$mp(p, C)$ で表す.

例 16 (改変プログラム) p をキュープログラム，予測 $(pre, post)$ = (キューのサイズが $1, 2$ つめの値で故障が発生する) とする．プログラム p では事前条件 pre を満たさない．そこで，キューのサイズを 1 に変更したプログラム p' を作成し，これを用いて再現実験を行う．前提条件 pre = キューのサイズが 1 を満たすように p を改変してできる p' は，$\{pre\}$ による p の改変プログラムであり，$mp(p, \{pre\})$ で表される.

故障再現フェーズでは，予測の事前条件を満たす改変プログラムにおいて、事後条件を満たす状態が存在するかどうかを，実験によって確認する．このような状態を予測再現故障と呼ぶことにする。

定義 17 (予測再現故障) $p = (S, T)$ をプログラム，$pd = (pre, post)$ を予測とする．$post \Vdash_{mp(p,\{pre\})} s$ を満たす状態 s を，pd に対する予測再現故障と呼ぶ.

例 17 (予測再現故障) p をキュープログラム，予測 $(pre, post)$ = (キューのサイズが $1, 2$ つめの値で故障が発生する) とする．改変プログラム $p' = mp(p, \{pre\})$ において,「2 つめの値で故障が発生する」状態 s が存在するならば，s は pd に対する予測再現故障である。pd に対する予測再現故障とは、事前条件を満たす改変プログラムにおいて，事後条件を満たす状態を示す.

確認フェーズ

予測再現故障が得られない場合，予測が成り立たなかったものとして，仮説を棄却する．予測再現故障が得られた場合，その状態となる原因となる状態を探し，これを予測再現フォールトと呼ぶ.

定義 18 (予測再現フォールト) $p = (S, T)$ をプログラム，$pd = (pre, post)$ を予測，状態 s を，pd に対する予測再現故障とする．$s' \triangleright_{mp(p,\{pre\})} s$ となる状態 s' が s の原因であるとき，s' を pd に対する予測再現フォールトと呼ぶ.

例 18 (予測再現フォールト) p をキュープログラム，予測 $(pre, post) =$ (キューのサイズが $1, 2$ つめの値で故障が発生する) とする．また，改変プログラム $p' = mp(p, \{pre\})$ において,「2 つめの値で故障が発生する」予測再現故障 fr' が観測されたとする．例えば,「2 つ目の値のキューへの格納が失敗したため値が捨てられた」状態 ft' が存在し，これが fr' の原因であるとすると，ft' は pd に対する予測再現フォールトである。

予測再現フォールトに対して，これを特徴づける条件を定義し，これを予測フォールト条件と呼ぶ.

定義 19 (予測フォールト条件) $p = (S, T)$ をプログラム，$pd = (pre, post)$ を予測，状態 s を，pd に対する予測再現フォールトとする．条件の集合 C について，$C \Vdash_{mp(p,\{pre\})} s$ が成立するとき，C を予測フォールト条件と呼ぶ.

例 19 (予測フォールト条件) p をキュープログラム，予測 $(pre, post) =$ (キューのサイズが $1, 2$ つめの値で故障が発生する) とする．例えば,「2 つ目の値のキューへの格納が失敗したため値が捨てられた」状態 ft' が予測再現フォールトであるとき,「2 つ目の値のキューへの格納が失敗したため値が捨てられた」ことが予測フォールト条件と言える.

予測 $(pre, post)$ に対して予測フォールト条件 Ct' が定義できることは，事前条件 pre を満たす改変プログラム $mp(p, \{pre\})$ について，予測フォールト条件 Ct' が満たされることで，事後条件 $post$ を満たす予測再現故障が発生することを意味する．すなわち，pre および Ct' の下で，$post$ が成立する状態が存在することが確かめられた．このように，予測が実験により確認されることで，仮説の確からしさが高まる．さらに，予測フォールト条件 Ct' が得られることで，当初の仮定におけるフォールト条件を精緻化できる．すべての予測が確認されると，仮説の確からしさが増す．開発者が仮説が故障の原因を十分説明できると確信する場合，プロセスは終了する．確信に至らない場合には，仮説を精緻化し，プロセスを繰り返す.

5.2.4 フォールトアナリシスプロセス

以上の議論を踏まえ，策定した仮説演繹法に基づく実験的フォールトアナリシスのアルゴリズム的記述を図 5.4 に示す.

5.3 POM/MC を用いた実験的フォールトアナリシスプロセスの実現

次に，POM/MC を用いた実験的フォールトアナリシスプロセスの実現を検討する.

5.3.1 POM/MC を導入した拡張フォールトアナリシスモデル

図 5.3 に示した拡張フォールトアナリシスモデルに，POM によるモデル抽出とモデル検査を導入したモデルを策定した．策定したモデルを図 5.5 に示す.

Require: $p = (S, T)$:ターゲットプログラム, fr:観測された故障状態 ($fr \in S$)
  repeat
    // 予測フェーズ
    $Cr \leftarrow C$ ただし $C \Vdash_p fr$ // 故障条件
    $\hbar \leftarrow fr$ の原因の仮説 // 故障の原因に関する仮説
    $\mathcal{Pd} \leftarrow \{pd | \hbar \vdash_p pd\}$ // 仮説から予測を導出
    for all $pd \in \mathcal{Pd}$ where $pd = (pre, post)$ **do**
      // 故障再現フェーズ
      $p' \leftarrow mp(p, pre)$ ここで $p' = (S', T')$ // 改変プログラム
      // 確認フェーズ
      $fr' \leftarrow s$ ただし $post \Vdash_{p'} s$ // 予測再現故障
      if fr' が見つからない **then**
        $\hbar$ を棄却し，3 に戻る
      end if
      $ft' \leftarrow s$ ただし $s \triangleright_{p'} fr'$ // 予測再現フォールト
      $Ct' \leftarrow C$ ただし $C \Vdash_{p'} ft'$ // 予測フォールト条件
      $ft \leftarrow s$ ただし $Ct' \Vdash_p s$
      if ft が見つかる **then**
        $\hbar$ の確からしさが増す
      else
        $\hbar$ を棄却し，3 に戻る
      end if
    end for
  until $\hbar$ の確からしさがフォールトを確信するに十分

図 5.4: 仮説演繹法に基づく実験的フォールトアナリシスプロセス

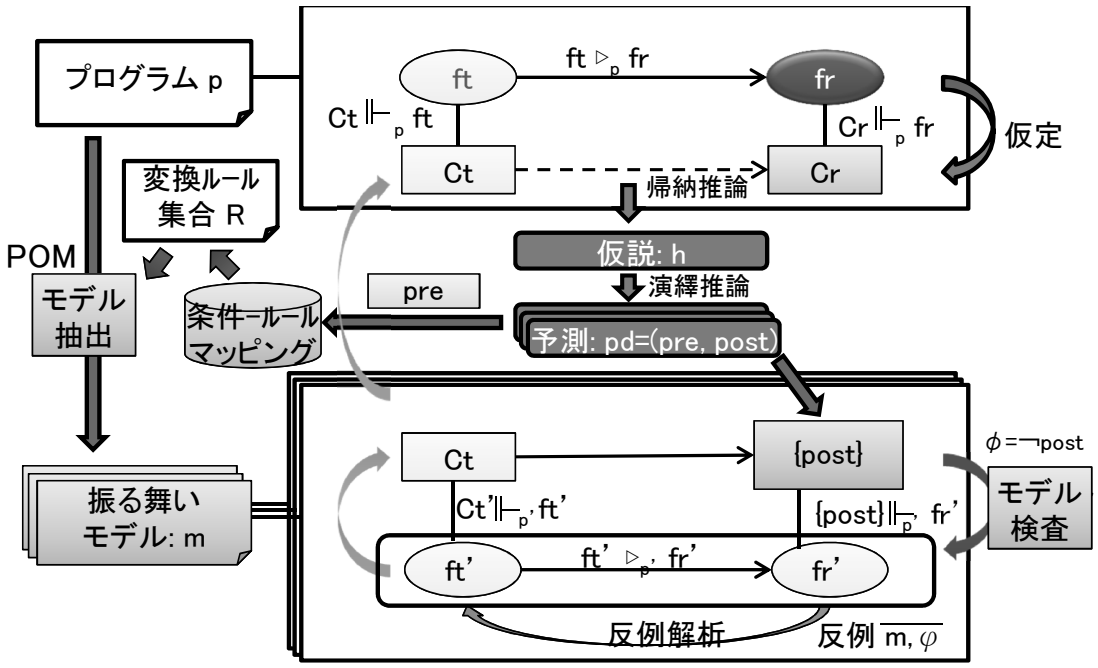

図 5.5: POM/MC を導入した拡張フォールトアナリシスモデル

予測 pd を得るところまでは，前出の拡張フォールトアナリシスモデルと同様である．まず，予測から得られる事前条件 pre に基づいてモデル変換ルールを選択し，POM により Promela モデル m を抽出する．モデル変換ルールの選択方法は後述する．次に，予測から得られる事後条件 $post$ の否定を性質 φ を定義する．次に，モデル m と性質 φ について，SPIN を用いてモデル検査を実施する．その結果，反例 $\overline{m,\varphi}$ が得られたならば，φ を満たさない状態 fr' が発見されたことになる．φ は $post$ の否定であったので，fr' は $post$ を満たす．そこで，反例 $\overline{m,\varphi}$ を解析し，fr' の原因となった ft' を探索する．ft' が見つかった場合は，前出の拡張フォールトアナリシスモデルと同様に，観測された故障との関係を確認する．

以上を踏まえたプロセスをフローとして表現すると，図 5.6 のようになる．

5.3.2 予測とモデル変換ルールのマッピング

POM/MC を用いて予測の事前条件に合ったモデルを抽出するためには，事前条件を満たすモデルを抽出できる変換ルール集合を得る必要がある．ここでは，変換ルール集合の選択手段を示す．

基本的な考え方は，予測の事前条件と変換ルール集合の対応表をあらかじめ用意しておき，対応表に基づいて対応する変換ルール集合を選択することとした．事前条件は主に故障の原因であるので，故障の原因の分類を検討し，事前条件に使われる要素を整理した．表 5.1 に分類結果を示す．

次に，故障再現に用いるモデル変換ルールを，予測の分類に従って従って分類した．図 5.2 に，例を示す．記載したルールの名称は，表 4.1 の記載に合わせた．

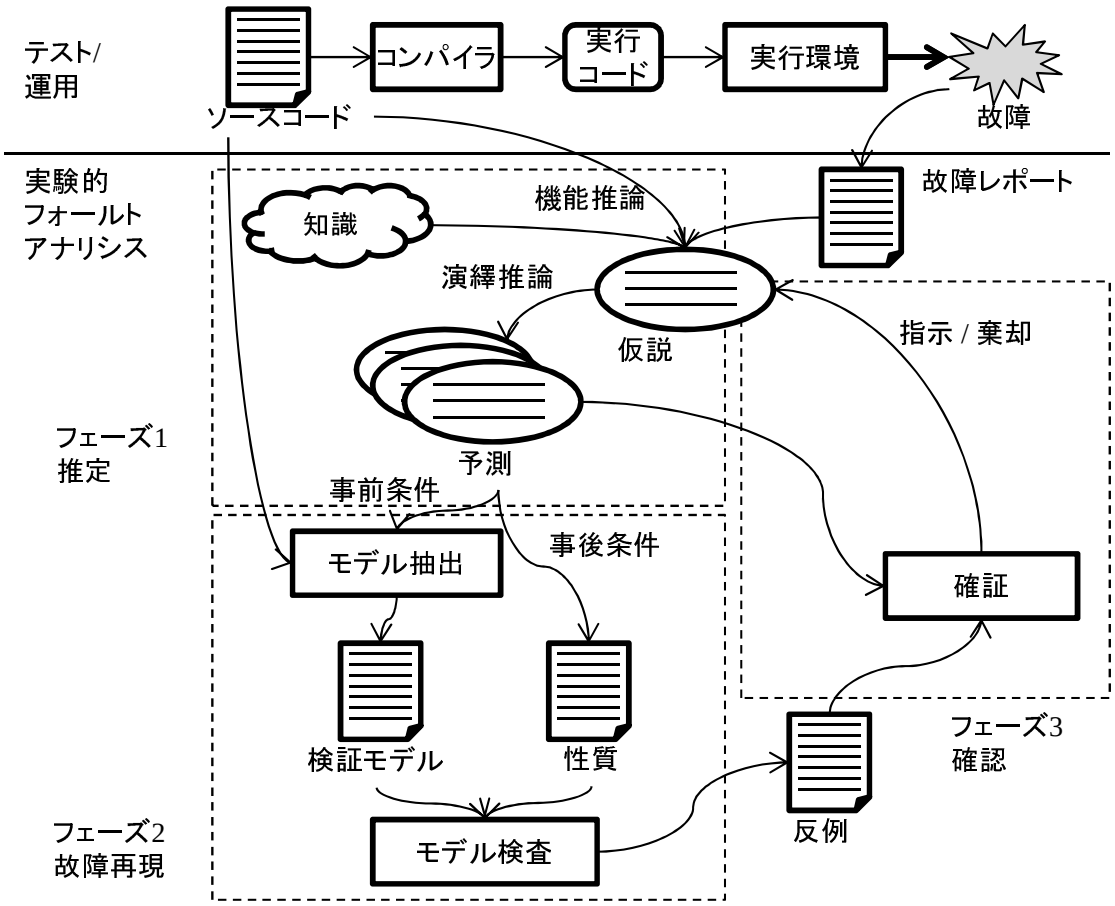

図 5.6: 実験的フォールトアナリシスプロセスの概略フロー

表 5.1: 予測の分類

分類	説明	事前条件の例
原因	故障を発生させる要因	並行性，非決定性，割り込み，ループ，値域
部位	故障を発生させるプログラム部位	特定のプロセス，スレッド，関数，ブロック，表明
フロー	故障を発生させるコントロールフロー	特定の条件，分岐
データ	故障を発生させるデータ	入力パラメタ，変数，アドレス，レジスタ

