博士学位論文

ソフトウェアの品質向上のための開発現場における モデル検査技術を用いたソースコード検証の研究

A Study on Source Code Verification Using Model Checking Technique for Improvement of Software Quality on Development Site

芝浦工業大学大学院　工学研究科

博士（後期）課程　機能制御システム専攻

青木　善貴

（指導教員　松浦　佐江子）

平成２６年　９月

松浦　佐江子　教授（博士　　（情報科学））

（主査）

長谷川　浩志　教授（博士　　（工学））

（副査）

新津　善弘　教授（博士　　（工学））

野田　夏子　准教授（博士　　（情報科学））

櫨山　淳雄　教授（博士　　（工学））

目次

図目次

表目次

要旨

ソフトウェアの品質向上のための開発現場における
モデル検査技術を用いたソースコード検証の研究

青木　善貴

開発現場では不十分な要件定義やシステムの実装時のミス・誤解等により大小多くの不具合が発生する．これらの不具合を解消するために多くの工数が費やされ，プロジェクトの進行が妨げられる．解消するために特に多大な工数を要する不具合が発生する．これは要件定義段階のような上流工程で不具合の原因が発生する場合と実装段階のような下流工程で不具合の原因が発生する場合がある．仕様の誤解や考慮不足等を上流工程において早期に発見することも企業にとって重要である．不十分な要件定義に対処するために，上流工程においても，客観的指標による早期の検証を行うことが必要である．

ある程度プロジェクトが進行した下流工程で発生する手戻りは，作業を再度やり直すことになるため工数の増大とスケジュールの圧迫を招く．大きな手戻りを発生させる不具合の原因は，システムで実現した機能と要求仕様に記述した機能との間に，小規模な修正では解消できない機能差が生じていることである．これは仕様に対するプログラムの整合性の検証が不十分な場合に発生すると考えられる．従ってこれを防ぐためには，要求仕様に対するプログラムの整合性を早期にもれなく検証する仕組みが必要である．

また，不具合の解消にはまず原因を特定して再現確認することが前提であるが，原因の特定が困難な不具合は再現確認できないため調査工数がかかる．通常不具合の解消にかけられる時間には制約があるため，調査が不十分なまま修正を行ってしまうと，完全な修正ではないため不具合の再発を招き再度調査・修正の繰り返しになり多大な工数がかかることになる．不具合に対応する開発者が，不具合の現象からその原因を想定できない場合，このような原因特定が困難な状態になる．従ってこのような状態は単純なコーディングのミス等ではなく，複雑な条件の組合せや同期・タイミングの問題で発生することがほとんどであり，仕様に記述された業務機能の振舞いと実際に作成されたプログラムにより実現された業務機能の振舞いの間で整合性が取れないことを示している．このような不一致の発見は，通常は単体・結合テストにより行うがテストケースが不十分な場合は見逃されてしまうため，もれなく業務機能の振舞いの整合性を検証する仕組みが重要である．

これらの不具合は成果物の仕様についての妥当性・整合性の検査がもれなくできれば発

見できると考える．近年，システム開発の上流工程においてシステムが取りうる様々な状態を満たすかを検査するモデル検査が注目されている．モデル検査は状態遷移系として定義されたシステム（検査モデル）に対し，要求する性質を論理式で記述し，状態遷移系がこの論理式を満たすことを検査する手法であり，システムがもたらす機能の振舞いを検査することに適した手法である．

開発現場で用いられているモデル検査は多くの場合，大規模な検査についてモデル検査の専門的な知識を持つ検査者が独自のやり方で対応している．また実施する検査はシステムをモデル化した検査モデルに対して到達可能性・デッドロックの検査を行うものがほとんどである．そして現状では，開発現場においてモデル検査は広く普及しているとはいえない状態である．その理由は，まずモデル検査の専門的な知識を持たない検査者がシステムの振る舞いをモデル化し，かつ検査要件を検査式として記述して検査を行うことが困難であること，さらに検査結果として出力される反例からその意味を読み解く知識が必要になるからである．

本研究は下流工程においてソースコードを検証することを主な目的としている．本研究ではJavaのソースコードに対して開発するシステムがもたらす業務機能の振舞いをモデル化することにより「システムがある条件下で正当でない状態に陥ることは決して起こらない」もしくは「特定の状態へ到達することができる」といった性質を検査できるソースコード検証手法を提案する．

提案手法では仕様形式化する手法としてデシジョンテーブルのように開発現場の開発者でも適切かつ容易に定義できる形式を利用して，システムの取りうるべき状態の検査モデル(本研究ではこれを仕様モデルと呼ぶ)に変換し，それをソースコードの制御フローに基づき作成した検査モデル（本研究ではこれをソースコードモデルと呼ぶ）と結合して検査する．先にあげた不具合はシステムをそのままモデル化して到達可能性・デッドロックを検査しても，システムがとりうるべき振舞いが定義されていないため仕様の妥当性・整合性の判断はできないが，本研究の提案手法ならば仕様とソースコード間の業務機能の振舞いの不一致が発見され，妥当性・整合性が確認できる．この作業を検査者が適切かつ容易に実施できるように検査には支援ツールを用いる．

またモデル検査は元々上流工程で仕様検証することを目的としたものである．提案手法をソースコードの代わりにUMLモデルへ適用すれば,要件定義の段階において仕様として記述されているシステムがもたらす業務機能の振舞いが想定外にならないことの検証ができると考えられるため，UMLモデルへの適用についても検討する．これをUMLモデル検証手法として提案する．

本研究の成果としては，モデル検査の専門的な知識を持たない開発者でも容易にモデル検査技術を用いて，不具合の原因の特定を安定した工数で行えるようにすることによりシステム開発の開発現場で広くプロジェクトの品質や作業効率の向上が望める．また，経験を重視する傾向が強い開発現場においてソフトウェア工学の考え方を導入することにより

経験的なアプローチだけではなく，工学的なアプローチが行われる土壌が養われることが期待される．

Abstract

A Study on Source Code Verification Using Model Checking Technique for Improvement of Software Quality on Development Site

Yoshitaka Aoki

Software programs often include many defects that are not easy to detect because of the developers' mistakes , misunderstandings caused by the inadequate definition of requirements, and the complexity of the implementation.

When failures are found after the service has been deployed, a significant number of person-hours are often required to detect the defects and the causes.

There are two kinds of failure that many man-hours are required to solve. One is a failure that occurs rework. Another is the failure that a specific of cause is difficult. It is required of a method that can determine the cause of the failure in stable man-hour on a development site.

In order to redo the work, a rework leads to a tight schedule and increase of man-hour. A cause of a failure that a large rework occurred is that a generated specification is not satisfied a function of system. This occurs if the verification of the validity of the program for the specification is insufficient. Therefore, in order to prevent this, it is important that validity of the program is verified early without omission.

In addition, it is possible to reproduce it to specify the cause, to resolve the problems is a prerequisite. Many man-hours it takes to research because the specific of cause is difficult by reproduce of failure. There are limit on the time taken to resolve the failure. If there is modify based incomplete investigation, a failure is occurred again. As a result, an investigation and modify occur repeatedly. Then a large number of man-hour is required. If developers of troubleshooting cannot assume the cause from the phenomenon of failure, Specific of the cause is a difficult situation. Thus such a

situation is not a mistake of a simple coding. Situations that occur in the timing and synchronization combination of complex conditions in most cases. It shows that the behavior of the system that is implemented by a program that is actually created and behavior of the system described in the specification do not match. The discovery of such inconsistency is carried out by combined test unit tests usually. However, since it would have been missed if the test case is insufficient, it is important that the proposed method verify the behavior of the system without omission.

When a Specification of a document is verified appropriateness and consistency without omission, these failures can be discovery. Model checking has been favored as a technique to improve the reliability earlier in the software development process. Model checking is a method which verifies that the expression is satisfied for system that is defined as state transition. This is a suitable method to checking the behavior of the system.

In many cases, a specialist in Model-Checking is working on the development site. Also an inspection is mostly reachability or deadlock. At present, model checking is not popular on the development site. The reason is that modeling of behavior of system is difficult for non-specialist in model-checking. The creation of the query equations is also difficult. This is because knowledge are required to decipher its meaning from the counterexample that is output as the results. Because a behavior of the system cannot be defined, we cannot determine the appropriateness and consistency of the specification for the failure of the two listed above.

We have proposed the source code verification method to find the discrepancy between the behavior of the source code and the specifications by using model checking technology. To use a decision table that can be easily defined and properly also the developer of the development site in the proposed method. Specification model is created on the basis of the decision table. Then an inspection is combined with source code models and the specification model. Mismatch of the behavior of the system of the source code between the specifications are discovered by this inspection, appropriateness and consistency can be confirmed. We also prepared the inspection support tool for developers as can be inspected easily.

The results of the research, using easily even developers who do not have specialized knowledge of model checking, so can the identification of the cause of the failure. Can be expected to improve the work efficiency and quality of deliverables widely in the development site of system development by doing so. In addition, soil of engineering approach is nourished.

This research is the primary purpose is to verify the source code in the downstream

process study. However to be found in the upstream process considerations and lack of misunderstanding of the specification is also important for companies. We propose a UML verification method for verifying whether or not there is an unexpected behavior in the system at the stage of requirements definition.

1. 序論

1.1 本研究の背景

一般に企業における基幹系の情報システムは, 紙ベースの業務をシステム化したものが多い．業務システム構築の目的は，業務を遂行する上で遵守すべき業務ルールのシステム化である．書類作業をシステム処理に置き換えているため，個々の処理は四則演算や承認処理など比較的単純なロジックで構成される．業務処理は，システムが実施するシステム機能とユーザが実施する手作業で構成される．これらの処理をつないだものが業務フローであり，このフローをシステム上で遂行できるようにすることで業務ルールのシステム化を実現している．

しかし開発現場では不十分な要件定義やシステムの実装時のミス･誤解等により大小多くの不具合が発生する．これらの不具合を解消するために多くの工数が費やされ，開発プロジェクトのスケジュール（納期）やコストを圧迫する．そして解消するために多大な工数を要する不具合がある．要件定義段階のような上流工程で不具合の原因が発生する場合と実装段階のような下流工程で不具合の原因が発生する場合がある．

ある程度プロジェクトが進行した下流工程で発生する手戻りは, 作業を再度やり直すことになるため工数の増大とスケジュール（納期）の圧迫を招く．大きな手戻りを発生させる不具合の原因は，システムで実現した機能と要求仕様にある機能との間に，小規模な修正では解消できない，機能差が生じていることである．これは仕様に対するプログラムの整合性の検証が不十分な場合に発生すると考えられる．従ってこれを防ぐためには，要求仕様に対するプログラムの整合性を早期にもれなく検証する仕組みが必要である．

また，不具合の解消にはまず原因を特定して再現・確認することが前提であるが，再現が困難な不具合は，まず再現する条件を特定するまでに多くの工数がかかる．通常不具合の解消にかけられるコスト・時間には制約があるため，調査が不十分なまま修正を行ってしまうことがあるが，その場合には修正が不完全となり不具合の再発を招き，調査・修正を繰り返すことになり多大な工数がかかることが多い．対応する開発者が不具合の現象からその原因を想定できない場合，こういった原因の特定が困難な状態になる．

Marc[4]は不具合原因の特定が難しい理由を議論している．そういった難しい状況の 53%が不具合現象と原因の乖離とデバッグ用のツールの適用不可により引き起こされると主張している．目に見える不具合現象と不具合の原因が乖離していて直感的に原因を想定し難いことが不具合の原因特定を難しくするといえる．

効率的な不具合原因の特定をするには開発者に不具合の原因を効率よく発見するスキルが要求される．不具合現象と原因が乖離していて直感的に原因を想定し難い場合でも開発経験の豊富な開発者ならば，経験により的中する確率が高い検査の知識があるため短期間に原因を突き止められる可能性がある．従ってこのような調査には本来は開発作業に専念すべき経験豊富な開発者があてられることが多く，開発プロジェクトの進捗に悪影響を及ぼす．しかし経験の浅い開発者を充てても， Ko[5]が述べているように原因の特定のため

にはしらみつぶし的な調査・テストを行うことになり，原因の特定までに多くの工数を要し開発プロジェクトの進捗に悪影響を及ぼす.

従ってこのような状態は, 不具合の現象を見ればすぐにわかる単純なコーディングのミスなどではなく，複雑な条件の組合せや同期・タイミングの問題で発生することがほとんどであり，要求仕様と実際に作成されたプログラムの間で，システムがもたらす業務機能の振舞いの整合性がとれていないことを示している．このような不一致の発見は，通常，単体・結合テストにより行う．しかし業務機能の振舞いを全て網羅するテストケースの作成は困難であり見逃しが発生してしまうため業務機能の振舞いの整合性を漏れなく検証する仕組みが必要である．上記２つの不具合は成果物（設計書・ソースコード）に表わされたシステムがもたらす業務機能の振舞いの妥当性・整合性の検査がもれなくできれば発見できると考える．開発現場においては，効率よくかつ正確に問題を解決する方法が求められている.

モデル検査は状態遷移系として定義されたシステムに対し, 要求する性質を時相論理式で記述し，状態遷移系がこの論理式を満たすことを検査する手法でありシステムがもたらす機能の振舞いの検査に向いている．本研究では一般的な開発者でもモデル検査技術を用いてシステムに想定外の振舞いがないかをソースコードを対象として検証するソースコード検証手法を提案する.

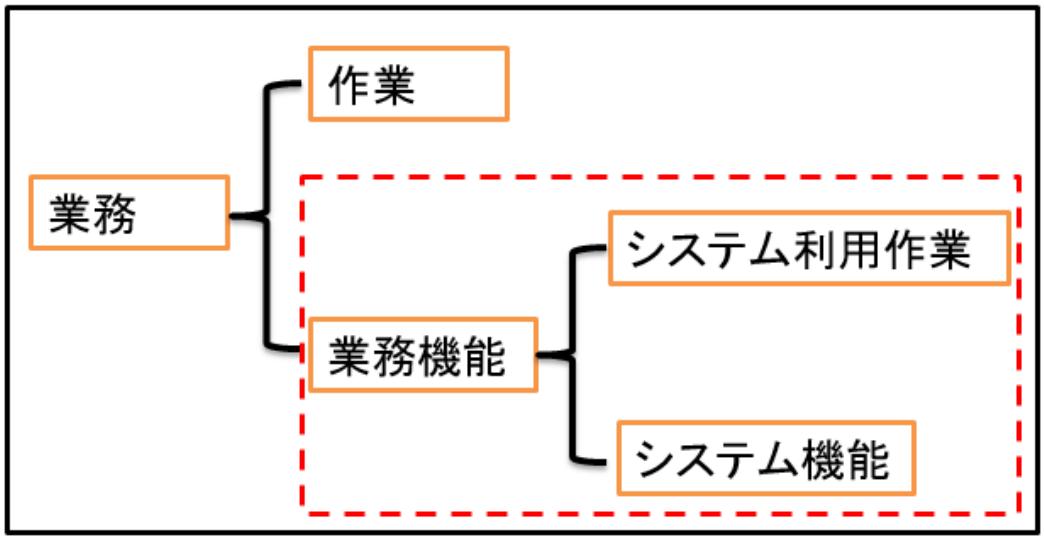
従来からある研究のアプローチは仕様もしくはソースコードを個別に扱い, 抽象化や検査の効率等，モデル検査の技術的な面を改善していく部分に注力されており現実の開発現場で利用するという部分への注力はあまりされていない．本研究はそうした従来からある研究とは違うアプローチを行っており，開発現場の開発者がもつ業務ドメインの知識や開発業務で一般的に使用される技術を用いて検査可能にする点に注力し，まずは技術的・知識的な面のハードルを下げて利用しやすくすることを目指している．これは開発現場において大きなメリットがあると考える.

本研究は下流工程においてソースコードを検証することを主な目的としているが, 仕様の誤解や考慮不足等は上流工程においても発生する．これは要求分析段階で要求仕様の非曖昧性，完全性，無矛盾性の実現が難しいためである．これを早期に発見することは企業にとっても重要である．UML（Unified Modeling Language）[71]は要求仕様を記述するための自然言語を含む柔軟な記述を許す道具として広く使用されているが，記述の自由度が高いため整合性・妥当性の検証は難しい．モデル検査は元々上流工程で仕様検証することを目的としたものであることもあり，ソースコード検証手法をソースコードの代わりにUML モデルへ適用できれば，要件定義の段階において仕様が表わすシステムがもたらす業務機能の振舞いが想定外にならないことの検証ができると考え，UML モデルの検証についても検討を行った．これを UML 検証手法として提案する.

1.2 本研究の適用対象

開発現場では不十分な要件定義や実装時のミス・誤解等により不具合がシステムに含まれる．そして綿密にテストケースの作成を行ってもテストケースには漏れが生じるため，不具合が発見できない場合は多々ある．早期に発見できないと大きな手戻りが発生して開発プロジェクトの進行を阻害する．そうした不具合が発見できないのは，複雑な条件の組合せや同期・タイミングの問題でそれらが発生するためであり，これはシステムがもたらす業務機能の振舞いに関係しているといえる．本研究の適用対象はこの「システムがもたらす業務機能の振る舞い」である．

機能要件の合意形成ガイド[1]の業務の捉え方を参考に本研究では業務の構造を図 1-1 のように考える．システムがもたらす業務機能の振舞いとはシステム化の範囲となる運用まで含めた開発対象の動作である（図 1-1 点線内）．業務機能はシステムにより自動化された業務，システム機能とユーザの手作業により行われる業務，システム利用作業の組み合わせで表される動作の連なりである．最上位にある業務を連携するつながりが業務フローである．



業務：人手で実施する業務（＝「作業」）とシステムによって自動化される業務（＝「システム機能」）と人とシステムのやりとりに関わる業務（＝「システム利用作業」）との総称

作業：人手で実施する業務

業務機能：システム化される業務　システムによって自動化される業務（＝「システム機能」）と人とシステムのやりとりに関わる業務（＝「システム利用作業」）との総称

システム利用作業：人とシステムのやりとりに関わる業務

システム機能：システムによって自動化される業務

図 1-1 業務の構成

モデル検査技術は，システムの振舞いを状態遷移とみなし，システムの満たすべき性質を，状態空間の探索により検証する技術である．テストでは実現できない網羅的検査に特徴があり，システム構築の上流工程において，その仕様の妥当性を検証するための形式検証技術として注目を集めている．「正当でない状態に陥ることは決してない」もしくは「いつか必ず特定の状態へ到達する」といった仕様に記述される性質はモデル検査技術を利用可能な形にしたモデル検査ツールで検証することができる．

本研究は基幹系の情報システムにおいて見られるような複雑な業務ルールに起因する不具合原因をモデル検査により特定することを目的とする．元々業務は図 1-1 にあるすべて人手で行われる作業により行われてきた．情報システムはこの作業の手順をプログラム上の制御フローに置き換えている．従ってこの制御フローさえ検査モデルに変換することができれば，モデル検査により厄介な不具合の原因の特定ができると考える．

また先に述べたように個々の処理が比較的単純であるため高度な抽象化を行わずに十分な粒度のモデルが作成できると思われる．これらを実現すれば経験の浅い一般的な開発者でも，仕様とソースコードもしくは UML モデル間のシステムがもたらす業務機能の振舞いの不一致を発見することができる．

1.3 本研究の目的

本研究の目的は仕様とソースコードのシステムの振る舞いの不一致を見つけることにより不具合を発見することである．本研究はモデル検査によるソースコードの検証と仕様の検証を組み合せることにより，開発現場に従来からあるモデル検査の適用方法よりも少ない工数でリーズナブルにモデル検査を適用できるようにするものである．

ソースコードの検証にモデル検査を用いる研究は従来からある．それらはソースコードを読んで理解してから検査モデルを作成し，さらに仕様も理解して実現すべき機能が満たすべき性質を時相論理式に置き換えて検査式を作成して検査するもので，主として対象を理解して検査モデルもしくは検査式をいかに作成するかの提案になっている．また検査を容易にするために定番的な API(例えばファイル I/O)を予めモデル化して組み込む等して検査を行うものである．しかしこれらの研究で提案されているソースコード・仕様の解釈の方法を理解して複雑な検査式を作成することは開発現場の開発者にとっては困難であり，定番的な API のモデル化では検査できる内容が機能に限定されてしまい，開発現場で発生するシステムにおける業務機能の振る舞いに関する不具合には適用が困難である．

業務機能はシステムによって自動化される業務・システム機能とシステムと人のやりとりにより構成される．本研究はこのシステム機能を定義した仕様からも検査モデルを作成し，これをソースコードの検査モデルと機能を接点として接合して振舞いの不一致を見つけるため，検査式は複雑にならない．またソースコードと仕様の振舞いの不一致を見つける検査式となるため検査できる内容が機能に限定されず，幅広い適用が可能となる．

本研究の提案により，一般的な開発者でもモデル検査技術を利用してシステムに想定外の振舞いがないかを検証できるようにして，通常のテストとは違う観点からの検査によりテ

ストを補完し，未検出の不具合を発見してシステム開発における成果物の品質の向上を目指す.

本研究の目的をまとめると以下の２つにある.

・モデル検査を利用してソースコードもしくは UML モデルに対して仕様を満たしているかを検証し、不具合を発見する

・開発現場でも利用できるように、モデル検査ツールを直接使わないでも検査を可能にする

これらの内容は主に以下の 2 件の論文で発表した.

Aoki, Y., Matsuura, S., "Verifying Business Rules Using Model-Checking Techniques for Non-specialist in Model-Checking", IEICE TRANSACTIONS on Information and Systems , Volume E97-D No.5, pp.1097-1108,2014.

Yoshitaka Aoki, Saeko Matsuura, "Verifying Security Requirements using Model Checking Technique for UML-Based Requirements Specification", RET 2014,pp.18-25, 2014.

これらの論文では，システムがもたらす業務機能の振舞いを表わすモデルを用いて，UML モデルもしくはソースコードと仕様の間の妥当性・整合性を検査することにより想定外の業務機能の振舞いがないことの検証できることを評価した.

1.4 提案手法の概要

本研究では自然言語で記述されたシステムがもたらす業務機能の振舞いを一般的な開発者でも適切にモデル化する手法と，検査モデルを段階的に詳細化して検査をする手法を提案する．図 1-2 は本研究におけるソースコード検証手法の概要を示している．

ソースコード検証手法は仕様において特定できる識別子に着目して，ソースコード上の識別子との対応関係を付けることにより，ソースコードの仕様の妥当性・整合性を検査する．検査はシステムがもたらす業務機能の振舞いの検査モデル(本研究ではこれを仕様モデルと呼ぶ)に変換し，それをソースコードの制御フローに基づき作成した検査モデル（本研究ではこれをソースコードモデルと呼ぶ）と結合して行う．

ソースコード検証手法ではモデル検査ツール UPPAAL[2]を用いる．モデル検査器への入力となる検査モデルと検査式の生成は，検査支援ツール Source2UPPAAL により，検査者が検査のモデルを記述せずに行う．検査対象は Java のソースコードである．このツールは独立行政法人情報処理推進機構技術本部ソフトウェア高信頼化センター（SEC: Software Reliability Enhancement Center）が実施した「2012 年度ソフトウェア工学分野の先導的研究支援事業」の一環で開発した．

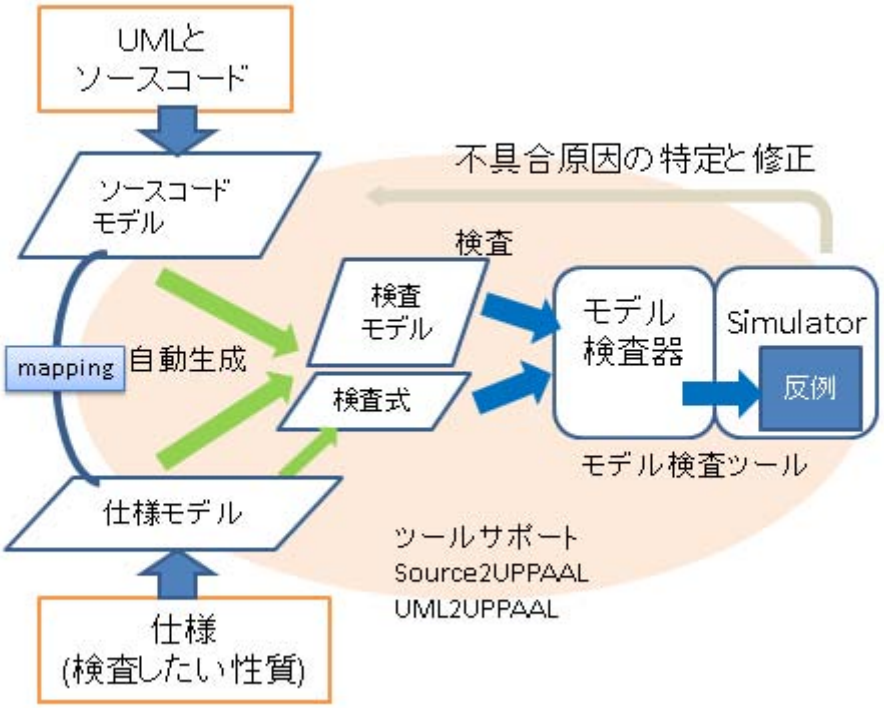


図 1-2 提案手法概要

モデル検査で常に問題となる状態数の削減については，ソースコードをメソッド単位で段階的にモデル化することとステートメントへの非決定性の値の割り当てによって対応する．システムがもたらす業務機能の振舞いを検査するためには，仕様を形式化した表現に直す必要がある．そのため本研究ではテストケースの作成で利用されるデシジョンテーブル[3]を実行結果まで記述するように拡張し，これを使って仕様の形式化を行い，業務機能の振舞いを表わす仕様モデルを作成している．仕様モデルとソースコードモデルを仕様の識別子とソースコード上の識別子の対応付けを用いて紐付けることにより検査モデルを構成する．

また上流工程において UML を対象にしたシステムの振舞いの検証もソースコードを UML モデルに置き換えて，同様の構成で行う(図 1-2)．要件定義段階の検査支援ツールとして UML2UPPAL を利用した．UML2UPPAAL は UML モデルの UPPAAL モデルへの変換，検査式の生成，検査の実行，反例のモデルへの反映の機能を持ち，UML モデリングツール astah*[57]のプラグインとして実装されている．このツールも先の Source2UPPAAL と同様に「2012 年度ソフトウェア工学分野の先導的研究支援事業」の一環で開発した．

本研究の提案手法を用いて検査対象のソースコード・UML モデルをモデル化すればその全状態を検査して性質を確認でき，経験の浅い開発者が適切な想定ができずにしらみつぶしに行ってしまう検査を肩代わりして行うことができ，それにより経験の浅い開発者の不具合の原因特定ができると考える．

1.5 論文の構成

まず，第 2 章ではモデル検査の概要を述べ，モデル検査を企業の開発現場で適用する利点と適用するにあたっての問題を説明する．

第 3 章ではモデル検査の研究動向について説明する．研究動向はソースコードのモデル検査，仕様のモデル検査，状態爆発，デッドロック・到達可能性の検査，モデル検査の非専門家による検査，反例解析，上流工程におけるモデル検査を対象にする．そして各種モデル検査ツールについて述べる．

次に第 4 章では本研究で提案する手法，ソースコード検証手法について説明する．提案手法の概要の説明をまず行い，続いてソースコードのモデル化のロジックの説明を行う．さらにソースコード検証手法のために開発した検査支援ツールについて説明を行う．次に仕様をモデル化する手法の説明，ソースコードモデルと仕様モデルを結合させた検査モデルの検査方法，検査結果の評価について説明する．

第 5 章では UML 検証手法について説明する．提案手法の概要の説明を行い，続いて UML モデルから検査モデルへの変換ロジックの説明を行う．次に UML 検証手法のために開発した検査支援ツールについて説明を行う．次にこの検査ツールを使った検査方法について述べる．

第 6 章から第 8 章では提案手法の適用事例を説明する．適用事例をとおして提案手法の有効性の評価を行う．第 6 章では実際にあった与信チェックプログラムの不具合をサンプルプログラムとして作成し，それへの適用をとおして規模に関する評価，業務要件への適用についての評価を行う．第 7 章では芝浦工業大学の授業における課題である長方形エディタプログラムへの適用をとおして実際のプログラムへの適用を評価する．第 8 章では上流工程の要件定義段階におけるセキュリティ案件への適用事例をとりあげる．本件の適用対象はソースコードではなく UML モデルであり，システムの振舞いが記述されているドキュメントがあれば適用可能であり，この応用性の高さについての評価を行う．

最終的に，第 9 章ではここまでに述べた内容についての考察とまとめ，さらに本研究で検討すべき今後の課題について説明を行う．

2　モデル検査を適用する利点と適用の問題

2.1 モデル検査の説明

形式手法は，数学的に厳密に意味付けられた言語を用いて情報システムの要求や設計等を記述し，情報システムがユーザの要求等を満たしているかなど論理的に推論するための仕組みを提供する手法である．欧米を中心に実際のシステムに対する適用事例は増えている[6]．通常のテストケースは業務フローの手続きの確認が主な目的であるため，テストで期待される結果は正常終了を想定したものになりがちで，システムが正常終了する検証はできるが，不具合がないことを証明することは難しい．テストケースをいくら綿密に作成しても全てのケースを網羅したことの保証はできない．形式手法はシステムが持つ性質そのものを検証するため一定の不具合が存在しないことの保証ができる．

形式手法は，大まかに以下の２つのアプローチに分けられる[41]．ひとつはモデル検査であり，システムの振舞いを状態遷移系のモデルとして表し，それが満たすべき性質を時相論理式（検査式）で表し，これらの充足関係を検証するものである．状態遷移を全自動で網羅的に探索して検証を行う（図 2-1）．性質が満たされない場合は反例を出力する．検査対象の状態は有限である必要がある．SPIN [7]，NuSMV [8]などが代表的である．

もうひとつは検証したい性質が仕様によって満たされることを推論規則に基づいて証明する演繹法である．VDM [50] や B-method [55]などが代表的である．検査対象の状態が有限である必要はない．検証には人による作業が必要であり，全自動で検証することはできない．

本研究は一般的な開発者が検証することを目的としているため高度な数学的な知識がなくても形式手法を利用して全自動で検証ができるモデル検査を利用する．

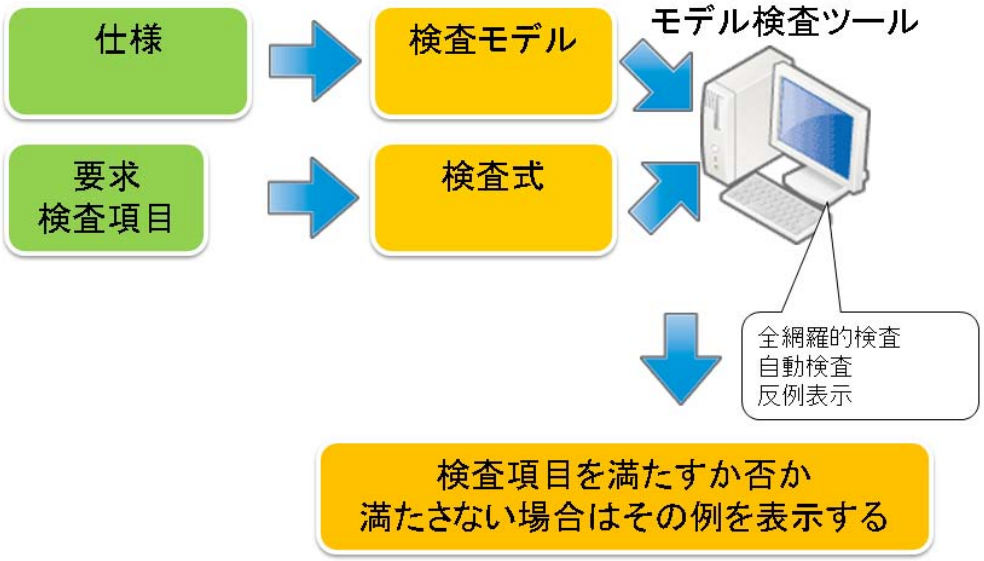


図 2-1 モデル検査ツール概要

モデル検査で検証できる性質は時相論理で記述可能なものとなり，以下の 4 つがある．

・到達可能性：

初期状態からある特定の状態へ到達できることを表す．

ソースコード上のデッドコードの検証ができる．

・安全性：

ある条件下で正当でない状態に陥ることは決して起こらないことを表す．

デッドロック，無限ループの検証ができる．

システムがある条件である特定の状態に決してならないことの検証ができる．

・活性：

ある条件下でいつか必ず特定の状態が起きることを表す．

システムがある条件である特定の状態に必ずなることの検証ができる．

・公平性：

ある条件下で特定の状態が無限回起きる．

本研究では主に「到達可能性」と「安全性」を用いて検証を行う．これは本研究が業務システムの不具合原因の特定を重視しているためである．業務システムは，業務が進むにつれて状態は変化し続けるため，特定の状態に必ずなる「活性」や特定の状態になり続ける「公平性」では不具合原因の特定がし難い．不具合を想定した状態に陥らない「安全性」と正常終了時を想定した状態になりうる「到達可能性」の検査をすれば業務システムの不具合の検証は可能である．

モデル検査では満たすべき性質を時相論理式（検査式）で表して検査を行う．この時相論理式には LTL(Linear Temporal Logic)と CTL(Computation Tree Logic)の 2 種類があり，どちらを使うかはモデル検査ツールにより違う．

LTL は状態遷移を直線的な構造として捉え解釈するものである．記述できる性質は現在の状態を基準にして「いつか状態 P が成り立つ」「常に状態 P が成り立つ」「次に状態 P が成り立つ」「状態 Q が成り立つまで P が成り立つ」である．これらの性質の否定形の記述も可能である．到達可能性を検査する場合は「いつか状態 P が成り立つ」を検査し，安全性を検査する場合は「常に状態 P が成り立つことはない」を検査する．

「いつか状態 P が成り立つ」は図 2-2 のようなイメージになる．

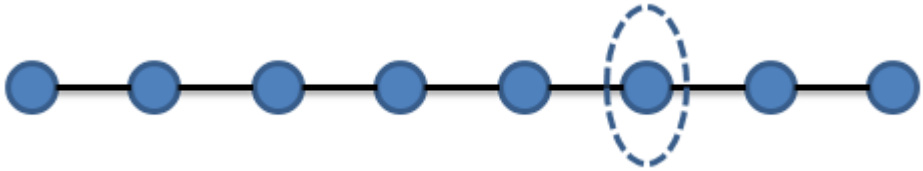


図 2-2　LTL による表現イメージ

CTL は状態遷移を木構造として捉え解釈するものである. 記述できる性質は LTL と同じであるが,「全ての経路について」または「ある経路において」という性質が付加される.「ある経路でいつか状態 P が成り立つ」は図 2-3 のようなイメージになる.

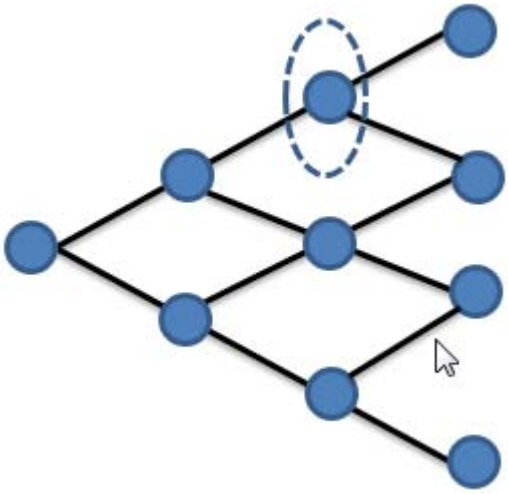


図 2-3　CTL による表現イメージ

以下に簡単なモデルの例を挙げる．変数 n を 3 までインクリメントする振舞いの Java ソースコードを検査モデルに変換すると図 2-4 になる. ステートメントを状態として捉えて制御フローをモデルとして表わしている．この検査モデルは，本研究で使用するモデル検査ツール，UPPAAL のものである．UPPAAL の記述の詳細については 4. 2 節でも説明する. エッジ（遷移）には状態遷移をする条件・更新処理・同期処理・選択処理を記述できる.