表 5.2: モデル変換ルール集合の分類

分類	説明	ルールの例
原因	故障要因の有無を変更する	Pthreads，スタブ
部位	実行する部位を変更する	エントリポイント，状態遷移ドライバ，スタブ
フロー	プログラムのフローを変更する	(現状該当なし)
データ	プログラムのデータを変更する	未使用宣言の削除

例えば，Pthreads と呼ばれる変換ルールは，Pthreads ライブラリによる並行実行を，Promela の並行機構を用いて模擬する．スタブと呼ばれる変換ルールは，関数呼び出しの呼び出し先をスタブに置き換える．この変換ルールは，事前条件分類の「並行性」に対応する．Pthreads に関するライブラリ関数の呼び出しを，Pthreads 変換ルールを用いて Promela で模擬するか，スタブ変換ルールを用いてスタブに置き換えるかによって，モデルの並行性の有無を制御できる．

事前条件と変換ルールを対応づけられることから，条件からから変換ルールへのマッピングを定義することができる．このマッピングを条件-ルールマッピングと呼ぶ．

定義 20 (条件-ルールマッピング) 条件-ルールマッピングは，条件のタイプから変換ルール集合へのマッピングである．C 条件の集合，R を変換ルールの集合とする．条件-ルールマッピング $\mathcal{C2R}$ は，$\mathcal{C2R} \subseteq C \times Power(R)$ である．各条件 $c \in C$ に対して，表記 $\mathcal{C2R}(c)$ は，c からマッピングされた変換ルール集合を示す．

5.3.3 POM/MC による実験的フォールトアナリシスプロセス

ここで，モデル抽出とモデル検査を用いた実験的フォールトアナリシスプロセスを示す．図 5.7 に同プロセスのアルゴリズム的記述を示す．

推定フェーズ

$Cr \Vdash_p fr$ となる故障条件 Cr を定義する．fr の原因に関する仮説 $\mathcal{h}$ を設定する．$\mathcal{Pd} = \{\mathcal{pd} | \mathcal{h} \vdash_p \mathcal{pd}\}$ となる予測集合 $\mathcal{Pd}$ を導出する．

故障再現フェーズ

$\mathcal{pd} \in \mathcal{Pd}$ となる $\mathcal{pd}$ を選択する．このとき，$\mathcal{pd} = (pre, post)$ とする．$\mathcal{C2R}$

$\mathcal{C2R}$ を用いて $R = \mathcal{C2R}(pre)$ を得る．もし pre が $\mathcal{C2R}$ の定義域に含まれない場合は，pre に対応する新たな変換ルール r を定義し，$\mathcal{C2R}$ を修正する．プロパティ $\neg post$ を定義する．モデル検査を実施し，反例 $pommc(sc(p), \mathcal{C2R}(pre), \neg post)$ を得る．

Require: $p = (S, T)$:ターゲットプログラム, $sc(p)$:p のソースコード, fr:観測された故障状態 ($fr \in S$), $\mathcal{C2R}$:条件-ルールマッピング
repeat
// 予測フェーズ
$Cr \leftarrow$ C ただし $C \Vdash_p fr$ // 故障条件
$\hbar \leftarrow fr$ の原因の仮説 // 故障の原因に関する仮説
$\mathcal{Pd} \leftarrow \{pd | \hbar \vdash_p pd\}$ // 仮説から予測を導出
for all $pd \in \mathcal{Pd}$ where $pd = (pre, post)$ **do**
// 故障故障再現フェーズ
$R \leftarrow \mathcal{C2R}(pre)$ // モデル変換ルールの取得
$\varphi \leftarrow \neg post$ // 性質の定義
$ce \leftarrow pommc(sc, R, \varphi)$ // モデル検査による反例の獲得
// 確認フェーズ
$fr' \leftarrow s$ ただし $\{post\} \Vdash_{pom(R,sc(p))} s$ // 予測再現故障
if fr' が見つからない場合 **then**
$\hbar$ を棄却し，4 に戻る.
end if
$ft' \leftarrow s$ ただし $s \triangleright_{pom(R,sc(p))} fr'$ // 予測再現フォールト
$Ct' \leftarrow C$ ただし $C \Vdash_{pom(R,sc(p))} ft'$ // 予測フォールト条件
$ft \leftarrow s$ ただし $Ct' \Vdash_p s$
if ft が見つかる **then**
$\hbar$ の確からしさを増す.
else
$\hbar$ を棄却して，4 に行く
end if
end for
until $\hbar$ の確からしさがフォールトを確信するに十分

図 5.7: POM/MC に基づく実験的フォールトアナリシスプロセス

確認フェーズ

反例が得られない場合は，仮説 $\hbar$ を棄却し，推定フェーズに戻る．反例が得られた場合には，反例によって得られた再現故障を fr' とする．

$ft' \triangleright_{pom(\mathcal{C2R}(pre),sc(p))} fr'$ となる ft' を再現フォールトとして選択する．$Ct' \Vdash_{pom(\mathcal{C2R}(pre),sc(p))} ft'$ となる条件 Ct' を定義する．Ct' はフォールト条件の候補となる．

$Ct' \Vdash_p ft$ を満たす状態 ft が，元のプログラム p に存在しうるか確認する．ft が存在するならば，ft は観測された故障に対するフォールトの候補であり，仮説 $\hbar$ が支持され，確からしさが増す．推定フェーズに戻り，仮説を精緻化する．ft がありえない場合には，h は棄却され，推定フェーズに戻って仮説を設定しなおす．フォールトを確信するまで，繰り返す.

第6章　ケーススタディ

6.1　キュープログラム

例題として用いたキュープログラムを用いて，実現性を評価するケーススタディとした．対象とする故障として，$stored\_element$ 配列に記録された値と，キューから取り出した値が異なる問題を用いた．2 度の試行により，この故障に関するフォールトを特定することができた．以下，試行のステップを示す．説明のため，仮説，予測およびプロパティは自然言語で記述する．

6.1.1　一度目の試行

キュープログラム p は，故障が発生した場合，表明 (`assert`) により，エラーを表示して停止する．本例において観測される故障のソースコード上の位置は表明の位置である．また，その時点での一部の変数の値がエラー表示として観測できる．対象とする故障状態 fr を次のように表現する．

$$fr = (\text{“}ERROR\ on\ t2\text{”},\ i = 2,\ stored\_element[i] = 0\ on\ t2)$$

推定フェーズ (1)

故障 fr を特徴づける $Cr \Vdash_p fr$ 故障条件 Cr を設定する．ここでは，エラーが表示されることを故障条件 Cr とする．

$$Cr = (\text{“}ERROR\ on\ t2\text{”})$$

キュープログラムの例では，様々な故障 fr の原因が想定できる．例えば，2 つのスレッドのレースコンディション，非決定的に選択される数値，キューの状態，ループ数，分岐条件の間違い，オーバーフローやアンダーフロー，アルゴリズムの間違い，コーディングミスなどである．

最初の仮説 $\mathcal{H}_1$ として，スレッド t_1 とスレッド t_2 との間の並行性が故障の原因であると設定した．

$$\mathcal{H}_1 \ = \ \{\text{“}Concurrency\ caused\ the\ ERROR\ on\ t2\text{”}\}$$

次に，$\mathcal{H}_1 \vdash \mathcal{PD}_1$ を満たす予測 $\mathcal{PD}_1$ として，並行実行する場合に故障が発生するとした．

$$\mathcal{PD}_1 \ = \ (\text{“}Concurrency\text{”}, \text{“}ERROR\text{”})$$

再現フェーズ (1)

$\mathcal{Pd}_1$ の事前条件である “$Concurrency\ relates$” に対して，C2R マッピングを用いて，“Pthreads”ルールを選択した．$Pthreads$ ルールは，4 章で述べたように，Pthreads の機能を Promela の並行機能で実現する環境マッピングルールである．

$$R_1 = (\text{“}Pthreads\text{”})$$

また，検査するプロパティとして，$\mathcal{Pd}_1$ の事後条件の否定である “$\neg ERROR$” を設定した．

$$\varphi_1 = \neg \text{“}ERROR\text{”}$$

これにより，“assertion”と呼ばれるモデル変換ルールを選択し，ソースコード中の ERROR フラグを表明に変換することとした．性質は LTL 式で表現することもできるが，キュープログラムのようにエラーを摘出するコードがソースコード中に記述されている場合には，表明を用いる方が簡便であると判断した．

$$R_1 = (\text{“}Pthreads\text{”}, \text{“}assertion\text{”})$$

選択したモデル変換ルール集合を用いて，Pthreads ライブラリを模擬し故障を検出する Promela モデルを生成することができた．反例 ce を得た．

$$ce_1 = pommc(R_1, sc(p), \varphi_1)$$

確認フェーズ (1)

反例 ce を解析することで，故障条件 Cr を満たす再現故障 fr' を得た．

$$fr'_1 = \quad (\text{“}Error\ in\ t_2\text{”}, i = 2, stored_elements[i] = 0\ in\ t_2)$$

また，$ft'_1 \triangleright_{pom(R_1,sc(p))} fr'_1$ を満たす状態として，スレッド $t1$ で整数値をキューに格納した数を示す変数 i の値と，スレッド $t2$ で参照する配列のインデックスを表す i の値が異なる状態が存在することが判明した．

$$ft'_1 = (i = 1\ on\ t_1, i = 2\ on\ t_2)$$

モデル検査によって反例を得ることができたが，並行性によって，なぜ故障が発生するのか詳細を理解することはできなかった．つまり，状態 ft' が fr' の原因であるフォールトであるとは確信が得られなかった．そのため，仮説 $\hbar_1$ は棄却はされなかったが，充分なもっともらしさを得ることはできなかった．

6.1.2 二度目の試行

推定フェーズ (2)

故障状態 fr と故障条件 Ct は維持する．一度目の結果により，並行性に関する予測は棄却されなかったため，$\hbar_1$ は仮説として維持し，一度目の試行で得られた新たな仮説を加え

ることとした．ft'_1 が故障に関係する状態であることから，関数 nondet_int による値の非決定性は故障に関係がないという仮説と，キューのサイズは故障に関係がないという仮説を加えて $\hbar_2$ とした．

$$\begin{aligned}\hbar_2 \;=\; \{&\text{“}Concurrency\ caused\ the\ ERROR\text{”},\\ &\text{“}Nondeterminism\ doesn't\ relates\ to\ the\ ERROR\text{”}\\ &\text{“}Size\ of\ queue\ doesn't\ relates\ to\ the\ ERROR\text{”}\}\end{aligned}$$

そこから，並行実行において，値を固定値にして，キューのサイズを最小にしても，故障が発生するという予測 $\mathcal{Pd}_2$ を導出した．

$$\begin{aligned}\mathcal{Pd}_2 \;=\; \{&(\text{“}Concurrency\text{”}\ and\ \text{“}fixed\ value\text{”}\ and\ \text{“}minimal\ queue\ size\text{”},\\ &\text{“}ERROR\text{”})\}\end{aligned}$$

故障再現フェーズ (2)

予測 $\mathcal{Pd}_2$ の事前条件に加えた “fixed value”は，`non_detint` 関数の動作を変更することで実現できる部位仮説であるため，対応するスタブ変換ルールを選択した．しかし，キューのサイズはマクロで定義されており，C 言語モデルで表現される ASG 上ではキューのサイズは具体値に展開されているため，モデルの変換ではキューのサイズの変更は困難であると判断した．そのため，キューのサイズについては，手作業でマクロ定義を変更した．

$$\begin{aligned}R_2 &= (Pthreads, StubFunction)\\ \varphi_2 &= \neg\text{“}ERROR\text{”}\end{aligned}$$

一度目と同様，φ_2 は表明を用いて検出できるため，対応する変換ルールを R_2 に加えた．

$$R_2 = (Pthreads, StubFunction, Assertion)$$

選択したモデル変換ルール集合を用いて Promela モデルを生成した．反例 ce_2 を得た．

$$ce_2 = pommc(R_2, sc(p), \varphi_2)$$

確認フェーズ (2)

モデル検査により，故障条件 Cr を満たす再現故障 fr'_2 を得た．

$$fr'_2 = \quad (\text{“}Error\ in\ t_2\text{”}, i = 2, stored\_elements[i] = 0\ in\ t_2)$$

また，反例 ce_2 を解析することで，$ft'_2 \triangleright_{pom(R_2,sc(p))} fr'_2)$ を満たす状態として，スレッド t_1 が動作する前に，スレッド t_2 が動作しカウンタ `i` を加算した状態 ft'_2 が見つかった．

$$ft'_2 = (\text{“}t_1\ not\ run\text{”}, i = 1\ in\ t_2)$$

図 6.1 に，正常動作時と故障発生時のシーケンス図を示す．本シーケンス図は，POM/MC が反例として生成したシーケンス図を簡約して作成したものである．反例の解析によって，