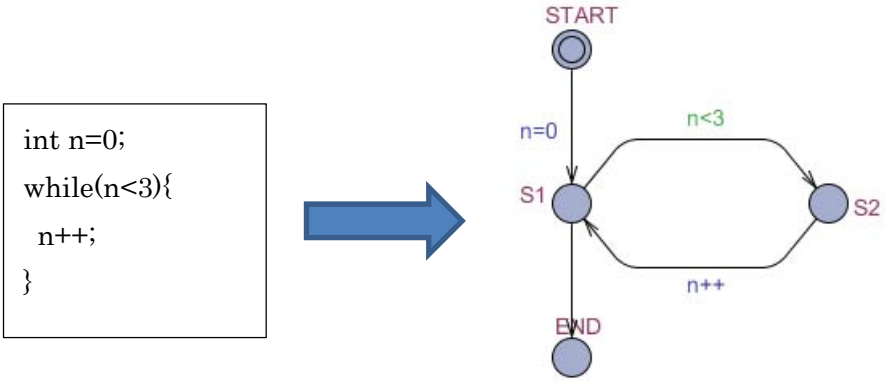


図 2-4 ソースコード変換例

この検査モデルにおいて到達可能性の検査は，例えば「ロケーション END に状態遷移

することがあるか」となり，安全性の検査は「変数 n は決して 3 を超えることは無いか」となる．この場合の検査結果はいずれの場合も「属性は満たされた」となり，検査内容が満足されることが確認できる．

またモデル検査の特徴として非決定性がある．非決定性とは「ある状態から別の状態に変わるときに，その変わる状態がひとつに定まらない」ことである．非決定性は様々なシステムの振舞いを単純に記述するのに役立つ．例えば外部から入力値を条件分岐の条件に使っている場合，入力値によって S2，S3 のいずれかの状態になるとする．この場合非決定性を使って検査モデルを作成すると図 2-5 のようになる．

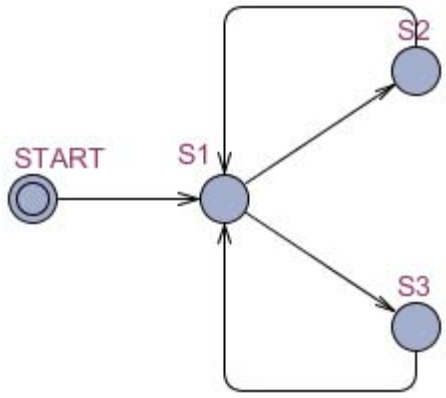


図 2-5 非決定性のモデル

モデル検査で常に意識しなければならない問題として状態爆発がある．モデル検査は基本的に「しらみつぶし」に状態を検査することで，性質を満たさない状態を発見するものである．このため，検証したいシステムのモデルが無限の状態に陥る場合は適用できない．また，有限の状態でも複雑なシステムでは検査モデル上で検査項目に対して識別したい状態数が多すぎることで，モデル検査ツールが規定時間内に検査を終えることができなくなる．

2.2 企業の開発現場においてモデル検査を使う利点

システム開発におけるコストオーバ・スケジュール超過を防ぐために，SIer（System Integrator）は開発成果物の品質を落とさずに，高効率なシステム開発を行うことを目指している．常に競合他社優位性の向上を意識している．そして業務システムはコスト・スケジュールについて多くの制限が設けられており，これを遵守して業務システムを計画とおりに開発することが求められる．

要件分析→設計→実装→テスト→本番開始が基本的なVモデルの開発工程（図 2-6）になるが，不具合の発生が後工程になるほど手戻り作業が大きくなりコスト・スケジュールに対する影響が大きくなる．

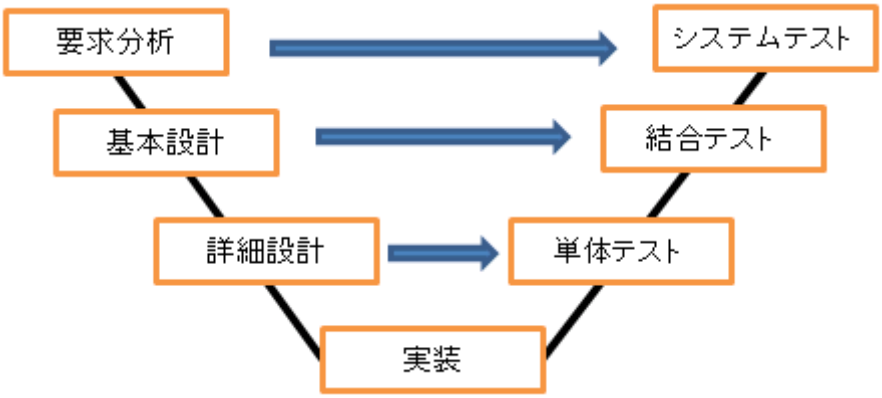


図 2-6　Vモデル

このような開発工程がとられる理由のひとつとして，各工程において顧客へ成果物の納品→検収→支払を行い，会計処理を確定させることがある．世の中からは経営資源のスピーディな把握・公開が強く要求されているためシステム工学の視点ではなく経営上の視点が影響していると言え，そういった意味でもコストオーバ，スケジュール超過の撲滅が強く期待されている．従って大きな手戻りが発生する不具合を早期に発見，修正することが非常に重要になる．

SIer（System Integrator）はシステム開発における成果物の品質を向上させるために注力しているが，どんなに綿密にテストケースを作ってもテストケースの漏れは発生し，思わぬところで不具合が発生する．経験が浅い開発者が経験豊富な開発者に較べて不具合原因の特定に工数がかかる理由は，経験豊富な開発者がある程度不具合箇所を想定して検査できるのに対し，適切な想定ができずソースコードを頭から制御フローに沿って順番に検査していくことが多いためである．経験豊富な開発者をこれらの不具合の解決にあてることは，開発プロジェクト全体の進捗を考えると望ましいことではなく，開発プロジェクトを計画通りに進めるためには開発者によらず一定の工数内で安定的に不具合を発見できる

方法が必要である．

本研究の提案手法は経験の浅い開発者が適切な想定ができずにしらみつぶしに行ってしまうテストを肩代わりして行うことができ，それにより経験の浅い開発者の不具合原因特定にかかる工数低減ができ，経験豊富な開発者をより重要な作業にあてることができるようになるため開発プロジェクトの円滑な進行に貢献できると考える．

また，システムの規模が大きくなるほど全体を把握することが難しくなるため，テスト作業はシステムのあるべき振舞いをテストするのではなく，業務における個々の手続きをテストして，それを積み重ねてシステム全体の正しさを確認するという方法を採ることが多くなる．

テストの期待結果は主に想定が容易な正常終了のケースになるので，あらかじめ想定することが困難な不具合をテストケースとして作成することは難しく，またユーザが実施する業務を全て網羅してテストすることは不可能であるため不具合が含まれたシステムがリリースされて問題を起こす．モデル検査は状態遷移系として定義されたシステムに対し，要求する性質を論理式で表しこの論理式が満たせるかを網羅的に検査できるので，検査モデルさえ構築できれば，テストケースの検討をせずとも検査モデル上でシステムの振舞いはすべて検査できるため，通常のテストでは見つけられない不具合を発見できる．

モデル検査を利用する本研究の提案手法を適用することにより通常のテストではカバーできていない部分を補完できるようになる．本研究ではこのモデル検査の性質を開発現場でも利用できるように，モデル検査適用の難しさを低減する手法を提案しシステムの品質の向上を目指す．

システムの品質は機能性，効率性，操作性，信頼性などがあるが，本研究は設計書に記述されるシステムの振舞いに着目して検査モデルを作成するため，対象になる品質は主にシステムの状態の遷移が明確な機能性，つまり作成された機能が仕様書上にある業務目的を網羅できている度合いとなる．

従って業務目的を達成できていない機能を発見してこれを修正することにより，機能が業務目的を網羅できている度合を上げることを品質向上と捉える．また，モデル検査の網羅的な検証により，顧客に対して不具合がないことを保障できる点も品質向上となる．

2.3 モデル検査の適用の問題

2.3.1 状態爆発の問題

状態爆発とは，有限の状態でも複雑なシステムでは検査モデル上で検査項目に対して識別したい状態数が多すぎることで，モデル検査ツールが規定時間内に検査を終えることができなくなる状態である．

複数のモデルが同時並行で動くが，図 2-7 のように同期処理でモデルを連携させた場合，同期した後の動きが呼び出し元と先で自由に状態遷移する．つまり呼び出し先で１つ遷移する間に呼び出し元は最後まで遷移してしまうこともありえる．その全てを検査するため，呼び出し先の１状態に対して呼び出し元の残り状態数が全て移動可能になってしまうため状態数が増大する．こういった状態の増加が状態爆発を招く．図 2-7 の例のシンプルなモデルの連携でも状態数は１０００万を超える．

本研究における課題はシステムがもたらす業務機能の振舞いの忠実なモデル化による検査モデルの作成及び作成した検査モデルでの不要な状態遷移を制限する仕組みである．

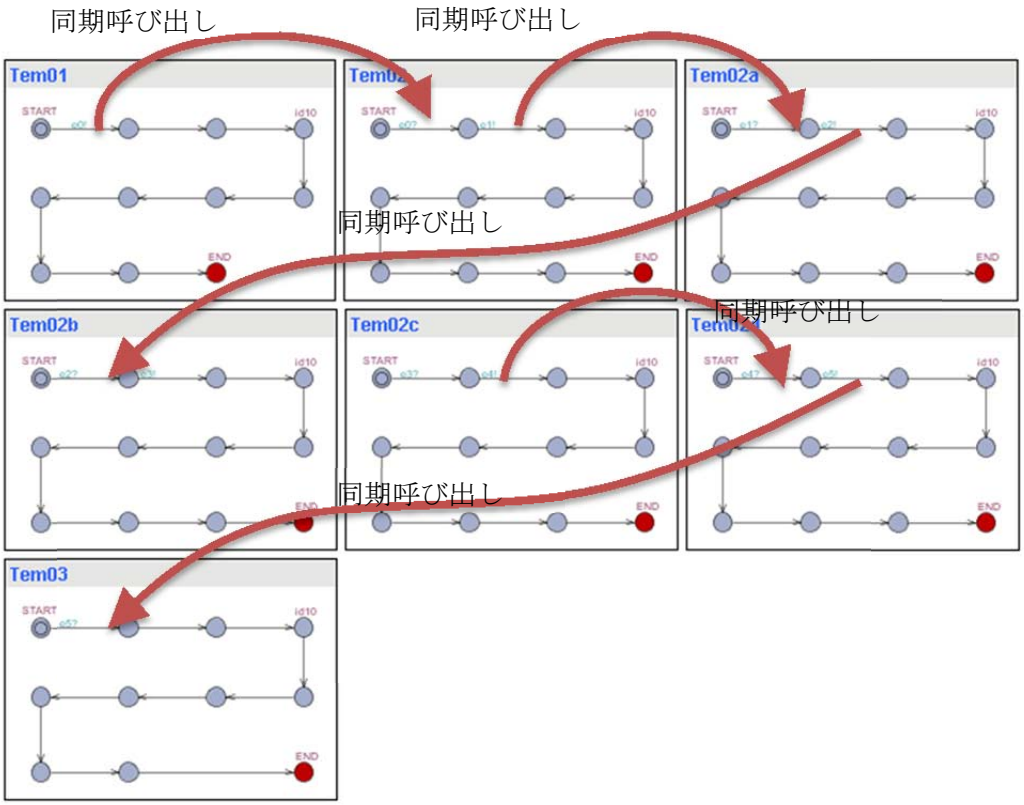


図 2-7 同期処理による連携のモデル

2.3.2 ソースコードのモデル検査についての問題

ソースコードのモデル検査では，ソースコードの制御フローに基づいてモデルを自動生成することができる．しかし，検査を行うには次の 2 つの問題点がある．第一に，ソースコードは規模が大きく，その制御構造は複雑であるため，ソースコードの全てのステートメントをモデル化すると状態爆発が発生し，検査ができないことである．第二 には，デッドロックや到達可能性の検査は，前者は何も指示する必要はなく，後者も到達すべき状態，すなわち，ソースコードの位置を指定すれば検査が可能である．しかし，アプリケーションの性質に依存した検査を行う場合には，検査したい性質をソースコードに記述されている識別子を用いて，複雑な検査式を定義しなければならない．いくつかの変数やメソッド名を検査モデル上の変数とロケーションに正しく関連付けて検査式を記述する必要がある．ソースコードの理解とモデル検査の知識が必要である．ソースコードを読み解いて，検査式を書く方法やアプリケーション固有ではない性質の検査の研究はあるが，これらの作業は，選択する基準・作業量が問題となり開発現場の技術者が容易には行えない．

本研究では上記の問題に取り組む．第一の問題に対しては，段階的なモデル化により必要なシステムの振舞いのみをモデル化する提案をするとともに，抽象化によっては検査したい性質が隠れてしまうので，それへの対応も提案する．第二の問題に対しては，複雑な検査式を記述しなくてもシステムの性質を検査できる方法を提案する．

2.3.3 仕様のモデル検査についての問題

一般的に自然言語で記述された仕様が要求を満たしているかを客観的に判断することは難しい．開発現場において，仕様の正しさを判断する基準は，レビューの実施回数であったり，不具合発見率であったりするのもそのためである．モデル検査でも自然言語への対応は困難と認識されている．自然言語で記述された仕様をモデル検査するには検査者が正確に仕様を理解する必要があるからである．そして開発現場において仕様を記述しているドキュメントの多くは自然言語で記述されており，これが開発現場におけるモデル検査の適用を難しくしている．仕様書（設計書）には自然言語で「何がしたいか」は必ず記述されているため，自然言語の仕様から定型的な手法でシステムがもたらす業務機能の振舞いを抽出する方法があれば検査モデルの作成は可能になる．

本研究で取り組む課題としては，一般的な開発者が通常行う設計・実装・テスト等の作業の延長上でモデル検査を用いた検査を行えるようにすることである．

2.3.4　モデル検査の非専門家によるモデル検査についての問題

開発現場においてモデル検査を使う場合，検査を実施する検査者はだれが適任であるか検討する必要がある．業務ドメインの知識を持つ検査者はモデル検査の技術を持っておらず，逆にモデル検査の技術を持っている検査者は業務ドメインの知識を持っていない(図2-8)．モデル検査の非専門家である一般的な開発者はモデル検査の知識がないためモデルの作成，検査式の作成が困難である．モデル検査の専門家はモデルの作成や検査式の作成については問題ないが業務ドメインの知識がないため単独で検査モデルを作成することはできず，業務担当者からのヒヤリングが必須となる．ヒヤリングの内容については実際にモデル検査を行って結果を検証してみるまで適切かどうかの判断も困難である．さらにヒヤリングするためのコスト・時間が発生する．

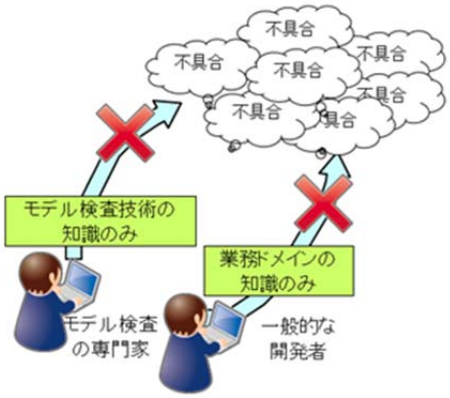


図 2-8 モデル検査適用の問題

検査者としては業務ドメイン知識を持つ一般的な開発者の方に優位性があると考える．その理由は，検査すべきシステムがもたらす業務機能の振舞いを仕様書とソースコードから導き出せるものに限定すれば，モデル検査技術の不足については手法及び支援ツールで補うことが可能だからである．従って本研究の取り組む課題は，モデル検査の知識不足を補う定型的な手法及び支援ツールの提案である．

2.3.5　検査結果である反例の解析について問題

モデル検査は検査対象となるシステムの振舞いを有限の状態空間にモデル化し，システムの満たすべき性質を表す論理式で検証する．もし与えられた性質が満たされなかった場合，モデル検査ツールはその満たされない状態に至る状態遷移の経緯を反例として出力する．不具合は検査モデル上の特定の状態遷移で表されるため，検査者はこの反例を解析することにより，不具合の原因を特定することができる．

しかし，反例は検査式を満たさない１つの状態遷移のトレースを表しているに過ぎない．そのため，その状態遷移のトレースから不具合の原因を特定するには，検査者が状態遷移における各状態を検査モデルと照らし合わせて意味を読み取る必要がある．これにはモデル検査の知識と検査対象に対する知識を併せ持つことが要求されるがそのような開発者は少ない．反例として出力される振る舞いの経路が少ない単純な構造の検査モデルの不具合ならば問題にならないが，複雑なモデルの場合，膨大な量の反例から原因を特定するのは解釈した内容を手作業で整理しながら行なわなければならず困難である．

本研究で取り組むべき課題は，状態遷移とモデルを照らし合わせて理解するようなことをしなくても不具合原因を想定できる表示方法の提案である．

2.4 本章のまとめ

本章では，本研究で使用するモデル検査とはどのようなものであるかを説明した．基本的な考え方，検査できる特性と簡単なモデル検査のモデルの例を示した．

次にこのモデル検査を企業の開発現場において利用した場合のメリットについて説明した．企業ではシステムを開発するにあたり最終的なシステムの機能確認は綿密なテストにより行われるがどうしても不具合の見逃しが発生してしまう現状がある．モデル検査を利用することによりこれを補完できることを述べた．

次にモデル検査を利用するにあたっての問題点について説明した．これは企業内でモデル検査が普及しない理由といえる．状態爆発の問題，ソースコード・仕様のモデル検査の問題，モデル検査の非専門家によるモデル検査の問題，そして検査結果である反例の解析の問題について述べた．これらの問題への対応が本研究の大きな課題である．

3　モデル検査の研究動向

本章では 2.3 節であげたモデル検査の利用にあたっての問題に関する研究動向と本研究における対応について述べる．さらに上流工程の要件定義段階におけるモデル検査利用の研究動向についても述べる．次に利用されるモデル検査ツールの特徴および UPPAAL を採用した理由についても説明する．

3.1　ソースコードのモデル検査

モデル検査は上流工程において仕様を検査する場合に多く用いられるが，プログラムのソースコードの検査にも用いられる．ソースコードの不具合検出にモデル検査を使用する研究は多い．ソースコードのモデル検査に関する研究は図3-1に示すように「仕様に関わる性質の検査」「ソースコードの制御構造の検査」に分類される．本研究は仕様に関わる性質の検査に入る．既存の研究はソースコードを解釈し，さらに仕様を解釈して複雑な検査式を作成して検査するものである[12][13][45]．また検査を容易にするために定番的なAPI(例えばファイルI/O)を予めモデル化して組み込む等して検査を行うものもある[46][56]．しかしこれらの研究で提案されているソースコード・仕様の解釈の方法を理解して複雑な検査式を作成することは開発現場の開発者にとって困難である．また定番的なAPIのモデル化では検査できる内容がAPIの機能の振る舞いに限定されて開発現場で発生する不具合には適用が困難である．

本研究はシステムによって自動化される業務・システム機能実行時に成立する状態を定義した仕様から検査モデルを作成し，ソースコードの検査モデルと機能を接点として接合して振舞いの不一致を見つけるためアプリケーション固有の性質に対する検査式は複雑にならず，またソースコードと仕様の振舞いの不一致を見つける検査のため検査対象が特定の機能に限定されず，幅広い適用が可能となる．

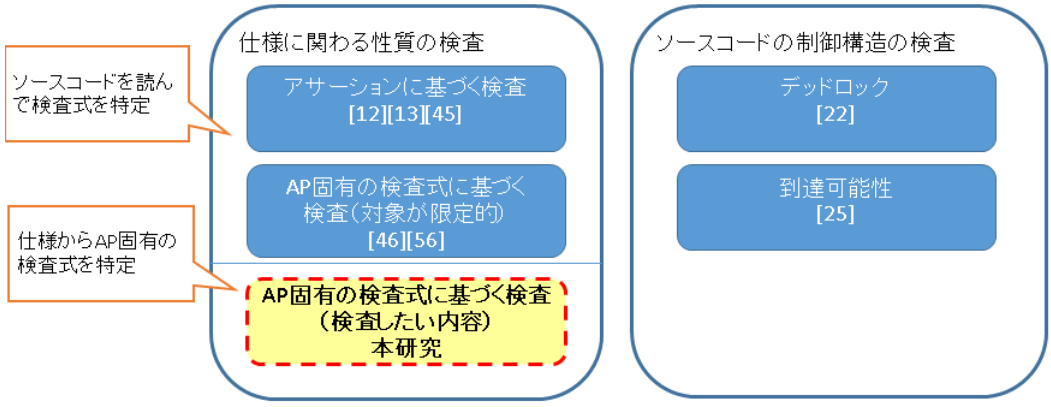


図 3-1　関連研究比較図　ソースコードのモデル検査

ソースコードの制御構造の検査はデッドロックや到達可能性による研究が特に盛んである．前者は何も指示する必要はなく，後者も到達すべき状態，すなわち，検査モデル上の

位置を指定すれば検査が可能である．モデル検査の基本的な検査であり，状態遷移が不能になる状態がないか，検査モデル上に到達できないロケーションがないかを検査する．もしこれらがある場合，検査モデルとして致命的な欠陥があることになる．さらにソースコード・仕様を忠実にモデル化しているとすればそれらにも致命的な欠陥があることが判明する．

John [22]らは Java ソースコードからモデル検査ツール Java Pathfinder（JPF）[9]のデッドロック検出機能を用いてデッドロックを見つけている．彼らはサンプルプログラムを用いて JPF によるデッドロックの検出は有効であることを主張しているが，その結果のトレースを追うことは容易でないとも言っている．また JPF は Java のバイトコードを検査対象としているため，あるシナリオに沿って検査をするにはテスト対象の呼び出しと結果を確認するプログラム(テストドライバ)の作成が必要であり，これが困難であり費用対効果の面でもよくないと言っている．

Beyer ら[12]は，eclipse のプラグインでモデル検査ツール BLAST[42]を用いて，到達可能性の検査を行っている．C のプログラムにアサーションを挿入してそこへの到達可能性を検査することによりデッドコードを発見するというものである．アサーションの挿入は，検証目的がはっきりしている場合は挿入する場所が明確でよいが，ミッションクリティカルなシステムで網羅的な検査を行うのには向いていない．

Lei ら[13]は BLAST を用いてアプリケーションのバッファオーバーフロー（変数）の脆弱性の検査を行っている．バッファに割り当て可能なバイト数の制限を表わす条件式のアサーションをソースコードへ挿入し，そこへの到達可能性を判定することにより検査をしている．この研究は変数の脆弱性を検査するために特化している．

Thompson ら[45]は彼らが開発したモデル検査ツールを用いて C++で作成された NASA のフライトソフトウェアを検査している．JPF と同様にバイトコードを検査してアサーションによりエラーを発見する．アサーションを挿入する場所の選定や発見後のステートメント単位での状態遷移の確認は，ソースコードをデバックすることに近い検査といえる．本研究が提案しているシステムがもたらす業務機能の振舞いの検査には向いていない．

Xiaoli ら[46]は組み込みシステム設計言語 Virgil のソースコードを中間言語へ変換し，そのブロック単位をロケーションに置き換えて UPPAAL モデルへ変換して検査を行っている．彼らが検査しているのは，モデル上でエラーになる条件が満たされると状態遷移するモデルを設けてそのロケーションへの到達可能性を検査してエラーを発見している．ブロック単位でモデルへ変換するため複数のステートメントがまとめてモデル化されるので，ステートメントとロケーションが一対一で対応していない．従って検査結果（反例）からソースコードの位置を特定することも難しい．

デッドロックや到達可能性の検査は，検査としては基本的なものであるが，不具合原因を特定するためには不具合の再現条件および不具合原因の場所を明確にしなければならないためその検査方法および検査結果の表し方には検討が必要である．本研究では検査支援ツ

ール Source2UPPAAL を用いて一般的な開発者でも検査モデルを作成できるようにし，到達可能性の検査式も自動生成する．

Markosian[10]は Java Pathfinder により NASA のフライトシミュレータ開発時に検査を行っている．リファクタリングにより状態数を減らすことにより効率的にモデル検査を行う手法を説明した．この手法は有用な技術である．

Hao ら[25]は SPIN を用いて並行システムにおいて効率的に到達可能性の検査を行う提案をしている．

Bandera[11][47]は Java ソースコードをモデル検査するためのツールである．抽象化仕様言語(BASL)を用いて指示する抽象化とスライシングにより，Java プログラムのソースコードから，コンパクトな検査モデルを作成する．検査モデルは SPIN や SMV 等の既存の検査モデルを作成できる．従って検査自体はそれらのツールで行う．このアプローチは，"状態爆発"には有効であるが，検査者には検査内容を解釈してモデル検査式へ変換する知識が必要となる．

Shanbhag ら[54]のモデル検査を用いずにプログラムを作成して静的に Java のライブラリを解析する研究もあるが使用目的が限定的で汎用性は高くない．

3.2 状態爆発の防止

ソースコード内の全てのステートメントを検査モデルに変換すると状態数が膨大になりモデル検査ツールが規定時間内に検査を終えることができなくなる状態爆発を起こす可能性が非常に高い．状態爆発をさけるために，ソースコードは抽象化してモデルに変換する必要がある．ソースコード上で捉えたい現象が現れる範囲で抽象化したモデルを状態爆発が生じない粒度で作成する必要がある．手動もしくは自動で行うかの違いはあるが基本的には，プログラムの顕著な特性を抽象化してモデルに変換することにより行う．状態爆発を防ぐ様々な方法が検討されており，現実的なモデル検査範囲設定の指針及び検査モデル作成の定型的な手順が構築できれば上記問題を解決でき，モデル検査の利用が可能になると考える．スライシングを使って無関係のコンポーネントを除外するなど状態爆発を防ぐ様々な方法が検討されている．状態爆発の防止に関する研究は図 3-2 に示すように「抽象化」「スライシング」「式の拡張」「並列実行」に分類される．本提案は抽象化の分類に入る．本提案では，検査時の状態数を減らすために，機能的構造をメソッド単位で捉え，モデル化対象に選択されたメソッド内のステートメントへの非決定値の付値または仕様モデルへの割り当てをすることによる段階的な検証手法を提案している．既存の研究は「述語抽象化」「データマッピング」等を用いてシステム全体の抽象化を行うが，本研究では業務ドメインの知識に基づいて機能一覧・機能定義書から検査対象となる機能に該当するメソッドを選択して，その選択されたメソッド内のステートメントを非決定性や仕様モデルを用いて抽象化する．抽象化の対象はソースコードの機能の振る舞いに限定的されるが，本研究の目的には合致する．また新たな学習なしで開発現場の開発者が実施可能である．

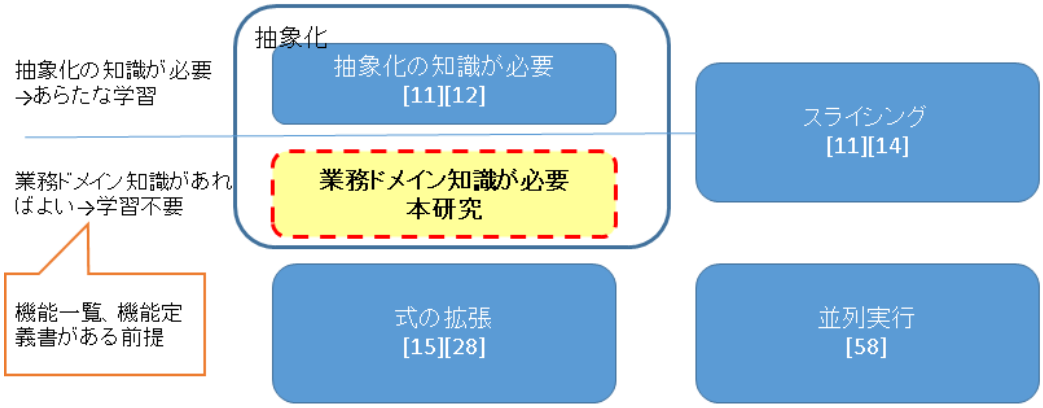


図 3-2　関連研究比較図　状態爆発の防止

Corbett[11]らが提案しているモデル検査ツール Bandera は抽象化とスライシングにより，Java プログラムのソースコードから，コンパクトな有限状態モデルの自動抽出が可能である．スライシングや抽象化エンジンを用いることにより無関係のコンポーネント除去，データ抽象化，コンポーネント制限を行っている．出力モデルは，SPIN[2]や SMV[4]等の既存のモデル検査ツールで検証することができる．このアプローチは，状態爆発には有効であるが，検査者はスライシングをするための知識，抽象化を行うために抽象化仕様言語（BASL）の知識が必要となる．

Beyer ら[12]はモデル検査ツール BLAST の抽象化機能を使って状態爆発を防いでいる．

Classen ら[15]は，ソフトウェアプロダクトライン（SPL）に対してモデル検査を適用している． featured transition systems (FTS)を用いて SPL を表しモデル検査器は NuSMV を用いている．状態爆発の問題に直面し CTL を拡張した独自のアルゴリズムを提案した．その結果かなり高速に処理できるようになったと述べている．

Wang ら[14]は JPF を用いたアサーションの到達可能性を検証によりシステムの脆弱性を検出する研究において，状態爆発を回避するためにスライシングによりアサーションに関係しないコードを除去している．

Salamah ら[28]は Dwyer らによって定義されたパターンとスコープを利用して PropertySpecification(Prospec)ツールを提案している．さらに LTL 式の実行効率を改善することを提案している．効率的に LTL 式を実行するために，状態数の削減は避けられない問題である．彼らの研究は LTL 式のスケールを削減する問題の解決に貢献している．

Achenbach[56]は様々なモデルチェックツールの抽象化テクニックを比較し，それぞれのツールを実際の問題に適用している．例題としてファイル IO のエラー処理をオープン，クローズ，エラーの 3 状態のモデルを使って抽象化しており，これを用いて抽象化の考え方を検討している．

Verstoep ら[58]は分散型のモデル検査 DIVINE[59]を用いて LTL モデル検査の並列アルゴリズムを提案している．状態爆発への対策としては有効であるが非効率な DIVINE のタ

イマ管理の改善など DIVINE に特化したものである．

状態爆発を防ぐには余計な状態数を減らすことが大切であり，方法はスライシングであったり，モデル検査ツールの機能であったり,個別のアルゴリズムによる抽象化であったりする．いずれにしても状態数を減らした検査モデルにもとの性質が残されているかの確認は難しい． 本提案は Java のソースコードのメソッド単位で段階的に変換を行う方法を採っているため，どの機能が抽象化されているのか理解し易い．

3.3 仕様のモデル検査

顧客との間で合意した仕様に基づき作成された大規模システムが業務ルールを完全に網羅しているかを確認することは困難であり，想定外の使い方により発生する不具合に対処することに多大な工数を費やしている．設計書を見ても業務ルールが漏れなく遵守されているかを確認するすべは無い．業務システムが満たすべき仕様を第三者からも客観的に評価ができる手法を開発することが必要である．

業務システムの制御条件は複雑であり，制御項目の多くは条件マスタに格納され，複数の条件マスタとプログラムのロジックを組み合わせて業務を実現している．そのため不具合原因が実データに依存している場合，テストデータでデバックしても再現できず，不具合の原因追求が難しい場合がある．このような場合，条件マスタのルールをモデル化して遵守しなければならない仕様を仕様モデルとすることでソースコードから生成したソースコードモデルに対して想定外の動作がないかを検査することができる．

業務ルールが遵守されているということは，想定外の業務機能の振舞いがないことである．従ってモデル検査を用いて業務ルールが遵守されていることを検査するには，その業務機能の振舞いを検査モデルもしくは検査式として定義する必要がある．これは業務機能の中身を特に考慮しないデッドロックや到達可能性の検査ではできない．

仕様のモデル検査に関する研究は図 3-3 に示すように「モデル検査ツールで直接モデルを記述しないで別の記述方法で作成されたドキュメントを変換」と「モデル検査ツールをそのまま利用する」に分類される．本研究は前者の分類に入る．

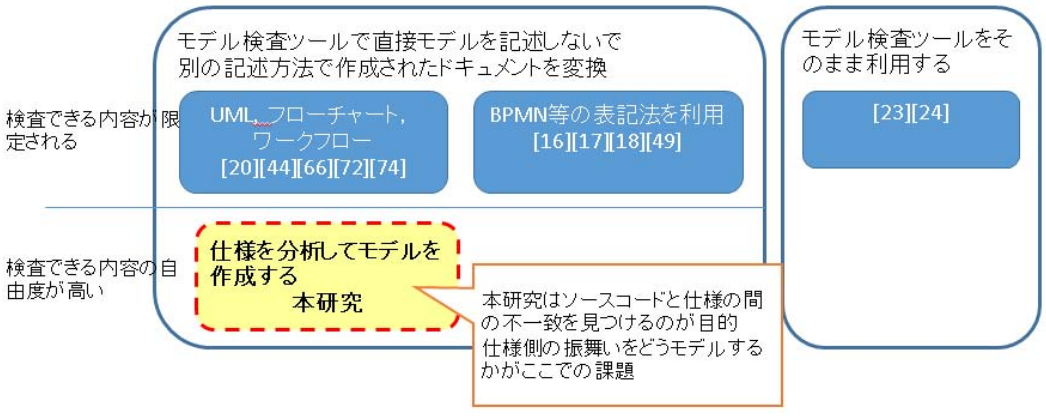


図 3-3 関連研究比較図 仕様のモデル検査

既存の研究としてはモデル検査ツールそのまま利用してモデルを作成するもの[23][24]があるが，これを実施するには検査内容を理解し，解釈してモデル化する必要があり難しく，そのため検査するために充分な情報を持つドメインに基づいて別の記述方法へ一旦変換し，それをモデル化する研究[16] [17] [18] [20] [44] [49] [66] [72] [74]が盛んである．しかし，既存のこれらの研究は変換結果がドメインに依存するため適用できる範囲がそのドメインで表わせる内容に限定される傾向がある．例えばビジネスプロセスの時間制約を BPMN で表わしてモデル化する手法は航空機用油圧システムのロジックのモデル化には利用できない．

本研究では，自然言語記述の仕様からシステムがもたらす業務機能の振舞いを理解しやすい定型的な手法（デシジョンテーブル等）で整理して仕様モデルを作成し，これをベースにして検査を行うためシステムの振る舞いに関する仕様があれば対応が可能であり，開発現場で起きる不具合への対応性は高いと考える．

以下で仕様のモデル検査の研究について述べる．([44][66][72][74]の研究については 3.6 上流工程におけるモデル検査で述べる．)

Dury ら[17]は，ビジネスワークフローを Role Based Access Control (RBAC)を用いてモデル化し，そのセキュリティ特性を SPIN により検査した結果について議論した．示されたモデル化方法は，モデル検査を適用する効果的な例である．しかしモデル検査の専門家ではない開発者のためのビジネスルールの幅広い用途のモデリングではない．

Roy ら[18]はエラーメトリックスおよびビジネスプロセスモデリング表記法（BPMN）との関係の実証分析を行った．制御フローのエラーや構文エラーに基づいて検出されている．彼らの手法は BPMN によるモデリングの知識を必要とするので，本研究の目的の一般的な開発者による検査実施には向いていない．

Watahiki ら[49]はビジネスプロセスモデリング表記法（BPMN）を UPPAAL にマッピングすることによりビジネスプロセスの時間およびリソースの制約を検証している．時間の制約の組み合わせの検証に特化しているため検証できる性質が限定的である．

Angelis ら[16]は BPEL(Business Process Execution Language)で記述した対象を Java に変換して Java Pathfinder に検査特性をわたしてモデル検査を行い，反例が出力された場合にその特性をテストケースとして導出する提案を行っている．既存のドキュメントがある場合でビジネスプロセスの成否を検査するのに有効であると思われる．一般的な開発者にとって BPEL 及び検査特性の作成は難しい．

Chen ら[19]はテスト用のセキュリティポリシーのためのシナリオで駆動されるフレームワークを提示する． モデル検査ツールは NuSMV を使用している．シナリオをオートマトンで記述してモデルへ変換，テストのカバレッジ基準から検査式を作成して検査を行う．そして不正なシナリオが見つかった場合，出力された反例に基づいてテストケースを生成している．最初にシナリオをオートマトンで記述するなどモデル検査に関する知識が必要である．

JAXA[23][24]では人工衛星の姿勢制御ソフトウェアの仕様記述に対して UPPAAL の時間特性の検査を用いて到達可能性の検査を実施して，タイミングの問題を発見している．到達可能性と不変条件が守られることの検査であり，タイミングのズレがないことの検証に特化した検査である．

Yanhua ら[20]はワークフローのプロセスをモデル検査ツール UPPAAL に変換して時間的制約の検証を行っている．ワークフローは複数のアクティビティ(活動)で構成され，各アクティビティ(活動)は実施するのに必要な時間の範囲を持たせて定義される(例えば 1-4time). UPPAAL は時間の概念が扱えるためアクティビティ(活動)で構成される業務のシナリオ（ワークフロー）が要求される時間内で終了できるかの検証を行っている．システムの振舞いの時間という性質にだけ特化して検証していると言える．