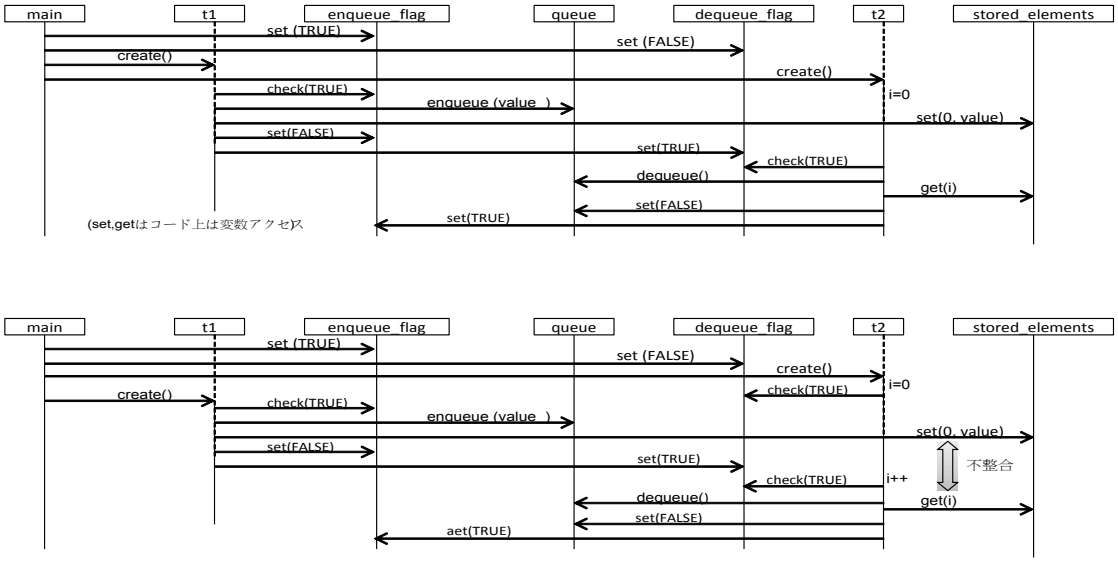

図 6.1: キュープログラムのシーケンス図

2つのスレッドが特定の順で実行される時に故障が発生することがわかった．つまり，スレッド t_2 がスレッド t_1 より先に実行した場合，`deque_flag` が `FALSE` のためデキューをしないまま，カウンタ `i` のインクリメントのみをすることがある．

そこで，スレッド t_2 が，スレッド t_1 より先に動作を開始することをフォールト条件 Ct とした．

$$Ct = \text{“}t_1\ not\ run\text{”}, \text{“}t_2\ run\text{”}$$

実際，元のソースコードを精査すると，Ct を満たす状態が発生しうることがわかった．これが故障の原因となるフォールトであると結論づけた．

まとめ

本ケーススタディは，Intel®Core™i7-3770 CPU (3.4 GHz) と 16GB のメモリを搭載し，Windows®7 Ultimate を搭載した PC で実施した．モデル検査に用いた計算機リソースは図 6.1 にまとめる．各試行において，SPIN のオプション 3 種を試した; エラーを 1 つのみ見つける (-c1)，最短パスのエラーを見つける (-i)，すべてのエラーを見つける (-e) である．SPIN が探索した状態空間 (SPIN の出力としては Transitions=stored states の数と matched states の数の合計) や，モデル検査にかかる時間は，1 度目の試行に対して 2 度目の試行は著しく減少している．1 度目の試行での，すべてのエラーを見つける場合には，メモリー不足によりモデル検査が失敗した．また，反例トレースの長さもモデル抽象化によって短縮した．短い反例トレースは長い反例トレースよりもフォールトの特定は容易であろうと考えられる．

表 6.1: キュー問題のケーススタディ結果

	1つのエラー		最短パスのエラー		すべてのエラー	
	1度目	2度目	1度目	2度目	1度目	2度目
遷移 (Transitions)	58,372	176	199,063	1,791	(failed)	1,791
メモリ量 (MB)	70	65	110	65		65
時間 (msec)	97	3	690	4		2
エラーの数	1	1	364	4		6
反例の長さ (lines)	287	82	88	82		-

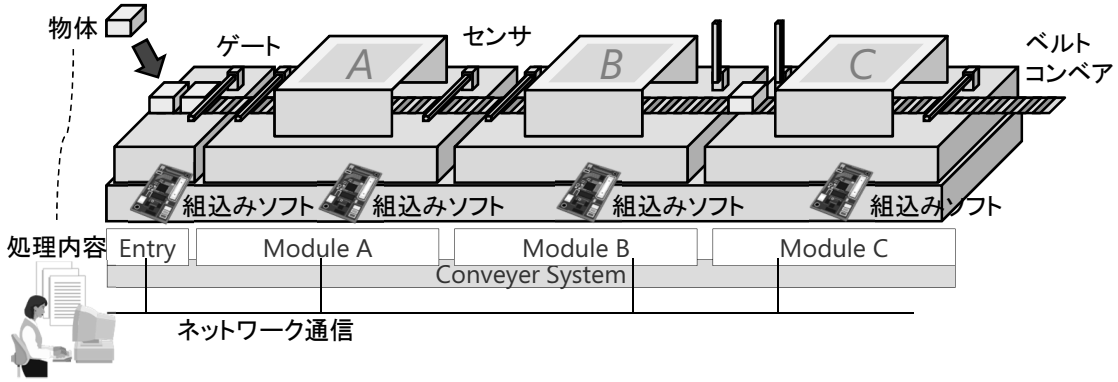

図 6.2: 搬送システムの概要 (再掲)

6.2 産業ソフトウェア

6.2.1 搬送システム

本節では，実製品に対する適用結果を示す．対象は 1.1.3 で組込みソフトウェアを搭載したシステムの例として示した搬送システムである．図 6.2 に，同製品の概要構成図を再掲する．各モジュールのソフトウェアは相互に影響関係のある並行動作するプログラムであり．各ソフトウェアはベルトコンベア，ゲート，物体に施す処理を制御する．また，ソフトウェアは，物体ごとにユーザが定義した処理内容を管理している．ベルトコンベアによって物体は搬送され，個々の物体の位置はセンサを用いてソフトウェアがリアルタイムに把握する．物体に対する処理内容を含む各物体に関する情報は，物体の移動と共に，モジュール間で通信によって受け渡される．各モジュールのソフトウェアは，この物体ごとの情報に基づいて物体に対する処理内容を決定し，ハードウェアモジュールに処理を指示する．

搬送システムは，物理と情報を同期してリアルタイムに制御し，かつ大量に複数の物体を並行して処理する複雑なシステムとなっている．実際の製品において，メッセージ送受信の特定シーケンスに起因するデッドロックに関する故障が発生し，実システムでは再現が困難であった．この故障について，フォールトアナリシスを実施した．

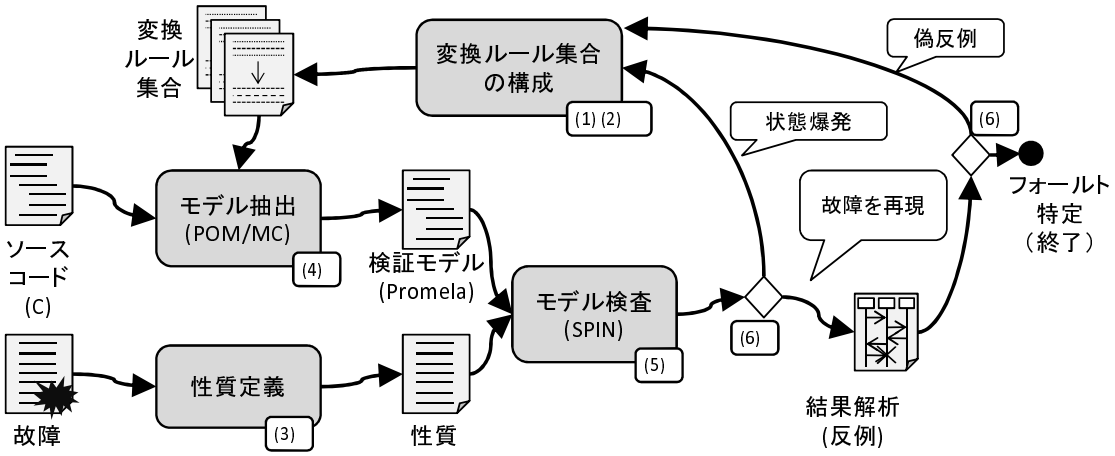

図 6.3: 実施した手順

6.2.2 実施条件

本ケーススタディに関わった参加者のタスクについて，表 6.2 に示す．第一の参加者は，QVTo の実装に関して 3 カ月の経験を持っていた．一日間のターゲットシステムの構造および運搬仕様の教育を受けた．第二の参加者は，ターゲットシステムの開発に参加し，システムの全体の仕様と設計を理解していた．両参加者は，モデル検査の経験を持っていた．分担としては，主に第一の参加者がルールの設計記述を行い，第二の参加者がモデル検査の実施と，抽象化の設計を行った．

ユースケースの際には，次の最低限の情報が与えられていた：

- デッドロックを発生したことで内部の状態遷移がストップした
- モジュール間のメッセージ送受信において特定のシグナルを送ったときに発生する

6.2.3 フォールトアナリシス手順

図 6.3 に，実施したフォールトアナリシスの手順を示す．

1. 抽出するモデルの設計: モデルとして抽出するソースコードの範囲と，外部環境として抽象化する範囲の境界を定める．例えば，ハードウェアを操作する関数はすべて空のスタブ関数に置き換える，あるいは，モジュール間の通信をグローバル変数の読み書きに置き換えるなどが考えられる．

表 6.2: ケーススタディの参加者

#	変換ルール	モデル検査	製品ドメイン
1	√	√	(√)
2	-	√	√

2. モデル変換ルール集合の構成: 上記のモデルの設計に従って，一般的な（POM/MC が標準で持っている）モデル変換ルールを選択するか，対象とするソフトウェアや問題にあわせてモデル変換ルールを新たに作成する．

3. プロパティの定義: 検査したいプロパティを LTL 式または assertion として定義する．

4. モデル抽出: POM/MC にソースコードと上記で構成したモデル変換ルール集合を与え，Promela コードを自動抽出する．

5. モデル検査: POM/MC から SPIN を起動し，抽出したモデルがプロパティを満たすか否かを検査する．

6. モデル検査結果の評価: モデル検査の結果を評価し，対象となる故障を発見できたか否かを決定する．SPIN がプロパティ違反を発見した場合，反例トレースが対象とする故障に至る振る舞いを表現していることになる．もし，反例トレースが示す振る舞いが対象のソースコードでは発生しないものである場合や，状態爆発が起きた場合には，ステップ (1) またはステップ (2) に戻る．ケーススタディにおいては，モデル検査が 2 時間で終了しない場合には，状態爆発が発生したと判断した．

6.2.4 結果

表 6.3 に示したように，7 回の試行錯誤によって，対象となる故障を再現した．特徴的なステップについて説明する．モデル検査が状態爆発で実施できない場合には，モデル変換ルールを再構成してターゲットモデルを抽象化することで，さらに状態空間を縮小する必要があった．例えば，表 6.3 のステップ 3 では，モジュール間通信に伴う状態数を削減するため，パラメタを受け渡すプロセスをバイパスしプロセス間で直接通信するモデル変換ルールを適用した．しかし，モデル検査によって得られた反例は，元のソフトウェアでは発生しえない偽反例であった．偽反例を得た場合には，抽象化を緩めるか，異なる抽象化の方法を採用する必要がある．ステップ 4 では，ステップ 3 での抽象化をやめ，ネットワークを単純化する抽象化を行った．結果，再び状態爆発が発生したため，ステップ 5 ではネットワークをさらに単純化した．このように，抽象化を試行錯誤的に繰り返すことで，ステップ 8 において故障の再現に至った．

図 6.4 に，各ステップでの Promela モデルの行数と，Promela モデルに含まれるシンボルの数の変化を示した．抽象化によりモデルを縮退していた様子が見てとれる．

以上のステップで故障を再現するのに，約 4 日間，工数にして 7 人日を要した．反例を分析するためのソースコードやドキュメントの調査時間は含んでいる．ただし，対象システムの調査や，POM/MC および SPIN の実行時間は含んでいない．元のソースコードは約 17,000 行であり，最終的に抽出されたモデルは，Promela で約 2,000 行（コメント行等を除く）であった．

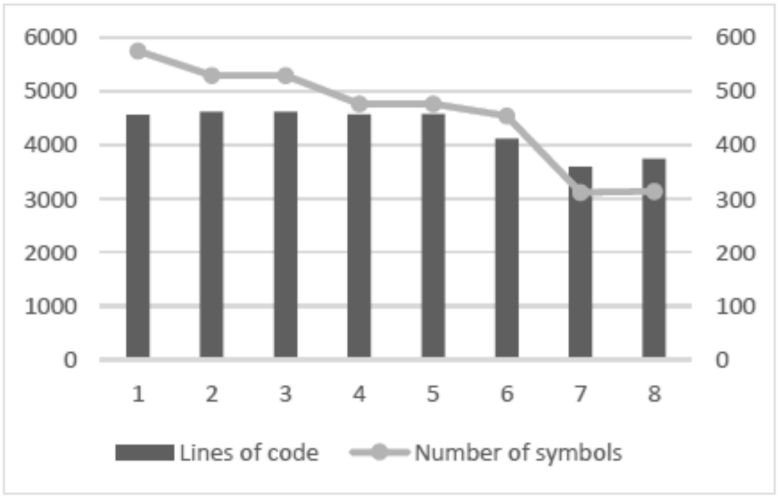

図 6.4: 段階的なモデルの抽象化

表 6.3: フォールトアナリシスの経緯

#	変換ルールの内容 (差分)	結果
1	ドライバ作成，ハードウェア動作模擬 (搬送制御, 通信, シグナルの発生)	状態爆発
2	ハードウェア動作模擬の追加 (シグナルの処理)	状態爆発
3	縮退の追加 (関数引数処理の簡略化)	偽反例
4	縮退の追加 (通信前処理の簡略化)	状態爆発
5	縮退の追加 (通信前処理のさらなる簡略化)	偽反例
6	ハードウェア動作模擬の追加 (モジュール間の同期処理)	状態爆発
7	縮退の追加 (タイムアウト処理の削除)	偽反例
8	縮退の追加 (タイムアウト処理の簡略化)	フォールト特定成功