Lindsay ら[21]は，自然言語で提供されるシステムの機能要件をモデル化するためのグラフィカルな表記法である行動ツリー（BT）によりシステムの振舞いを記述しそれをモデル検査ツールの検査モデルへ変換する手法を提案している．BT 表記はノードとエッジで構成される．ノードの種類は状態設定，選択，ガード，内部イベント，外部イベントである．これらを組み合せてモデルを構成する. 適用事例としてエアバス A320 航空機用油圧システムの設計ロジックを検証している．検査モデルを直接作成するのに比べれば容易かもしれないが，行動ツリー（BT）は手動で作成することになるため，記述するには専門知識の学習が必須となる．示された適用事例のモデルは比較的単純なものであったが，もっと複雑なものになった場合，記述された行動ツリー（BT）はシステムの制御フローと取りうるべき状態が一緒に記述されているため，記述が正しいか確認するのは難しいと思われる．本研究では仕様の整理にはデシジョンテーブルを使うなど一般的な開発者が持つ知識の範囲で作業できるため，特別な学習は不要である．またシステムの取りうるべき状態（仕様）ごとにモデル化するため確認がし易い．

Trcka[65] はペトリネットを利用して，read, write, delete といった振舞いを特定する予め用意した 9 つの検査式を用いて検査を行う方法を提案している．この研究も本研究のアプローチと類似しているが，本研究では仕様モデルを用いてシステムがもたらす業務機能の振舞いに着目した検査をすることにより，様々な他の性質の検査への拡張が可能である．

3.4　モデル検査の非専門家による検査

業務ドメインの知識を持つ一般的な開発者はモデル検査の技術を持っておらず，逆にモデル検査の技術を持っている外部の専門家は業務ドメインの知識を持っていない．モデル検査の非専門家である一般的な開発者はモデル検査の知識がないためモデルの作成，検査式の作成が困難である．またモデル検査の専門家はモデルの作成や検査式の作成については問題ないが業務ドメインの知識がないため単独で検査モデルを作成することは困難で，業務担当者からのヒヤリングが必須となる．モデル検査の専門家は業務ドメイン知識がないためモデル化に欠落があっても単独では気づけないため，ヒヤリングの内容が適正に検査モデルに反映されない可能性がある．さらにヒヤリングするためのコスト・時間が発生する．従って本研究では，検査はモデル検査の非専門家である業務ドメイン知識を持つ一般的な開発者が行うべきと考える．

モデル検査の非専門家による検査の研究は図 3-4 に示すように「手動」と「自動」に分類される．本研究は「手動」と「自動」の両方の分類に入る．既存の研究[25]はモデル検査の検査式の作成が難しいため，それを容易にするために適用する仕様を分類してパターン化したプロパティ仕様パターンを作成し，これを検査式と結びつけることにより，検査したい仕様がどのプロパティ仕様パターンに合致するか決められれば検査式を決定できるというものである．既存の研究[26]はプロパティ仕様パターンの決定を容易にするためツール化したものである．彼らが提案するパターンは検査式の記述を助けることには確かに有効だが，それを利用するためにはプロパティ仕様パターンの意味やスコープの意味を理解する必要がある．形式手法の非専門家にとって厳密な数学的形式に則り，正しく有限オートマトンに関連した検査式を記述することは困難と考える．

本研究は，検査モデル作成する支援ツールと定型的で理解しやすい仕様分析，例えばデシジョンテーブルによりシステムの振舞いの把握により，適用範囲は限定されるが容易に検査式を作成できる．詳細は後述するが本研究の適用範囲は既存の研究[25]の 80%を網羅しており有効性は充分にある．

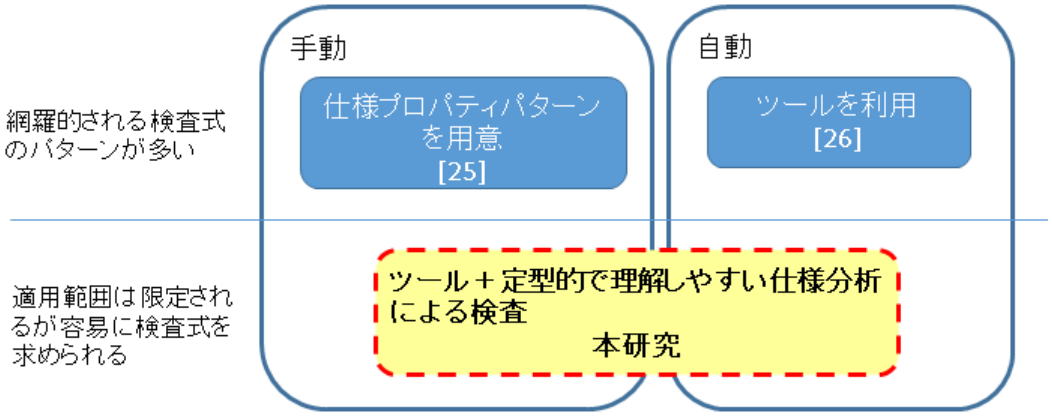


図 3-4　関連研究比較図　モデル検査の非専門家による検査

Dwyer ら[25]はプロパティ仕様パターンによるアプローチを提案している.プロパティ仕様パターンは，システムの振る舞いのいくつかの重要な側面の構造を記述し，LTL，CTL の一般的な形式手法の範囲で，この振る舞いに対する検査式を提供する．５つの基本的なスコープとして global, before, after, between, after-until がある．スコープは状態／イベントの開始と終了を指定することにより決定される．例えば global はプログラム全体にわたる実行を意味し，before はある状態／イベントに至るまでの実行を意味している．彼らは 555 個の仕様を集め，それぞれの仕様を検討して手動でプロパティ仕様パターンにマッチするか確認している．結果として，555 個のサンプルのうち 511 個(92%)がマッチすることを確認している．仕様の大部分を占めるサンプルのパターンは Response, Universality, Absence で，これらの 3 つのパターンでサンプルの 80%を網羅している．Response はスコープ内で状態/イベント P は状態/イベント Q に続いて必ず発生することを意味する．Universality はスコープ内を通じてある状態/イベントが必ず発生することを意味する．Absence はスコープ内である状態/イベントは発生しないことを意味する．本研究のデシジョンテーブルを利用した手法から生成された検査式は，この Global スコープにおける，Response, Universality, Absence などの最も一般的なパターンに該当している．

本研究では，デシジョンテーブルによりシステムの想定される状態，もしくは想定されない状態について把握する．これは Global-Universality，Global-Absence のパターンに該当する．また Global-Response についてもデシジョンテーブルで業務ルールを遷移条件に変換してモデル化するため，特定の状態／イベントへの到達可能性を調べることにより同様の検査を行うことができる．これはデシジョンテーブルによるソースコード検証手法が Dwyer らの提案とほぼ同じ範囲，サンプルの 80%で利用可能であることを示している．

Rachel ら[26]は Dwyer の提案しているプロパティ仕様パターンとスコープの考えを基に PROPEL システムを提案している．ただし，Dwyer のプロパティ仕様パターンでは非専門家が扱うには不十分と考え，有限オートマトン(FSA)と disciplined natural language (DNL)をテンプレートとして用いてプロパティ仕様パターンの記述性の向上を提案している．DNL は明確な意味の捕捉を意図した自然言語の制限されたサブセットである．PROPEL はプロパティの記述や理解をこの提供されたテンプレートを用いて簡単に行えるようにすることを狙っている．しかし，テンプレートを用いて仕様プロパティを正確に記述する難しさは残る．また彼らの研究は Universality 等，Dwyer の全てのパターンについて言及されておらず，非専門家の検査者は当てはまるパターンが見つけられない可能性がある．

Guglielmo ら[27]は Drag and Drop PSL (DDPSL) を提案している．これはテンプレートライブラリ(DDTempletes)と特性仕様言語 (PSL) ベースのテンプレートを利用したプロパティ定義を簡素化するためのツール(DDEditor)である．ユーザが DDEditor 上でドラッグ＆ドロップすることにより専門家ではなくても PSL プロパティを定義できるようにする

ことにより，複雑なプロパティの定義に必要な労力を削減する方法を提供している．定義されたプロパティを基に DDEditor はルールファイルを生成し，それを既存の検証ツールにかけてシミュレーションを行い検査している．適用事例では産業用ベーカリーオーブンコントロールアプリケーションの解析を行い，IBM フォーマルチェッカーツールにルールファイルを入力して不具合を発見している．生成されるルールはモデル検査における検査式にあたるが，これらの式(ルール)は開発者が正確に書き込む必要がある．

3.5 反例解析

モデル検査ツールは検査対象となるシステムの振舞いを有限の状態空間にモデル化し，システムの満たすべき性質を表す論理式で検証する．もし与えられた性質が満たされなかった場合，モデル検査ツールはその満たされない状態に至る状態遷移の経緯を反例として出力する．検査者はこの反例を解析することにより，満たされない状態にいたる原因を特定することができる．しかし，反例はモデル検査ツールが出力する状態遷移のトレースファイルであるため単に状態遷移の過程を表しているに過ぎない．そのためそこから不具合の原因を特定するには，検査者が状態遷移を検査モデルと照らし合わせて意味を読み取る必要がある．これは単純な構造の検査モデルの不具合ならば問題にならないが，複雑なモデルの場合，膨大な量の反例から原因を特定するには解釈した内容を手作業で整理しながら行なわなければならず困難である．

反例解析に関する研究は図 3-5 に示すように「反例を理解しやすくする」と「反例を収束させる」に分類される．本研究は両方の分類に入る．

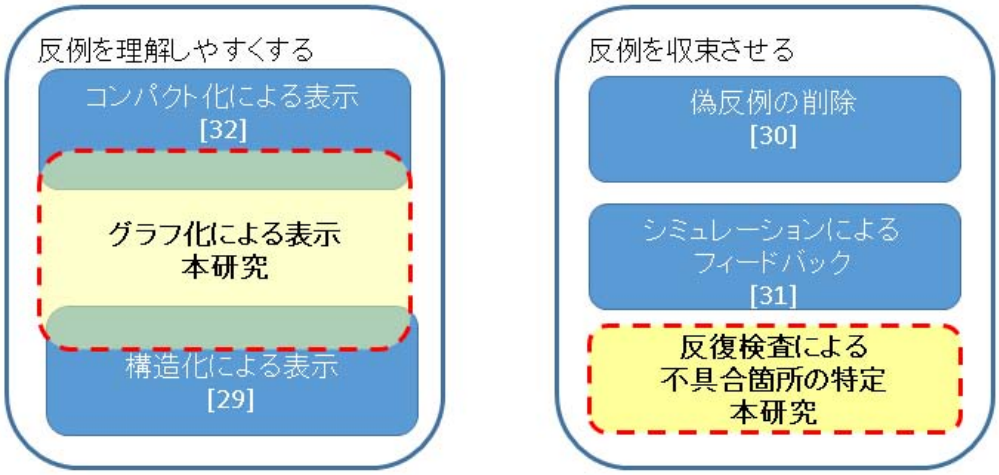


図 3-5　関連研究比較図　反例解析

既存の「反例を理解しやすくする」研究では理解しやすくするために余分な状態遷移数を減らしてコンパクト化する研究[32]と複雑な状態遷移を構造化して見やすくする研究[29]が行われている．本研究では，状態遷移とモデルを照らし合わせて理解するようなことをしなくても不具合原因を想定できるようにするため，UPPAAL の反例を解析して状態遷移の特徴がわかるようにグラフ化する試みを行っている．

また反例は満たされない状態になるひとつの過程を表わしているに過ぎないため，既存の「反例を収束させる」研究では，不要な反例を淘汰して収束させている．モデル上では成立するが実際の環境では成立しない反例である偽反例を見つける研究[30]等がある．本研究では同じ検査式から複数パターン出力される反例に対して基の検査式を修正して再度検査するというサイクルで問題となる状態遷移の特定を容易にする手法の提案も行っている．

Clarke ら[29]は木構造の反例がモデル検査における反例機能の自然で実用的な拡張であると主張し，反例を木構造として表示する手法を提案している．反例機能を拡張することが目的であり，今後の課題は実装と提示方法の実用的な改善とある．反例をより見やすい形にしようとする点は本研究と同じであるが，新たに記述方法を学ぶ必要がある．

Lerda ら[31]はシミュレーションの技術を用いて反例を検索することを提案している．組込みシステムを想定した研究であり，実測値から自分らが見つけたい反例になるようにシミュレーションを行い検索する．トレースファイルをグラフ化する点は本研究と同じであるが，シミュレーションを行うためには，あらかじめ目標が明確である必要がある．検査の目標が不明確なソースコードの不具合発見の検査には向いていないと言える．

Chan ら[32]は CTL 式からの反例生成のアルゴリズムを検討している. 彼らの実験結果は反例の大きさが過剰であることを示している．コンパクトな表現を得るために反例として作用することができる正規表現を生成するための簡単なアルゴリズムを提示しており，理解しやすい反例の作成に役立つと考える．

Cong ら[30]は偽反例の見つける手法について提案している．偽反例を効率よく検出することにより検査の精度をあげることを目指している．偽反例の判定は実際のシステムの運用との対比が必要であり難しいため，判定の簡易化は本研究でも検討すべき課題と認識している．

3.6 上流工程におけるモデル検査

システム開発の初期段階において仕様に関する検証を行い, 仕様誤解や仕様の実現可能性の不具合を上流工程において早期発見して修正することは重要である．上流工程で早期に不具合を発見すれば，手戻りによる工数増大とスケジュール遅延を防ぐことが可能になる．ここでいう上流工程とは主に要件定義段階を指す．しばしば変更されるユーザの曖昧な要求から作成した要求定義を限定された時間内で検証することは難しい. 本研究では UML モデルのうちアクティビティ図，ステートマシン図をモデル化して検査を行う．UML モデルから作る検査モデルは検査支援ツール UML2UPPAAL により自動生成するため頻繁に変更が入っても対応が容易である．検査式も自動生成される．

矢竹ら [61]はソフトウェア開発の上流工程において，複数のオブジェクトの協調動作の中で，オブジェクトの状態が不変条件を満たすかどうかを定理証明する手法を提案している．しかし，実現するには，非常に多くの厳密な定義が必要となる．要求分析段階のような不明確・不確定な要素が多い段階で，厳密な定義を行うことは難しい．

Choi ら[72]は Web ベースのシステム上におけるページの遷移の仕様とフローチャート間の整合性を検証する方法を提案した．これは画面遷移図とフローチャート間の整合性に特化した検証である．画面遷移図では画面 a→画面 c が可能だがフローチャートでは画面 a →画面 b→画面 c となる場合等が検出された．

Bose[33]や Jing [34] の研究では，UML モデルを PROMELA へ変換し，SPIN を利用

して，モデル検査を行う方法を提案している．しかし，検査を行うためには，一般の開発者が直接モデル検査ツールを操作する必要があり，検査者は UML と SPIN の両方の知識を要求される．本研究で用いる検査支援ツール UML2UPPAAL は UML の知識だけで検査を実行できる．

Rik[66][74]はアクティビティ図に対してのモデル検査手法を提案している．検査対象はアクティビティ図の整合性及び固有の仕様(アクティビティ図上のアクションの実行順序等)の整合性である．アクティビティ上における個々のアクションのつながり方を検査するもので，仕様が持っている性質を検査するものではない．

Konrad ら[44]は電子制御ステアリングシステムのタイミングに関する仕様について，タイミング情報を付加した UML モデルに対して SPIN を用いて分析をしている．SPIN モデルは Dwyer[25]の仕様パターンを使って定義している．検査者はプロパティ仕様パターンの意味を理解する必要があり，学習が必要である．

3.7 モデル検査ツール

以下に主なモデル検査ツールの特徴，適用例を述べ，本研究において利用するモデル検査ツールとして UPPAAL を採用した理由を述べる．

3.7.1 UPPAAL

モデル検査ツール UPPAAL[1][40][78][81] はスウェーデンの UPPSALA 大学とデンマーク AALBORG 大学によって開発されたリアルタイムシステムのモデリング, シミュレーション，検証のための統合ツールである．時間的制約を扱うことができるのが大きな特徴である．エディタでシステムを構成するオートマトンを記述し，グラフィカルなシミュレータによって分かり易く確認することが可能になっている．デッドロックなどの性質や，オートマトンがある状態に到達するかどうかの到達可能性のチェックができる．与えられた論理式が満たされないと，その満たされない状態系列を反例として出力する．UPPAAL モデルは状態を表すロケーションとその状態遷移を表すエッジから構成される．

エッジには遷移の条件である「ガード」， 遷移に伴う「更新」，他プロセスとの「同期」，プロセス内の変数に選択的に値を決定する「選択」が定義できる．学術目的での使用は無償だが，ビジネスで使用する場合は有償となる．モデル駆動開発（MDD）のモデルとして UPPAAL を使う研究[43]もある．性質の検証を行う検査式は CTL のサブセットである TCTL (timed computation tree logic)を用いる．

UPPAAL の事例として以下のものがある[6]（形式手法の実践ポータルの事例）.

・JAXA - 人工衛星の姿勢制御ソフトウェア： モデルの不変条件や到達性を検証

3.7.2 SPIN

SPIN[7][42][53][79][80] はシステムの仕様記述に PROMELA(Process Meta Language) という高水準言語を利用する．UML モデルを PROMELA へ変換し，SPIN を利用してモデル検査を行う方法を提案している研究がある[33][34]．時間的な制約定義はできない．オープンソースである．性質の検証は LTL 式で行う．

SPIN の実施事例として以下のものがある[6]（形式手法の実践ポータルの事例）[69].

・ Selex Communications - 船舶通信システム：デッドロックや到達不能パスの検査
・日立ソリューションズ - Blu-Ray ディスク：デッドロックの検出
・日立ソリューションズ - 入退室管理システム：要求の妥当性確認
・Logica Nederland B.V. - オランダ運河のマエスラント防潮可動橋（水門）：ライブロック及びデッドロックの検査

研究としては SPIN を拡張して C プログラムの動的なメモリ割り当てを検査する研究[51]や SPIN の反例からテストケースを作成する研究[52]等もある．

3.7.3 Java PathFinder（JPF）

Java PathFinder（JPF）[35][42][53]は実行可能な Java バイトコードよりプログラムを検証してデッドロックとアサーションエラーを検出する．JPF は複雑で大規模なソースコードに対しては"状態爆発"の注意が特に必要である．NASA で開発されたオープンソースソフトウェアであり，火星探査機の制御システムの検証などにも使われた実績を持つ．Markosian ら[10]は NASA のフライトシミュレータの検査を JPF により行っている．検査対象は Java のみである．ソースコードの解析に特化したものといえる．

また一つ前の状態に戻って別の手順を試す BackTrack の機能がある．Java のクラスファイルは，Java VirtualMachine によって解釈され，実行されるユーザ定義による検査も可能であり，基本的には安全性の検査となる．時間的な制約定義はできない．

Tudose ら[48]はアサーションによるエラーの検出だけではなく，その状態に至る状態遷移を表す試みを行っているが，実行ステップをシンボリックに表示するだけなので他の有限状態マシンで記述されているモデル検査ツールに比べると出力内容の表現は限定的である．JPF の事例として以下のものがある[6]（形式手法の実践ポータルの事例）．

・NASA-フライトシミュレータの開発：

3.7.4 NuSMV

NuSMV[8][76][80]は状態遷移図や状態遷移表のモデル検査に適している．大規模なモデルを高速で検査することができる．カーネギーメロン大学，イタリアの大学と研究機関 FBK-IRST(Fondazione Bruno Kessler-Istitutoper la Ricerca Scientifica e Tecnologica)によって SMV を拡張して開発された．

二分決定木（Binary Decision Diagram：BDD）ベース だけでなく，充足可能性判定（Satisfiability：SAT） ベースの有界モデル検査(有限長でのモデル検査)もできるようになっている．また CTL, LTL の双方を性質記述に利用できる．オープンソースである．

NuSMV の実施事例として以下のものがある[6]（形式手法の実践ポータルの事例）．

・Rockwell Collins 社 - パイロット用のディスプレーを管理するシステム：563 個の性質に対して検査

3.7.5　ツールの比較

以下で主なモデル検査ツールを比較する．ツールの比較内容の詳細は表 3-1 に示す．

検査対象はJPFのみJavaに限定されるが他のツールについてはそれぞれの書式で作成される検査モデルとなるため特に制限はない．GUI を用いたモデルの作成，結果の表示については標準でできるのはUPPAALのみで他は拡張アプリを利用すれば可能になるものがある．部分的なモデル検査については，JPF はバイトコードを検査するため難しく，他については部分的な検査モデルの作成により可能である．時間的制約を扱えるのはUPPAALのみである．検査者が定義した特性をモデルとして追加することは基本どのツールでも可能である．

表 3-1 モデル検査ツール比較表

項目	Model Checking tool			
	UPPAAL	Java PathFinder(JPF)	SPIN	NuSMV
検査対象	GUI 上でモデル記号により記述されたモデル	JavaByte Code	PROMELA 記述言語により定義したモデル	NuSMV 記述言語により定義したモデル
検査対象の制限	特になし	Java のみ	特になし	特になし
GUI による作成	オートマトンを GUI により記述	ByteCode 解析のため不可	専用言語 PROMELA により記述するため不可	不可
GUI による結果表示	記述したオートマトンに結果表示	拡張アプリによりオートマトン表示	拡張アプリによりオートマトン表示	不可
部分的なテストの容易さ*1	GUI のオートマトン抽出により可能	ByteCode であるため切出し困難	PROMELA の記述抽出により可能	NuSMV の記述抽出により可能
時間制約を考慮した検査	クロック変数の利用により定義可能	時間定義不可	時間定義不可	時間定義不可
使用する時相論理式	CTL		LTL	CTL と LTL
検査特性の追加	新規モデル追加	Property, listener class に定義	新規モデル追加	新規モデル追加

*1：複雑な検査モデルの一部分を切り出して検査する場合の容易さ

JPF はバイトコードを検査するため，デバック要素が強い，他のツールについては有限オートマトンによる検査モデルを構築して検査するというスタイルは同じであり，機能的には大きな違いはないと言える．但しモデルの定義方法は大きく違うため検査する内容の性質に合わせて記述しやすいツールを選ぶことが重要である．

3.7.6 UPPAAL を採用した理由

本研究で使用するモデル検査ツールを UPPAAL とした理由を以下に述べる．一般的に UPPAAL の特徴というと時間制約を扱えることがあげられるが他にも優れた特徴がある．

本研究では，一般的な開発者でも検査可能にすることを検討していたため，モデル検査を支援するツールを開発することが必須であった．

そのため以下の条件を重視した．

・デバックがしやすいこと

作成した検査モデルがうまく動作しない可能性がある．また想定した通りの動きをしない可能性もある．その場合モデル検査ツールでデバックが必須となるが，GUI で動作を確認できる UPPAAL は便利である．

・出力結果の編集がしやすいこと

反例を解析する場合，ロケーションに定義されている変数を抽出編集する必要がある．その場合ツールを使って編集することになる．UPPAAL は GUI 上の定義を xml ファイルで出力する．xml の構造さえわかれば記述が容易で，編集・分析もしやすく，既存のデータ変換技術も利用できる．従ってモデル作成・反例解析ツールも作りやすい．

・状態遷移が表現しやすいこと

SPIN，NuSMV は規定の言語で直接作成者が書くため状態遷移の記述方法も，プログラミングと同じでいろいろな書き方ができる．UPPAAL は GUI の記述を xml で保存するので状態遷移の記述方法は 1 通りしかない．従って xml の自動生成も容易である．

本研究が提案するソースコード検証手法は,仕様モデルを利用することにより，仕様モデルの特定の状態への到達を検査すればよいため，複雑な検査式を必要としない．

そのため UPPAAL の欠点としてよくあげられる検査式の表現の制限や計算速度は問題にならないと認識している．但し今後，検査式の制限や計算速度が問題になってきた場合は，他のモデル検査ツールへの変更も考えている．モデル検査ツールは検査器として利用しているだけなのでツールのインターフェースを変更すれば他のモデル検査ツールでも対応可能である．

3.8 本章のまとめ

本章ではモデル検査の研究動向とモデル検査ツールについて説明した．研究動向は主に２章であげたモデル検査を使う上での問題に対する研究を取り上げた．それぞれの研究の内容と本研究との比較を述べた．

またモデル検査ツールについてはよく使われるツールについて，その特徴の説明と実用事例を説明し，取り上げたツールの比較と本研究でモデル検査ツール UPPAAL を利用することにした理由を述べた．

4　ソースコード検証手法

4.1 検査手法の概要

本研究が提案するソースコード検証手法は，モデル検査技術を用いて，ユーザがシステムの仕様に基づき作成した検査モデル(仕様モデルと呼ぶ)とソースコードから制御フローに基づき作成した検査モデル（ソースコードモデルと呼ぶ）を結合して検査することで，仕様とソースコード間の不一致を発見する手法である．本研究では業務システムを念頭に置き，ユースケースで定義されるレベルのシステムがもたらす業務機能の振舞いを検査の対象としている．業務機能の振舞いはシステム内の処理とユーザの手作業の組み合わせで表される動作の連なりである．従って，この業務機能の振舞いはユースケースを構成するシステムの基本機能を表すアクションと基本フロー，事前・事後・分岐条件であり，設計書に記述されている．図 4-1 は提案手法の概要図である．使用するモデル検査ツールは UPPAAL である．

検査を行うまでの手順は以下の(1)～(8)になる．(1)～(3)は基本的に手動で作成する部分であり，定式化する方法を検討中である．(4)は支援ツールにより自動で作成する部分である．(5)～(6)，(7)～(8)は手動で作成，定式化しつつある部分である．

基本的に手動で行っている部分は処理内容についての判断が必要になるため難しい部分である．特に(1)～(3)の部分については設計書の記述内容によって作業量が大きく変わる．これらの部分については今後ツールを作成して支援する，手動で行う場合でも定式化させていく予定である．

(1)業務機能の振舞いは設計書に記述されているので，そこからアクションと条件（事前・事後・分岐条件），実行結果（状態）を抽出する（図 4-1①).

(2)(1)の内容をデシジョンテーブルで整理する(図 4-1②).

(3)この整理結果を基に業務機能の振舞いを状態遷移に置き換えて仕様モデルを作成する(図 4-1③).

(4)ユースケースレベルの機能(アクション)はメソッドに対応すると仮定できるので，ソースコードモデルはメソッド単位で機能を捉え，ソースコードの階層構造に基づいたモデル変換を検査支援ツールにより行い，検査モデルを作成する(図 4-1④).

(5)メソッドをソースコードと仕様の接点として取らえて，これを機能の中継のみを目的とするアクションモデルとして定義する(図 4-1⑤).

(6)ソースコードモデルのメソッド呼び出し部と仕様モデルの状態変化を表す部分をアクションモデルで紐付けして検査モデルを構成する(図 4-1⑥).

(7)仕様モデルが想定外の状態遷移を起こさないことを検証する検査式を作成する(図 4-1⑦).

(8)モデル化されたステートメントは検査モデル上で状態を表すロケーションと紐付く．この検査モデルを検査式により検査する(図 4-1⑧).

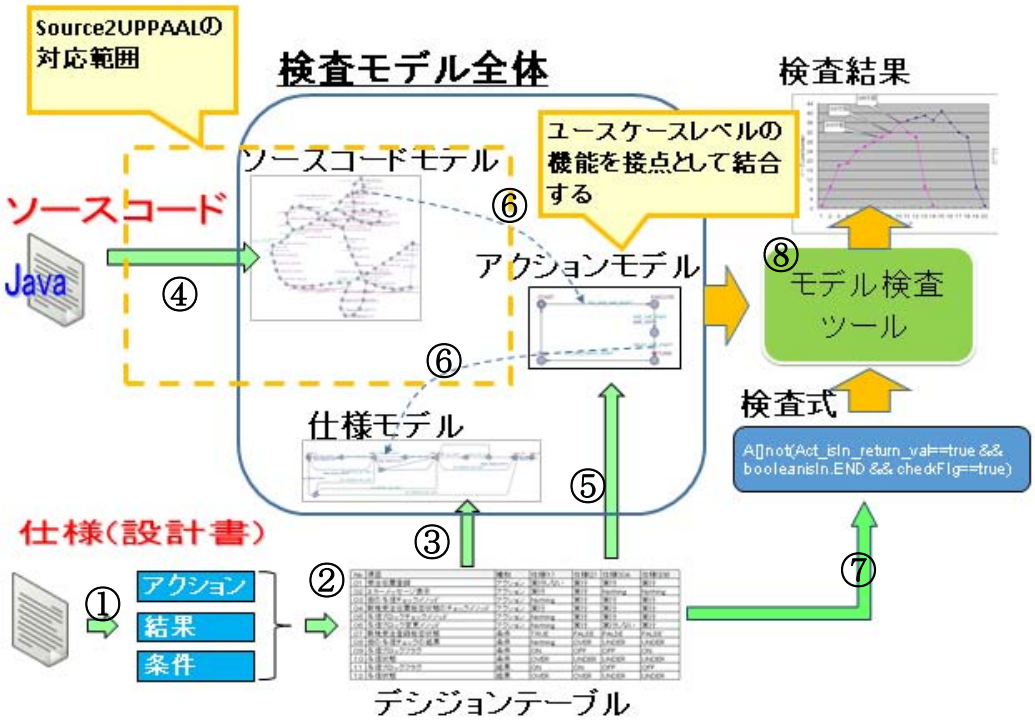


図 4-1 提案手法　処理手順

検査は以下の手順で行う．図 4-1⑥で構成された検査モデルはユースケースレベルの機能を表したものになる．手順１はこの検査モデルがソースコードの制御フローの動きを再現できていることを到達可能性の検査で確認する．手順２は作成した検査式による検査で業務機能の振舞いに合致しない状態遷移が起きないことを確認する．ただしアクションモデルの基になったメソッドのロジックに問題がある場合，これまでの検査では不具合は発見できない．その場合は，アクションモデルをソースコードモデルへ再変換して，新たなソースコードモデルを追加した検査モデルで上記の手順を繰り返す．そうすればモデルが段階的に詳細化され，該当メソッドのロジックについても検査できる．本研究はユースケースレベルのシステムがもたらす業務機能の振舞いの仕様をデシジョンテーブルで整理してモデル化し，想定外の振舞いがないことを検証する．従って振舞いの仕様以外については検査対象外となる．

また eclipse プラグインを用いてモデル検査における非専門家が不具合の原因を特定するための検査支援ツール Source2UPPAAL を開発した．このツールは検査モデルおよび検査式を作成するモデル検査の知識がなくてもソースコードの制御フローを基に UPPAAL の検査モデルを生成できるものである．

4.2 ソースコードを検査モデルへ変換するロジック

4.2.1 検査モデルへの変換

まず変換先となる UPPAAL モデルについて説明する．UPPAAL モデルは状態を表すロケーションとその状態遷移を表すエッジから構成される図 4-2 のようなプロセスの集合体として定義される．エッジには遷移の条件である「ガード」，遷移に伴う「更新」，他プロセスとの「同期」，プロセス内の変数に選択的に値を決定する「選択」が定義できる．設定した値のリストからランダムに値を選択することにより変数の値を非決定的に設定する定義ができる．

図 4-2 の START，LOC1 および LOC2 はロケーションと呼ばれるシステムの状態を表わす要素である，条件式である「i1== 0」と「i1>0」は「ガード」に記述し，更新式である「flg=true」と「flg=false」は「更新」に記述する．

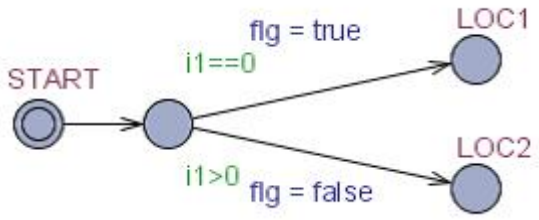


図 4-2 UPPAAL モデル

「同期」は複数のオートマトン間のチャネルを提供する．図 4-3 は同期処理の例である．状態遷移は図上の①～⑤の順番で行われる．「sen」「ret」は同期処理のためのチャネルの変数である．変数名は任意に設定できる．「sen!」が実行されると「sen?」に同期されて状態遷移が開始される．同様に「ret!」が実行されると「ret?」に同期されて状態遷移が開始される．「？」で同期呼び出しを待っている状態であり状態遷移は停止している．「！」で同期呼び出しを行った場合，呼び出し側の状態遷移は継続される．また同名複数の「？」がある場合はどれが同期されるかは非決定である．全チャネルに対して同時に同期呼び出しを行う設定も可能である．

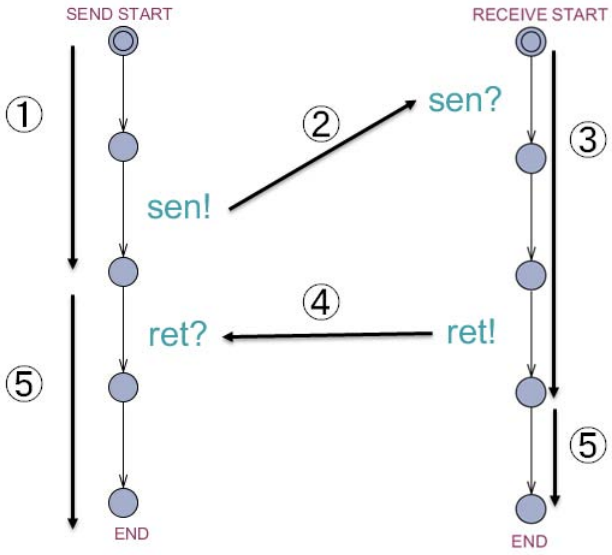


図 4-3 同期処理の動き

モデル検査ツールでは非決定性の設定が可能である．制御構造の条件式に非決定性の値を与えるなどして検査モデルを複雑にせずに記述することができる．UPPAAL では整数型と論理型の変数に非決定的に値を設定することができる（図 4-4）．エッジの編集画面の「選択」で非決定に発生させる値の範囲を設定する．「a=int[0,10]」とあれば 0~10 の値が非決定にこのエッジ内でのみ有効な変数「a」に設定される．そして「更新」で「a」の値を UPPAAL 上の変数に設定する．ソースコードのステートメントは UPPAAL のロケーションとエッジに置き換えられる．本研究ではこの非決定の設定は検査支援ツール Source2UPPAAL 上で行い検査モデルに反映させるため，検査者が直接 UPPAAL 上で編集する必要はない．

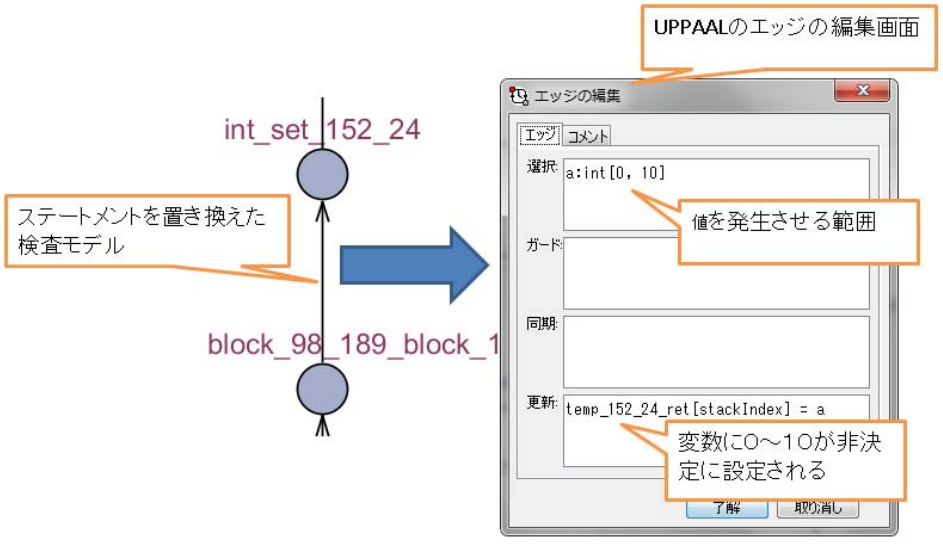


図 4-4 非決定な値の設定

モデル検査においては，システムの状態を識別し，その状態遷移として対象システムの振舞いを有限状態モデルとしてモデル化する．UPPAAL では，状態の遷移をロケーションとエッジのつながりで表現する．状態が持つ性質は整数型と論理型のグローバル変数で表わすことができる．ソースコードはステートメント単位で逐次実行され，制御構造に応じて，その状態を遷移している．そこで，ソースコードの UPPAAL モデルへの変換は次のように行う．ソースコードのステートメント単位の実行ステップを状態遷移と捉え，これを UPPAAL モデル上でロケーションとエッジに変換する．次にソースコードの上で定義されている整数型と論理型の変数はそのまま, UPPAAL モデルのグローバル変数に置き換える．これらの変数への代入式は，UPPAAL モデルにおける変数の「更新」に置換する．

このようにしてソースコード上に現れる状態遷移を UPPAAL モデルでも同等に表現できる．さらに，制御構造に従って状態を変化させる値（整数型と論理型の変数に対する値）を与えることで，ソースコードの振舞いを再現する．

以下の変換規則によりソースコードを UPPAAL モデルに変換するものとする．全てのメソッドを一度に変換するのではなく，検査者が 1 つもしくはいくつかのメソッドを選択して変換する．

UPPAAL モデル内で，状態はロケーションと整数型及び論理型のグローバル変数により定義される．そこで変換ルールを以下のように定義する．

・ソースコードの各ステートメントは UPPAAL モデル内のロケーションに変換する
・整数型および論理型の変数は UPPAAL モデルでグローバル変数に変換する
・整数型および論理型の値への代入式は UPPAAL モデルの更新式に変換する
・各メソッドおよび API 呼び出しに UPPAAL の選択式を用いて単数もしくは複数の非決定的な値を変数へ割り当てる．これらの値は取りうる状態を表すものである
・状態遷移の分岐条件は上記の変数を使用する

以下に簡単な変換例を示す（図 4-5）．Java ソースコードは入力した数値を判別して print 文を実行するものである．変換されたモデルは IF 文による分岐する制御フローが反映される．変数 val は UPPAAL 内のグローバル変数 var_0_133_10 に変換され条件分岐に使用されている．var_0_133_10 への値の設定は UPPAAL の選択式による非決定性の機能を用いて設定している．今回は仮に 0〜10 の間の値を設定する．

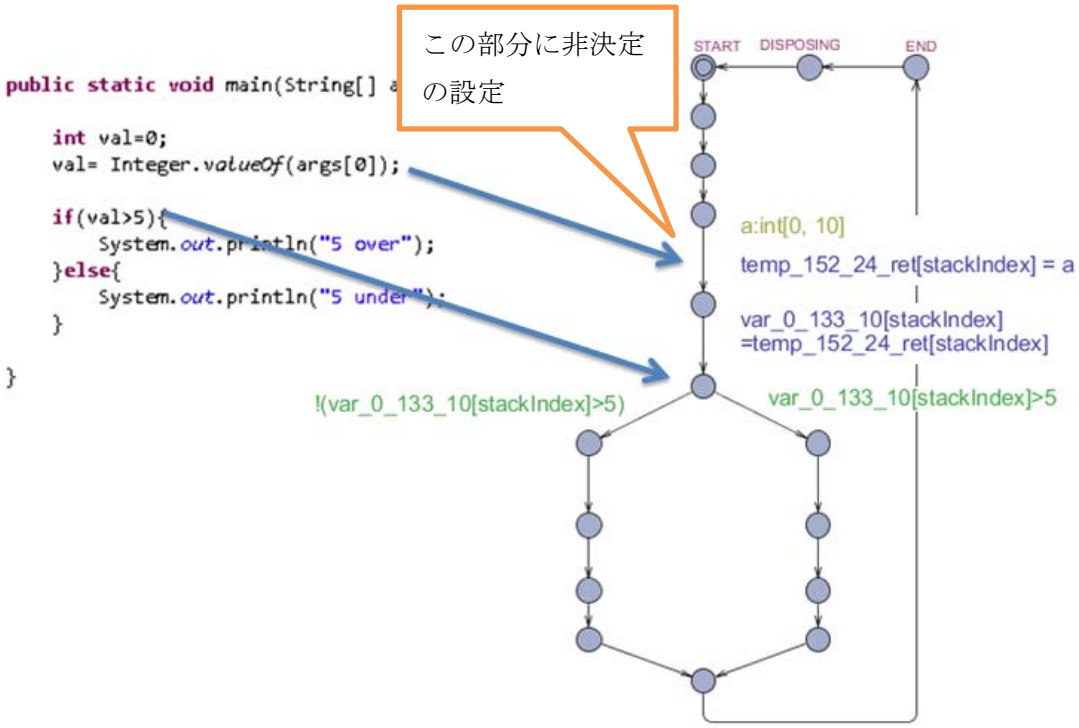


図 4-5 ソースコードからのモデル変換

検査モデル上で状態識別に用いている変数の値を与えることにより，例えば if 文の条件判定結果に真偽値による非決定性を与えて，異なる状態遷移を検査することや，整数で制御される複数の状態に非決定性を与えて検査を行うことができるため，変数として使える型が整数型・論理型でも本手法は妥当な方法であると考える．

すなわち，ソースコードを本稿で示した変換方法により，UPPAAL モデルに変換することは可能であるが，モデル検査を実行するためには，検査の範囲を限定できる機能と，非決定性を与える手段が必要となる．Source2UPPAAL はこれらの機能を提供し，規模に依存しないソースコードを検査対象として扱えるようにしている．

さらに，ソースコードを UPPAAL モデルに変換することによって，到達可能性のように，必ずソースコードのある箇所へ到達することや，無限ループのような「ソースコードの仕様に関する性質の定義を必要としない検査」は無限ループの特性を表す仕様モデルを UPPAAL モデルのループ構造を制御する式に割り当てることにより可能になる．

無限ループのケースでは同じプロセスを予想回数より多く繰り返された場合に検知できるモデルを定義した．この仕様モデルを UPPAAL の同期を使用して疑わしい制御フローに割り当てて，無限ループが発生しないことの安全性を検査することにより無限ループの原因を特定できた．

しかし，「ソースコードの仕様に関する性質の定義を必要とする検査」は，このようには検査できない．

一般的なプログラム構造からわかる式ではなく，この性質を割り当てる式を対象システムの仕様に従って決めなければならない．そこで，後述するデシジョンテーブルを用いたソースコードの仕様に基づいて満たすべき性質を検査する方法を提案した．適用事例でこの手順の詳細を説明する．ここでは，ソースコードの仕様に基づく満たすべき性質の UPPAAL モデルと，この性質を定義するソースコード上のメソッドの UPPAAL モデルをデシジョンテーブルから生成する．UPPAAL のモデルでは，メソッドの呼び出しをチャネルによる同期によって定義することができ，Source2UPPAAL を用いて，メソッド呼出しに非決定性を与えていた代わりに，この後者のモデルを割り当てることで，ソースコードのモデルと，ソースコードの仕様に基づく満たすべき性質のモデルを同期させ，ソースコードが満たすべき性質の検査を行うことができる．

一般的にソースコードは，自然言語で記述された仕様書に基づいて開発されている．本研究では検査者が文書で記述されたプログラムの仕様を理解できることを前提としている．

4.2.2 検査式の作成

検査式の種類は大きく 2 種類ある．ひとつは特定の状態への到達可能性を検査する式である．もうひとつは，検査者が仕様を理解して認識できたシステムの振舞いについて，そのシステムの振舞いが想定外の動きをすることがないか検査する式である．

到達可能性の検査は次のようになる．図 4-6 の検査モデルを例にすると以下のようになる．検査式の意味は「ロケーション block_158_94_START_ELSE にいつか到達する」である．従って期待される検査結果は「属性は満たされました」であり，もし「属性は満たされませんでした」となった場合は到達できない場合があることになる．ソースコード上に到達できない場所があることになる．自動生成されるモデル名＋ロケーション名により定義されるので検査式の自動生成が可能で，ロケーションが多数あっても対応可能である．

E<> SYSTEM_Test0925_main(0).block_158_94_START_ELSE

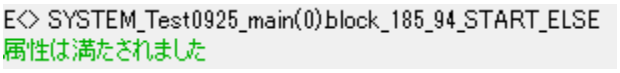


図 4-6 到達可能の検査

システムの振舞いを検査する検査式は，基本的にシステムの振舞いとしてあり得ない状態になることがないことを確認する検査式となる．仕様モデルをソースコードモデルに紐付けした状態で検査を実行する．例えば無限ループを検査する場合は無限ループの状態を定義したモデルを追加する．図 4-7 のモデルは想定繰り返し回数を超えた場合ロケーションが START－＞END へ状態遷移する．cnt が繰り返した回数，limit が想定繰り返し回数である．

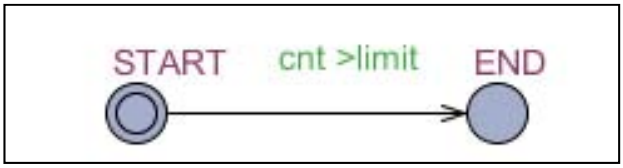


図 4-7 無限ループ判定用モデル

従って検査式は次のようになる.

A[] not LOOP.END

検査結果が「属性は満たされませんでした」となった場合，無限ループが発生することがわかる.

4.3 検査支援ツール　Source2UPPAAL

4.3.1　概要

ソースコードを一度に UPPAAL モデルに変換すると状態数が非常に多くなり，検査を実行できないことがある．そこで，検査モデルが必要以上に大きくならないように段階的な UPPAAL モデルへのモデル化を実現する必要があり，本稿では Source2UPPAAL により，ソースコードのメソッドの階層構造に基づいた段階的なモデルへの変換，制御構造に従って状態を変化させる値の式への付値を容易にできるように支援する．

ツールは eclipse[36]のプラグインとして作成した．検査する対象は Java ソースコードとし，UPPAAL の検査ファイルおよび，到達可能性を検査する検査式を作成する．検査はメソッド単位で行う．ツールはソースコードを AST パーサ[37]で解析し該当プロジェクト内のソースコードとそのメソッドをツリー構造で表示する．検査者はその中から検査対象のメソッドを選択しモデル化する．メソッド内のステートメントはモデル化対象項目として表示される．

モデル化の対象になる Java のソースコードは，いくつかのクラスを含むパッケージで構成されている．後述の 4.3.2 で述べる変換ルールに基づいてソースコードから作成される検査モデルであるソースコードモデルは半自動的に生成される．図 4-8 は支援ツールを用いた変換プロセスを示している．図 4-9 に示すサンプルソースコードを使用して変換プロセスを説明する．

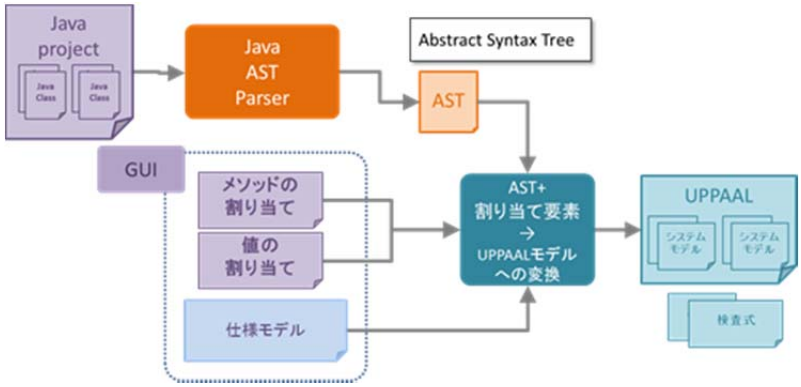


図 4-8 変換処理の構成

まず，図 4-9 に示したソースコードは，AST パーサを使用して抽象構文木（AST）に変換される．図 4-10 はソースコードの抽象構文木を示している．各ノードはメソッド，if 文，開始ノードと終了ノードから構成される．

```
public static void main(String[] args) {

    boolean flg=false;
    int val=0;
    Check chk = new Check();
    val= Integer.valueOf(args[0]);
    flg= chk.chkVal(val);
    if(flg==true){
        System.out.println("Even number");
    }else{
        System.out.println("Odd number");
    }

}
```

```
public class Check {
    public boolean chkVal(int inVal){
        boolean flg=false;
        if(inVal%2==0){
            flg= true;
        }else{
            flg= false;
        }
        return flg;
    }
}
```

図 4-9 サンプルソースコード

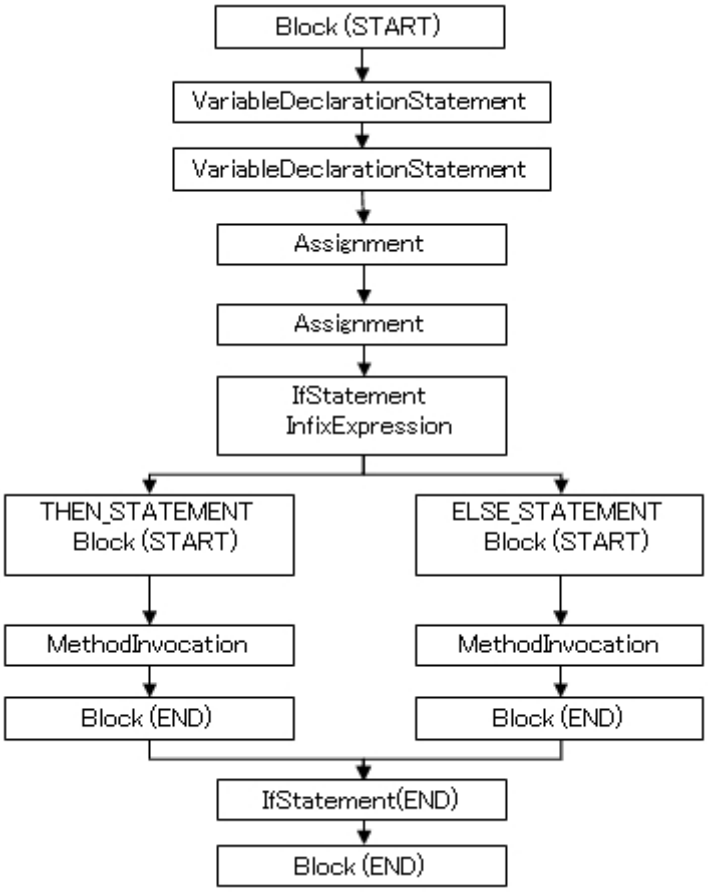


図 4-10　AST から作成されたグラフ

図 4-11 に示すように，グラフは，UPPAAL モデル内の位置に，グラフ内のノードをマッピングすることにより，UPPAAL モデルに変換される．

二つのフィールド変数，flg と val は UPPAAL モデルでグローバル変数に変換される．

これらの変数が整数型または論理型(UPPAAL の boolean)であるので，それらは UPPAAL モデルのエッジの「更新」へ無変更で移行できる．

一方，「char CH1='A'」「double d =0.0」等の変数は UPPAAL モデル内のグローバル変数として使用することはできない．しかしながら，代入式である「ch1 = line1.charAt(1);」や中置式である「CH1== 'B'」は，UPPAAL モデルでの真または偽の状態に抽象化でき，手動で非決定的な値をこれらの文を設定することができる．その結果これらのステートメントは UPPAAL モデル内の「選択」に変換される．

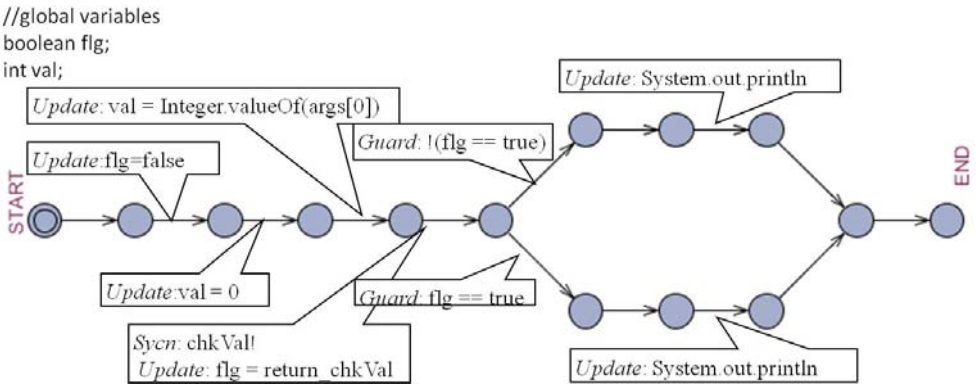


図 4-11 UPPAAL モデル

「System.out.println(“Even number”);」のようなステートメントは，アプリケーションのロジックには影響しないため，式のステータスとしては True を割当てればよい．

不具合と認められる業務機能の振舞いの条件分岐等に関係するメソッドがある場合，その不具合の原因が呼び出しているメソッド内にあると想定できる．例えば chkVal()のようなメソッドを UPPAAL モデルに変換する．第 1 の検証ステップでは戻り値の型に基づいて非決定的な値の設定を行う．Integer クラス内の"valueOf"などの API メソッドの場合，任意の整数のリストを設定する．

上記の様に非決定性を利用してメソッドのロジックを抽象化した場合，不具合が現れなくなる場合はそのメソッドも UPPAAL モデルに変換して同期処理で紐づけることにより検査モデルを構築する．

API メソッドの場合は，その API の仕様に基づいて期待される返値を返す等の動作を行うモデルを定義する必要がある．

4.3.2　変換ルールまとめ

表 4-1 の変換規則で，ソースコードの振舞いをそのまま UPPAAL モデルへ変換する．

表 4-1 変換ルール

Java の statement		UPPAAL の要素
boolean 型の変数宣言		bool 型の global 変数
その他の基本型の変数 数値型（byte、short、int、long、float、double）及び char 型		int 型の global 変数
参照型の変数		ツール上での割り当てなし⇒変換しない ツール上での割り当てあり⇒非決定の値の設定（true/false もしくは int 型数値），割当先の UPPAAL モデルからの返値(bool もしくは int 型)
int 型および boolean 型の変数への代入式		変換後の変数の更新式
expression	条件式の論理式	式内の変数がすべて int または boolean であればそのままガードとする．その他は値割り当てまたは仕様モデルの割り当てを行う．
	参照型のオブジェクトに対するメソッド呼び出し	channel による送受信行う状態遷移．代入文の右辺にある場合には，値割り当てまたは仕様モデルの割り当てを行う必要がある．
	参照型のオブジェクトに対するフィールドの参照	無変換
	インスタンスの生成	無変換または仕様モデルの割り当てを行う．

4.3.3　Source2UPPAAL の機能と画面

検査者のモデル化作業を最小限に抑えるために次のような機能を持つ検査支援ツール Source2UPPAAL を作成した.

- メソッドを選択し,グラフィカルユーザインタフェース（GUI）を使用してメソッドコールのツリー構造から検証する.
- GUI を使用してステートメントの非決定性の値の設定もしくは呼び出しメソッドの指定を行うことにより検査モデルを構成する.
- API やビジネスルールの仕様に基づいて定義した仕様モデルを紐付けることにより検査モデルを構成する.
- AST データに基づいてソースコードモデルを作成し，上記処理との組み合わせにより検査モデルと検査式を生成する.

変換作業が容易になるように作業は主にドラッグ＆ドロップによって行う．ツールの画面構成は図 4-12 のようになる．画面は左ペインと右ペインにより構成される．右ペインにはプロジェクト内にあるすべてのメソッドクラス単位で表示される．左ペインにはモデル化するメソッドを表示する.

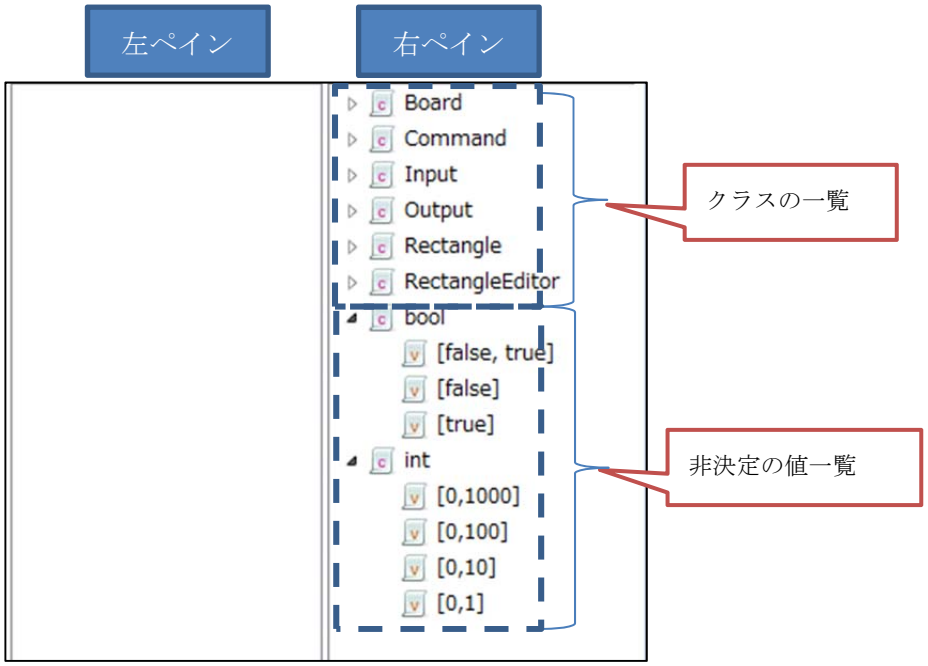


図 4-12 Source2UPPAAL 初期画面

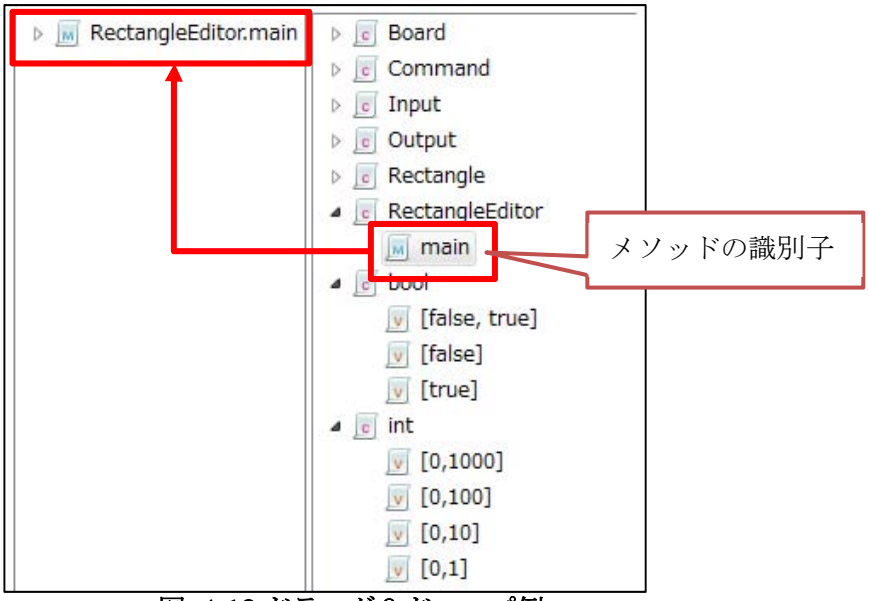


図 4-13 ドラッグ＆ドロップ例

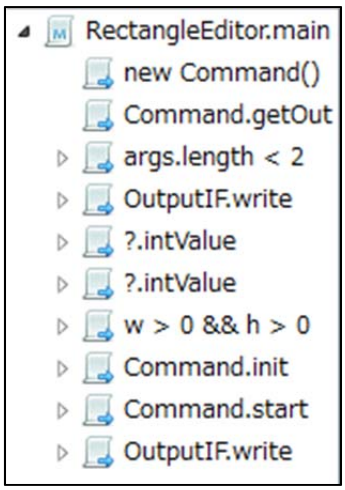


図 4-14 メソッドの展開

初期状態では，右ペインにプロジェクト内のメソッド一覧と非決定値の一覧を表示する，左ペインは空白とする（図 4-12）．

右ペインの識別子を左ペインへドラッグ＆ドロップすることにより左ペインにも識別子を表示する（図 4-13）．左ペインの識別子（メソッド）は内部にある識別子を展開できる（図 4-14）．

4.3.4 割り当て処理

Source2UPPAAL はプロジェクト内に定義されているソースコードファイルを AST パーサにより解析して抽象構文木を作成する．図 4-15 はツールの画面を表す．モデル化はメソッド単位で行うため，ツールの右ペインにはモデル化の候補となるメソッドの呼出し階層が表示される．さらに付値可能な非決定の値のパターンとプロジェクト内に登録されて

いる UPPAAL モデルが表示される．これは本論文での仕様モデルに相当する．

検査するには，最初に右ペインよりモデル化したいメソッドを選び，左ペインにドラッグ＆ドロップする．そうすると左ペインにはそのメソッドの内に定義されたメソッド呼出し，分岐条件，繰り返し条件が表示される，メソッドをまたがった検査をしたい場合は，左ペインにあるメソッド呼出しの式に右ペインにあるメソッドの階層構造の中からモデル化したいメソッドを選択して割り当てることにより当該メソッドを展開して，メソッドをまたがった変換を行うことができる．

以下に例を示す．図 4-17 は図 4-9 で提示したメインメソッドのソースコードと新たにその中で呼ばれているメソッド chkVal()を追加したソースコードである．このプログラムの機能は，入力した数値が偶数か奇数かを判定するものである．chkVal は判定機能を実装しており，メインメソッド内の図 4-17①で実行される．仮に正しい判定が表示されない不具合が発生した場合，このメインメソッドの検査は，まずこのメソッドの基本的な振る舞いを検査するため最初は図 4-18 に示す通り，右画面の Check.chkVal に非決定の値[false,true]を付値し，検査モデルを生成する．メインメソッド上の条件分岐の問題ならばこのモデルで到達可能性の検査をすれば原因を特定できるが，もし chkVal 内部に不具合があり正しい判定結果が返ってこない場合，この検査では原因は特定できない．その場合 chkVal 内部の振る舞いを，図 4-19①のように右画面にあるモデル化候補メソッドの chkVal を左画面の Check.chkVal に割り当てる．chkVal 内のモデル化の候補が左画面に展開されるので必要に応じて，その他の非決定性の割り当てを行う．この例では chkVal の基本動作を検査するため，図 4-19-②のように非決定値を割り当てることができる．

図 4-16 の検査式により検査を行う．検査式(図 4-16)の意味は「IF 文の分岐ポイントに到達した時点で，入力値が偶数の場合，変数 flg は false になる．または，入力値が奇数の場合，変数 flg は true になることは決してない．」である．検査結果は「属性は満たされました」となれば問題がなく，「属性は満たされませんでした」となった場合は不具合があることになる．

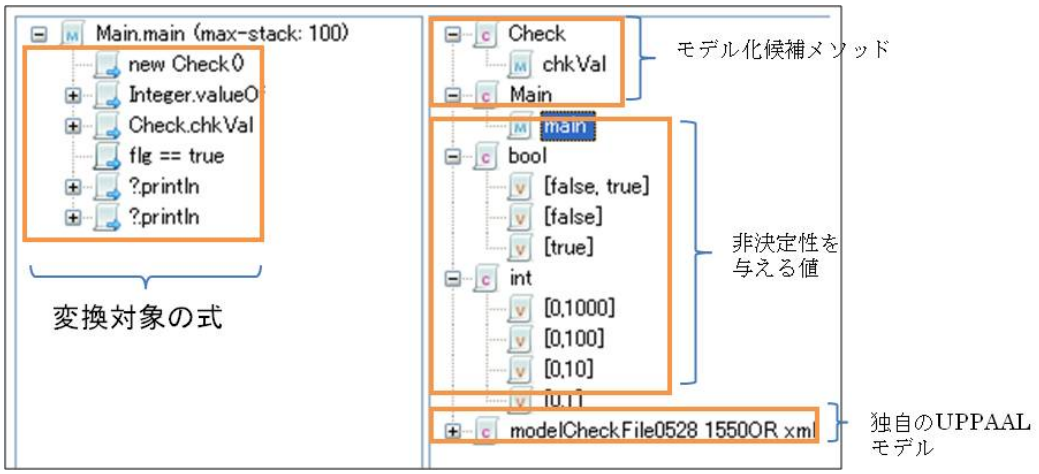


図 4-15 Source2UPPAAL の画面

同様の処理を繰り返していくことにより，必要に応じて検査モデルを詳細化していくことができる．各メソッドは UPPAAL 上では別のモデルとして定義され，チャネルを用いた同期処理により関連付けられる．また Java のライブラリのメソッドや業務ルール等，ソースコードが存在しない対象の振る舞いの検査は，図 4-20 のように，その機能を抽象化して作成した UPPAAL モデルを登録し，式にドラック＆ドロップで割り当てることにより検査することができる．

A[] not (SYSTEM_Main_main(0).block_215_106_START_IF &&((temp_161_24_ret[0]%2== 0 &&var_0_92_18[0]==false) || (temp_161_24_ret[0]%2> 0 &&var_0_92_18[0]==true)))

※.block_215_106_START_IF：IF 文の分岐ポイント
※temp_161_24_ret：入力値を格納する変数 val
※var_0_92_18：偶数奇数の判定結果を格納する変数 flg

図 4-16 検査式

```
public static void main(String[] args) {

    boolean flg=false;
    int val=0;
    Check chk = new Check();
    val= Integer.valueOf(args[0]);
    flg= chk.chkVal(val);
    if(flg==true){
        System.out.println("Even number");
    }else{
        System.out.println("Odd number");
    }

}
```

①

```
public class Check {
    public boolean chkVal(int inVal){
        boolean flg=false;
        if(inVal%2==0){
            flg= true;
        }else{
            flg= false;
        }
        return flg;
    }
}
```

図 4-17 サンプルソースコード

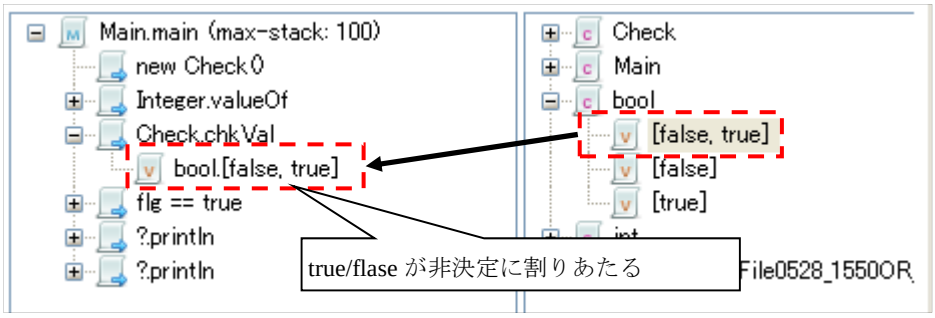


図 4-18 非決定性を付値する場合

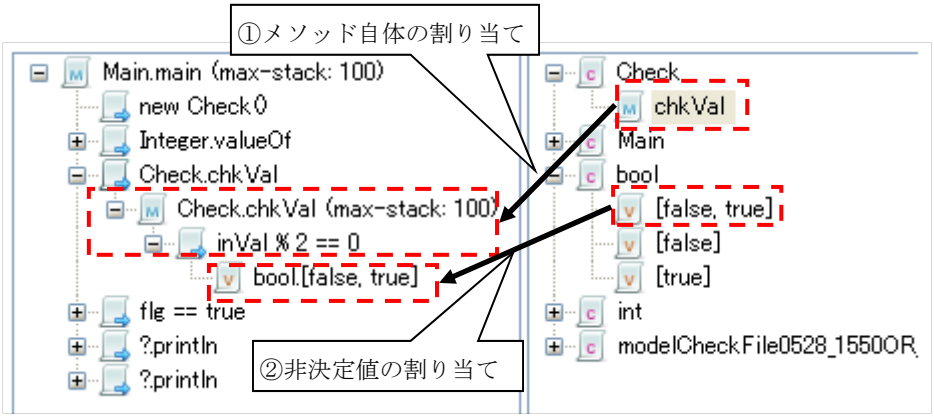


図 4-19 メソッドを割り当てる場合

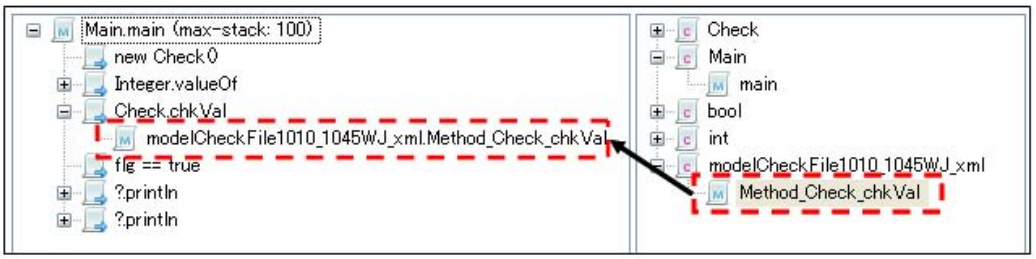


図 4-20　UPPAAL モデルを割り当てる場合

4.3.5　UPPAAL モデル作成

前節までの処理によりモデル作成の準備が整うので，モデル作成処理を実行する．左ペインにある識別子群が UPPAAL モデルとして生成される．生成されるファイルは UPPAAL の記述に準拠した xml である． 1 ファイルで出力され，内部的にメソッド単位でモデルが複数定義される．図 4-21 はソースコードを基に UPPAAL モデルを生成した場合の UPPAAL GUI 上での表示例である．

メソッド main()と Check()はそれぞれ独立した UPPAAL モデルとして作成される．ネーミングルールは"System_"+メソッド名（空白は"_"置き換え）である．MAIN とあるモデルは全体を統合するモデルである．これはデフォルトで必ず生成される．この UPPAAL モデルから状態遷移が開始される．

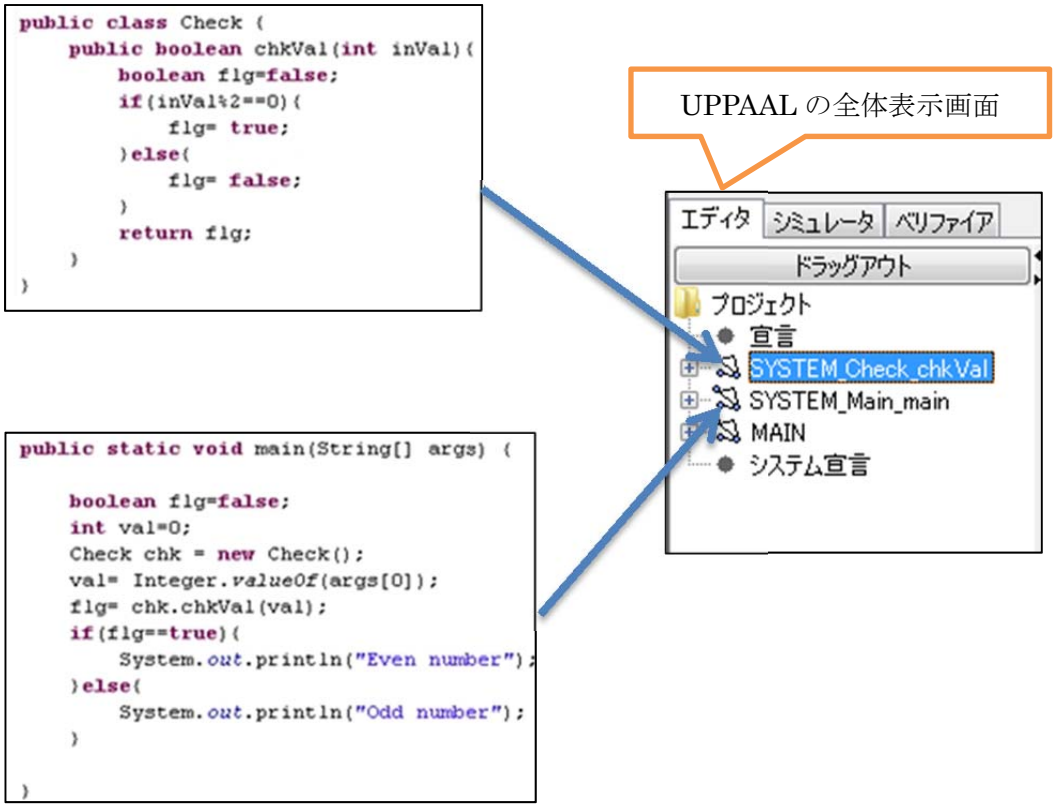


図 4-21 ソースコード変換

4.4 検査モデルの作成

4.4.1 作成手順概要

本研究の提案手法では検査モデルは次の手順で作成する．

・ソースコードより Source2UPPAAL を用いてソースコードモデルを作成する
・仕様書（設計書）に記述されている仕様をデシジョンテーブルで整理して仕様モデルを作成する
・ソースコードモデルと仕様モデルをつなぐアクションモデルを作成する
・アクションモデルを接点としてソースコードモデルと仕様モデルを結合して検査モデルを生成する