第7章　評価・考察および今後の課題

7.1　提案手法の有効性に関する考察

7.1.1　ケーススタディの結果の評価

提案手法によるフォールトアナリシスの効率性

実製品である搬送システムにPOM/MCを適用した例では，実際に発生した故障について，モデル検査による故障再現を行い，その原因を特定した．これにより，提案したフォールトアナリシスプロセスとそれをモデル抽出とモデル検査で実現したPOM/MCは，実製品開発におけるフォールトアナリシスに有効であることを示した．実際のソフトウェア開発では，本故障を再現するためにモデル検査を用いず約1カ月を要した．また，類似のシステムに対して手作業で振る舞いモデルを作成してモデル検査を適用した場合の実績を基づき，本故障について手作業でモデル検査を用いた場合に要するコストは20人日と見積もられた．提案手法では，7人日でフォールトの特定を完了した．以上より，提案手法は，再実行によるフォールトアナリシスや，手作業でのモデル検査によるフォールトアナリシスに比べて，コスト削減においても有効であることが示された．本研究により，従来手法よりも効率的なフォールトアナリシス手法を確立した．

コスト削減の要因に関する考察

このようなコスト削減が得られた要因を考察する．

第一には，POM/MCによる自動的なモデル抽出が挙げられる．千行を超えるモデルを手作業で記述する場合と比べて，ツールで自動抽出する場合は，モデル記述に要する時間が大きく異なることは自明である．同時に，20人日から7人日への削減率は，手動と自動のモデル記述に要する時間の差に比べると小さく，モデル記述そのものはコストに対して支配的ではなかったことを示唆する．

そこで，モデル抽象化の内容を比較する．搬送システムの例で用いた抽象化の手法は，同規模の類似システムでの手動のケースに比べると，単純なものが多かった．手動のケースでは，20以上の抽象化手法を用い，それらはアルゴリズムやデータ構造の変更の変更など複雑なものが多かった．搬送システムのケースでは，約10の抽象化手法を用い，それらは，関数のスタブ化など，単純なものが中心であった．モデル抽出を自動化することによりモデルを記述する時間が短縮されただけでなく，第二の要因として抽象化の検討そのものの時間が減ったと考えられる．

この抽象化の違いは，次の２つの要因から発生したと考えられる．一つ目の要因は，ソフトウェアアーキテクチャの違いである．搬送システムのソフトウェアアーキテクチャは，

テスト時にシミュレーションを行うために，ハードウェア操作などの低レベルの操作は統一されたレイヤのインタフェースとして定義されており，抽象化がしやすかった．この点は，提案手法によるコスト削減ではない．

２つ目の要因は，モデル抽象化の方向性である．手で検証用モデルを構成していく場合には，小さなモデルからはじめて徐々に詳細を追加していく方式をとることが多い．これはモデル記述の観点ではリーズナブルであるように思えるが，元となるソースコードからの抽象化という観点で見ると，最初に大規模な抽象化を行い，徐々に抽象度を下げていく（ソースコードに近づけていく）ことになる．初期の抽象化は大規模なだけでなく，複数の抽象化を同時に行い，整合性を取る複雑な作業となり，モデル規模は小さい割にはコストを要する方法である．他方，POM/MC による抽象化プロセスは，実装レベルのモデルからはじめて，徐々に抽象度を上げていく．これは，最初に単純な抽象化を適用して，複雑な抽象化を後で適用していくことになる．そのため，各ステップでの抽象化のコストを減らすことができたと考えられる．

このように，POM/MC によるモデル抽出は，モデル記述作業の単なる自動化ではなく，実施者の思考過程を効率化する効果があったと考えられる．

7.1.2 POM/MC によるモデル抽出

変換可能な C 言語に関する考察

POM/MC は，すべての C 言語ソースコードを Promela モデルに変換することを目的としていない．モデル変換ルール集合で指定された情報をモデルとして抽出することが目的であり，モデル変換ルール集合に定義されない情報は抽出されない．本研究において作成したモデル変換ルール集合により，C 言語コードが Promela コードにどう変換されるかは，表 4.2 に示した．

POM/MC では完全に扱えない C 言語の要素の代表例として，ポインタがある．POM/MC では，図 4.1 に示す “ポインタのエイリアス解析” ルールを用意し，ポインタによる参照を，変数の実参照へと変換する．この変換は，ポインタの参照先が静的に決定するという前提の下でのみ健全であり，動的に参照先が変わるポインタについては，妥当な Promela モデルを生成することができない．

しかし，本研究が対象としたソフトウェアにおいては，ポインタの参照先が静的に決定するという前提は，現実的なものであった．組込みソフトウェアにおいては，メモリリソースの節約や，処理速度の高速化のためにポインタが用いられることは多い．その一方，利用できるメモリが限られている上に仮想記憶が利用できないこともあり，動的にメモリを確保することは動作時に故障の原因となる．そのため，ソースコード上では `malloc()` 関数等を用いて動的にメモリ確保をしていたとしても，実質的にはあらかじめ定められたメモリマップを構成するものであり，ポインタについても定められた領域の参照だとみなすことができるケースが多い．そのため，C 言語に対する一般性は損なわれても，ポインタ参照を含むソースコードについて，フォールトアナリシスを実施することができた．

一方，本論文に示したケーススタディには含まれなかったが，動的なポインタとしてよく使われる例としては，ポインタを使ったリスト構造などがある．このような動的なポインタを汎用的に Promela で表現する方法の一つとして，メモリモデルを Promela により

模擬する方法が考えられる．しかし，メモリモデルの模擬は，状態数を爆発的に増大させることになる．POM の考え方としては，汎用性のある言語間の変換を実現するのではなく，対象となるソースコードの特徴に合わせた抽象化を行うことで，実用性を確保する．例えば，前述のポインタを使ったリスト構造に対しては，ポインタ操作を Promela で実現するのではなく，リストという論理的な要素を実現して，これに変換する．つまり，リストに対する値の格納や取り出しといった単位を論理要素として抽象プログラムモデルに定義し，これを実現する Promela コードに変換することが効果的であると考える．しかし，どのような論理要素を用意すればよいか，どのような論理要素がよく使われるか，という点については分析できておらず，今後の課題である．

また，6.1 節のキュープログラムの例に現れたように，実装モデルはソースコードに対してプリプロセスを施した後の ASG を表現しているため，マクロ定義などで表現されている意味はモデルに抽出することができない．C 言語プログラミングでは，可読性や保守性を確保するために，システムの特徴を示す定数値などを一方，プリプロセス前のソースコードは，C 言語の構文を維持しておらず，構文解析が困難である．マクロ定義等で表現されている意味の扱いも，今後の課題である．

POM コンセプトフレームワークの言語変更可能性

POM コンセプトフレームワークの中間モデルは，実装モデル，抽象プログラムモデル，ターゲットモデルの 3 つで構成している．そのため，C 言語以外のプログラム言語に対して実装メタモデルを，Promela 以外のモデル言語に対してターゲットメタモデルを定義し，メタモデル間の対応関係を QVTo で定義することで，他の言語にも対応できる汎用性を指向している．しかし，本研究では入力言語を C 言語と仮定し，抽象プログラムメタモデルは C 言語の ASG をベースに抽象化し，論理要素を加えたものである．そのため，現状の抽象プログラムメタモデルで任意の言語を表現できるわけではない．たとえば，Java 言語を入力言語とする場合，現状の抽象プログラムメタモデルでは，クラスの継承などについて完全に表現することはできない．逆に言えば，抽象プログラムモデルを拡張することにより，C 言語以外のプログラム言語で書かれたコードを入力とすることが可能となっている．この際には，入力言語に依存する拡張を行うのではなく，言語のパラダイムを表現する拡張（例えば Java 言語に対してはオブジェクト指向）をすることが，汎用性維持の点で望まれる．

ターゲットモデルについても同様である．他のモデル検査ツール用の言語への変換には，その言語に特有の要素を中間プログラムメタモデルの論理要素として用意することで，変換が容易になると考えられる．たとえば NuSMV[13] のような状態遷移に基づくモデル言語に変換するためには，言語間の差異を吸収するとともに，例えば手続き型言語を状態遷移で表現するという，意味論の変換が必要である．任意のプログラムを一般的に状態遷移で表現することは難しい．しかし，状態遷移モデルに基づいて設計されたプログラムによっては，状態を示す変数や状態遷移を行う関数，あるいはイベントの操作がコーディング規則やフレームワークで規定されている場合がある．そのような場合には，論理要素として状態遷移系を構成する状態やイベントなどを示す構造を定義し，状態遷移に関して規定されたプログラム要素を定義した論理要素に変換する変換ルール集合を作成することで，QVTo で定義できると考えられる．

このようにPOMフレームワークでは，入力言語およびターゲットモデルが変更可能な汎用性を指向している．しかし，本研究においては，他言語の実現性は評価しておらず，今後の課題である．

モデル変換ルール集合の管理

これまでに示したように，POM/MCのモデル変換ルール集合は予めすべて用意されることを想定しておらず，必要に応じて追加・変更されるものである．提案手法では，モデル変換ルール集合のパターンを定義したが，モデル変換ルール集合を追加・変更する場合には，該当のモデル変換ルール集合を適切なパターンに分類し，仮説のパターンと関連付ける必要がある．あるいは，モデル変換ルール集合や仮説のパターンも変化するものと考えられる．

また，モデル変換ルール間には，依存関係や順序関係も存在しうる．モデル変換ルールは，モデル上の要素の置き換えであるので，複数の変換ルールの適用順序を変更すると，得られるモデルが異なることが考えられる．したがって，複数のモデル変換ルールの間の関係も管理する必要がある．しかし，モデル変換ルール集合とそのパターン，モデル変換ルール間の関係の維持管理について，本研究では研究の対象としておらず，今後の課題である．

7.1.3 製品開発における適用性

1.3節に示したように，本論文で示したフォールトアナリシス手法は，製品知識を有する開発者と，提案したフォールトアナリシス手法の知識を有する開発支援者の共同作業によって実施することを想定している．実製品に対するフォールトアナリシスの適用においては，表6.2に示した2名の参加者によって実施した．一方が開発者，他方が開発支援者に相当する．したがって，実製品に対するフォールトアナリシスの適用結果は，想定する体制においてフォールトアナリシスが適用可能であることを示唆する．ただし，本論文で示した適用は，両者がモデル検査ならびにフォールトアナリシスについて知見を持っていた．実際の製品開発の場合には，必ずしも開発者らが充分な知見を持っているとは限らない．以下，製品開発における適用を想定し，必要とされる知見や前提，企業における製品開発で実践する際の課題について考察する．

対象システムに関する知見

本手法を実践するユーザは，テスト者から報告された故障状態を理解できる必要がある．また，報告された故障状態に対して，故障状態を特徴づける故障条件を想定できなければならない．さらに，故障の原因に対する仮説を設定し，対象システムを前提とした予測を導出しなければならない．これらを実施するためには，ユーザは対象システムの振舞いを理解している必要がある．一方，対象システムの振舞いを詳細かつ完全に理解しているならば，フォールトアナリシスをしなくても．故障の原因を推測できる可能性がある．しかし，現実の大規模かつ複雑なシステム開発において，システムの振舞いを完全に理解して

いることを期待することはできない．また，故障の原因が推測できたとしても，それが正しいかどうかの確認は必要であり，従来はそれをソフトウェアの再実行で行っていた．提案手法は，対象システムの概要を理解し故障の原因を想定した上で，その想定の正しさについて確信するための実験手段を提供するもの，あるいは充分に原因を想定できない場合にも実験を繰り返すことで，原因を絞り込む手段を提供するものと考えている．

産業ソフトウェアを用いたケーススタディでは，参加者のうちの 1 人が対象システムの開発に関わったことがある研究者という条件の下で行った．また，故障の原因について，おおよその見当が付いている状態で実施した．その場合においても，故障再現までに 8 回の試行錯誤を要した．

開発者が対象システムの振舞いについての知識をどの程度保有していれば，本手法を実践可能であるかについては，評価できていない．この点を明らかにすることで，製品開発部署において提案手法をより実践しやすくなると考える．

対象システムの構造

提案した抽象プログラムメタモデルの特徴の一つに，論理要素がある．論理要素は，プログラム言語で既定された言語要素のほかに，ソースコード中の特定の記述を意味のある要素として扱うものである．例として，`pthread_create()` 関数の呼び出しを，スレッド生成という意味のある言語要素に変換することを示した．これにより，C 言語の規格上は関数呼び出しの一つでしかない `pthread_create()` 関数について，関数内部を解析することなく，スレッド生成という意味を与え，Promela の `run` 文への変換を実現している．このような抽象化は，プログラム解析および変換を楽にすると共に，モデル検査における状態爆発の回避にも役立っている．

しかし，このような論理要素による抽象化を行うためには，論理要素として扱うコードのまとまりがソースコード上に存在し，それを特定するモデル変換ルールを記述できなければならない．そのため，ソースコードが意味のある単位で構造化されていることが期待される．また，そのような構造をユーザが理解している必要がある．

モデル検査に関する知見

提案手法では，POM/MC によって C 言語ソースコードからの Promela モデルの抽出，抽出した Promela モデルに対する SPIN を用いたモデル検査，モデル検査によって生成される反例の可視化までを自動化している．そのため，モデル検査を実施する点に限定すれば，モデル検査に関して多くの知識を要しない．しかし，モデル検査の課題の一つである状態爆発を回避するためには，どのようなモデルを抽出すれば状態数を削減できるかという一般的な知識を要する．また，提案手法では，故障状態をモデル検査の入力の一つである性質として定義するが，この定義には LTL 式か表明を用いることが必要である．これらを利用するためには，モデル検査に関する基本的な知識を有していることが必要である．

産業ソフトウェアでのケーススタディでは，二名の参加者の両方がモデル検査に関する知識を有していた．想定する体制においても，開発者がモデル検査の基本的な知見を有していることを想定はしているが，その理解度は個人差があるものと考えられる．想定するように，開発者と開発支援者の両方がモデル検査の知見を持っている必要があるのか，開

発支援者がモデル検査の知見を持っていれば，開発者がモデル検査の知見を持っていなくても実践は可能であるのか，といった点については検証できていない．一層の検討が必要である．

モデル変換に関する知見

提案手法では，仮説のパターン化と，モデル変換ルール集合のパターン化を行い，それらの対応関係を用意することで，モデル変換ルール集合の選択を支援する方法をとった．しかし，ありうる仮説のすべてに対して，モデル変換ルールをあらかじめ用意できるわけではない．仮説に対応して，既存のモデル変換ルール集合がない場合には，ユーザがモデル変換ルールを定義する必要がある．また，入力となるC言語のすべての要素について変換できるわけではないため，用いる言語要素によってモデル変換ルール集合を定義する場合もある．したがって，想定する体制としては，開発支援者が，QVTo を用いてモデル変換ルールを作成できることが期待している．

モデル変換ルールの記述については，モデル変換ルールの断片をコンポーネント化し，その組み合わせによってモデル変換ルールを定義できるようにするなど，プログラミングにおける部品化や再利用の技術を応用することで，より簡便にルールを記述をする支援ができると考えられる．しかし，そのようなモデル変換ルールの記述支援については，今後の課題である．

故障状態を示す性質の定義

また，モデル検査のための性質の定義のためにも，モデル変換に関する知識を要する．なぜならば，モデル検査はモデル変換によって抽出されたモデルに対して行うため，それに対応する性質も抽出されたモデルの要素を用いて表現するためである．想定では，これも開発支援者が担うことを期待している．

性質の定義については，入力言語である C 言語の要素を用いて記述した LTL 式を，モデル抽出に用いたモデル変換ルール集合を用いて，Promela の要素で記述した LTL 式に変換するという解決アプローチが考えられる．しかし，入力ソースコードと違い，LTL 式の定義に使われる C 言語の要素はコード断片であり，ソースコードと同様のプログラム解析を行うことができない．ソースコードのプログラム解析および変換の結果を用いて，LTL 式における C 言語のコード断片と抽出した Promela モデルの断片の対応を得ることができれば，C 言語の語彙で書かれた LTL 式を Promela の語彙で書かれた LTL 式に変換できる可能性がある．このような性質定義の支援方法については，今後の課題である．

7.2 ソフトウェア開発へのモデル検査の適用性に関する考察

モデル検査は，ソフトウェア開発を高度化する技術として長く注目されている．特に，モデル検査の網羅的な状態探索は，ソフトウェアに内在する未知の問題を摘出し，ソフトウェアの品質を保証する技術として期待される．一方，現実のソフトウェア開発では，モ

デル検査の利用はミッションクリティカルなソフトウェアなど，限定的であることも事実である．

モデル検査を実用する上での課題として，以下が考えられる．(1) 振る舞いモデルの作成，(2) 状態爆発の回避，(3) 性質の定義，(4) 結果の解析である．以下，それぞれの観点から，モデル検査の課題と提案手法について考察する．

7.2.1 振る舞いモデルの作成

実際の製品開発においては，仕様書は断片的であったり，人が理解することを目的としているために網羅性や厳密性に欠けていたりしており，モデル検査をはじめとする形式的な検証には利用できないことも多い．これは，仕様書を記述する技術が未成熟であったり，仕様書のメンテナンスが行われていなかったりすることが原因である場合もある．しかし，仕様書とは開発者の特定の視点に着目したモデルであるため，その視点とは異なる視点での検証は本来的に困難であるという問題もある．厳密な検証に耐えられる仕様書を記述することがコストがかかるだけでなく，詳細な仕様は，人が設計を理解するという仕様書の本来の目的を果たさない可能性もある．ソフトウェアに存在するあらゆる側面について，十分な仕様書を作成することは，現実的に不可能である．そのため，モデル検査をソフトウェア開発で利用する場合には，検証対象とするモデルをいかに作成するかが課題であった．

POM/MC では，検証対象モデルはソースコードから抽出されるため，検証のために振る舞いモデルを記述する必要はない．一方で，ソースコードからどのようなモデルを抽出するかについては，モデル変換ルール集合としてユーザが指定する必要がある．提案手法では，モデル検査の目的を故障再現というユースケースに絞ることで，故障のパターンとモデル変換ルール集合のパターンを対応づけることで，ルール選択を支援する方法を提案した．故障再現以外の目的でモデル検査を用いる場合にも，目的ごとにモデルの抽象化を行う方法をパターン化することが有効ではないかと考える．

7.2.2 状態爆発の回避

状態爆発は，モデル検査を利用する上での最も大きな障害の一つである．状態爆発を回避するために過度に抽象化を行うと，元のソフトウェアでは起こりえない振る舞いを見つけて，偽反例を得ることがある．逆に，本来摘出すべき特性が抽象化によって取り除かれ，見つけるべき振る舞いが見つけられない場合もある．このような，状態爆発のための抽象化と，目的を果たすための情報の保持のトレードオフが，モデル検査を利用する上での難しい作業となる．

本研究では，このトレードオフを解消するための試行錯誤は必要な作業であるとして，試行錯誤を効率的に行う手法を検討した．モデル変換ルールの構成を変更することによる試行錯誤プロセスを策定し，実製品システムにおいて，７回の試行錯誤でフォールトを特定するという結果を得た．状態爆発そのものは回避していないが，状態爆発を回避する作業を支援するという点で，実用的な提案であると考えている．

7.2.3 性質の定義

モデル検査を用いてシステムの正しさを検証しようとすると，正しいシステムの性質を漏れなく記述する必要がある．例えば，[40] では，ゴール指向要求分析の結果から，モデル検査のための性質を導く方法を提案した．しかし，ソフトウェアが満たすべき性質は仕様レベルの性質から実装レベルの節まで多岐にわたり，それらのレベルに合わせて性質を定義することは困難である．そのため，システムの正しさを検証する，または，未知の問題を摘出するためのモデル検査利用は難しさが残っている．

本提案手法では，故障再現に目的を絞ることで，検査性質は観測された故障を表現するという明確な基準を持っている．そのため，故障再現のためのモデル検査の利用は，未知の故障を見つけ出す検証目的でのモデル検査の利用に比べて簡便であり，故障再現はモデル検査のソフトウェア開発での有効な利用形態として期待される．

一方で，提案手法においても，予測の事後条件をどのように性質として定義するかは規定しておらず，実施者に依存している．予測から得られた条件を LTL 式や表明として定義する難しさのほか，ツールが自動抽出したモデルに合わせた表現で LTL 式を書く難しさも残っている．後者については，C 言語ソースコードに対応して記述した LTL 式を，ソースコードと同じモデル変換ルールでモデルに対応した式に変換する方法などが考えられるが，今後の課題である．

7.2.4 結果の解析性

モデル検査の結果は，性質の充足性と充足しない場合の反例として得られる．本提案では，反例から再現フォールトを見いだすため，人が反例を理解し解析できることが求められる．POM/MC では，反例トレースからシーケンス図を生成する機能を有することで，反例の理解性を向上した．また，詳細は述べていないが，反例の検索やフィルタリングの機能を有している．

また，付録 A.2 に示した Promela コードで示されるように，POM/MC で抽出する Promela モデルには，次のようなコメント文を挿入している．

```
/* @trace:file://queue_BUG.c- at line 12 */
```

これは，C 言語ソースコードのどの部分が該当する Promela コードに変換されたのかを示している．上記の例では，元のソースコードの名前が queue_BUG.c であり，その 12 行目が対応するコードであることを意味している．この情報をモデルに付加することにより，モデル検査の結果が Promela モデルで表現されるのに対して，人が理解可能な C 言語ソースコードへのトレーサビリティを確保できる．例えば，反例表示を C 言語のコード断片を用いて表現することが可能である．ただし，C 言語による反例表示は，現在のところツールとしては実装はしていない．

しかし，そのようなツールの機能だけでなく，予測に基づくモデル抽出も，反例の解析性向上に寄与していると考えられる．まず，モデル抽出によりモデルが縮退し，それに伴って反例も縮小し理解しやすくなる．キュープログラムのケーススタディでは，一度目の故障再現ではフォールトは特定できなかったが，中間的な異常な状態を発見し，その情報を元に仮説を加えて絞り込むことで，２度目にフォールトを特定した．これは，繰り返しを

前提とするプロセスを取ったことで，一度に反例のすべてを解析できなくても，プロセス全体で徐々に絞り込んでいくことができることを示している．また，予測がある状態でモデル検査を実施しているため，反例解析においても，予測に関連する部分から解析していくことができる．本提案では，直接的に反例の解析性を向上する手段は多くは提供していないが，フォールトアナリシスプロセスが反例解析を支援していると考えられる．

一方，8.2 節で示すように，モデル検査の反例解析を行う技術が提案されており，それらを提案手法と併用することは有効だと考えられるが，今後の課題である．

7.3 仮説演繹法の有効性に関する考察

本研究では，仮説演繹法に基づき，フォールトアナリシスのプロセスを定式化した．この定式化によって，開発者はこれまで経験的に行ってきたフォールとアナリシスをシステマティックに行うことができるようになり，さらにモデル抽出とモデル検査によりツール支援が可能となった．本研究で得られた知見の一つは，帰納的手法（故障で観測される事実と開発者の知識に基づく仮説設定）と，演繹的手法（仮説に基づく予測の導出と，モデル抽出とモデル検査による故障再現）との組み合わせが，現実的なフォールトアナリシスに適当であることを示したことにある。

仮説演繹法とは，科学哲学分野における，確証のための基本的なモデルである．既に述べたように仮説演繹法では，まず仮説を設定し，仮説から予測を導き出し，予測を実験で確認するというステップを繰り返す．仮説演繹法を現代的に解釈したのは Popper[41][42] や Hempel[26] が代表的であるが，Sprenger[43] は仮説演繹的確証 (H-D 確証; H-D confirmation) を次のように定義した

定義 21 背景知識 K に関して，観測される現象 *(Evidence)* E が，仮説 H を *H-D* 確証するのは，次の条件を満たすとき，かつそのときに限る：*(1)*$H.K$ が無矛盾　*(2)* $H.K$ から E が導出される *(3)*K 単独では E を導出できない．

開発者は，対象システムや，過去に起きた故障，ソフトウェア技術に関する知識を持っており，これが背景知識 K に相当する．K の下で設定されるフェイラーの原因に関する仮説が H に相当する．実験で得られた結果を E とする．Srenger の定義によれば，結果 E によって，仮説 H を確証するには，次の条件を満たす必要がある．(1)H と K 無矛盾，(2)H と K から E が導出される，(3)K 単独では E を導出できない．まず，仮説 H は開発者の前提知識 K に基づくものであるので，(1) は担保される．また，前提知識 K から実験結果 E が導かれるならば実験は不要であるので，(3) も担保される．本提案手法では，前提知識 K と H の下で予想 Pd を導き出し，これを実験で確認する．実験結果 E が予測 Pd と一致するとき，E が H と K から導かれるものと言える．つまり，(2) が成り立つ．このように，フォールトアナリシスのプロセスは仮説演繹法で説明することができ，実験により仮説を確証することが妥当であると言える．

7.4 開発者知識の活用に関する考察

従来の手作業によるモデルの作成では，モデルの記述方法や，モデルの抽象化方法は，モデルを作成する開発者の知見に依存していた．一方，POM/MC で用いるモデル変換ルー

ルは，そのような開発者の知見を形式知化したものだとみなせる．すなわち，従来は特定の開発者に内在していた知識を表出化することで，経験の少ない開発者でも故障再現のためのモデルを抽出することを可能にした．複数人で行うソフトウェア開発において，開発者の知識共有は重要な課題であり，変換ルール集合はモデル抽出を自動化するだけでなく，知識のデータベースとしての重要な役割も果たすと言える．現状の C2R マッピングは，単純な対応表である．より柔軟な関係を扱える構造を持たせることで，フォールトアナリシスに関する知見の表現がより豊かになるだろう．このような知識ベースを整備することで，経験が少ない開発者でも簡単にフォールトアナリシスが可能となることが期待される．

ツールによる作業の自動化は魅力的なキーワードである．一方で，開発者が知識を有しているならば，それを活用することもソフトウェア開発の高度化にとって有効な方法である．本研究では，モデルの抽出とモデル検査という，計算機による処理に適した部分を POM/MC として実現し，故障の特徴づけや再現した故障からのフォールトの推定など，人の知見と思考が有効な部分をプロセスとして規定した．この際，人に依存するプロセスに対しては，フォールトアナリシス自体の形式化や，仮説演繹法によるプロセスのモデル化を行った．ソフトウェア開発のような人が関与する事象に対して実践的な手法を確立するためには，本研究のように，計算機による自動化と人の知識の活用の双方に対して科学的なアプローチを取ることが重要であると考えている．

このように開発者の知見を蓄積する仕組みとしての提案手法を考えると，次のような点で検討すべき課題が残っている．まず，QVTo による変換ルールの記述は難易度が高い．QVTo に関する知識に加えて，POM/MC のメタモデルを理解する必要がある．また，QVTo を用いて記述された変換ルールは，可読性が低い．抽象プログラムモデルを用いることで，入出力言語の影響を抑える工夫はしているが，QVTo で手続き的に記述した変換ルールの理解は，通常のプログラムの理解と同様の困難さを伴う．これらを容易とするためには，通常のプログラム言語と同様に，変換ルールに対する構造化が求められる．

また，フォールトアナリシスの実施中には，対象となるシステムを前提とした抽象化を検討し，これを変換ルールとして記述する．しかし，変換ルールを知識として再利用するためには，変換ルールの汎化が必要である．例えば，変換ルールの適切な分割や，パラメタライズによる共通化である．実際，POM/MC の開発においては，変換ルールの再構成も行っており，言語変換およびフォールトアナリシスに向けたリファクタリングや，変換ルールを知識として管理する手法が求められる．