4.4.2 検査モデル全体の作成手順

ソースコードモデルの作成対象となるのは，検査対象となる機能を表わすメソッドである．その配下にあるメソッドで仕様モデルの状態変化に関係するものはアクションモデルに置き換える．仕様モデルの状態変化に関係ないメソッドは非決定性の値を付値する．これらの設定を検査支援ツール Source2UPPAAL 上で行い検査モデルを作成する．この作成されたモデルで不具合が検出できない等，充分な検査ができなかった場合，先ほどのアクションモデルをソースコードモデルへ変換して順次検査モデルを詳細化していく．

4.4.3 デシジョンテーブルの作成

開発において，業務機能を実現するシステム機能ごとに仕様は与えられるため，これをデシジョンテーブルにより整理する．デシジョンテーブルはビジネスロジックに含まれる条件を表形式で表現し，各条件のとり得る値の組み合わせを整理するのに用いるもので，単体テストをブラックボックステストで行う場合等，そのテストケースを洗い出すためのツールとして使用されることが多い．一つのテストケースでテストする機能は一つである．つまり機能についての実行条件，実行方法（アクション）がシンプルにまとめられているということであり，本研究において行うアクションとその結果としてのシステムの状態変化の紐づけに適している．

自然言語で記述された仕様のモデル化が難しい理由は，日本語の表現の自由度が高いため会話で伝える場合と同じつもりで仕様を記述してしまうことが大きい．そのため文章を処理しやすいように変換する．

はじめに主語と述語の関係を明確にするとともに文章を単文化する(図 4-22　手順１)．主語がない場合は非生物であっても追加する．さらに日本語で多く使われる「存在文」(・・ある)は関係性が不明確になりがちなので，モデル化する対象への影響が明確になる「行為文」(・・する)に人手で変換を行う(図 4-22　手順１)．次に動作動詞・可算名詞の抽出を行う(図 4-22　手順２)．英文における動作動詞がメソッドに該当し，可算名詞がクラスに該当するとする研究がある[38][39]．この考え方を基に仕様書(設計書)からモデル化対象を抽出

する．動作動詞はアクション(システムの動作)に置き換えられ，可算名詞はモデル化の対象に置き換えられると考えられる．ドキュメント内から動作動詞と可算名詞を抽出して一覧を作成する(図 4-22　手順３)．

次にモデル化すべき文章の特定を行う．作成した一覧を用いて動作動詞・可算名詞の両方が含まれている単文化した文章を抽出する(図 4-22　手順４)．可算名詞をモデル化の対象，動作動詞をアクション(システムの動作)と捉えられるので，この両方が記述されているということは，システムの状態変化を伴う振舞いを記述していると判断できる．この文章を基にデシジョンテーブルを作成する(図 4-22　手順５)．

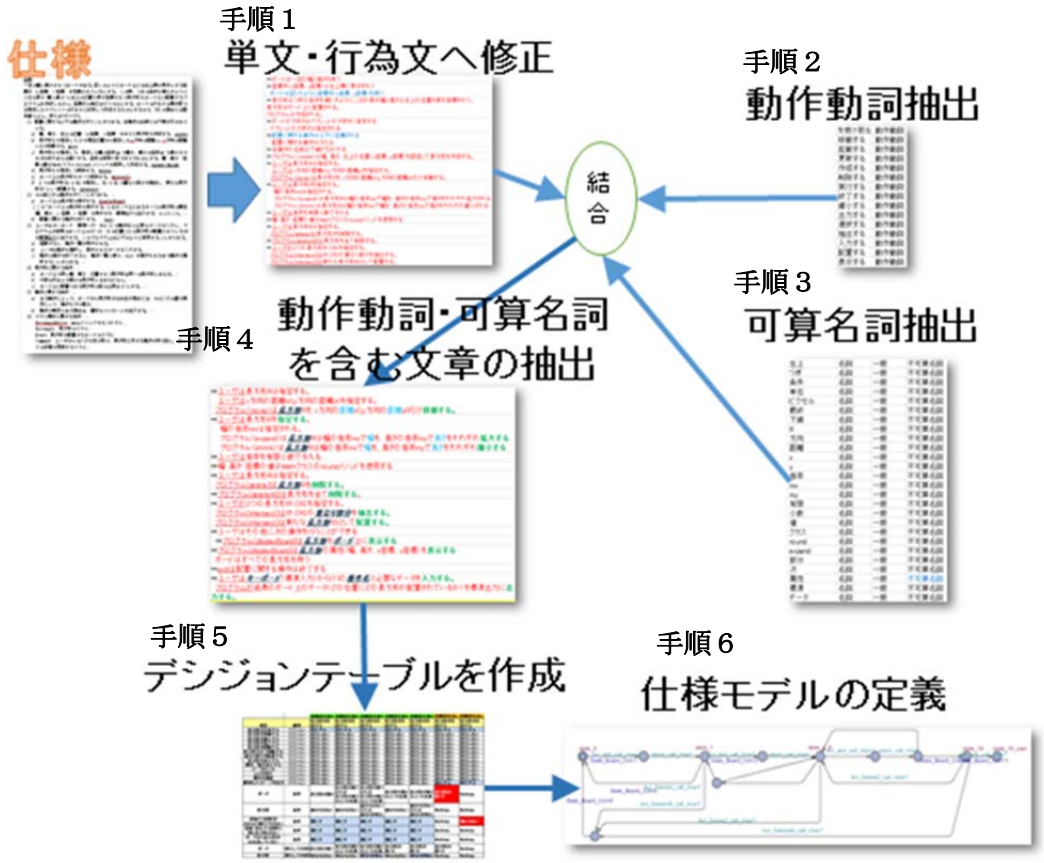


図 4-22 分析手順概要

4.4.4　仕様モデル作成

仕様は自然言語や UML といったモデリング言語で定義される．そして開発者は自然言語で記述された仕様から業務ルールを理解できるので，同様に検査者も業務ルールを理解してデシジョンテーブル[4]を定義できることを前提としている．デシジョンテーブルは条件とアクションの組合せによって複雑な業務ルールの振舞いを表せる方法として知られている．検査者は，まず 4.4.3 で特定した仕様書の文章から業務を実現するための条件とアクション及び結果を抽出する．次に検査者は，仕様から抽出した条件・アクション・結果(システムがもたらす業務機能の振舞いの状態)を関連する仕様毎に並べて結合させてデシジョンテーブルを作る．仕様は前節で述べたように単文化・行為文に変換しているので，条件・アクション・結果(システムがもたらす業務機能の振舞いの状態)の関連は明確になっている．テーブルの各列が仕様を構成し，アクションの中のどのアクションをどの条件で実行した時に，結果がどのように期待されるかを定義する．このテーブルを作成することにより，システムの取りうるアクションの結果として期待される業務機能の結果（状態）が，デシジョンテーブルの例（表 4-2）の点線内のように明確になる．これを「想定される状態」と呼ぶ．この特定した状態をモデル検査ツール UPPAAL のロケーションに置き換え，システムの動作をエッジに置き換える．このロケーション間をシステムの動作のエッジで結ぶことにより仕様モデルを作成する．ここに現れなったシステムの振る舞いの状態は，仕様上はありえないものであるため「想定されない状態」と呼ぶ．

項目	仕様1	仕様2	仕様3	仕様4	仕様5	仕様6
アクション1	実行する	実行する	実行する	実行する	実行する	実行する
条件	A==B	A==C	A==B	A==C	A==B	A==C
条件	状態==未設定	状態==未設定	状態==FALSE	状態==TRUE	状態==TRUE	状態==FALSE
結果	状態==TRUE	状態==FALSE	状態==TRUE	状態==FALSE	状態==TRUE	状態==FALSE

表 4-2 デシジョンテーブルの例

アクションは検査するプログラム内のメソッドおよび API に対応すると仮定できる．これらのアクションによって，ソースコードモデルと仕様モデルを Source2UPPAAL 上で紐付けて(例図 4-20)，検査モデルを構成する．紐付けは UPPAAL の同期処理で行う．表 4-2 の業務機能の振舞いの「状態」を仕様モデルにすると図 4-23 になる．アクション 1 の同期呼び出しを受けて（Action1_Call?）状態遷移を開始する．条件 A==B の場合はロケーション TRUE に到達し，A==C の場合はロケーション FALSE へ到達する．その時同期呼び出し（Action1_Ret!）を実行して呼び出し元へ同期呼び出しを行い，状態遷移を戻す．すでに TURE もしくは FALSE のロケーションに到達している場合も同様の処理を行っている．常に条件 A==B の場合はロケーション TRUE に到達し，A==C の場合はロケーション FALSE へ到達する．

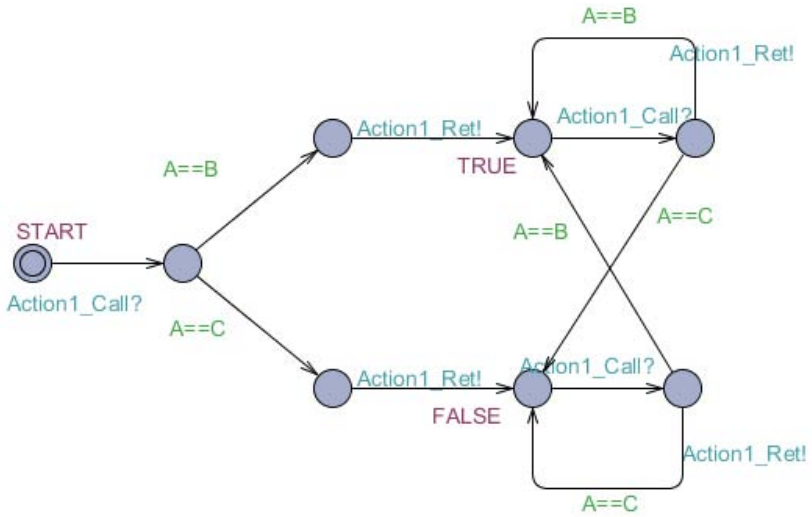


図 4-23 仕様モデルサンプル

現状，この部分は検査者の手作業で行っており，今後モデル化の手順を明確にして自動化する予定である．

仕様モデルの作成手順をまとめると以下のようになる．

- デシジョンテーブルで特定した状態をロケーションに置き換える
- ロケーション間の状態変化を起こすアクションがあればエッジでつなぐ
- アクションが直接状態に影響するならばロケーション⇒ロケーションを直接つなぐ
- アクション＋外部の条件(例：人の選択)で次の状態が決まる場合はアクションで状態遷移を開始し非決定性で分岐させて状態を決める
- アクション＋自他の仕様モデルの状態との組合せで次の状態が決まる場合はアクションで状態遷移を開始し，自他の仕様モデルの状態で条件分岐させて状態を決める
- 状態遷移後は必ず状態のステータスをグローバル変数に設定する
- アクションが状態遷移を起こさない(例：値の取得だけ)場合は同じ状態に戻る遷移にし，状態のステータス値を返す．
- 初期状態が未定の場合は非決定性を用いて１回だけ動く状態遷移で状態を決める

4.4.5　アクションモデルの作成

通常，仕様で記述されているアクションはシステムの状態を変化させる行為であり，予め存在する何らかのメソッド及び API で実行される．従ってアクションは検査するプログラム内のメソッドおよび API に対応すると仮定できる，メソッドおよび API をソースコードと仕様の接点として取らえて，これを機能の中継だけをするアクションモデルとして定義する(図 4-24).

アクションモデルはソースコードモデルからの同期処理を受け，それをトリガにして仕様モデルへ同期処理を行うモデルである．図 4- 24①で他のソースコードモデルから同呼び出しをうけ遷移を開始し,図 4-24②③で処理を行う．図 4-24 では仕様モデルに状態遷移を起こさせる場合はここに同期処理のチャネルを設定して送受信を行う．もし値を取得するだけの場合は，非決定で値を取得するか,他のモデルに設定されている変数を取得する等して返値の変数に設定する．図 4-24④で呼び出し元のモデルへ同期呼び出しを行い，処理を戻す.

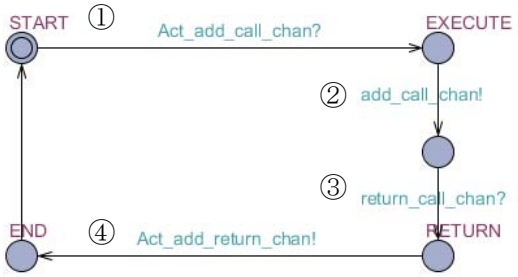


図 4-24 アクションモデル例

アクションモデルの作成手順をまとめると以下のようになる.

・同期処理の受信を受け(図 4- 24①)，状態遷移を開始し，最後に送信元へ同期処理を送信(図 4- 24④)するモデルにする

- 開始ロケーションから開始ロケーションへ戻るモデルにする
- 仕様モデルに状態遷移を起こさせる場合は，アクションと対応する同期処理の送信チャネルを設定する(図 4- 24②)
- 次の同期処理の戻りを待つため，同期処理の受信のチャネルを設定する(図 4- 24③)
- 仕様モデルに状態遷移を起こさせない場合は，同期処理の値の設定は行なわない
- 仕様モデルから値の取得をするが値が常に一意ならばグロバール変数から直接取得する
- 仕様モデルから値の取得をするが一意にならないならば，仕様モデル側に状態遷移を記述して値を決定しそれを取得する
  但し一意にならない原因が仕様モデル側にない時(例 : 引数により変わる)はアクションモデル側に条件分岐を設け対応する

図 4-25 に連携している例を示す. アクションモデルはソースコードモデルから同期処理を受け状態遷移を開始する. この間ソースコードモデルの状態遷移は停止している. アクションモデルは仕様モデルへの同期処理を行い仕様モデルはそれを受け状態遷移を開始する. この間アクションモデルの状態遷移は停止している. 仕様モデルは状態遷移が終了して特定の状態になったらアクションモデルへの同期処理を行う. アクションモデルは状態遷移を再開しソースコードモデルへの同期処理を行う. ソースコードモデルは状態遷移を再開する.

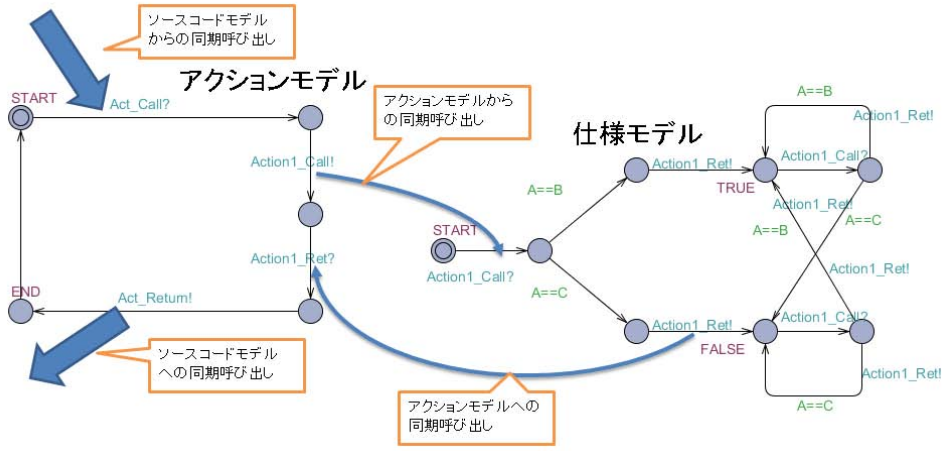


図 4-25 アクションモデルと仕様モデル

4.5 検査の実施

ソースコードモデルと仕様モデルを同期処理で結合して検査モデルを完成させる．作成された検査モデルはシステムがもたらす業務機能の振舞いを表しているので，モデル上の全ての状態へ到達できるはずである．従って検査モデルに対して各ロケーションへの到達可能性の検査を行い検査モデルとして問題がないことを確認する．まず各検査モデルの終点のロケーションに到達できるかを確認する．これはソースコードモデルならメソッドの終点のステートメントに到達できるかの確認にあたる．

次に整理したデシジョンテーブルより導き出される「想定される状態」にいつかなること確認する検査式を実行する．システムの振舞いが検査モデル上で再現できていることを確認する．

最後にデシジョンテーブルより導き出される「想定されない状態」には決してならないことを確認する検査式を実行する．この検査により業務機能の振舞いに合致しない状態遷移が起きないことを確認する．ただしアクションモデルの対象にしたメソッドのロジックに問題がある場合，これまでの検査では不具合は発見できない．その場合は，アクションモデルをソースコードモデルへ再変換して，新たなソースコードモデルを追加した検査モデルで上記手順を繰り返す．そうすればモデルが段階的に詳細化され，該当メソッドのロジックについても検査できる．本研究はユースケースレベルのシステムがもたらす業務機能の振舞いの仕様をデシジョンテーブルで整理してモデル化し，想定外の振舞いがないことを検証している．従ってハードウェア・開発環境・パッケージソフト等に関する不具合，つまり仕様に記述されている振舞い以外については本研究の提案手法の検査対象外となる．

4.6 検査結果の検証

4.6.1 反例解析について

検査対象となるシステムの振舞いを有限の状態空間にモデル化し，システムの満たすべき性質を表す論理式で検証する．もし与えられた性質が満たされなかった場合，モデル検査ツールはその満たされない状態に至る状態遷移の経緯を反例として出力する．不具合は検査モデル上の特定の状態遷移で表されるため，検査者はこの反例を解析することにより，不具合の原因を特定することができる．

UPPAAL の場合，反例は GUI でグラフィカルに状態遷移がモデル上でシミュレーションされて表示される．複数のモデル間にまたがった状態遷移も全て表され，初期状態から開始して検査式で満たされない状態になる時点まで実行される．また各モデル間の連携の状態も表示される．従ってこの状態遷移を追っていけば不具合の再現条件の確認は可能である．他のモデル検査ツールの状態遷移のリストによる反例よりは直感的に判り易いと言えるが，検査モデルが非常に複雑になってきた場合や反例のデータを解析しようとする場合には，状態遷移が多くなり意味を読み取ることが難しくなる．

図 4-26 は UPPAAL モデルの反例の表示のサンプルである．ロケーション id00 から状態遷移を開始して，4 つのいずれかの末端のロケーションに到達する．その時，変数 flg に値を設定するモデルである．変数 flg==100 になることはないという検査式(A[]not flg==100)の実行結果としての反例である．図 4-27 はその時の検査式とその結果表示である．

ロケーション id00－＞id11—＞id12 と遷移することで flg==100 になることが確認できる．

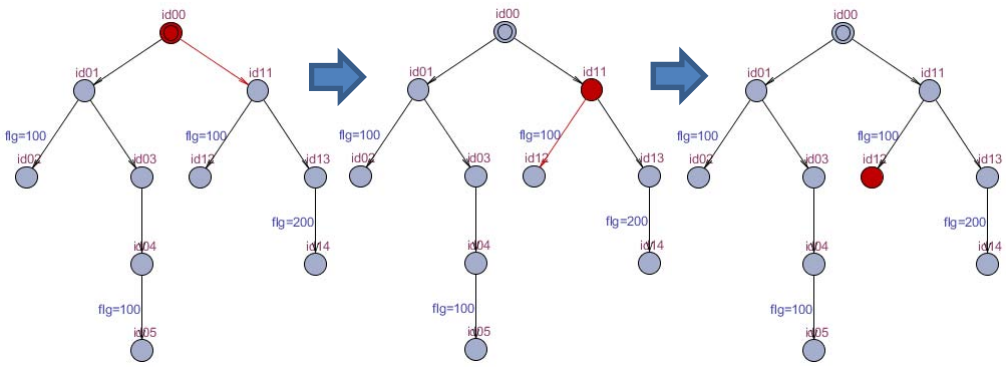


図 4-26 反例シミュレーション

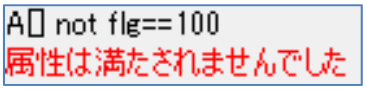


図 4-27 検査結果の表示

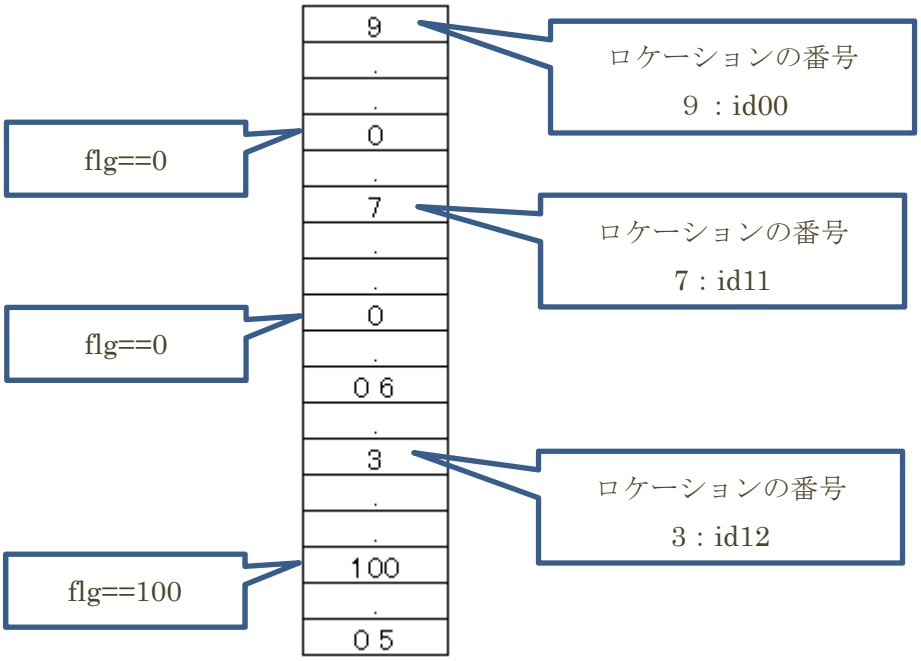


図 4-28 反例のトレースファイル

反例のトレースファイル自体は図 4-28 のように作成されている．トレースは 1 状態遷移ごとに記録されている．ロケーションは UPPAAL 内の内部番号に置き換えられている．変数は UPPAAL 内で定義されている順番で値だけが表示される．これだけを見ても反例を理解して不具合の原因を特定することは困難である．

他のモデル検査ツールを見ても状況はほとんど同じでありロケーション名，変数が状態遷移の単位でトレースファイルとして出力される．図 4-29 は SPIN の検査モデル，図 4-30 は SPIN の反例，図 4-31 は NuSMV の検査モデル，図 4-32 は NuSMV の反例である．

```
mtype = { id00, id01, id02, id03, id04, id05, id11, id12, id13, id14};
mtype state = id00;
int flg = 0;
active proctype proc(){
        do
        ::state==id00 -> state=id01;
                flg=1;
        ::state==id00 -> state=id11;
                flg=-1;
        ::state==id01 -> state=id02;
                flg=100;
        ::state==id01 -> state=id03;
        ::state==id03 -> state=id04;
        ::state==id04 -> state=id05;
                flg=100;
        ::state==id11 -> state=id12;
                flg=100;
        ::state==id11 -> state=id13;
        ::state==id13 -> state=id14;
                flg=99;
        od

]
```

図 4-29 SPIN のモデル

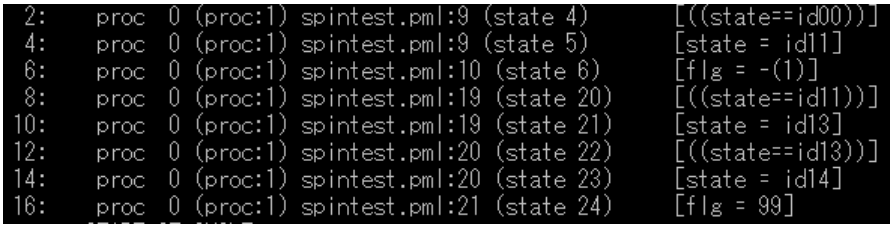


図 4-30 SPIN の反例表示

```
MODULE main
VAR
        State:{id00,id01,id02,id03,id04,id05,id11,id12,id13,id14};
        flg:-1..100;

ASSIGN
        init(State):=id00;
        init(flag):=0;
        next(State):=case
                State=id00:{id01,id11};
                State=id01:{id02,id03};
                State=id03:id04;
                State=id04:id05;
                State=id11:{id12,id13};
                State=id13:id14;
                TRUE:State;
        esac;
        next(flg):=case
                State=id02:100;
                State=id05:100;
                State=id12:100;
                State=id14:99;
                TRUE:flag;
        esac;
DEFINE
```

図 4-31NuSMV のモデル

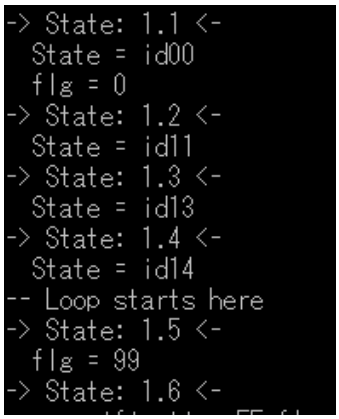


図 4-32NuSMV の反例表示

簡単な検査モデルの場合や反例のレコード数が少ない場合は特に問題にならないが，検査モデルが複雑になり反例のレコード数が膨大になるとグラフィカルに表示をされても初期状態から問題の状態に至るまでの状態数が膨大になり理解するのは困難になる．

また，反例はある特定の状態にいたる 1 つの例を示しているにすぎず，別の状態遷移を行って同じ状態に至ることもある．図 4-26 の例では flg==100 になるパターンは終点となるロケーションが id02,id05,id12 の 3 パターンあり，反例はその一つを示したに過ぎない．一般的な開発者が検査を行っても見落としがないようにしなければならない．

一般的な開発者でも反例の解析を可能にするために反例解析を容易にする仕組みが必要である．

4.6.2　反例解析の容易化

学習用のサンプルの検査モデル程度ならば反例として出力されるレコード数も少なく，示される状態遷移の経路も少なく問題にならないが，実際の不具合に適用した場合，検査モデルは複雑になるため，反例のレコード数も膨大になる．原因を特定するには反例を解析し，解析内容を手作業で整理し，そこから不具合と照らし合わせて不具合原因の特定を行わなければならず，困難である．反例の状態遷移と不具合の内容を照らし合わせて理解することをしなくても不具合原因を想定できる仕組みが必要である．

取り組み方としては2つある．ひとつは反例を直感的に理解できるようにすることであり，もうひとつは反例の結果の精度の向上である．

１つ目の直感的な理解を助ける仕組みとしては反例結果のグラフ化を提案する．反例のトレースファイルを解析して検査者が確認したい任意の項目をグラフ化することにより，状態変化を追い易くなり，特徴的な変化も俯瞰できるため反例の理解を助ける．図 4-26 の反例をグラフ化したのが図 4-33 である．横軸はレコード数，縦軸は各項目の整数の値，折れ線グラフが変数 flg を表し，プロットがロケーションの番号を表す．このグラフよりロケーション番号 9 (id00)からスタートして状態遷移 3 回目のロケーション番号 3 (id12)で flg==100 になることがわかる．

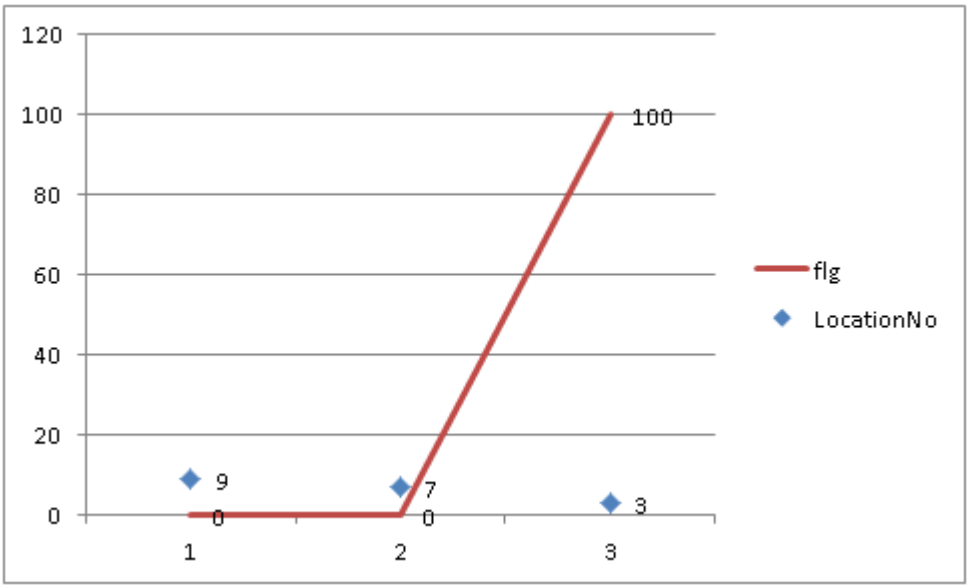


図 4-33 グラフ化した反例

このグラフはツールにより生成している．ロジック部分は C#のプログラムで作成し，グラフ表示部分は Excel を使用している．実際にグラフ化するデータは Excel 上で加工・編集可能であり，グラフの表示方法もカスタマイズ可能である．

本研究では，ソースコードを UPPAAL の検査モデルに変換する．この時，検査モデルのロケーションはソースコードのステートメントと紐付くため，ロケーションからソースコードの行数を特定することが可能である．反例の結果をソースコードの行数と紐づけられるため不具合箇所の特定が容易になる．図 4-34 はソースコードの行数と紐づけたグラフの例である．縦軸がソースコードの行数，横軸が反例のレコード数である．処理が成功した場合と失敗した場合の反例のグラフを重ね合わせて分岐ポイントを見つければそこに不具合原因があると想定できる．図 4-34 の左にある数列は成功・失敗の場合のソースコードの遷移を表している．ソースコードの位置も判明することにより不具合原因の特定の支援になると考える．

成功	失敗
1	1
9	9
18	18
19	19
24	24
26	26
28	28
30	30
32	32
36	36
37	32
38	30
39	9
37	1
41	
36	
32	
30	
9	
1	

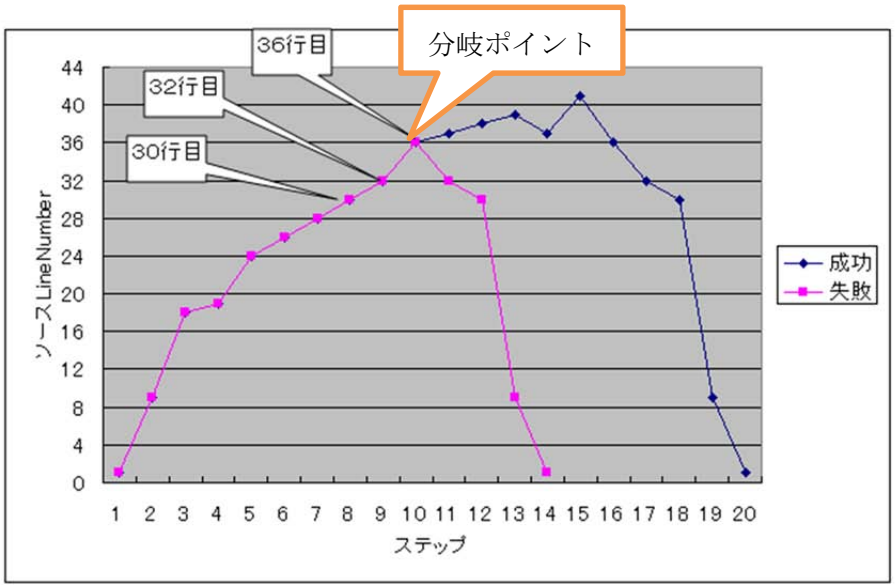


図 4-34 ロケーションとソースコード行数による反例表示

２つ目の反例結果の精度の向上については，出力された反例を基に状態遷移を限定する式を生成し，検査式に付加して再検査するというサイクルを繰り返すことにより反例の精度を上げる（不具合箇所の特定をしやすくする）手法を提案する．

反例は状態遷移の一つの例を示しているに過ぎないので，図 4-34 の例でも分岐ポイント以降の成功の状態遷移でもまだ失敗の状態に陥る可能性がある．

「想定される状態」になった場合と「想定されない状態」になった場合について共に反例をグラフ化して出力し，それらを比較して差異が出来る部分を表示することにより反例解析を支援できる．「想定されない状態」になるか検査で反例がでた場合，それと対になる「想定される状態」になるかの検査を行う．単純に「想定される状態」の検査を行ってもまったく関係ない状態遷移になる可能性が高い．そのため「想定されない状態」の反例よりシステムの仕様で特定できる識別子に着目してその識別子に関係する UPPAAL の変数を検査式に取り込む．業務機能の振舞いを特定できる状態を揃えられるため類似した状態遷移

になり，比較が可能になる．そうすると「想定される状態」「想定されない状態」で分岐ポイントが見つけられる(図 4-35).