第8章　関連研究

8.1　モデル検査のためのモデル抽出

ソースコードからの Promela モデルの生成に関していくつかの提案がなされている．Modex (FeaVer)[28] は，特定のプラットフォームを前提として C 言語で書かれたソースコードから Promela モデルを生成する．Modex では，対象となるソースコードをターゲットコードと外部（すなわちプラットフォーム）のコードに分割する．ターゲットコードはあらかじめ定義された「変換マップ」に従って Promela コードに変換される．その後，変換された Promela コードと外部モデルとを結合する．変換マップは，シンタクスレベルのパターンのみを定義できる．この変換アプローチでは，変換マップを定義する際，言語間の対応と，モデルの抽象化という 2 つの観点を同時に考慮する必要がある．試行錯誤を要するフォールトアナリシスにおいては，この変換マップ定義の作業は，大きなコストがかかると共に，誤りが混入するリスクが高い．本提案では，抽象プログラムモデルを導入することで，言語間の対応のための変換と，モデル抽象化のための変換を分離した．また，抽象プログラムに論理要素を導入することで，抽象化のためのモデル変換ルールはプログラム言語よりも論理的なレベルで記述できる．

別の抽象化アプローチとしてコンパイラアプローチがある．Bandera[16] は Java 言語で書かれたソースコードからのモデル抽出を行い，Promela を含む多種の出力形式をサポートする．Bandera は，コンパイラの最適化に用いられる Jimple 中間表現を利用して，モデルを最適化する．Jimple の制御フロー，データフロー，プログラム依存関係を利用して，Bandera はプログラムスライシングとデータ抽象（すなわち同値分割）を行う．このアプローチでは，効率的で健全な抽象化ができるが，実験的フォールトアナリシスで行うような意図的かつ破壊的なプログラム構造の変更を定義することはできない．実験的フォールトアナリシスにおいても，Bandera で行っているような解析的な抽象化は有効であり，POM/MC 上で同様の抽象化を実現することは，POM/MC によるモデル検査をより効率的なものにするだろう．

Androutsopoulos らは，状態遷移モデルに対する抽象化手法を提案した [4]．彼らの手法では，特定の状態やイベントと関係のないモデル要素を自動的に特定し，モデルから削除する．POM/MC でもモデルの一部を削除するような抽象化を定義することができるが，抽象構文木上で定義できる単純なものに限られる．データフローなどの振る舞いを解析することで，より効果的な抽象化ができるだろう．

8.2 モデル検査

CBMC[15], BLAST[27], CPAChecker[9], ESBMC[17], LLBMC[38], SLAM[7] など，ソースコードを対象とした様々なモデル検査ツールが提案されている．これらのツールは主に，自動的に未知の故障を見つけ出す（すなわち望ましい振る舞いからの逸脱を発見する）ことを目的としている．本論文では，テスト等で発見された故障について，それを再現するためにモデル検査を用いており，目的が異なる．提案手法では，モデル検査器として SPIN，モデル記述に Promela を用いているが，それらに依存せず，他のモデル検査器やモデル記法を用いることができる．ただし，モデル記法として C 言語を用いる場合，言語仕様が複雑な C 言語上での抽象化を定義しなければならないため，変換ルール定義が煩雑になることが予想される．

CEGAR[14] は，モデル検査で生成される反例を利用して繰り返しモデルを自動的に構成していく手法である．本論文の提案手法では，開発者の知見に基づいて構成されたモデル変換ルールに基づいてモデルを抽出する．フォールトアナリシスでは，開発者が対象システムや故障に関する知見を持っており，それらを利用することで効果的なフォールトアナリシスを実現できると考えているが，過抽象化による偽反例への対処として，提案手法と CEGAR を組み合わせることは有効だろうと考えられる．

8.3 デバッグプロセス

荒木らは，デバッグプロセスのフレームワークを提案し [5]，Xu らがそれを再編した [47]．Zeller は，仮説をベースとした「科学的デバッグ」について，著書 [48] で説明している．これらのフレームワークは，デバッグプロセスを仮説とテストの観点から説明している．ただし，プロセス全体の枠組みが示されているのみであり，本研究で示したような詳細にフォールトアナリシスのプロセスの形式化がなされ，具体的な手順とツールまでを実現したことは過去に例がないと考える．

8.4 故障再現とフォールトローカライゼーション

観測された故障を再現する手法が提案されている．BugRedux[32] は，運用環境で発生した故障を社内でデバッグするアプローチである．運用環境での実行データを収集し，再実行することにより運用環境での故障を模擬する．バグレポートに基づく手法もある．BugLocator[49] は，バグを修正するためにそれに関係するファイルを特定するための情報検索ベースの手法である．この手法では，バグレポートとソースコードの文字列の類似性に基づき，全ファイルについてバグとの関連性をランキングする．

また，モデル検査やテストの結果に基づいて，故障の原因を推定するフォールトローカライゼーションの技術も提案されている [46]．アプローチの一つは，故障を起こす場合の実行系列と，故障を起こす場合の実行系列を比較し，故障を起こす場合に特徴的な実行を抽出することで，故障に関係するコード断片を見つけ出す方法である [6][36][35][45]．異なるアプローチとして，フォールトローカライゼーションを充足可能性問題に帰着するものがある．例えば BugAssist[33] では，有界モデル検査で得られる故障時の実行系列を充

足不能論理式として表現する．この論理式を満たす最大の節の集合を最大充足可能性問題 (MAX-SAT) として解く．この集合から故障の原因の候補を得る．

これらのフォールトローカライゼーションの技術は，実行系列からフォールトの候補を自動的に得ようとするアプローチであり，本論文で示した開発者の知識を活用して仮説と予測によってフォールトを絞り込んでいく手法とはアプローチが異なり，比較対象とはならない．ただし，提案手法では得られた故障から再現フォールトを導く方法については言及しておらず，これにフォールトローカライゼーションの手法を組み込むことは可能であると考える．

8.5 POM の応用

本研究では，POM に基づいたモデル検査環境 POM/MC を開発し，フォールトアナリシスに応用した．POM は，モデル検査に特化しない汎用的なフレームワークである．

POM/EQ[50] は，モデル変換によりリファクタリングが適切に行われたことを検証する技術である．リファクタリングパターンを特定しリファクタリングによるソースコードの変更を変換として定義することで，パターンを用いて行われたリファクタリング後のソースコードが，逆変換により元のソースコードと同型になることを確認することで，リファクタリングの妥当性を検証する．

POM/Viz[52] は，ソースコードの理解のための可視化技術である．ソースコードの構文解析に加えて，テストコードの解析と，ソフトウェアメトリクスの取得を行い，これらを用いてソースコードの注目点を推定することによって，抽象化したプログラム可視化を実現する．

第9章　まとめ

本論文では，プログラム実行では再現が困難な故障に対して，故障の原因となるフォールトを特定するフォールトアナリシスの手法を提案した．従来の故障再現では，故障発生時と同じ条件下でプログラムを実行するが，システムの振る舞いに非決定性や並行性のために，不具合が再現されない場合が多々あり，組込みソフトウェア開発の課題の一つであった．本提案では，フォールトアナリシスのための故障再現にモデル検査を用いる．モデル検査では，定義したモデルに対して与えられた性質を満たさない振る舞いを網羅的に探索するため，システム非決定性や並行性に影響されず，故障に至る振る舞いを反例として得られる．

筆者らは，仕様書やソースコードを参照し手作業でモデルを作成する方法で故障再現をモデル検査で行う取り組みを実践してきた．しかし，モデル検査における状態爆発を回避するためのモデル抽象化や，過抽象化による偽反例の回避など，モデル作成の試行錯誤に要するコストが問題となっていた．また，ソフトウェア開発で作成される仕様書は，モデル検査に用いるには精度が不十分という問題もあった．そこで，本研究ではまず，ソースコードからモデルを抽出する手法を検討した．特に，モデルの抽象化のための試行錯誤を効率的に行うために，ユーザが抽象化の方法を指定できる POM(Program-Oriented Modeling) フレームワークを考案した．さらに，POM に基づいて，C 言語のソースコードから Promela モデルを抽出する POM/MC ツールを開発した．POM/MC ツールでは，モデル抽出のほか，SPIN によるモデル検査および反例の可視化もサポートする．POM/MC ツールを実現するため，3 種のメタモデル (C 言語，抽象プログラム，Promela) と，モデル変換ルール集合を策定した．抽象プログラムモデルにより，抽象化方法を言語に依存しない変換ルールとして実現し，モデル作成の試行をツールで行うことができる．

さらに，故障再現を含むフォールトアナリシスを実践できるようにするため，POM/MC を用いたフォールトアナリシスプロセスの形式化を行った．そのために，まず，故障やフォールトの概念を整理し，フォールトアナリシスモデルを策定した．次に，科学哲学において確証に関する基本概念である仮説演繹法の観点からフォールトアナリシスを分析し，仮説に基づく予測と，予測に基づく実験という観点でフォールトアナリシスモデルを捉えなおし，実験的フォールトアナリシスプロセスを策定した．さらに，実験的フォールトアナリシスプロセスに POM/MC による故障再現を融合したプロセスを確立した．

提案手法を評価するために，ケーススタディを実施した．モデル検査ツールのコンペティションである SVCOMP のベンチマークプログラムのほか，座席予約システムを模擬したサンプルコードについて，POM/MC を用いて Promela モデル抽出とモデル検査が実現可能であることを確認した．また，SVCOMP ベンチマークの一つと，実製品である搬送システムにおいて発生した再現困難な故障について，POM/MC を用いたフォールトアナリシスを行い，フォールとアナリシスが可能であることを確認した．搬送システムの例では，

7 回の試行錯誤を経て，約 13,000 行の C 言語ソースコードから約 2,000 行の Promela モデルを生成し，フォールトを特定できた．本作業には，2 名で 7 人日を要した．従来の手作業でのモデルを作成した方法では同規模の問題について 20 人日を要しており，POM/MC を用いたフォールトアナリシスが効率的であることを確認した．

本研究では，ユーザが抽出方法を指定可能なソースコードからのモデル抽出手法 POM，POM を用いた C 言語モデル検査ツール POM/MC，POM/MC を用いた実験的フォールトアナリシスプロセスを提案した．本提案のために，フォールトアナリシスを仮説演繹法の観点から捉えなおし，フォールトアナリシスモデルを策定した．フォールトアナリシスについて綿密に形式化した研究はこれまでにないと考えている．また，従来は未知の問題を発見するために用いられてきたモデル検査について，既知の故障の分析にモデル検査を用いるユースケースを提案した．未知の問題の発見に比べ，既知の故障の分析は適用が容易であり，モデル検査のソフトウェア開発における実用化に寄与するものである．今後は，本提案を含めてモデル検査や形式手法の製品開発における利用を進め，ソフトウェア開発の高度化を進めていきたい．

謝辞

本研究を行うにあたり，終始ご指導を賜った北陸先端科学技術大学院大学の青木利晃准教授，日本工業大学の粂野文洋准教授に深謝いたします．また，本研究のベースとなるソフトウェア工学の知見をご教授いただき，博士課程入学のきっかけとなったトップエスイープログラムにつきまして，国立情報学研究所の本位田真一教授はじめ同プログラムの先生方，共に受講し議論し刺激をいただいた一期生・二期生ほか関係者各位に御礼申し上げます．

入社以来ご指導をいただいた小泉忍氏，島袋潤氏，共に研究を推進してきた研究ユニットメンバー，サイエンスへの探求心をかきたてて叱咤激励いただいた小島啓二氏ほか，博士課程へのご理解およびご支援いただいた株式会社日立製作所ならびにグループ会社の皆さんに御礼申し上げます．

最後に，研究活動に理解を示し協力してくれた妻と，気分転換のお手伝いをしてくれた２人の子どもたちに心より感謝します．

参考文献

[1] JIS X 0014:1999. 情報処理用語—信頼性，保守性及び可用性. 日本規格協会, 1997.

[2] ISO/IEC 2382-14:1997. *Information technology – Vocabulary – Part 14: Reliability, maintainability and availability.* ISO/IEC, 1997.

[3] IEEE Std 610.12-1990. *IEEE Std 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology.* IEEE, 1990.

[4] Kelly Androutsopoulos, David Binkley, David Clark, Nicolas Gold, Mark Harman, Kevin Lano, and Zheng Li. Model projection: simplifying models in response to restricting the environment. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, pp. 291–300. ACM, 2011.

[5] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. A general framework for debugging. *Software, IEEE*, Vol. 8, No. 3, pp. 14–20, 1991.

[6] Thomas Ball, Mayur Naik, and Sriram K Rajamani. From symptom to cause: localizing errors in counterexample traces. In *ACM SIGPLAN Notices*, Vol. 38, pp. 97–105. ACM, 2003.

[7] Thomas Ball and Sriram K Rajamani. The slam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, Vol. 37, pp. 1–3. ACM, 2002.

[8] Dirk Beyer. Competition on software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 504–524. Springer, 2012.

[9] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*, pp. 184–190. Springer, 2011.

[10] David Binkley. Source code analysis: A road map. In *Future of Software Engineering*, pp. 104–119. IEEE Computer Society, 2007.

[11] Inernational Software Qualifications Board: 翻訳 Japan Software Testing QualificationsBoard. テスト技術者資格制度 foundation level シラバス 日本語版. [Online]. Available: `http://www.jstqb.jp/dl/JSTQB-SyllabusFoundation_Version2011.J02.pdf`, 2011.

[12] M Busch, R Chaparadza, ZR Dai, A Hoffmann, L Lacmene, T Ngwangwen, GC Ndem, H Ogawa, D Serbanescu, I Schieferdecker, et al. Model transformers

for test generation from system models. In *Proc. of Int. Conf. on Quality Engineering in Software Technology (Conquest)*, 2006.

[13] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pp. 359–364. Springer, 2002.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pp. 154–169. Springer, 2000.

[15] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176. Springer, 2004.

[16] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Pasareanu, Hongjun Zheng, et al. Bandera: Extracting finite-state models from java source code. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, pp. 439–448. ACM, 2000.

[17] Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. Context-bounded model checking with esbmc 1.17. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 534–537. Springer, 2012.

[18] Patrice Godefroid and Nachiappan Nagappan. Concurrency at microsoft: An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[19] Peter Godfrey-Smith. *Theory and reality: An introduction to the philosophy of science*. University of Chicago Press, 2009.

[20] Richard C Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, 2009.

[21] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/.

[22] Object Management Group. Xml metadata interchange (xmi). [Online]. Available: `http://www.omg.org/spec/XMI/`, 2005.

[23] Object Management Group. Meta object facility (mof) core specification. [Online]. Available: `http://www.omg.org/spec/MOF/2.0/PDF`, 2006.

[24] Object Management Group. Mof model to text transformation language, v1.0. [Online]. Available: `http://www.omg.org/spec/MOFM2T/1.0/PDF`, 2008.

[25] Object Management Group. Omg unified modeling lanuage (omg uml). [Online]. Available: `http://www.omg.org/spec/UML/Current`, 2013.

[26] C. Hempel. A aspects of scientific explanation and other essays in the philosophy of science, 1965.

[27] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Model Checking Software*, pp. 235–239. Springer, 2003.

[28] Gerard J Holzmann and Margaret H Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. on Software Engineering*, Vol. 28, No. 4, pp. 364–377, 2002.

[29] Gerrard J. Holzmann. The spin model checker: Primer and reference manual. *ISBN: 0-321-22862-6*, 2004.

[30] Spin Homepage. Language reference. [Online]. Available: `http://spinroot.com/spin/Man/promela.html`.

[31] ISO/IEC/IEEE24765:2010. *ISO/IEC/IEEE24765:2010 Systems and software engineering - Vocabulary*. IEEE, 2010.

[32] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, pp. 474–484. IEEE Press, 2012.

[33] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, Vol. 46, No. 6, pp. 437–446, 2011.

[34] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pp. 1–2, 2008.

[35] Stefan Leue and Mitra Tabaei Befrouei. *Counterexample explanation by anomaly detection*. Springer, 2012.

[36] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *Software Engineering, IEEE Transactions on*, Vol. 32, No. 10, pp. 831–848, 2006.

[37] Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-driven architecture. In *Advances in Object-Oriented Information Systems*, pp. 290–297. Springer, 2002.

[38] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *Verified Software: Theories, Tools, Experiments*, pp. 146–161. Springer, 2012.

[39] Hideto Ogawa, Hajo Eichler, Huri Glickman, Ranganai Chaparadza, and Andreas Hoffmann. Mda based approach for generation of ttcn-3 test specifications. In *TTCN-3 Users Conference*, 2005.

[40] Hideto Ogawa, Fumihiro Kumeno, and Shinichi Honiden. Model checking process with goal oriented requirements analysis. In *Proc. of Asia-Pacific Software Engineering Conf. (APSEC)*, pp. 377–384. IEEE, 2008.

[41] Karl R Popper. The logic of scientific discovery, 1959.

[42] Karl R Popper. Conjectures and refutations: The growth of scientific knowledge, 1962.

[43] Jan Sprenger. A synthesis of hempelian and hypothetico-deductive confirmation. *Erkenntnis*, Vol. 78, No. 4, pp. 727–738, 2013.

[44] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[45] Cong Tian and Zhenhua Duan. Detecting spurious counterexamples efficiently in abstract model checking. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, pp. 202–211. IEEE Press, 2013.

[46] W Eric Wong and Vidroha Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45-09*, 2009.

[47] Shaochun Xu and Vaclav Rajlich. Cognitive process during program debugging. In *Proc. of IEEE Int. Conf. on Cognitive Informatics*, pp. 176–182. IEEE, 2004.

[48] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[49] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, pp. 14–24. IEEE, 2012.

[50] 市井誠, 小川秀人. モデル変換を用いたリファクタリング検証手法. ソフトウェア工学の基礎研究会 (FOSE2014). ソフトウェア科学会, 2014.

[51] 独立行政法人情報処理推進機構. 2012年度「ソフトウェア産業の実態把握に関する調査」調査報告書. [Online]. Available: `https://www.ipa.go.jp/files/000026799.pdf`, 2013.

[52] 福井大輔, 是木玄太, 小川秀人. テストコード解析とソフトウェアメトリクスを用いたプログラム抽象化・可視化技術の研究. 電子情報通信学大会講演論文集, p. 24, 2013.

パブリケーションリスト

本研究に関連する著作

主筆論文

- Ogawa, H., Ichii, M., Myojin, M.,Chikahisa, M., Nakagawa, Y. A Model-checking Approach for Fault Analysis based on Configurable Model Extraction, Vol.E98-D, No.6, Jun, 2015, pp. 1150-1160.

主筆査読付国際会議

- Ogawa, H., Kumeno, F., Honiden, S. (2008, December). Model checking process with goal oriented requirements analysis. In 15th IEEE Asia-Pacific Software Engineering Conference(APSEC'08), 2008, pp. 377-384.
- Ogawa, H., Ichii, M., Kumeno, F., Aoki, T. (2013, December). A Practical Study of Debugging Using Model Checking. In 20th IEEE Software Engineering Conference (APSEC'13), 2013, pp. 134-139.
- Ogawa, H., Ichii, M., Kumeno, F., Aoki, T. (2015, July). Experimental Fault Analysis Process implemented using Model Extraction and Model Checking. In IEEE Computer Software and Applications Conference (COMPSAC2015), 2015, pp. 95-104.

主筆査読付国内会議

- 小川秀人, 市井誠, 粂野文洋, 青木利晃. POM/MC を用いた仮説ベースモデル検査デバッグ手法, ソフトウェア工学の基礎 XX (FOSE2013), (pp.137-142). ソフトウェア科学会.

共著査読付国際会議

- Ichii, M., Myojin, T., Nakagawa, Y., Chikahisa, M., Hideto, O. (2012, October). A Rule-based Automated Approach for Extracting Models from Source Code. In 19th IEEE Working Conference on Reverse Engineering (WCRE), 2012, pp. 308-317.

共著国内会議は省略する.

そのほかの著作

主筆解説論文

- 小川秀人, 企業におけるソフトウェア開発とソフトウェア工学, コンピュータソフトウェア, 28(3), ソフトウェア科学会, pp.2-11.

主筆国内会議

- 小川秀人, ソフトウェアテストプロセスに関する一考察　-V⇒W⇒V 3 −, ソフトウェアテストシンポジウム (JaSST 07), 2007.
- 小川秀人,　設計検証におけるモデル検査適用に関する一考察,　情報処理学会ウィンターワークショップ 2010・イン・倉敷 論文集,　2010.
- 小川秀人, 市井誠, 新原敦介, 鈴木康文, Phan Thi Thanh Huyen, 坂井田真也, 形式手法のソフトウェア開発利用の課題と事例, 情報処理学会ウィンターワークショップ 2015・イン・宜野湾　論文集, 2015, pp.43-44.

共著査読付国際会議

- Busch, M., Chaparadza, R., Dai, Z. R., Hoffmann, A., Lacmene, L., Ngwangwen, T., Ndem, G.C., Ogawa, H., Serbanescu, D., Schieferdecker, I. and Zander-Nwicka, J., Model transformers for test generation from system models. In Proceedings of 10th International Conference on Quality Engineering in Software Technology (Conquest 2006).
- Ogawa, H.,Eichler, H., Glickman, H., Chaparadza, R. and Hoffmann, A., MDA based approach for generation of TTCN-3 test specifications, TTCN-3 Users Conference, 2005.

付 録A　コード例

A.1　C言語ソースコード

例として，キュー問題のソースコードを示す．本コードはSCVOMP2012[8] のベンチマークプログラムの１つであるが，実行可能なプログラムとするため以下の点を改変してある．

- 関数 *nondet_int*() の実装を加えた
- 実体のない ERROR ラベルへの処理に関数 exit() の呼び出しを加えた

```
#include <pthread.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

#define SIZE    (20)
#define EMPTY    (-1)
#define FULL    (-2)
#define FALSE    (0)
#define TRUE    (1)

typedef struct {
    int element[SIZE];
    int head;
    int tail;
    int amount;
} QType;

pthread_mutex_t m;

int nondet_int()
{
    int r;

    r=rand();
    return r ;
}

int stored_elements[SIZE];
_Bool enqueue_flag, dequeue_flag;
QType queue;

int init(QType *q)
{
  q->head=0;
  q->tail=0
  q->amount=0;
}

int empty(QType * q)
{
  if (q->head == q->tail)
  {
    printf("queue is empty\n");
    return EMPTY;
  }
  else
    return 0;
}

int full(QType * q)
{
  if (q->amount == SIZE)
  {
    printf("queue is full\n");
    return FULL;
  }
  else
    return 0;
}

int enqueue(QType *q, int x)
{
  q->element[q->tail] = x;
  q->amount++;
  if (q->tail == SIZE)
  {
    q->tail = 1;
  }
  else
  {
    q->tail++;
  }

  return 0;
}

int dequeue(QType *q)
{
  int x;

  x = q->element[q->head];
```

```
  q->amount--;
  if (q->head == SIZE)
  {
    q->head = 1;
  }
  else
    q->head++;

  return x;
}

void *t1(void *arg)
{
  int value, i;

  pthread_mutex_lock(&m);
  value = nondet_int();
  if (enqueue(&queue,value)) {
      printf("t1_1\n");
      goto ERROR;
  }

  stored_elements[0]=value;
  if (empty(&queue)) {
      printf("t1_2\n");
      goto ERROR;
  }

  pthread_mutex_unlock(&m);

  for(i=0; i<(SIZE-1); i++)
  {
    pthread_mutex_lock(&m);
    printf("t1 ");
    if (enqueue_flag)
    {
      value = nondet_int();
      enqueue(&queue,value);
      stored_elements[i+1]=value;
      enqueue_flag=FALSE;
      dequeue_flag=TRUE;
      printf("%d\n",value);
    }
    pthread_mutex_unlock(&m);
  }

  return NULL;

  ERROR:
  exit(1);
}

void *t2(void *arg)
{
  int i;

  for(i=0; i<SIZE; i++)
  {
    pthread_mutex_lock(&m);
    printf("t2 ");
    if (dequeue_flag)
    {
      if (!dequeue(&queue)==stored_elements[i]) {
          printf("t2_1 %d %d\n",i,stored_elements[i]);
          goto ERROR;

          ERROR:
          exit(1);
      }
      dequeue_flag=FALSE;
      enqueue_flag=TRUE;
      printf("%d %d\n",i, stored_elements[i]);
    }
    pthread_mutex_unlock(&m);
  }

  return NULL;
}

int main(void)
{
  pthread_t id1, id2;

  while(TRUE){
  enqueue_flag=TRUE;
  dequeue_flag=FALSE;
  srand((unsigned)time(NULL));

  init(&queue);

      printf("start\n");
  if (!empty(&queue)==EMPTY) {
      printf("main\n");

      goto ERROR;
      ERROR:
      exit(1);
  }


  pthread_mutex_init(&m, 0);

  pthread_create(&id1, NULL, t1, &queue);
  pthread_create(&id2, NULL, t2, &queue);

  pthread_join(id1, NULL);
  pthread_join(id2, NULL);
  }
  return 0;
}
```