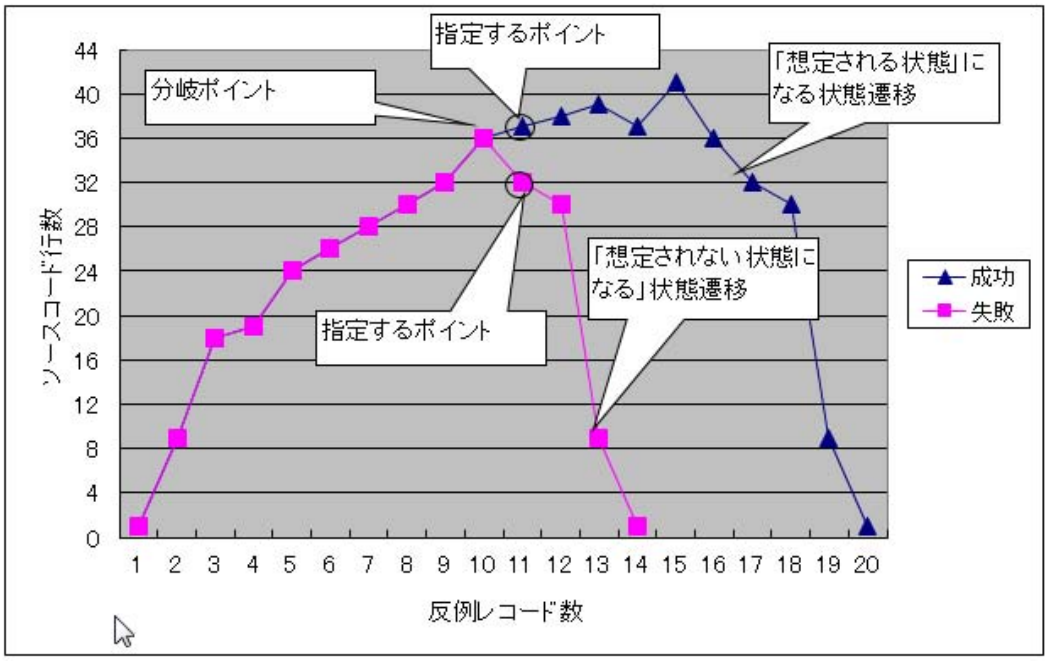


図 4-35 状態遷移のグラフによる比較例　1

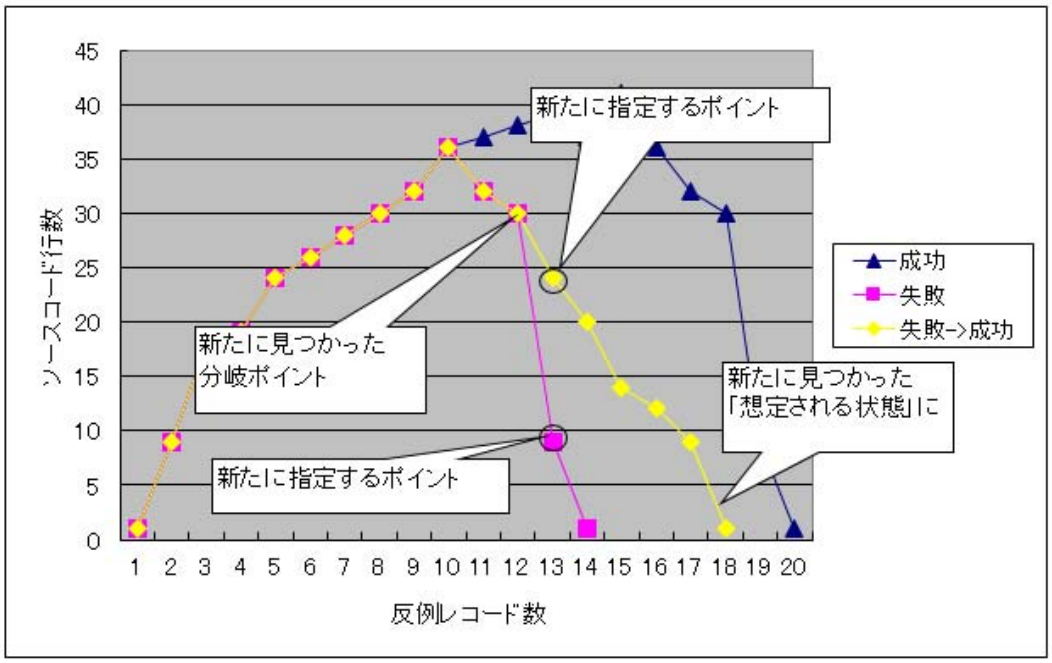


図 4-36 状態遷移のグラフによる比較例　2

しかしその分岐後で最終的なシステムの振る舞いの状態が特定できるとは限らない．分岐ポイント以降の状態遷移を規定するため，分岐ポイントより先のポイント（図 4-35 指定するポイント）を指定した式を追加した検査式で再度検査する．最終的なシステムの振る舞いの状態が変化しない分岐ポイントが確定するまで検査を行う．こうすることにより検査結果の精度が高められる．図 4-36 のように新たな分岐ポイントが見つかるかもしれず，最終的に見つかった分岐ポイントに不具合の原因があると考えられる．

4.7 本章のまとめ

本章では，一般的な開発者でもモデル検査技術を用いてシステムに想定外の振舞いがないかをソースコードを対象として検証するソースコード検証手法を提案した．手法の提案は以下のようにまとめられる．

ソースコードをモデル検査ツール UPPAAL の検査モデルへ状態爆発を避けながら段階的に変換してソースコードが表すシステムがもたらす業務機能の振舞いをモデル化したソースコードモデルの作成方法を提案した．また仕様書（設計書）に記述されている仕様から拡張したデシジョンテーブルを用いて「想定される振舞い」を抽出し，それをモデル化した仕様モデルの作成方法を提案した．

そしてモデル検査の非専門家である一般的な開発者でもモデル検査技術を利用して不具合原因の特定ができるようにするために検査支援ツール Source2UPPAAL を提案した．さらに検査結果として出力される反例を理解し易くするための解析手法を提案した．

5　UML 検証手法

5.1 概要

多くの IT プロジェクトでは「要求仕様の決定遅れ」「要求分析作業が不十分」が工程遅延や費用増大の大きな要因となっている[73]．IEEEstd.830[75]には要求仕様で定義すべき項目が述べられているが，項目の相互作用を十分理解しながら，要求仕様の目的を満たすように自然言語で理解性の高い仕様書を定義することは難しい．その上，レビューのような非形式的な検証以外の検証を行うことが困難である．このため，要求分析段階で要求仕様の非曖昧性，完全性，無矛盾性といった品質を保証することが難しい．

VDM [50] や B-method [55]のような形式仕様化技術は要求仕様を形式化する 1 つの技術とも考えられるが，一般には，曖昧な要求から，要求抽出，要件定義を行う混沌とした工程において，はじめから厳密な形式手法を用いることは困難である．

UML（Unified Modeling Language）[71]は要求仕様を記述するための自然言語を含む柔軟な記述を許す道具として広く多くの開発者に使用されている．クラス定義のように識別子を構造的に定義することは可能であるが，例えばアクティビティ図においても，アクションの記述や分岐の条件を表すガードの記述の自由度は高く，そのままでは形式手法のように機械的な検証はできない．

要求分析手法では，開発者が業務遂行に必須な業務フローと業務データを UML モデルであるアクティビティ図とクラス図により定義する．本研究はこのアクティビティ図をモデル化して検査することによりシステムに想定外の振舞いがないかを一般的な開発者でも検証できるようにする．

5.2 UML モデルから検査モデルへの変換

図 5-1 は本手法における全ての UML モデルからモデル検査器で実行される UPPAAL モデルへの変換の全体像を示している．UML モデルのうちナビゲーション，アクティビティ図，ステートマシン図の各モデルを入力として，これらのモデルを関連付けることによって UPPAAL モデルと検査式を自動生成する．アクティビティ図は対象システムのユースケースを定義したものであり，ナビゲーションはユースケースをどのような手順で実行するか定義したものである．ステートマシン図は業務ルール等に定義されている遵守されなければならないシステム振舞いである．

はじめに，ユースケースを定義したアクティビティ図が各ノードをロケーションに対応付けることでアクティビティ図のフローを踏襲した遷移経路を持つ 1 つの UPPAAL のモデルに変換される．

この時，オブジェクトの属性の更新アクションは同期チャネルを持つ 3 つのロケーションとその遷移に変換される． 図 5-1 はアクティビティ図から変換した UPPAAL モデルを示している．アクション，オブジェクト，デシジョン，マージ，開始，終了のすべてのノー

ドは UPPAAL のロケーションに変換される．そして， オブジェクトの属性更新アクション以外は，すべての制御フローとデータフローがロケーション間の遷移に，そのままの順序で変換される．

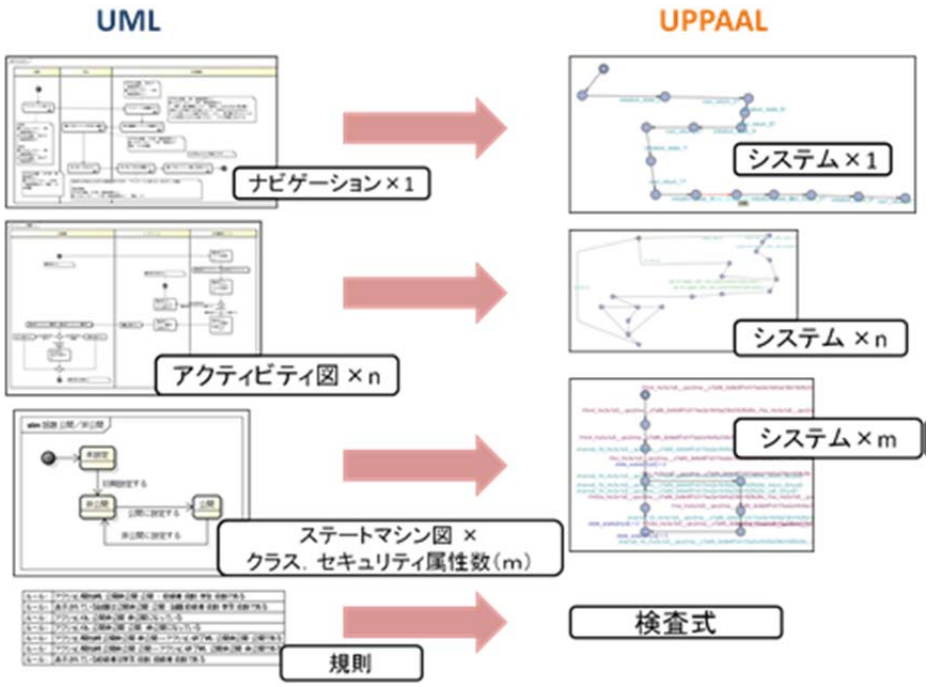


図 5-1 モデルから UPPAAL モデルへの変換

5.3 モデル検査支援ツール　UML2UPPAAL

本ツール（UML2UPPAAL）の構成を図 5-2 に示す．UML2UPPAAL は UML 要求分析モデルを作成する astah* プラグインとして構築されている．開発者の指示によって要求分析モデルから UPPAAL モデルを自動的に変換し，その変換結果（UPPAAL モデル）を UPPAAL に引き渡す．UPPAAL は渡された変換結果を検査し，その検査結果を UML2UPPAAL に戻す．UML2UPPAAL は検査結果を astah*上に表示する．構成上は UML2UPPAAL と UPPAAL は分離しているが，検査者から見た場合には UPPAAL は隠蔽されており，検査者が直接操作する必要はない．

UML2UPPAAL は次の 3 つの機能を提供する．

① システムの性質検査機能

システムの入出力データを表すアクティビティ図上のエンティティ・データに対する操作に関して，エンティティ・データごとに定められたステートマシン図が許容しないような操作をおこなっていないことを検査する機能である．

② 通り得ないフローの検査（到達可能性）

アクティビティ図に関して，すべてのアクションへと到達する可能性があるかどうかを検査する機能である．複数のアクティビティ図間の包含関係や事前・事後条件を考慮する．

③　不適切なアクション

モデルにおけるアクションの自然言語記述，アクションとオブジェクトノードの構造などの整合性をチェックし，問題箇所を指摘する機能である．

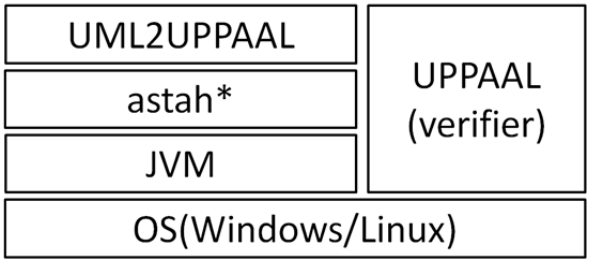


図 5-2　UML2UPPAAL の構成

5.4　モデル検査の実施と検査結果の検証

ツールの利用例を説明する．UML2UPPAAL は図 5-3 の赤線で囲まれた部分であり，4 つのタブから構成される（通り得ないフロー，不適切なアクション，UPPAAL 検証，コンソール）．図 5-3 は，システムの性質検査機能を実施した時の画面表示例である．変換結果の一部を示す．右上の UML モデルに対して，検査を実行した結果が右下に表示されている．図中，緑色のハイライトが検査に成功したものを表し，赤色のハイライトが検査に失敗したことを表している．図 5-4 は，②通り得ないフローの検査を行った時の画面表示例である．UML モデル中のどの部分が該当するかを色で識別することができる．

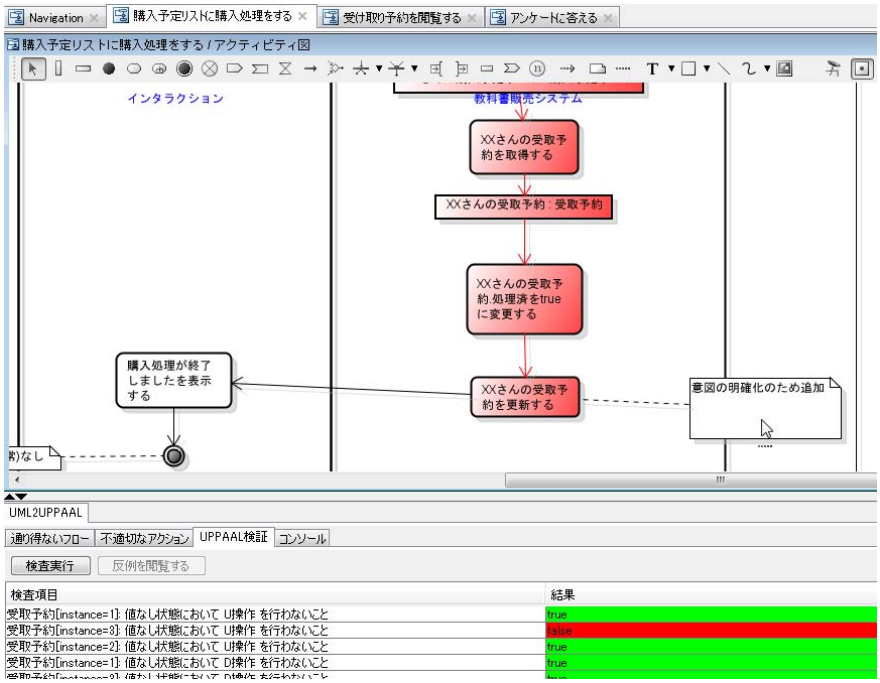


図 5-3 UML2UPPAAL の実行結果（その１）

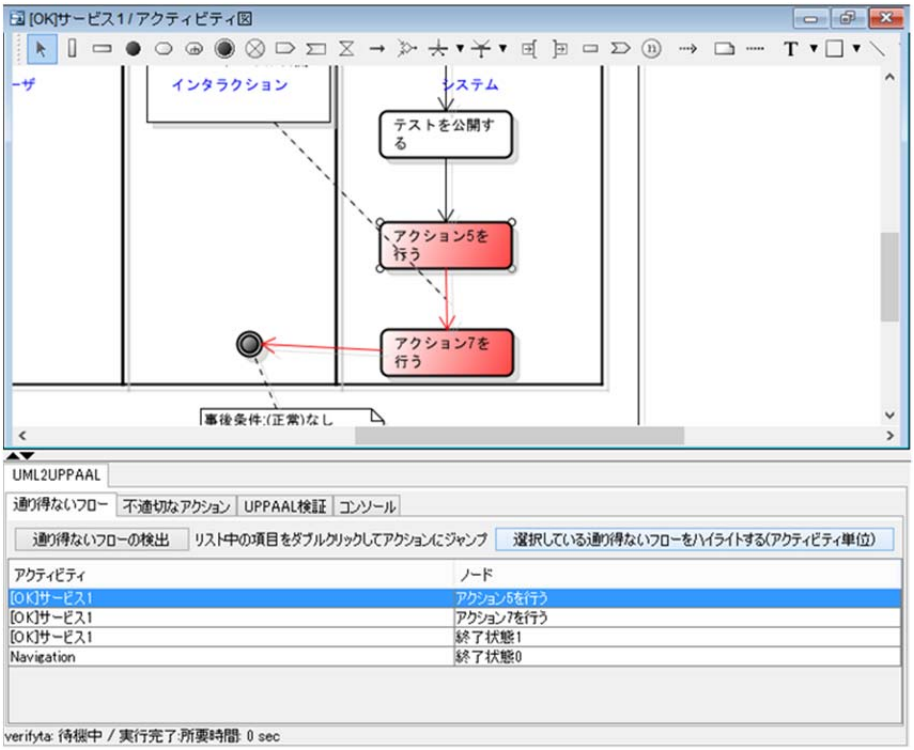


図 5-4UML2UPPAAL の実行結果（その２）

5.5 本章のまとめ

本章では，一般的な開発者でもモデル検査技術を用いて，要件定義の段階においてシステムに想定外の振舞いがないかを UML を対象として検証する UML 検証手法を提案した．手法の提案は以下のようにまとめられる．

UML モデルをモデル検査ツール UPPAAL の検査モデルへ変換する方法を提案した．

そしてモデル検査の非専門家である一般的な開発者でもモデル検査技術を利用して不具合原因の特定ができるようにするために astah* プラグインで開発した検査支援ツール UML2UPPAAL を提案した．UML2UPPAAL は３つの機能があり直接 UPPAAL の画面を操作する必要がなく，astah*上での検査および検査結果の表示が可能である．

6　適用事例　開発現場で実際に発生した不具合と同じ振舞い検出の検証

6.1 目的

本研究が提案するソースコード検証手法の有効性の確認を行う．実際に発生した不具合の原因を基にサンプルプログラムを作成する．そのプログラムへソースコード検証手法を適用することにより，開発現場で実際に発生した不具合と同じ振舞いを検出できるかを検証し，提案手法が開発現場でも有効であることを確認する．

6.2 適用事例概要

検証の対象は，流通の業務における販売システムの一部である受注伝票登録処理において使用される与信チェックの機能である．与信チェックとは企業が顧客に対してどれだけ商品を貸し売りしてよいかをチェックする機能である．この機能は予め設定してある与信限度に貸し売りした金額が達した場合に伝票登録をブロックする．

販売システムとは顧客より注文を受注し，在庫にある品物を引き当てして注文先へ出庫後に顧客に請求処理を行う一連の業務フローを行うシステムである(図 6-1).

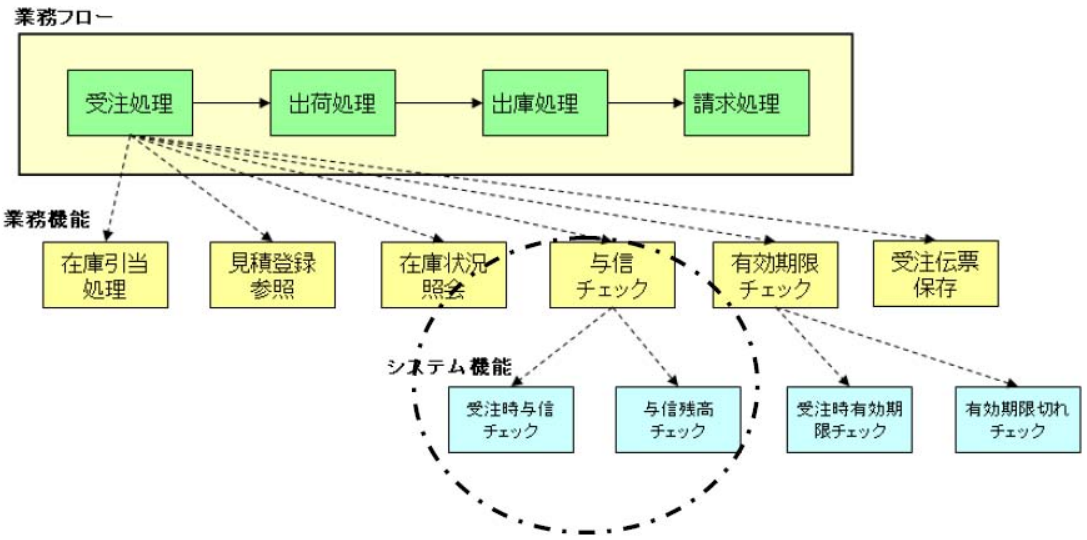


図 6-1 販売システム概要

検査対象となるのは，図 6-1 の点線内の機能である．与信管理マスタに格納されている与信限度を超過する新しい受注を受けた場合，販売システムは受注伝票の登録をブロックする．この仕様は実際の業務で使用されたものに則したものである．

プログラムの機能は次のとおりである.

- 販売システムにおいて受注伝票作成時に与信のチェックを行い, 受注伝票登録をブロック/解除する

- 受注伝票の登録により与信状態が OVER になった場合、与信ブロックフラグを ON にして，以降の受注伝票登録を拒否する（図 6-2）.

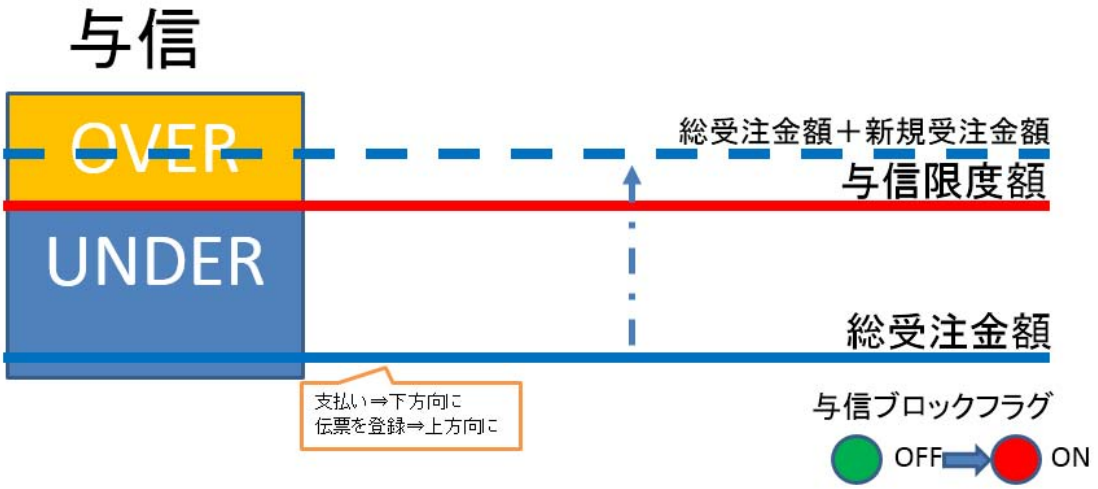


図 6-2 与信状態 UNDER－＞OVER

- 与信限度額の引き上げが行われ与信状態が UNDER になった場合，受注しても与信限度を超えないことを確認後，登録を行い，与信ブロックフラグを OFF にする（図 6-3）.

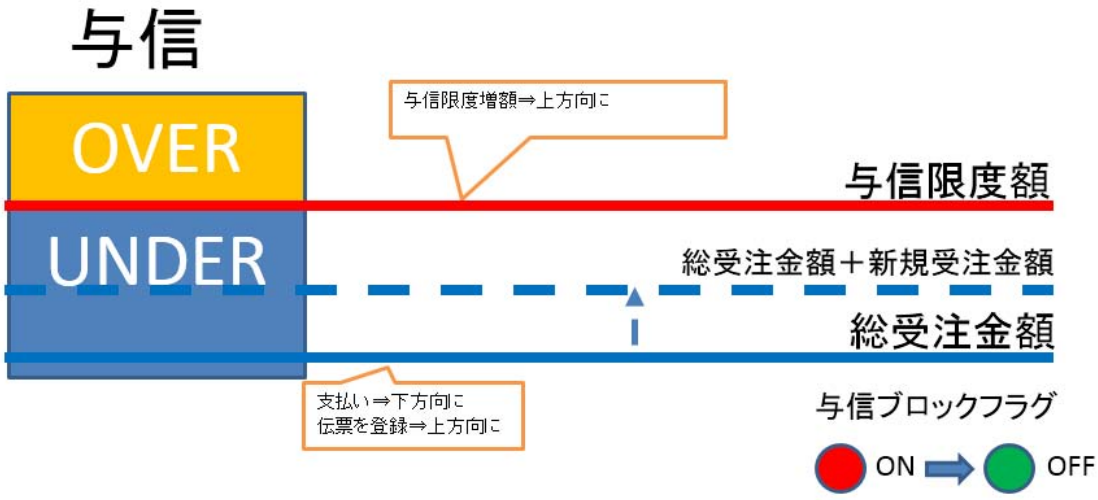


図 6-3 与信状態　OVER－＞UNDER

提示された仕様は以下の通りである．(1)~(3)が与信チェック機能の仕様であり，(4)~(9)がこの機能の中で使用されるメソッドの仕様である．

(1)新規受注登録拒否状態の場合は，伝票登録は行わず，エラーメッセージ表示する．
(2)新規受注登録拒否状態ではなく(受注新規登録が可能な状態)かつ仮の与信チェックの結果が OVER の場合は，受注伝票登録を行い，エラーメッセージを表示する．与信ブロックフラグを ON にする．
(3)新規受注登録拒否状態ではなくかつ仮の与信チェックの結果が UNDER の場合は，受注伝票登録を行い，さらに与信ブロックフラグが ON の場合は OFF にする．
(4)新規受注伝票拒否状態のチェックは新規受注伝票拒否状態のチェックメソッドを利用する．与信状態が OVER かつブロックフラグが ON の場合 true を返し，それ以外は false を返す．与信状態には新規受注金額は含まれない．
(5)エラーメッセージの表示はエラーメッセージ表示メソッドを使用する．
(6)与信のチェックには仮の与信チェックメソッドを使用する．これは既存の債権と新規の受注金額の合計が与信限度額を超えているかどうかをチェックするメソッドである．与信限度額を OVER する場合は false を返し，UNDER な場合は true を返す．
(7)与信ブロックフラグのチェックには与信ブロックフラグチェックメソッドを使用する．これは与信ブロックフラグが ON 場合は true を返し，OFF の場合は false を返すメソッドである．
(8)受注伝票の登録は受注伝票登録メソッドを使用する．受注伝票登録により債権は増加する．
(9)与信ブロックフラグの変更は与信ブロックフラグ変更メソッドを使用する．引数が true の場合は ON に false の場合は OFF に変更するメソッドである．

6.3 ソースコードモデル及び検査モデルの生成

検査対象となる機能は与信チェックであるのでソースコードモデルへの変換対象は main メソッドである．これをソースコードモデルへ変換する．その配下にあるメソッドは，仕様モデルの状態に関係するものはアクションモデルに変換し，関係しないものは非決定性の値を付値する．割り当ては表 6-1 になる．これを検査支援ツール Source2UPPAAL でモデル化する．

検査者は図6-4に示すように検査支援ツールSource2UPPAALを用いて対象のソースコードに非決定的な値を割り当てるとともに作成したアクションモデルを登録してドラッグ&ドロップで割り当てる．Source2UPPAAL の検査モデル生成機能を実行することにより，ソースコードモデルが生成されると同時に仕様モデル，アクションモデルを取り込んで検査モデルが生成される．

表 6-1　アクションの割り当て

アクション	メソッド	検査モデル種類	関係する仕様モデル
メイン	main	ソースコードモデル	–
与信ブロック変更	changeCreditBlockFlag	アクションモデル	与信ブロック
受信伝票登録	createSalseOrder	アクションモデル	与信状態
新規受注伝票拒否状態のチェック	checkNesSalesOrderRefu	アクションモデル	与信ブロック/与信状態
仮の与信チェック	checkTemparayCregit	アクションモデル	与信状態
与信ブロックチェック	checkOrditBlockFlag	アクションモデル	与信ブロック
エラーメッセージ表示	displayErrorMessage	非決定性の値	–

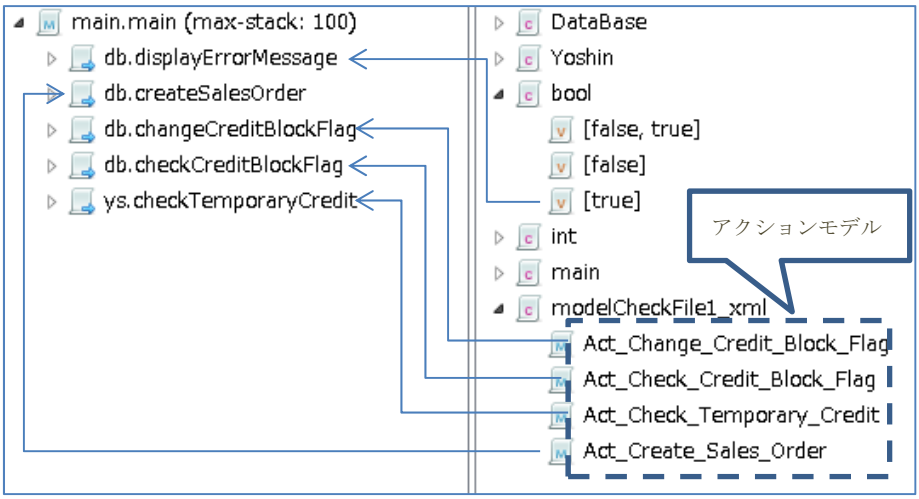


図 6-4　検査モデルへの割り当て

6.4 仕様モデルの作成

6.4.1 デシジョンテーブルの作成

サンプルプログラムを作成するために整理した仕様を基に，デシジョンテーブルを作成する．サンプルコードを作成する過程で 4.4.3 にある「文章の単文化及ぶ存在文への変換」,「動作動詞・可算名詞の抽出」,「モデル化すべき文章の特定」までは完了しているため，「条件とアクション及び結果を抽出」から始める．条件（表 6-2），アクション（表 6-3），結果（表 6-4）を抽出してデシジョンテーブル（表 6-5）を作成する．

表 6-2 条件

No.	条件	取り得る値
con01	新規受注登録拒否状態	true／false
con02	仮の与信チェックの結果	UNDER/OVER
con03	与信ブロックフラグ	ON/OFF
con04	与信状態	UNDER/OVER

表 6-3 アクション

No.	アクション	取り得る値
act01	受注伝票登録	実行／実行しない
act02	エラーメッセージ表示	実行／実行しない
act03	仮の与信チェック	実行／実行しない
act04	新規受注伝票拒否状態のチェック	実行／実行しない
act05	与信ブロックチェック	実行／実行しない
act06	与信ブロック変更	実行／実行しない

表 6-4 結果一覧

結果	取り得る値
与信ブロックフラグ	ON/OFF
与信状態	UNDER/OVER

デシジョンテーブルは次のように作成する．仕様(1)~(3)に記述されているアクション・条件・結果を抽出しそれをデシジョンテーブルの１列として記述する（表 6-4）．複数条件が記述されている場合はデシジョンテーブルも複数列になる．仕様(1)におけるアクション・条件・結果の対応は図 6-5 のようになる．この対応するアクションと条件のステータスをデシジョンテーブルに記述する．仕様(1)ではシステムの状態の変化はないので初期の状態が引き継がれる．初期の状態は新規受注登録拒否状態であり，これは(4)で「与信状態が OVER かつブロックフラグが ON」 と記述されているので，これを結果として記述する．

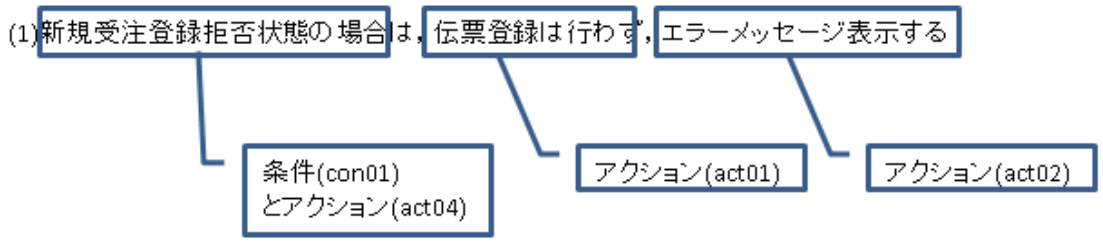


図 6-5 仕様(1)アクション・条件の対応

表 6-5　デシジョンテーブル

No	項目	種別	仕様(1)	仕様(2)	仕様(3)A	仕様(3)B
01	受注伝票登録	アクション	実行しない	実行	実行	実行
02	エラーメッセージ表示	アクション	実行	実行	Nothing	Nothing
03	仮の与信チェックメソッド	アクション	Nothing	実行	実行	実行
04	新規受注伝票拒否状態のチェックメソッド	アクション	実行	実行	実行	実行
05	与信ブロックチェックメソッド	アクション	Nothing	実行	実行	実行
06	与信ブロック変更メソッド	アクション	Nothing	実行	実行しない	実行
07	新規受注登録拒否状態	条件	TRUE	FALSE	FALSE	FALSE
08	仮の与信チェックの結果	条件	Nothing	OVER	UNDER	UNDER
09	与信ブロックフラグ	条件	ON	OFF	OFF	ON
10	与信状態	条件	OVER	UNDER	UNDER	UNDER
11	与信ブロックフラグ	結果	ON	ON	OFF	OFF
12	与信状態	結果	OVER	OVER	UNDER	UNDER

デシジョンテーブルからシステムの振舞いが読み取れる(表 6-5 の点線内)．以下の 2 点が想定されるシステムの振舞いである．これを否定にしたものが想定されないシステムの振舞いとなる．

- 与信ブロックフラグ ON かつ与信状態 OVER
- 与信ブロックフラグ OFF かつ与信状態 UNDER

6.4.2　仕様モデルの作成

特定した状態をモデル検査ツール UPPAAL のロケーションに置き換え，システムの動作をエッジに置き換える(図 4-22 手順 6). 状態のロケーション間を関係するシステムの動作のエッジで結ぶことにより仕様モデルを作成する.

モデル化すべき状態は与信ブロックフラグと与信状態である．与信ブロックフラグの仕様モデルは図 6-6 である. ソースコードモデルが状態遷移を開始する前に初期設定が終わっている必要があるため時間の制限を利用する．「t<10」と「t>5」はクロック変数 t が 5 より大きく 10 未満の間に START からアージェントのロケーション（図 6-6①）に状態遷移することを表している．ソースコードモデルの方も同様の設定を行い，t が 10 より大きく 20 未満と設定すれば初期のステータスの遷移が必ず先に行われる．アージェントは時間経過しないロケーションである．ステータスは ON と OFF の二つがあり，初期のステータスは与信データの状況により変わるため非決定で設定する（図 6-6②のどちらか）．仕様(2)(3)より状態を変化させるのは与信ブロックフラグ変更メソッドであることがわかる．このメソッドが実行された場合 ON->OFF もしくは OFF->ON に変更する（図 6-6③）．ステータスの状態は UPPAAL のグローバル変数(Credit_Block_Flag)に格納する.

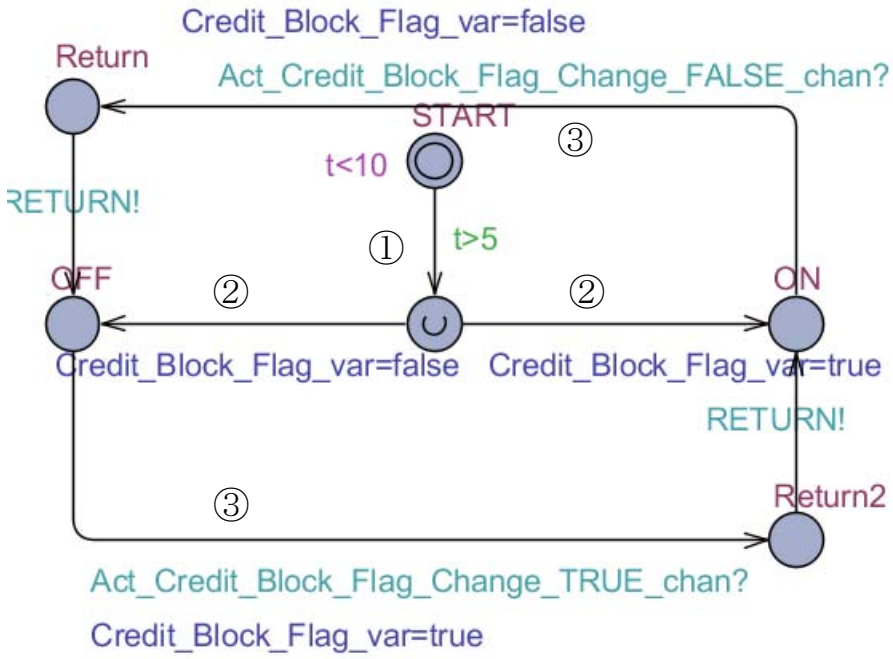


図 6-6 仕様モデル　与信ブロックフラグ

与信状態の仕様モデルは図 6-7 である．与信状態を取得するアクション(表 6-5　No03「仮の与信チェックメソッド」)がある．ステータスが UNDER の場合は仮の与信のステータスは一意に決まらないため，ロケーション UNDER および OVER から自ロケーションに戻る状態遷移を追加して仮の与信のステータスを返すようにしてある.