A.2 抽出したPromelaコード

```
/* queue_BUG */
```

```
chan active_pid_queue_ = [6] of {pid};
inline mutex_lock_ ( mutex_ ) {
    if
    :: atomic {
        (mutex_==false)->
        mutex_ = true
    }

    fi
}

inline mutex_unlock_ ( mutex_ ) {
    mutex_ = false
}

inline proc_join_ ( proc_id_ ) {
    !(active_pid_queue_??[proc_id_])
}

inline rand_ ( max_, rand_val_ ) {
     atomic {
        rand_val_ = 0;
        do
        ::(rand_val_<(max_-1))->
            if
            ::true->
                rand_val_++;
            ::true->
                break

            fi;
        ::else->
            break

        od
    }
}

chan proc_index_queue_ = [6] of {byte};
inline assign_proc_index_ ( index_ ) {
    proc_index_queue_?index_
}

inline release_proc_index_ ( index_ ) {
    proc_index_queue_!!index_
}


/* @trace:file://queue_BUG.c- at line 12 */
typedef unnamed_stuct_t_0 {
    int element_[20];
    int head_;
    int tail_;
    int amount_
};


/* @trace:file://queue_BUG.c- at line 19 */
bool m_;

/* @trace:file://queue_BUG.c- at line 20 */
inline nondet_int_ ( ret__ ) {
    rand_(10,ret__)
}
```

```
/* @trace:file://queue_BUG.c- at line 21 */
int stored_elements_[20];

/* @trace:file://queue_BUG.c- at line 22 */
bit enqueue_flag_;

/* @trace:file://queue_BUG.c- at line 22 */
bit dequeue_flag_;

/* @trace:file://queue_BUG.c- at line 23 */
unnamed_stuct_t_0 queue_;

/* @trace:file://queue_BUG.c- at line 25 */
inline init_ ( q_, ret__ ) {
    int init_proc_index_;

    /* @trace:file://queue_BUG.c- at line 27 */
    q_.head_ = 0;

    /* @trace:file://queue_BUG.c- at line 28 */
    q_.tail_ = 0;

    /* @trace:file://queue_BUG.c- at line 29 */
    q_.amount_ = 0
}


/* @trace:file://queue_BUG.c- at line 32 */
inline empty_ ( q_, ret__ ) {
    int empty_proc_index_;

    /* @trace:file://queue_BUG.c- at line 34 */
    if
    ::(q_.head_==q_.tail_)->

        /* @trace:file://queue_BUG.c- at line 36 */
        skip;
        ret__ = -1;
        goto End_of_empty;
    ::else->
        ret__ = 0;
        goto End_of_empty

    fi;
    End_of_empty:    skip
}


/* @trace:file://queue_BUG.c- at line 54 */
inline enqueue_ ( q_, x__, ret__ ) {
    int enqueue_proc_index_;

    /* @trace:file://queue_BUG.c- at line 54 */
    int x__54;
    x__54 = x__;

    /* @trace:file://queue_BUG.c- at line 56 */
    q_.element_[q_.tail_] = x__54;

    /* @trace:file://queue_BUG.c- at line 57 */
    q_.amount_++;

    /* @trace:file://queue_BUG.c- at line 58 */
    if
    ::(q_.tail_==20)->
```

```
        /* @trace:file://queue_BUG.c- at line 60 */
        q_.tail_ = 1;
    ::else->

        /* @trace:file://queue_BUG.c- at line 64 */
        q_.tail_++

    fi;
    ret__ = 0
}


/* @trace:file://queue_BUG.c- at line 70 */
inline dequeue_ ( q_, ret__ ) {
    int dequeue_proc_index_;

    /* @trace:file://queue_BUG.c- at line 72 */
    int x__72;

    /* @trace:file://queue_BUG.c- at line 74 */
    x__72 = q_.element_[q_.head_];

    /* @trace:file://queue_BUG.c- at line 75 */
    q_.amount_++;

    /* @trace:file://queue_BUG.c- at line 76 */
    if
    ::(q_.head_==20)->

        /* @trace:file://queue_BUG.c- at line 78 */
        q_.head_ = 1;
    ::else->
        q_.head_++

    fi;
    ret__ = x__72
}

proctype t1_() {
    active_pid_queue_!_pid;

    /* @trace:file://queue_BUG.c- at line 88 */
    int value_;

    /* @trace:file://queue_BUG.c- at line 88 */
    int i_;

    /* @trace:file://queue_BUG.c- at line 90 */
    mutex_lock_(m_);

    /* @trace:file://queue_BUG.c- at line 91 */
    nondet_int_(value_);

    /* @trace:file://queue_BUG.c- at line 92 */
    bit ret_enqueue_6_;

    /* @trace:file://queue_BUG.c- at line 92 */
    enqueue_(queue_,value_,ret_enqueue_6_);

    /* @trace:file://queue_BUG.c- at line 92 */
    if
    ::(ret_enqueue_6_!=0)->

        /* @trace:file://queue_BUG.c- at line 93 */
        assert(false);
    ::else->
```

```
    skip

fi;

/* @trace:file://queue_BUG.c- at line 96 */
stored_elements_[0] = value_;

/* @trace:file://queue_BUG.c- at line 97 */
int ret_empty_2_;

/* @trace:file://queue_BUG.c- at line 97 */
empty_(queue_,ret_empty_2_);

/* @trace:file://queue_BUG.c- at line 97 */
if
::(ret_empty_2_!=0)->

    /* @trace:file://queue_BUG.c- at line 98 */
    assert(false);
::else->
    skip

fi;

/* @trace:file://queue_BUG.c- at line 101 */
mutex_unlock_(m_);

/* @trace:file://queue_BUG.c- at line 103 */
i_ = 0;
do
::(i_<(20-1))->

    /* @trace:file://queue_BUG.c- at line 105 */
    mutex_lock_(m_);

    /* @trace:file://queue_BUG.c- at line 106 */
    if
    ::enqueue_flag_->

        /* @trace:file://queue_BUG.c- at line 108 */
        nondet_int_(value_);

        /* @trace:file://queue_BUG.c- at line 109 */
        enqueue_(queue_,value_,_);

        /* @trace:file://queue_BUG.c- at line 110 */
        stored_elements_[(i_+1)] = value_;

        /* @trace:file://queue_BUG.c- at line 111 */
        enqueue_flag_ = 0;

        /* @trace:file://queue_BUG.c- at line 112 */
        dequeue_flag_ = 1;
    ::else->
        skip

    fi;

    /* @trace:file://queue_BUG.c- at line 114 */
    mutex_unlock_(m_);
    i_++;
::else->
    break

od;
```

```
    /* @trace:file://queue_BUG.c- at line 117 */
    goto End_of_t1;

    /* @trace:file://queue_BUG.c- at line 120 */
    skip;
    active_pid_queue_??<_pid>;
    End_of_t1:    skip
};
proctype t2_() {
    active_pid_queue_!_pid;

    /* @trace:file://queue_BUG.c- at line 125 */
    int i_;

    /* @trace:file://queue_BUG.c- at line 127 */
    i_ = 0;
    do
    ::(i_<20)->

        /* @trace:file://queue_BUG.c- at line 129 */
        mutex_lock_(m_);

        /* @trace:file://queue_BUG.c- at line 130 */
        if
        ::dequeue_flag_->

            /* @trace:file://queue_BUG.c- at line 132 */
            int ret_dequeue_5_;

            /* @trace:file://queue_BUG.c- at line 132 */
            dequeue_(queue_,ret_dequeue_5_);

            /* @trace:file://queue_BUG.c- at line 132 */
            if
            ::(!ret_dequeue_5_==stored_elements_[i_])->

                /* @trace:file://queue_BUG.c- at line 133 */
                assert(false);

                /* @trace:file://queue_BUG.c- at line 135 */
                skip;
            ::else->
                skip

            fi;

            /* @trace:file://queue_BUG.c- at line 137 */
            dequeue_flag_ = 0;

            /* @trace:file://queue_BUG.c- at line 138 */
            enqueue_flag_ = 1;
        ::else->
            skip

        fi;

        /* @trace:file://queue_BUG.c- at line 140 */
        mutex_unlock_(m_);
        i_++;
    ::else->
        break

    od;

    /* @trace:file://queue_BUG.c- at line 143 */
    goto End_of_t2;
```

```
    active_pid_queue_??<_pid>;
    End_of_t2:    skip
};

/* @trace:file://queue_BUG.c- at line 146 */
init {

    /* @trace:file://queue_BUG.c- at line 148 */
    pid id1_;

    /* @trace:file://queue_BUG.c- at line 148 */
    pid id2_;

    /* @trace:file://queue_BUG.c- at line 150 */
    enqueue_flag_ = 1;

    /* @trace:file://queue_BUG.c- at line 151 */
    dequeue_flag_ = 0;

    /* @trace:file://queue_BUG.c- at line 153 */
    init_(queue_,_);

    /* @trace:file://queue_BUG.c- at line 155 */
    int ret_empty_1_;

    /* @trace:file://queue_BUG.c- at line 155 */
    empty_(queue_,ret_empty_1_);

    /* @trace:file://queue_BUG.c- at line 155 */
    if
    ::(!ret_empty_1_==-1)->

        /* @trace:file://queue_BUG.c- at line 156 */
        assert(false);
    ::else->
        skip

    fi;

    /* @trace:file://queue_BUG.c- at line 162 */
    skip;

    /* @trace:file://queue_BUG.c- at line 164 */
    id1_ = run t1_();

    /* @trace:file://queue_BUG.c- at line 165 */
    id2_ = run t2_();

    /* @trace:file://queue_BUG.c- at line 167 */
    proc_join_(id1_);

    /* @trace:file://queue_BUG.c- at line 168 */
    proc_join_(id2_);
    bit ret_main_;
    ret_main_ = 0;
    goto End_of_main;
    End_of_main:    skip
};
```

付 録B　モデル変換ルールの例

B.1　実装モデルから抽象プログラムモデル

例として，C 言語の for 文を抽象プログラムモデルのイテレーションに変換するルールを示す.

```
mapping CLanguage::ForStatement::toIterationStatement() : AbstractProgram::CompoundStatement
inherits CLanguage::CodeElement::toCodeElement
{
    items := Sequence{
            self.initializeExpression.map toExpressionStatement(),
            object AbstractProgram::IterationStatement {
                body := object AbstractProgram::CompoundStatement {
                    items := self.body.map toStatement()
                };
                condition := self.condition.map toExpression();
                continueExpression := self.postIterationExpression.map toExpression()
            }
        }->collect(i|i.oclAsType(AbstractProgram::BlockItem));
}
```

B.2　抽象プログラムモデルから対象モデル

例として，抽象プログラムモデルの文を Promela の文 n に変換するルールを示す．特に，抽象プログラム上のイテレーションから do 文への変換に関する部分を抜粋する.

```
mapping AbstractProgram::Statement::toStatement() : Promela::Statement {
    init {
        if (self.isEmpty()) then {
            result := object SkipStatement{};
        } else switch {

            case (self.oclIsKindOf(AbstractProgram::IterationStatement)) {
                if (self.hasContinueStatement() and self.oclAsType
                   (AbstractProgram::IterationStatement).continueExpression->isEmpty())
                then {
                    result := object Promela::CompoundStatement {
                        steps := object Promela::LabeledStatement {
                            name := 'Jump_of_loop_' + self.fromLine.toString();
                            statement := object Promela::SkipStatement {};
                        }.oclAsType(Promela::Statement)->asOrderedSet()
                        ->append(self.oclAsType(AbstractProgram::IterationStatement).map
                         toDoStatement());
                    }
                } else {
                    result := self.oclAsType(AbstractProgram::IterationStatement).map
                     toDoStatement();
                } endif;
```

```
            }

            -- IterationStatement 以外のステートメントは論文上は省略

        }

    }
}

mapping AbstractProgram::IterationStatement::toDoStatement() : Promela::DoStatement {
    var hasContinue:Boolean = self.hasContinueStatement();
    var labelStatement:Promela::LabeledStatement;
    if (hasContinue) then {
        labelStatement := object Promela::LabeledStatement {
            name := 'Jump_of_loop_' + self.fromLine.toString();
            statement := object Promela::SkipStatement {};
        };
    } endif;
    options += object Promela::Option {

        if (self.continueExpression != null) then {
            if (hasContinue) then {
                steps := self.condition.map toStatement()->asOrderedSet()
                        ->append(self.body.map toStatement())->append(labelStatement)
                        ->append(self.continueExpression.map toStatement())
            } else {
                steps := self.condition.map toStatement()->asOrderedSet()
                        ->append(self.body.map toStatement())
                        ->append(self.continueExpression.map toStatement())
            } endif;
        } else {
            steps := self.condition.map toStatement()->asOrderedSet()
                    ->append(self.body.map toStatement())
        } endif;
    }->asOrderedSet()
    ->append(object Promela::Option{
        steps := object Promela::ElseStatement{}->oclAsType(Promela::Statement)
        ->asOrderedSet()->append(object Promela::BreakStatement{});
    });
    if (hasContinue) then {
        result.assignGotoDestination(labelStatement);
    } endif;
}
```

B.3 抽象プログラムモデル上での変換

例として，pthread ライブラリによるスレッド生成に対する変換を示す.

```
transformation Pthread(inout apg: AbstractProgram);

main() {
    enableTraceLog:=true;
    builtinFunctionMap.subAttributes +=
     object SimpleAttribute{name:='pthread_mutex_lock'; value=''};
    builtinFunctionMap.subAttributes +=
     object SimpleAttribute{name:='pthread_mutex_unlock'; value:=''};
    builtinFunctionMap.subAttributes +=
     object SimpleAttribute{name:='pthread_create'; value:=''};
    builtinFunctionMap.subAttributes +=
     object SimpleAttribute{name:='pthread_join'; value:=''};
```

```
    assert (builtinFunctionMap.subAttributes->size()=4);
    assert (builtinFunctionMap.subAttributes->first().name = 'pthread_mutex_lock');
    emptyStubFunctions += 'pthread_init';
    emptyStubFunctions += 'pthread_mutex_init';

    apg.objectsOfType(Module)-> map detectStub();
    apg.objectsOfType(FunctionReference)->map resolveDeclaration();

    apg.objectsOfType(VariableDeclaration)->map detectPthreadTypeSpecifier();
    apg.objectsOfType(CompositeTypeSpecifier)->map detectPthreadTypeSpecifier();
    apg.objectsOfType(FunctionDeclaration)->map detectPthreadTypeSpecifier();
    apg.objectsOfType(CastExpression)->map detectPthreadTypeSpecifier();
    apg.objectsOfType(FieldAccess)->map detectPthreadTypeSpecifier();

    apg.objectsOfType(FunctionCall)->map detectPthreadFunctionReference();

    apg.objectsOfType(FunctionCall)->map removeStatelessStubFunctionCallArguments();

    apg.objectsOfType(TypeDefTypeSpecifier)->select(s|s.declaration.name=
     'pthread_mutex_t' or s.declaration.name='pthread_t')->forEach(t) {
        trace("XXX:" , t.container().oclAsType(CodeElement));
    };
}

mapping inout FunctionCall::detectPthreadFunctionReference() {
    population {
        self.getIdentifier().reference := self.getReference().map
         convertToPthreadFunctionReference(self);
    }
    end {
        if (self.getReference().oclIsKindOf(ProcessRunningFunctionReference)) then {
            var oldTarget: FunctionDeclaration :=
             self.arguments->
               at(3).getCastedExpression().oclAsType(Identifier).reference.
               oclAsType(FunctionReference).declaration.oclAsType(FunctionDeclaration);
            apg.objectsOfType(Module)->map detectProcessFunction(oldTarget);
            self.getReference().oclAsType(ProcessRunningFunctionReference).target
                := oldTarget.resolveone(ProcessFunctionDefinition);
            apg.objectsOfType(FunctionReference)->map reassignDeclaration(oldTarget);
            if (self.arguments->at(1).getCastedExpression().oclIsKindOf(AddressReference))
             then {
                self.getReference().oclAsType(ProcessRunningFunctionReference).processHandler
                    := self.arguments->
                      at(1).getCastedExpression().oclAsType(AddressReference).expression
                      .getCastedExpression();
            } else {
                self.getReference().oclAsType(ProcessRunningFunctionReference).processHandler
                    := self.arguments->at(1).getCastedExpression();
            } endif;
            if (self.arguments->at(4).getCastedExpression().oclIsKindOf(AddressReference))
             then {
                self.getReference().oclAsType(ProcessRunningFunctionReference).arguments
                    := Sequence{self.arguments->
                      at(4).getCastedExpression().oclAsType(AddressReference).expression
                      .getCastedExpression()};
            } else {
                self.getReference().oclAsType(ProcessRunningFunctionReference).arguments
                    := Sequence{self.arguments->at(4).getCastedExpression()};
            } endif;
        } endif;
    }
}
```