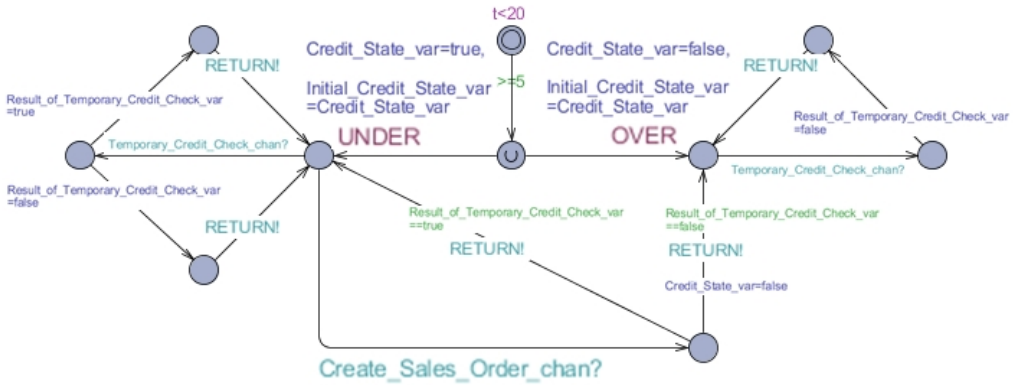


図 6-7 仕様モデル　与信状態

6.5 アクションモデルの作成

アクションモデルを作成する．表 6-5 デシジョンテーブルから与信ブロックフラグ変更のアクションの状態を抜き出す．そこから作成されるモデルは図 6-8 である．引数による条件分岐を持ち，与信ブロックフラグを ON/OFF させる同期呼び出しを送信する．他のアクションについても同様にモデル化する．

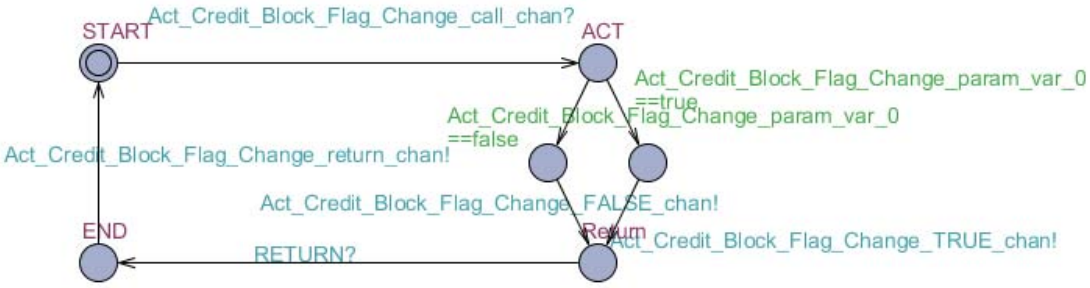


図 6-8 与信ブロック変更メソッドのアクションモデル

6.6 検査実施

6.6.1 到達可能性の検査

生成した検査モデルに構造上の問題がないことを確認するために到達可能性の検査を行う．図 6-9 の検査式を実行する．この検査式の意味は「メソッド main()の終点にいつか到達する」である．

```
E<> ( SYSTEM_main_main(0,0).END)
```

図 6-9 到達可能性の検査

6.6.2 システムの振舞いの検査

以下の 2 つの「想定されない状態」になることがあるかという安全性の検査を行う．

「想定される状態」次のとおりである．

- 処理実行後は与信ブロックフラグ ON かつ与信状態 OVER
- 処理実行後は与信ブロックフラグ OFF かつ与信状態 UNDER

図 6-10 の検査式を実行する．検査式の意味は「与信状態が UNDER もしくは与信ブロックフラグが OFF　かつ与信状態が OVER もしくは与信ブロックフラグが ON　になることは決してない」というものである．具体的には「与信ブロックフラグ ON かつ与信状態が UNDER」もしくは「与信ブロックフラグが OFF かつ与信状態が OVER」になるかの検査である．

```
A[] not (Rst_Credit_State.UNDER || Rst_Credit_Block_Flag.OFF)
and (Rst_Credit_State.OVER || Rst_Credit_Block_Flag.ON)

※State_Credit_Cntrol
  ：与信状態
※State_Credit_Block
  ：与信ブロック
```

図 6-10 検査式：与信状態と与信ブロックフラグ

検査結果は「属性は満たされませんでした」となり反例が出力された．反例では「与信ブロックフラグ ON かつ与信状態 UNDER の状態」になることが示された．

次に期待される結果になることがあるかを検査する．検査式は図 6-11 のとおりである．検査式の意味は出力された反例と同じ初期状態から開始して「main()メソッドに到達時に与信状態が UNDER かつ与信ブロックフラグが OFF になることは決してない」である．

```
A[] not ((vInitBlock==true && vInitCheck==true) &&
(State_Credit_Cntrol.UNDER && State_Credit_Block.OFF) &&
SYSTEM_main_main(0).END)

※vInitBlock
   : 与信ブロックフラグの初期値 true ON
※vInitCheck
   : 与信状態の初期値 true UNDER
```

図 6-11　検査式

検査結果は「属性は満たされませんでした」となり反例が出力された．正常に終了する場合もあることが判明する．

二つの反例を比較することによりソースコードのどの部分が成功と失敗の分岐点になるか調査する．ソースコードモデルのロケーションはソースコードのライン数と対応している．従って UPPAAL のトレースファイルにある状態遷移を実行ステップに変換できる．成功例と失敗例をこの実行ステップにより比較すると図 6-12 になる．36 行目で処理が分かれている． ソースコード(図 6-13)を見ると，失敗例 30->32->36 と遷移しており，if 文の処理を全てすり抜けてしまっていることがわかる．与信限度引き上げした直後の処理において条件設定が不十分であることが判明した．

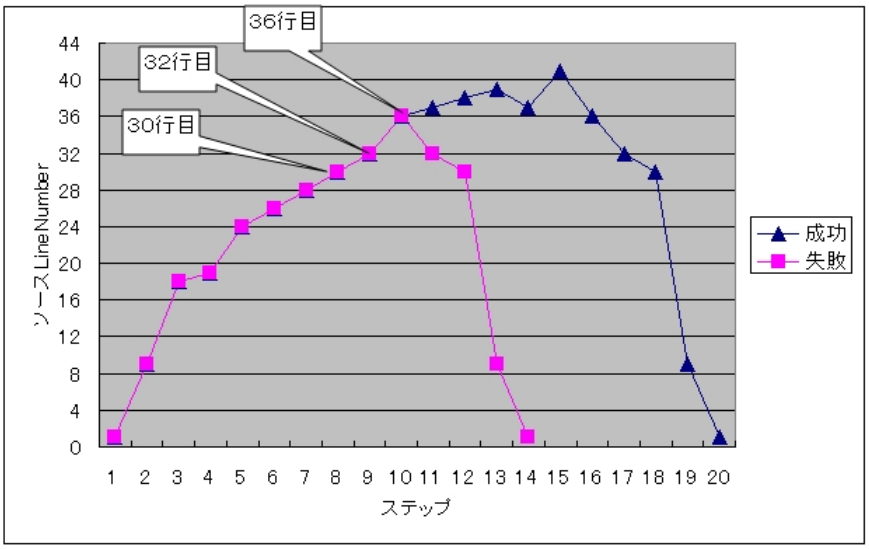


図 6-12 反例のグラフ

```
30        if(rejectSalceOrderFlag==true ){
31            db.updateErrorTable("credit block on");
32        }else if(checkResult==false && blockFlag==false){
33            db.updateErrorTable("Credit line over");
34            db.changeBlockFlag(ON);
35            db.saveSalesOrder();
36        }else if(checkResult==true){
37            if(blockFlag==true){
38                db.updateErrorTable("credit block off");
39                db.changeBlockFlag(OFF);
40            }
41            db.saveSalesOrder();
42        }
```

図 6-13 ソースコード

6.7 本章のまとめ

適用事例の検査では表 6-6 の不具合を発見することができた，不具合の原因は，与信限度を上げた直後の受注処理への考慮不足である．

表 6-6 不具合の一覧

No	不具合の内容	原因
01	与信状態と与信ブロックフラグが想定外の組み合わせになる場合がある	与信限度を上げた後の最初の受注処理に対する考慮不足による条件分岐の条件式の不備

モデル検査技術の知識を殆ど持たない一般的な開発者を支援するためのツールとして eclipse プラグインで実装された Source2UPPAAL があり，それを使用して与信管理の Java プログラムを検証しての不具合原因を特定した．

本研究の提案の大きな特徴はデシジョンテーブルを作成することにより，仕様モデルを作成することができ，その結果として複雑な検査式を作成せずに検証できる点である．実際に発生した不具合原因についても特定できることが確認できた．

7　適用事例　開発現場と同じ仕組みで発生する不具合検出の検証

7.1 目的

本研究の提案するソースコード検証手法の有効性の確認を行う．実際に開発現場で発生する不具合の多くは仕様が正しくプログラムに反映されないために発生する．プログラムを作成する開発者も不具合を検出する開発者も，どのような不具合がシステムに内在しているかは認識していない．この不具合の発生の仕組みはプログラムの規模とは無関係である．従って実際に仕様からプログラムを作成して発生した不具合の原因を検出できるかを検証し，提案手法が開発現場でも有効であることを確認する．

7.2 適用事例概要

長方形エディタプログラムへ提案手法を適用した．これは芝浦工業大学のプログラミング演習で使われた課題である（図 7-1）．具体的には学生に提示されたプログラムの仕様書とそれを基に作成された Java プログラムである．仕様書の内容は次のとおりである．

仕様

一定の幅と高さをもつボードがある。図 1 のようにボード上には左上隅を原点とする座標系（x 座標、y 座標）が定義されているとする。この時、つぎの条件を満たすように二辺の長さ(幅と高さ)と左上の位置を表す座標をもつ長方形をボード上に配置するプログラムを作成しなさい。座標系の単位はピクセルとする。ボードの大きさは最終版では指定したアプレットの大きさに依存して決定するものとするので、CUI の場合には固定値でよい。例えば 600×500。

(1)配置に関する以下の操作を行うことができる。各操作の名前には下線が引かれている。

a) 幅、高さ、左上の位置（ x 座標、 y 座標）を与えて長方形を作成する。create

b) 長方形 R を指定して、R を現在位置から指定した x 方向の距離 x0, y 方向の距離 y0 だけ移動する。move

c) 長方形 R を指定して、指定した幅の倍率 mx で幅を、高さの倍率 my で高さをそれぞれ拡大または縮小する。倍率は有限小数で与えるものとする。幅・高さ・座標の値は Math クラスの round メソッドを使用して決定する。expand/shrink

d) 長方形 R を指定して削除する。delete

e) ボード上の長方形をすべて削除する。deleteAll

(2) その他に次の操作を行うことができる。

a) ボード上の長方形を表示する。displayBoard

ここで「ボード上の長方形を表示する」とはボード上にあるすべての長方形の属性（幅、高さ、x 座標、y 座標）を表示する（標準出力に出力する）ということ。

b) 配置に関する操作を終了する。　　exit

(3)ユーザはキーボード（標準入力）から(1)の操作名と必要なデータを入力し、プログラムが結果のボード上のデータ（どの位置にどの長方形が配置されているか）を標準出力に出力する。このプログラムは以下のように使用することができる。

a) 起動すると、操作一覧が表示される。

b) ユーザは操作を選択し、要求されるデータを入力する。

c) 操作の実行が終了すると、操作一覧に戻り、exit が実行されるまで操作を選択することができる。

(4)長方形に関する条件：

a) ボード上で同じ幅、高さ、位置をもつ長方形は同一の長方形とみなす。

b) 今回は点および線分は長方形とはみなさない。

c) ボード上に配置できる長方形の数の上限は 10 とする。

(5)操作に関する条件：

a) ある操作によって、ボードから長方形がはみ出す場合には、その入力の値を無効として、操作をやり直す。

b) 操作が無効である場合は、適切なメッセージを出力する。

(6)クラス構成に関する条件：

RectangleEditor：main メソッドをもつクラス

Rectangle：長方形のクラス

Board：長方形を配置するボードのクラス

Command: ユーザからの入力を受け取り、長方形に対する操作を呼び出し、ボードの状態を更新するクラス

プログラムの機能は次の通りである．

- 長方形の作成
- 長方形の移動
- 長方形の削除
- 長方形の拡大・縮小

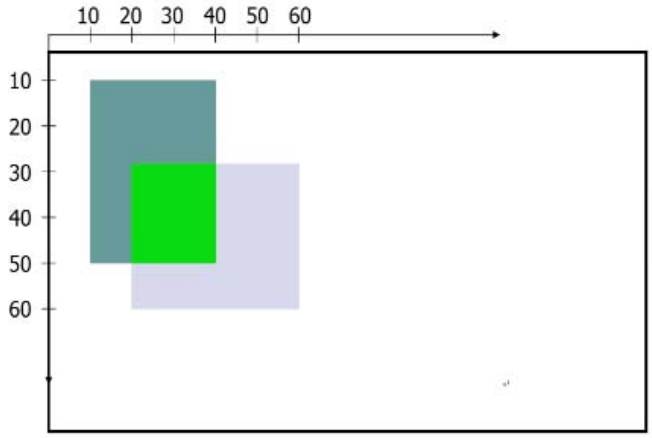


図 7-1 長方形エディタ　イメージ

7.3 ソースコードモデルと検査モデルの生成

検査対象となるソースコードは学生が作成した Java ソースコードを用いる．検査するソースコードは長方形エディタのロジック部分である．ソースコードは 8 個のクラスファイルで構成される．

このプログラムは機能を選択して実行するメニュー形式であるため，検査対象となる機能はメニュー上で ID を割り当てられている個々の機能となる．従ってソースコードモデルに変換するメソッドは個々の機能に対応するメソッドとなる(表 7-1)．

これらのメソッドの配下にあり，仕様モデルの状態に関係するものはアクションモデルとし，関係しないものは非決定性の値を割当てる．割り当ての状態は create の場合を表 7-2 になる．検査者は図 7-2 に示すように検査支援ツール Source2UPPAAL を用いて対象のソースコードに非決定的な値を割り当てるとともに作成した仕様モデルを登録してドラッグ＆ドロップで割り当てる．これを各処理において行い Source2UPPAAL でモデル化し検査モデルを作成する．

表 7-1 動作動詞とメソッドの対応

ID	メソッド	機能
1	create	作成
2	move	移動
3	expand	拡大
4	shrink	縮小
5	delete	削除
6	deleteAll	全削除

表 7-2 アクション作成の場合の割り当て

アクション	メソッド	検査モデルの種類	関係する仕様モデル
重複チェック	isDuplicate	アクションモデル	長方形の重複
はみ出しチェック	isIn	アクションモデル	長方形のはみ出し
長方形個数チェック	isMore	アクションモデル	長方形の個数
追加	add	アクションモデル	長方形の個数
長方形インスタンス作成	new Rectangle	アクションモデル	長方形のはみ出し/重複
出力	write	非決定性の値	–
実数の入力	inputDouble	非決定性の値	–

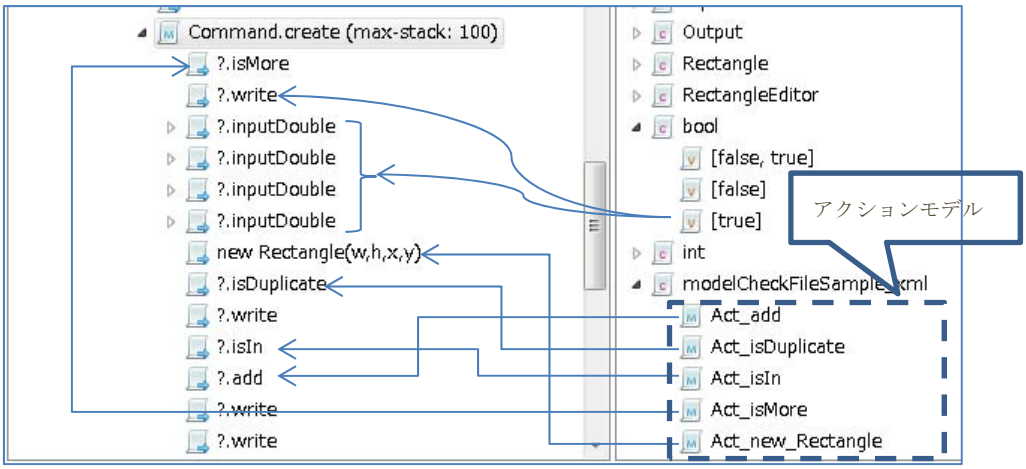


図 7-2 モデルの統合

7.4 仕様モデルの作成

7.4.1 仕様の解析

日本語の表現の自由度が高いため会話で伝える場合と同じつもりで仕様を記述してしまうことが多い．自然言語で記述された今回の長方形エディタの仕様は，まず主語と述語の関係が明確になるように単文化する．主語がない場合は非生物であっても追加する．単文化の例を図 7-3 に示す．

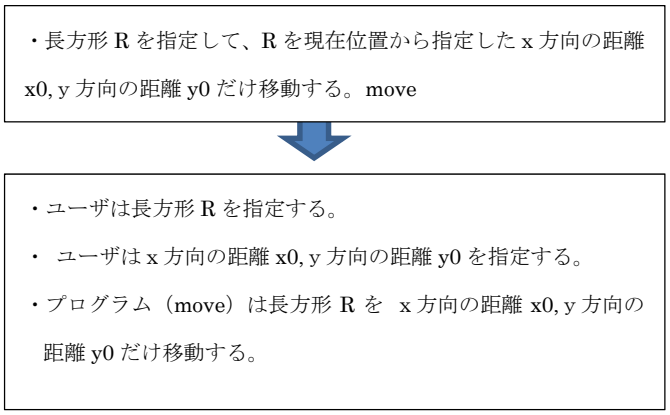


図 7-3 単文化の例

さらに日本語で多く使われる「存在文」(・・ある)は関係性が不明確になりがちなので，モデル化する対象への影響が明確になる「行為文」(・・する)へ変換を行う．今回，存在文から行為文への変換はなかった．主語のみ追加する場合はあった．

次に動作動詞・可算名詞の抽出を行う．英文における動作動詞がメソッドに該当し，可算名詞がクラスに該当するとする研究がある[12]．この考え方を仕様書(設計書)からのモデル化対象抽出する(図 4-22-手順 2 手順 3)．動作動詞はアクション(システムの動作)に置き換えられ，可算名詞はモデル化の対象に置き換えられると考えられる．ドキュメント内から動作動詞と可算名詞を抽出して一覧を作成する．

長方形エディタの仕様書より動詞を抽出し，動作動詞かどうか判別をおこない，抽出された動作動詞の一覧を作成する(表 7-3)．仕様書より名詞を抽出し，可算名詞かどうか判別をおこない，抽出された可算名詞の一覧を作成する(表 7-4)．

表 7-3　動作動詞一覧

移動する
起動する
更新する
作成する
削除する
実行する
終了する
縮小する
出力する
選択する
抽出する
入力する
配置する
表示する

表 7-4　可算名詞一覧

仕様
幅
ボード
図
原点
座標
長方形
アプレット
名前
メソッド
ユーザ
キーボード
点
線分
メッセージ
定義
位置
プログラム
作成
操作

モデル化すべき文章の特定を行う．前節で作成した一覧を用いて動作動詞・可算名詞の両方が含まれている単文化した文章より抽出する（図 4-22-手順 4）．可算名詞をモデル化の対象，動作動詞をアクション(システムの動作)と捉えられるので，この両方が記述されているということは，システムの状態変化を伴う振舞いを記述していると判断できる．長方形エディタの仕様書から整理，抽出した仕様を以下にあげる．

(1) プログラム（create）は幅，高さ，左上の位置（ｘ座標，ｙ座標）を設定して**長方形**を**作成する**.
(2) プログラム（move）は**長方形** R をｘ方向の距離 x0，ｙ方向の距離 y0 だけ**移動する**.
(3) プログラム(delete)は**長方形** R を**削除する**.
(4) プログラム(deleteAll)は**長方形** R を**全て削除する**.
(5) プログラム（expand）は**長方形** R は幅の倍率 mx で幅を，高さの倍率 my で高さをそれぞれ**拡大する**
(6) プログラム（shrink）は**長方形** R は幅の倍率 mx で幅を，高さの倍率 my で高さをそれぞれ**縮小する**.
(7) プログラム(displayBoard)は**長方形**をボード上に**表示する．**
(8) ボード上で同じ幅，高さ，位置をもつ長方形がある場合は，**作成しない**.
(9) 点および線分の場合は**作成しない**.
(10)ボード上の長方形が 10 を超える場合は長方形を**作成しない．**

(11)ボードから長方形がはみ出す場合には，その処理は実行しない.

(8)～(11)については長方形作成の条件である．また仕様には明示されていないが，考慮しなければならない長方形の個数の条件がある．作成する場合，長方形は 0 個でもよいが，削除・移動・拡大・縮小を実行する場合，長方形は 1 個以上必要である．従って長方形の個数の取りうる状態は，0 個， 1 個，2～9 個，仕様上ゆるされる最大の 10 個，エラーとなる 11 個で表わす.

7.4.2　デシジョンテーブル作成

条件（表 7-5），アクション（表 7-6），結果（表 7-7）を抽出してデシジョンテーブル（表 7-8）を作成する．条件は満たされない場合エラーとみなされるもので，正常動作の場合には影響しない

表 7-5 条件

No.	条件	取り得る値
con01	長方形の数が10以下である	満たす／満たさない
con02	追加する図形が点および線分ではない	満たす／満たさない
con03	追加･変化する図形と同じ長方形はない	満たす／満たさない
con04	ボードから長方形がはみ出していない	満たす／満たさない

表 7-6 アクション

No.	アクション	取り得る値
act01	作成する	実行／実行しない
act02	移動する	実行／実行しない
act03	削除する	実行／実行しない
act04	すべて削除する	実行／実行しない
act05	拡大する	実行／実行しない
act06	縮小する	実行／実行しない
act07	表示する	実行／実行しない

表 7-7 結果

結果	取り得る値
長方形の数	0個、1個、2～9個、10個

デシジョンテーブルは次のように作成する．仕様(1)~(9)に記述されているアクション・条件・結果を抽出しそれをデシジョンテーブルの１列として記述する（表 7-8）．複数の条件，結果がある場合はデシジョンテーブルは複数列になる．仕様(1)におけるアクション・条件・結果の対応は図 7-4 のようになる．

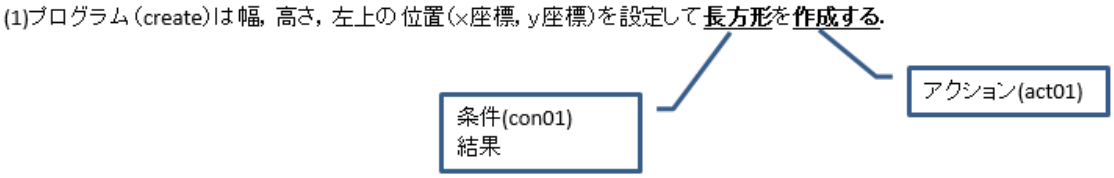


図 7-4　アクション・条件の対応

表 7-8 デシジョンテーブル

項目	種別	仕様(1)	仕様(1)	仕様(1)	仕様(1)	仕様(2)	仕様(3)	仕様(3)	仕様(3)	仕様(3)
長方形を作成する	アクション	実行	実行	実行	実行	実行しない	実行しない	実行しない	実行しない	実行しない
長方形を移動する	アクション	実行しない	実行しない	実行しない	実行しない	実行	実行しない	実行しない	実行しない	実行しない
長方形を削除する	アクション	実行しない	実行しない	実行しない	実行しない	実行しない	実行	実行	実行	実行
長方形を全て削除する	アクション	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない
長方形を縮小する	アクション	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない
長方形を拡大する	アクション	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない
長方形を表示する	アクション	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない	実行しない
長方形の数	条件	長方形の数0	長方形の数1 または 長方形の数2以上10未満	長方形の数2以上10未満	長方形の数10	長方形の数10以下	長方形の数1	長方形の数2以上10未満	長方形の数2以上10未満	長方形の数10
長方形の数	結果	長方形の数1	長方形の数2以上10未満	長方形の数10	長方形の数11以上	長方形の数10以下	長方形の数0	長方形の数1	長方形の数2以上10未満	長方形の数2以上10未満
追加する図形が点および線分ではない	結果	満たす	満たす	満たす	満たす	Nothing	Nothing	Nothing	Nothing	Nothing
追加・変化する図形と同じ長方形はない	結果	満たす	満たす	満たす	満たす	Nothing	Nothing	Nothing	Nothing	Nothing
ボードから長方形がはみ出していない	結果	満たす	満たす	満たす	満たす	満たす	Nothing	Nothing	Nothing	Nothing

デシジョンテーブルから業務機能の振舞いが読み取れる．想定される振舞いは以下のとおりである．

- 点および線分ではない
- 長方形の数は１０個以下
- 長方形ははみ出さない
- 同じ長方形は存在しない（重複しない）

検査式は上記の「想定される状態」を否定した「想定されない状態」にならないことを検査する式になる．

7.4.3　仕様モデルの作成

特定した状態をモデル検査ツール UPPAAL のロケーションに置き換え，システムの動作をエッジに置き換える(図 4-22 手順 6). 状態のロケーション間を関係するシステムの動作のエッジで結ぶことにより仕様モデルを作成する（図 7-5）.

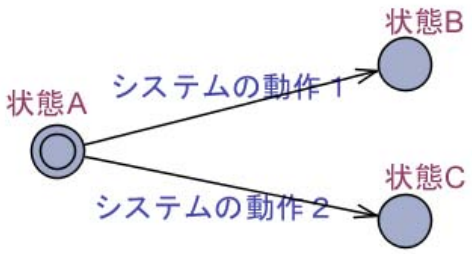


図 7-5 仕様モデルサンプル

長方形の個数が永続的に状態変化するものであるのでこれを仕様モデルとして定義する(図 7-6)．長方形の個数の取りうる状態は 0 個，1 個，2～9 個，10 個，11 個以上なのでこれらをシステムの動作(作成・削除)でつなげばモデルは作成できる．初期設定は必ず長方形０個であるので，6.4 節の与信ブロックのモデルのような非決定性による設定は行なわない. 状態を変化させるアクションは「作成する」と「削除する」であるので各長方形の個数のロケーションをこれらのアクションで接続すればよい. 状態遷移の組み合わせは表 7-9 のとおりである.「すべて削除する」の場合は長方形の個数が１～１０個どの位置にあっても０個のロケーションに遷移する.

表 7-9 状態遷移の組み合わせ

遷移元ロケーション	遷移先ロケーション
0個	1個
1個	2～9個
1個	0個
2～9個	1個
2～9個	2～9個
2～9個	10個
2～9個	0個
10個	11個
10個	0個

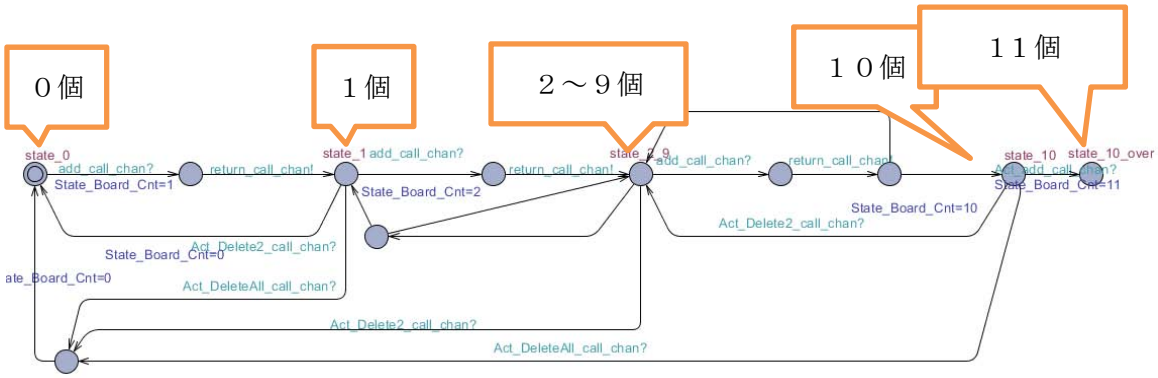


図 7-6 仕様モデル　長方形の個数

「長方形がはみ出さない」と「長方形が重複しない」についてもモデル化する．いずれも長方形の位置・サイズ確定時に決定する．図 7-7 は，はみ出し状態の仕様モデルである．「はみ出している」「はみ出していない」のどちらかの状態に必ずなる．初期状態は「はみ出していない」となる．同様に「長方形が重複しない」のモデルも作成する．

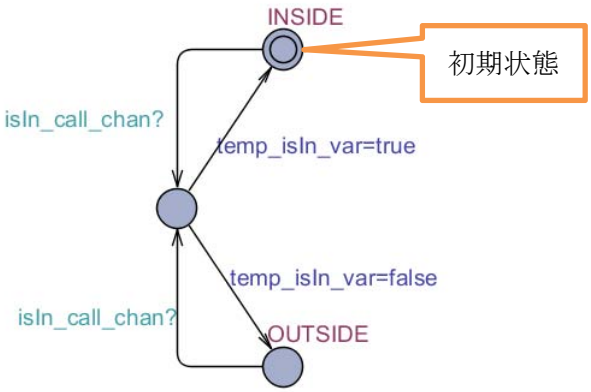


図 7-7　仕様モデル　はみ出し状態

7.5　アクションモデルの作成

アクションモデルを作成する．動作動詞により決定されたメソッドの配下にあり制御フロー及び長方形の状態に関係あるメソッドをアクションモデルに変換する．図 7-8 は長方形の重複を検査するメソッドのアクションモデルである．重複のステータスは UPPAAL のグローバル変数 Act_isDuplicate_return_var に格納される．

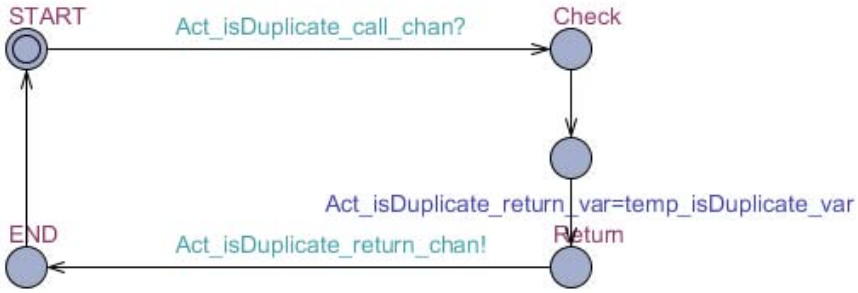


図 7-8 重複チェックのアクションモデル

7.6 検査実施

7.6.1 到達可能性の検査

生成した検査モデルに構造上の問題がないことを確認するために到達可能性の検査を行う．これによりシステムの振舞いを考慮した制御フローの整合性を確認する．

以下のような検査式を実行する．この検査式の意味は「メソッド start()の終点にいつか到達する」である．検査式は図 7-9 のとおり．

```
E<> ( SYSTEM_Command_start(0,0).END)
```

図 7-9 到達可能性の検査式

このような検査を各ロケーションに対して実行する．検査では到達できることが確認でき，不具合は発見されなかった．

7.6.2 システムの振舞いの検査　重複

先にあげた４つの「想定されない状態」に決してならないという安全性の検査を行う．そのうちのひとつ「長方形は重複しない」の検査について反例が出力された．検査式は「機能制御プログラム(start()メソッド)の終了場所において長方形は，重複した状態（Act_isDuplicate_return_var =true）になることは決してない」である．検査式は図 7-10 図 7-10 である．

A[] not (Act_isDuplicate_return_var==true && State_Board_Cnt==2 && SYSTEM_Command_start(0,0).END)

※Act_isDuplicate_return_var

：長方形重複のステータス　重複あり true

※SYSTEM_Command_start(0,0).END

：start()メソッドの終了場所のロケーション

※State_Board_Cnt

：長方形の数

図 7-10　検査式：長方形が重複することの検査

今回は検査結果が「属性は満たされませんでした」となり不具合が発見された．拡大処理で同じ長方形が存在する場合があるとの結果が得られた．UPPAAL が出力した反例であるトレースファイルを解析した結果のグラフが図 7-11 となる．左縦軸がコマンドの実行回数，右縦軸がコマンドの ID 番号である．横軸はトレースファイルのレコード数である．このグラフから作成処理(ID 1)を 2 回実行した後に拡大処理(ID 3)を実行したときに長方形の重複が発生しうることがわかる．

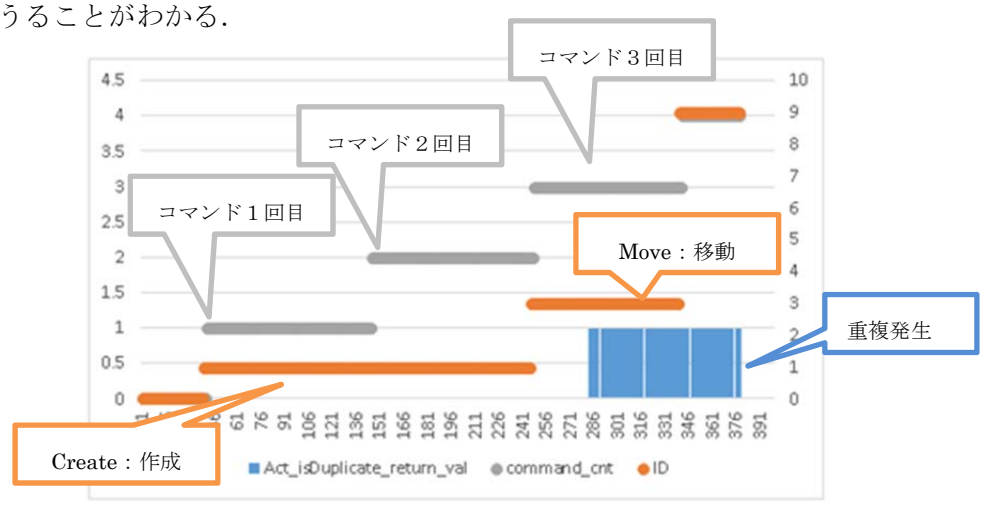


図 7-11　反例解析

実際にプログラムをこの再現条件で実行して不具合の確認ができた．また UPPAAL のロケーションの位置でソースコード位置が把握できるため不具合の原因の特定もできた．原因は長方形の重複をチェックする機能が抜けていたことである．他に移動処理，縮小処理でも同様の検査で同じ不具合が発見できた．

7.6.3　システムの振舞いの検査　はみ出し

先にあげた４つの「想定されない状態」のうちのひとつ「長方形ははみ出していない」についての検査を行う．検査式は「機能制御プログラム(start()メソッド)の終点において長方形は，はみ出した状態（Act_isIn_return_var =false）になることは決してない」である．検査式は図 7-12 である．

A[] not(Act_isIn_return_var==false && SYSTEM_Command_start(0,0).END)

※Act_isIn_return_var

：長方形はみ出しのステータス　はみ出しあり false

※SYSTEM_Command_start(0,0).END

：start()メソッドの終点のロケーション

図 7-12　検査式：長方形がはみ出すことの検査

検査結果が「属性は満たされませんでした」となり不具合が発見された．反例を解析して仕様モデルである長方形の個数の状態とアクション(システムの動作)を確認する．解析結果をグラフにすると図 7-13 になる．

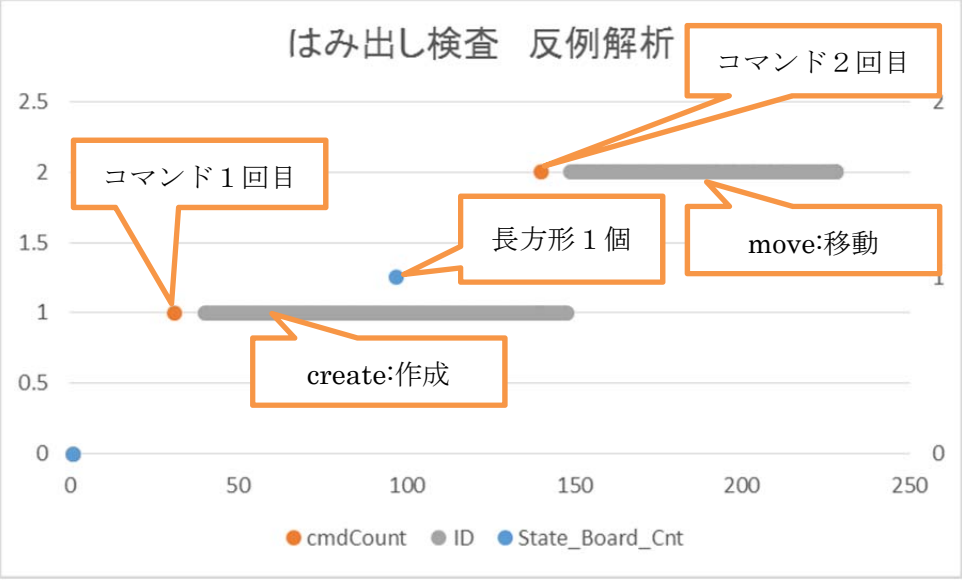


図 7-13　反例のグラフ

確認された動作は以下のとおり．

- 長方形の数は１個
- コマンドの実行回数は２回
- 実行したコマンドは１回目　作成，２回目　移動

この条件だけでは不具合が再現できない可能性が高いため，さらに検査を行う．

次の検査では「長方形エディタを実行する(start()メソッド)の終点において長方形は，はみ出さない状態（Act_isIn_return_var =true）になることは決してない」を検査する．検査式は図 7-14 である．システムの振舞いを再現するために検査式は，先の検査で判明した既知の状態も追加する．

A[] not(Act_isIn_return_var==false && SYSTEM_Command_start(0,0).END && State_Board_Cnt==1 && ID==2 && cmd_cnt==2)

※State_Board_Cnt

：長方形の数

※ID

：コマンド番号 1:create 2:move

※cmd_cnt

：コマンドの実行回数

図 7-14 検査式：長方形ははみ出さないことの検査

この検査でも反例が出力され，長方形がはみ出す場合とはみ出さない場合の両方が存在することがわかる．

反例を解析して不具合の原因の特定を行う．UPPAAL のロケーションはソースコードのステートメントに紐付いているためソースコードの動きも反例を解析することにより判明する．図 7-15 は求めた反例から各メソッドの状態遷移を時系列に並べて，Main メソッドから開始して「長方形がはみ出した」状態で処理が終了するまでの過程を，ひとつのグラフにして表わしたものである．反例は状態遷移のトレースである．横軸は反例のトレースのレコード数，縦軸はロケーションの番号である．

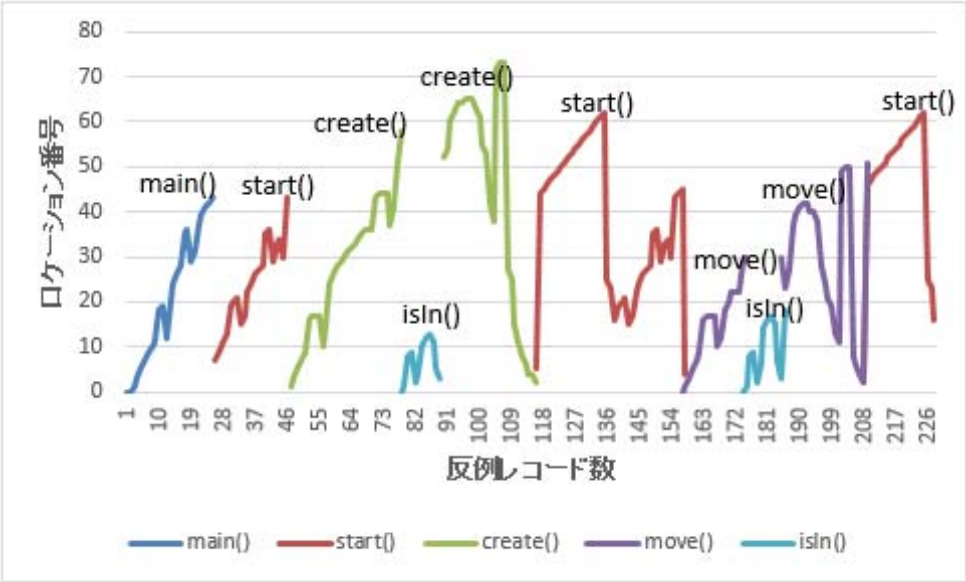


図 7-15 メソッド単位の状態遷移のグラフ

同様に 2 番目の反例（はみ出しが発生しない）をグラフ化する．はみ出しが発生する場合と発生しない場合のグラフを重ねると図 7-16 のグラフができる．実践と点線で表されているように，途中から状態遷移が分岐することがわかる．

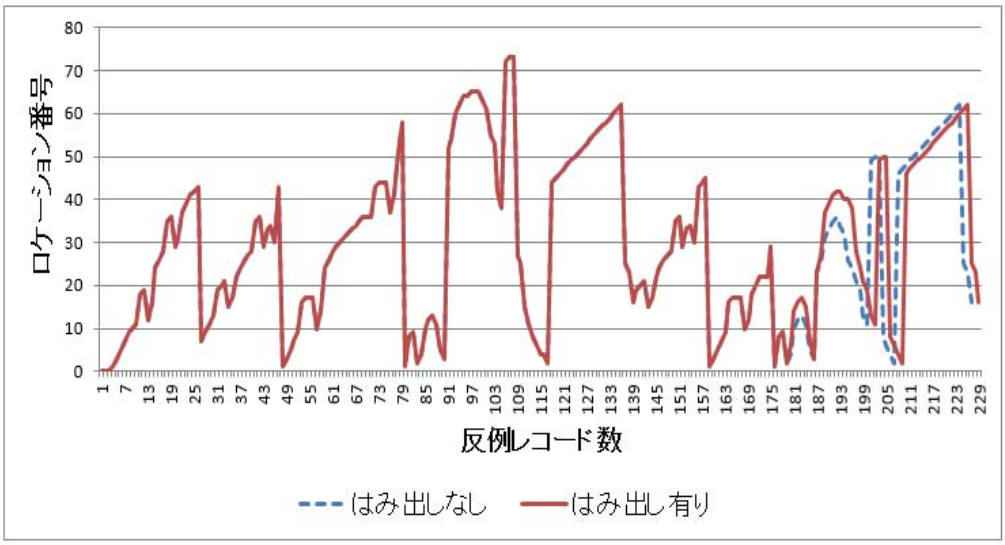


図 7-16 グラフの比較

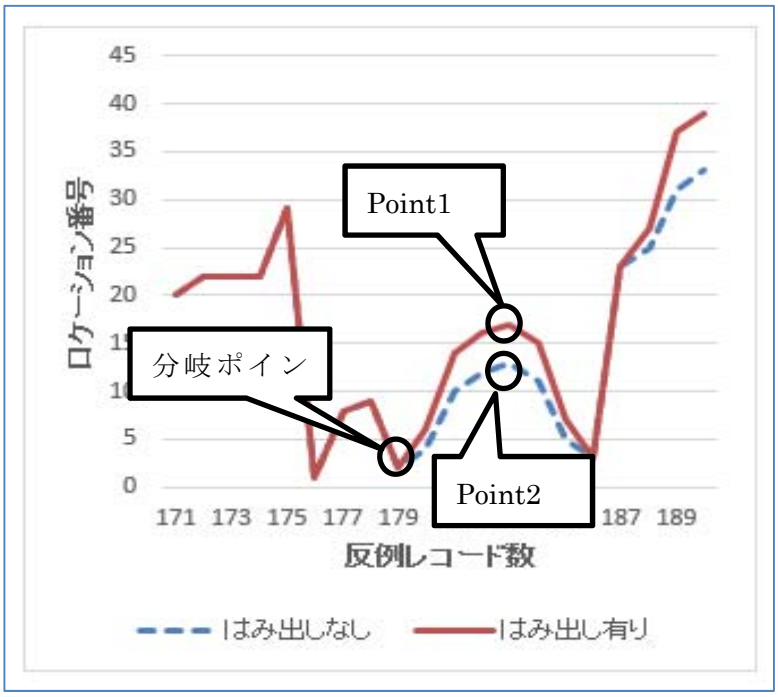


図 7-17 分岐ポイントのグラフ

長方形がはみ出す場合とはみ出さない場合の分岐ポイントがわかるので，その分岐ポイント以降で，はみ出しの判定が変わる場合があるかを検査する．検査式に分岐後のロケーション通過の有無の変数を付加する．図 7-17 にあるはみ出しが発生する場合のロケーション Point1，はみ出しが発生しない場合のロケーション Point2 に通過の有無を表す変数を設けて，再検査を行う．

Act_isIn_return_val==false ⇒true に変換した場合の式は図 7-18 の上段の検査式に，Act_isIn_return_val==true ⇒false に変換した場合の式は下段の検査式になる．

```
A[]  not(Act_isIn_return_var==true  &&  SYSTEM_Command_start(0,0).END
&& State_Board_Cnt==1 && ID==2  && cmd_cnt==2  && checkPoint1=true)
※checkPoint1
  : 図 7-17  Point1 通過有無の変数
```

```
A[]  not(Act_isIn_return_var==false  &&  SYSTEM_Command_start(0,0).END
&& State_Board_Cnt==1 && ID==2  && cmd_cnt==2  && checkPoint2=true)
※checkPoint2
  : 図 7-17  Point2 通過有無の変数
```

図 7-18 検査式：Point１，２を通過する

両検査とも反例が出力されないため，現在見つかっている分岐ポイントが最終的な分岐ポイントであることが判明する．この分岐ポイントに該当するソースコードの部分を調査すればよいということがわかる．ソースコードを調査すると条件判定式の図 7-19 の印の部分に等号が不足していたため移動先の座標が x==0 もしくは y==0 場合，はみ出していると誤って判定していることがわかった．

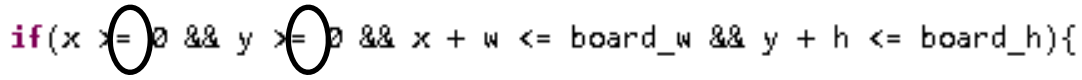


図 7-19 条件式

7.6.4　システムの振舞いの検査　長方形の数は１０以内

先にあげた４つの「想定されない状態」のうちのひとつ「長方形の数は１０個以下」についての検査を行う．検査式は「長方形の数が１０個を超えることはない」である．検査式は図 7-20 である．

A[] not (State_Board_Cnt>10)
※State_Board_Cnt：長方形の数

図 7-20　検査式：長方形の数は１０個を超えることは無い

検査結果は「属性は満たされました」となり，問題がないことが確認できた．

7.6.5　システムの振舞いの検査　点及び線分ではい

この性質については機能がメソッドとして定義されていなかったため，検査対象外とした．

7.7 本章のまとめ

事例の検査では表 7-10 の不具合を発見することができた，不具合の原因は，仕様に記述されている遵守しなければならないシステムの性質のチェック処理が行われていなかったことと，システムの性質を判定する条件式に誤りがあったことである．

表 7-10 不具合の一覧

No	不具合の内容	原因
01	重複した長方形を作成できる	移動する，拡大する，縮小するの処理で重複チェック処理が抜けていたため
02	はみ出しの誤判定 (はみ出していないのに作成できない)	はみ出しを判定する条件式で等号が抜けていて 範囲内でもはみ出していると判定されたため

今回，「想定しない状態」を表す反例と「想定される状態」を表す反例をグラフ化して比較することにより複数のメソッドにまたがる場合でも，システムの振舞いを比較検討することができ，反例の理解を助けられることを示した．

反例が出なくなるまで検査式を繰り返して精度を上げることができるため，不具合の原因

となる分岐ポイントを見つける方法は有効であると考える．また現状追加ポイントの設定は手作業で行っており，設定方法は今後の検討課題である．

本章では，ソースコード検証手法を大学の演習課題で与えた仕様とそれを基に作成されたソースコードへ適用する事例を示した．

自然言語で実際に記述された仕様を提案手法で分析・整理することでモデル化できた．また学生が作ったソースコードを検査対象としたことにより，実際の適用場面における提案手法の有効性も確認できた．今後の課題はより業務寄りのプログラムへの適用による有効性の確認である．

8　適用事例　要件定義段階の UML への適用評価

8.1 目的

本研究が提案する UML 検証手法の有効性の確認を行う．ソースコード検証手法が下流工程のプログラム構築以降を対象としているのに対し，UML 検証手法は上流工程における要件定義段階を対象としている．

UML は記述の自由度が高いため機械的に特定の性質を検証するのは難しく，不正確で曖昧な要求を分析している段階において，開発者が検証可能な要求仕様を定義することも一般的には困難である．これを UML 検証手法で対応できることを確認する．

また今回の適用事例では，非機能要求の 1 つであるセキュリティ要求を対象とした．セキュリティ要求のような非機能要求は，対象システムのアクターとそのアクションと対象データに対する制約として捉える事ができるため，UML 検証手法でシステムの振舞いをステートマシン図として定義すれば検証できる．

セキュリティの規則をモデル検査で検証する研究[62][63][64]は行われているが，いずれの研究もセキュリティの規則を制約として捉えて検査式に変換しており，その変換のアプローチにそれぞれの特徴がある．但しその変換は，検査者がセキュリティ規則を解釈して行っているものである．また検査対象となるシステムのモデル化も手動で直接記述しており，本研究のように UML の知識を持っている開発者であれば，検査できるものではなく，実際に適用する場合，モデル検査のモデル及び検査式を作成する知識が要求される．

8.2 適用事例概要

芝浦工業大学で稼働中の LMS(Learning Management System)である LUMINOUS[70]の BBS 機能の UML モデルを対象とした．機能は，2 つのアクターである学生と教員に対して，ユースケースは「学生が質問を投稿する」「教員が質問に回答する」「学生・教員が話題（質問・回答のセット）を閲覧する」である．ここで，質問・回答にファイルを添付し，それをダウンロードすることができる．本機能のセキュリティ要求は，教員の回答や添付ファイルが不必要に質問者以外の学生に公開されないことと，学生の質問が公開されても個人が特定できないことである．教員は回答や添付ファイルに対して，その公開／非公開を設定できる．

本事例では，情報セキュリティの国際評価基準（ISO/IEC15408）[75]である Common Criteria[60]の用語の意味に基づき，対象システムのアクターやデータのセキュリティ属性に関する規則を定義し，機能要求の振舞いのモデルと対応付けることにより，その規則を満たしているかを検証する．

Ogata ら[67][68]は UML を用いたモデル駆動要求分析手法を提案してきた．本研究は彼らが提案した UML 要求分析モデルに基づきモデルを作成する．

8.3 Common Criteria について

CC は，Part 1：概要，Part2:セキュリティ機能コンポーネント，Part3：セキュリティ保証要件から構成されている．本研究では，「概要」に定義された用語から，「セキュリティ機能コンポーネント」を定義する「セキュリティ機能方針(SFP)」は図 8-1 の左のモデルのようにモデル化される．セキュリティ機能コンポーネントを解釈し，対象システムのモデル要素との対応を定義する．操作にはアクションが対応し，操作の実行主体であるサブジェクトにはアクターが対応する．さらに，操作の対象であるオブジェクトはオブジェクトノードとして定義されている．

図 8-1 は「SFP は複数の規則から構成されており，規則はあるサブジェクトが実行主体となる操作に対して適用される．操作はオブジェクトをその対象とし，規則はサブジェクトとオブジェクトの特性として定義されるセキュリティ属性によって制御される」ことを表している．

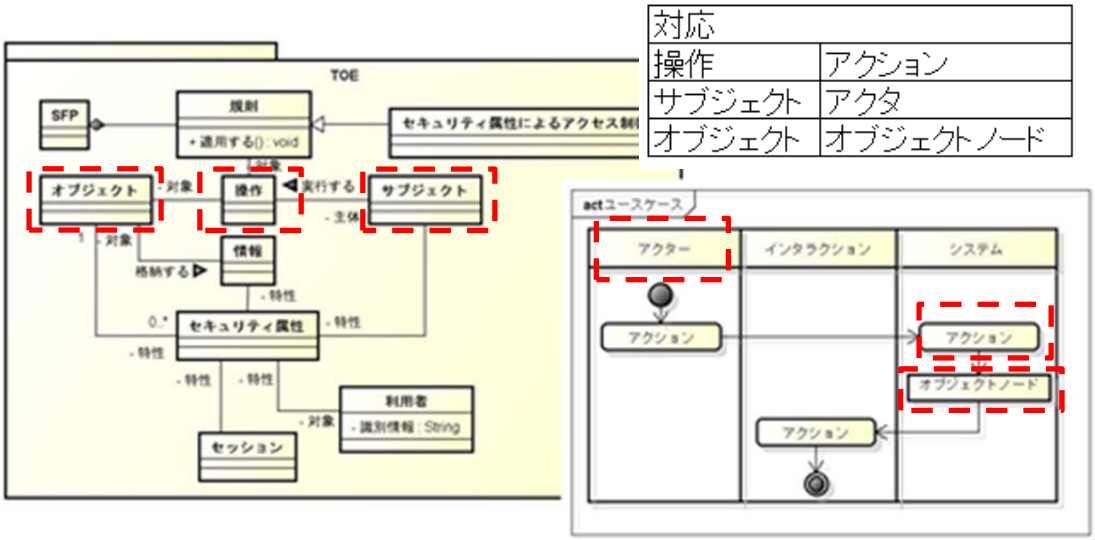


図 8-1　SFP とユースケースの対応

8.4 仕様モデルの作成

つぎに，対象システムの「セキュリティ属性によるアクセス制御」をサブジェクト・操作・オブジェクト・サブジェクトのセキュリティ属性・オブジェクトのセキュリティ属性からなる規則の集合として定義する．CC のセキュリティ機能コンポーネントはクラス，ファミリ，コンポーネントの階層構造を持ち，コンポーネントは複数のエレメントと他のコンポーネントとの依存関係を持つ．このことから，アクセス制御のセキュリティ要求をアクセス制御機能のコンポーネント（FDP_ACF）に基づき，定義する場合，依存関係を持つ全てのコンポーネントに対して，規則を整理することで，セキュリティの深い知識を持たない開発者でも，表 8-1 の SFP を漏れなく定義することができる．表 8-1 は，本学で稼働中の

LMS(Learning Management System)である LUMINOUS の BBS 機能のモデルから定義された SFP である．

表 8-1 の定義手順は以下のとおりである．まず，対象システムのサブジェクト（アクター）とオブジェクトに対してセキュリティ属性を定義する．要求分析モデルのアクティビティ図より，すべてのオブジェクトを抜き出し，それぞれを対象とするアクションを抽出する．上記のセキュリティ要求を満たすためには，学生アクターが個人を識別する情報をもたなければならない．この場合は，対象システムにおける学生の属性である「学籍番号」がそのまま，個人を識別するセキュリティ属性とみなせる．話題や添付ファイルは教員の意思により，その公開／非公開を決定し，学生は公開されているものは閲覧でき，非公開のものは閲覧できないことになる．すなわち，これら 2 つのオブジェクトがこの制御を識別できるように「公開／非公開」というセキュリティ属性を持つ．これらのセキュリティ属性を含め，表 8-1 のようにセキュリティ機能方針（SFP）を整理する．

表 8-1 セキュリティ機能方針

サブジェクト		オブジェクト		操作		ルール		
アクタ	セキュリティ属性	クラス	セキュリティ属性	ユースケース	アクション	FDP_ACF.1	FMT_MSA.3	FMT_MSA.1
		…						
学生	役割(学籍番号)	話題	公開非公開	質問を投稿する	話題を生成する		ルールB1	
		日時		質問を投稿する	現在日時を取得する			
		投稿者	役割	質問を投稿する	投稿者を取得する			
		投稿内容		話題を閲覧する(学生)	投稿内容を取得する			
		添付ファイル	公開非公開	話題を閲覧する(学生)	添付ファイルをダウンロードする	ルールA		
				質問を投稿する	添付ファイルを生成する		ルールB2	
教員	役割(教員)	…						
		話題	公開非公開	質問に回答する	<質問番号>により選択された話題を取得する			
				質問に回答する	回答を追加して話題を更新する			
					話題の公開非公開を公開に変更する			ルールC1
					話題の公開非公開を非公開に変更する			ルールD1
		日時		質問に回答する	現在日時を取得する			
		投稿者	役割	質問に回答する	投稿者を取得する			
		投稿内容		話題を閲覧する(教員)	投稿内容を取得する			
		添付ファイル	公開非公開	話題を閲覧する(教員)	添付ファイルをダウンロードする			
				質問に回答する	添付ファイルを生成する		ルールB3	
					添付ファイルの公開非公開を公開に変更する			ルールC2
					添付ファイルの公開非公開を非公開に変更する			ルールD2

つぎにセキュリティ属性の持つべき状態を識別し，これらの間の遷移をステートマシン図により図 8-2 ように定義する．ここでの状態遷移のイベントが，「話題」というオブジェクトの属性「公開／非公開」の値（未設定・公開・非公開の 3 つ）を遷移させる属性の更新アクションに対応する． このイベント名が対象システムのアクティビティのアクション名と対応付けられる．規則は，表 8-1 の要素を用いて，CC の機能コンポーネントの構造ならびに依存関係から選定されたコンポーネント毎に，制御が必要なアクションに対して，表 8-2 のように定義する．

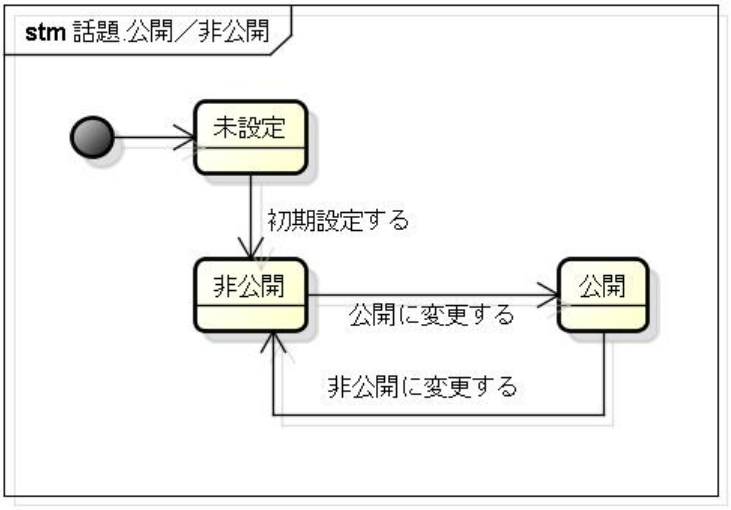


図 8-2 セキュリティ属性の定義

表 8-2 アクセス制御規則の定義

ルールA	アクション開始時，添付ファイル.公開／非公開==公開 ‖ 投稿者.役割==学生.役割
ルールB1	アクション終了時，話題.公開／非公開==非公開
ルールB2	アクション終了時，添付ファイル.公開／非公開==非公開
ルールB3	アクション終了時，〈話題.公開／非公開==公開ならば添付ファイル.公開／非公開==公開 ‖ 添付ファイル.公開／非公開==非公開〉 && 〈話題.公開／非公開==非公開ならば添付ファイル.公開／非公開==非公開〉
ルールC1	アクション開始時に話題.公開／非公開==非公開ならば，アクション終了時に話題.公開／非公開==公開
ルールC2	アクション開始時に添付ファイル.公開／非公開== 非公開ならば，アクション終了時に添付ファイル.公開／非公開== 公開
ルールD1	アクション開始時に話題.公開／非公開== 公開ならば，アクション終了時に話題.公開／非公開==非公開
ルールD2	アクション開始時に添付ファイル.公開／非公開== 公開ならば，アクション終了時に添付ファイル.公開／非公開==非公開

8.5 UPPAAL モデルへの変換

図 8-3 に示すように，アクションの対象となるデータとその主体であるアクターのセキュリティ属性に関わる基本的な属性操作の結びつきに着目して，UML モデルを UPPAAL モデルに変換し，そのセキュリティ機能方針（SFP）の定義から検査式を自動生成して，到達可能性，安全性に関する性質を検査する．UPPAAL モデルの作成，検査式の作成・実行には，モデル検査の非専門家でも実施できる支援ツール UML2UPPAAL を利用した．

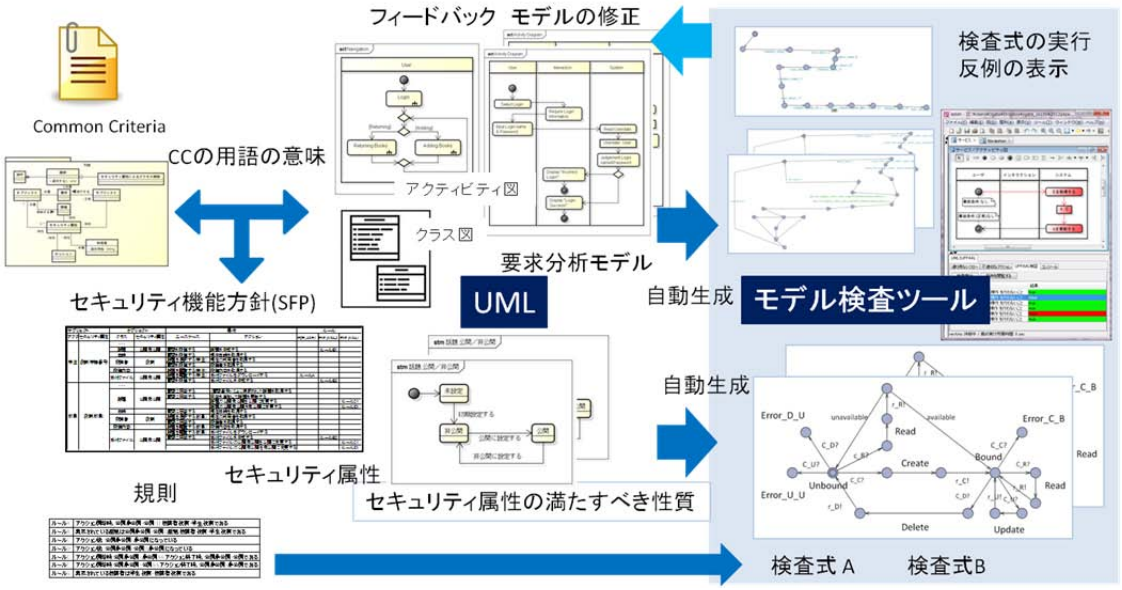


図 8-3UML モデルへのモデル検査の適用

8.6 検査実施

すべてのユースケースを統合した UML モデルは UPPAAL モデルの検査モデルに変換される．このモデルに対して検証を行い，ユースケースモデルの問題点を洗い出す．

8.6.1 到達可能性の検査

UPPAAL モデルの各ロケーションへの到達可能性の検査を行うことにより，アクティビティ図上の全てのフローが通過可能であることを確認する．検査式内のロケーション名は，アクティビティ図（ここでは識別子 ACT02）のアクション等のノードに対して，UML2UPPAAL 内で付与されたロケーション名である．

初期設定が行われた場合，そのモデル上で状態遷移が発生するので，初期設定を表すロケーションへの到達可能性の検査で実行可能であるか確認する．検査式は図 8-4 のとおり．検査式の意味は「話題の初期化のロケーションにいつか到達するか」である．ここで state_topic(0).Topic_init は，「話題」オブジェクトのセキュリティ属性のステートマシン図の「未設定」から「非公開」への遷移に対応するロケーションである．

```
E<> ACT02.llufz_h5x4bhag_fclelv_1g4fo_bf18c559adbf09632ad
```

図 8-4 到達可能性の検査式

今回の検査では「属性は満たされませんでした.」となり到達できないことが判明した，不具合が発見された．このような検査を各ロケーションに対して実行する．

8.6.2　システムの振舞いの検査

表 8-2 に定義したセキュリティの規則の検査を行う．セキュリティの規則は特定の時点にて満たされなければならない，セキュリティ属性の状態である．特定の時点は表 8-1 より特定のユースケース（アクティビティ図）の特定のアクションの開始前または終了後の遷移を選択することにより指定可能であることから，これを検査モデルのロケーションに置き換える．またセキュリティ状態は，UML2UPPAAL がモデル生成時に自動的に変数を割り当てる(例　公開/非公開　1: 未設定　2: 非公開　3: 公開)ので，これらのロケーションと変数を用いて検査式を生成する．この検査式によりセキュリティ規則の遵守について確認できる．検査式は図 8-5 のとおり．検査式の意味は「アクション開始時には常に話題は非公開　アクション終了時には常に話題は公開」である．

検査対象セキュリティ規則：
アクション開始時 公開／非公開＝公開 && アクション終了時 公開／非公開＝非公開(ルール C1)

A[] (ACT03.Assert01 imply state_topic[0]==3)
A[] (ACT03.Assert02 imply state_topic[0]==2)

図 8-5 セキュリティ規則　検査式

特定の時点の例：アクション開始時，終了時はアクティビティ図上で図 8-6 のようにノートを付加することにより UML2UPPAAL でモデル化する時にロケーションが追加される．
今回の検査では「属性は満たされました」となり不具合は発見できなかった．

図 8-6 アクション開始時，終了時の位置指定

8.7 本章のまとめ

本章ではセキュリティ要件の中でもアプリケーションのロジックに深くかかわるアクセス制御方針が要求分析の段階において，適切に反映できているかを UML モデルに対してモデル検査技術を使って検証する方法を示した．BBS 機能モデルへの適用実験では，「公開／非公開の初期設定の検査」により，初期設定アクションの漏れが発見できた．その他の表 8-2 の性質については，満たされることが確認できた．

表 8-3 不具合の一覧

No	不具合の内容	原因
01	公開／非公開の初期設定の漏れ	UML記述もれ

UML2UPPAAL は UML で書かれた要求仕様書を UPPAAL モデルに変換し，その仕様に登場するアクターやデータのセキュリティ属性に関する規則を検査することを支援するツールである．本ツールの特徴は UML の知識を持っている開発者であれば，特別な訓練なしにモデル検査の網羅的な検査の恩恵を受けられることであり，今回不具合を発見できたことによりその有効性が確認できた．

9　考察とまとめ

9.1 考察

本研究の提案は，開発現場において開発者が仕様上で読み取れる「システムがもたらす業務機能の振舞い」について，想定外の振舞いをすることがあるかを検証する．開発現場でモデル検査を利用するためには 2.3 節で述べた問題があり，本研究はこれらについても検討を行った．

9.1.1　ソースコード検証手法の検証対象

本研究における提案，ソースコード検証手法は，メソッドをモデル化の単位にして検証している．そのため，仕様上で使われている機能がメソッドとして定義されていない場合は，ソースコードを読んで該当する機能が記述されているステートメントを特定して，仕様モデルを紐づける必要がある．これはソースコードの量が多いと難しい作業になると思われるが，実際の開発現場で開発されるシステムは複数の開発者が開発を行うため，よほどの低水準な機能でない限り，業務処理に関わる機能はあらかじめそのプロジェクトの規約に則って定義され，メソッド化されている．従って機能とメソッドが紐付けできないことは発生しないはずである．

9.1.2　検証できる規模

業務機能は業務処理の最小構成単位であり，システムで行われるシステム機能とそれを利用した手作業であるシステム利用作業を合わせたものである(図 1-1)．デシジョンテーブルの作成はテストケースの作成と同様にこの業務機能の単位に行っている．業務機能は複数のシステム機能とそれを利用するエンドユーザの操作から成り立っているため，エンドユーザが現実的な時間内で入力可能な条件とその際に期待される動作を記述することになることから，デシジョンテーブルの大きさは記述可能な範囲に留まると考える．関係がある機能が増えれば，デシジョンテーブルの数は増えるが，個々のデシジョンテーブルの大きさは大きくは変わらない．各業務機能をその依存関係を考慮して，順番に検査することにより，システム全体をモデル検査することができると考える．モデル検査を適用できる規模は，テストを行うことと同様の単位と計画によって実施できると考える．

9.1.3　状態爆発への対応

本研究ではメソッド単位で抽象化したソースコードの段階的なモデル化により対応した．検査対象として不要なメソッドを抽象化することにより状態の削減を行った．適用事例では状態爆発を起こすことなく検査できた．

本研究の提案手法を用いても状態爆発を起こすことはある．例えばメニュー機能をモデル化した場合，何度でも繰り返し処理が可能になるため状態爆発の状態になることがある．そういった場合は通常のテストと同様にシナリオを考えてモデルの動きに制限を加えるこ

とにより状態爆発を防げる．すべてをモデル検査に任せるのではなく検査するシナリオを検討することも必要である．

他の研究ではスライシングを利用して検査対象を絞っているものもある．本研究ではまず仕様の検証を検討することを優先したため，利用していないが今後検討していく予定である．

9.1.4 ソースコードのモデル化への対応

検査支援ツール Source2UPPAAL によりメソッド単位にして制御フローをベースに検査モデルを作成して対応を行った．言語は Java に限定されるが複雑な制御フローであってもモデルは可能になった．但しモデル化するメソッドの選択作業は検査者が行っており，大規模なソースコードのモデル化を行った場合，作業量が増加してモデル化が困難になる可能性がある．

ソースコードと仕様をともに理解した複雑な検査式を作成しなくても検査できるようすることは，デシジョンテーブルからシステムの振舞いを読み取り，その否定系を検査式にすることで対応を行った．例えば「状態 P になる」というシステムの振舞いが読み取れれば検査式は「not　状態 P になることは決してない」となる．適用事例ではこの方法で検査式を作成して検査を行うことができた．現状，システムの振舞いを読み取るためには，本研究でデシジョンテーブルを拡張した項目である「結果」を検査者が整理する必要がある．基本的には重複する「結果」を排除しつつ，順列組合せで考えればシステムが実現する業務機能の振舞いになるので検査式作成の自動化は可能である．

9.1.5 仕様のモデル化への対応

本研究では自然言語で記述された仕様をデシジョンテーブルで整理して，それをモデル化することにより仕様に記述されている業務機能の振舞いを検査できるようにした．デシジョンテーブルの作成は業務機能の単位に行う．業務機能は複数のシステム機能とそれを利用するエンドユーザの操作から成り立つため，エンドユーザが現実的な時間内で実施可能な動作を記述することになる．

自然言語で記述された仕様を整理するためにデシジョンテーブルに実行した「結果」の項目を追加している．拡張項目「結果」よりありうるべき状態を抽出しロケーションに置き換え，それらの状態遷移がデシジョンテーブルの「アクション」により起こるものと捉え，それらをつなぐことにより仕様モデルを作成できる．この方法ならば一般的な開発者でも理解してモデルを作成することができる．また仕様モデルを検査するため検査式が複雑にならないため，UPPAAL の CTL はサブセットであるが問題にならないと考える．

またデシジョンテーブル作成の労力もテストケースを作成する場合のデシジョンテーブルの作成の労力と同じと考える．デシジョンテーブルにはシステムがもたらす業務機能の振る舞いを構成するシステム機能の状態が変化のみ記述すればよく，エラーメッセージ等

の表示のみでシステムの状態が変化しない場合は記述する必要はない．従って本研究のデシジョンテーブル作成の労力は正常終了を期待結果とするテストケースを作成する場合のデシジョンテーブルの作成の労力と同じと考える．

本論文の適用事例である「長方形エディタ」では１アクションについての状態変化は最大で４パターンであった．デシジョンテーブル作成自体の労力よりも仕様書・設計書からアクション・条件・結果（状態）を取り出す部分にかかる労力の方が大きく，これはドキュメントの記述方法により大きく左右される．この点は通常のデシジョンテーブルも同様である．

9.1.6　検査を行うモデル検査の非専門家の開発者への対応

モデル検査の非専門家である開発者がモデル検査を使った検査ができるようにするため知識不足を補う試みをおこなった．現状の一般的な開発者が持っているドメイン知識，開発に関する知識に基づいた定型的な方法で行うことがよいと考えた．モデル検査の大まかな流れは以下の 3 つのとおりである．

・モデル検査を行うためにはモデルを作成する
・検査式を作成する
・検査結果を考察する

モデルの作成は対象を Java もしくは UML モデルと限定して，直接モデル検査器をさわらなくても検査できる検査支援ツールの導入により対応した．これにより一般的な開発者でも業務ドメインの知識に基づいてモデル化の指示を行い，制御フローに基づいた検査モデルは自動生成することができるようになった．但しモデル化指示の煩雑さは課題として残っている．

検査支援ツールは以下の二つである．それぞれ Java ソースコード，UML モデルを UPPAAL モデルへ変換する．

Java ソースコード：Source2UPPAAL(eclipse プラグイン)
UML モデル：UML2UPPAAL(astah*プラグイン)

これらのツールは独立行政法人情報処理推進機構技術本部ソフトウェア高信頼化センター（SEC: Software Reliability Enhancement Center）が実施した「2012 年度ソフトウェア工学分野の先導的研究支援事業」の一環で開発したものである．

検査式の作成については，仕様を分析して，拡張したデシジョンテーブルに記述することによりシステムの振舞いを捉えることにより対応した．デシジョンテーブルは通常の開発作業でもテスト時に作成するものであり，一般的な開発者でも作成することに違和感は少ない．デシジョンテーブルに拡張項目として「結果」を記述して，整理することによりシステムの振舞いが記述できる．それを検査式に変換すればよいだけなので特別な知識は必

要ない．デシジョンテーブルの項目の内容と検査モデルの項目の紐付けができていないため手動による変換を行っているが，検査モデル作成時に自動化できると考える．

9.1.7　検査結果（反例）の解析への対応

反例が持つデータ項目を選択してグラフ化する対応を行った．ロケーションの状態遷移を追うことと，他のロケーションとの対比が容易になった．表示パターンは継続的に状態変化する場合には折れ線表示が適しており，一度状態変化した後その状態が継続するような場合は，ポイント表示が適している．値が true/false のような二者択一になる表示は，現状棒グラフで表示しているが見やすいとは言えず表示パターンの検討が必要である．

また，想定される状態を表す反例と想定されない状態を表す反例を比較することにより反例の理解を助ける手法を提案した．状態遷移を規定する式を付加して繰り返し検査することにより反例の精度が上がるため，不具合の原因がある条件分岐を見つける方法として有効であると考える．また現状，追加ポイントの設定を手作業で行っている．作業をもっと効率的にする方法の検討が必要である．

今回の適用事例は単純な例であった．有効性を明確にするため，もっと複雑なモデルへ適用する必要がある．

9.1.8　本研究の有効性

本研究は，通常テストの「業務処理手続き」の検査の積み重ねによるシステム機能の検証ではなく「システムが持つべき性質(状態)」を検査するという別の視点からのシステム機能の検証により，通常テストでは発見できなかった不具合を発見する．

開発現場では不十分な要件定義や実装時のミス・誤解等により人手では試すべき想定ケースが多すぎて網羅しきれず原因の特定が困難な不具合が発生する．原因の特定が困難な不具合は，仕様に記述されたシステムの振舞いとソースコードが実現したシステムの振舞いの不一致が原因であることが多い．こうした不具合は，どんなにテストケースを綿密に作成しても見逃しが出るため発生する．これは業務における個々の手続きをテストして，それを積み重ねてシステム全体の正しさを確認するという方法をとる限り完全に逃れることは困難である．なぜならばテストケースの網羅性の保証が業務フローの手続きを追うことに依っており，そのためテストの期待結果は正常終了を想定したものになりがちで，網羅性が保証されているとは言い難いからである．

このようにしてテストケースから漏れたケースで発生する不具合は原因の想定が難しく解決に時間がかかる．テストの期待結果が正常終了を想定したものになるため，「システムがある条件下で正当でない状態に陥ることは決して起こらない 」という視点での検証 が網羅されていないからである．入力データの整合性以外の原因で「正当でない状態に陥る」ことを想定したテストケースを作ることは非常に困難である．

モデル検査は状態遷移系として定義されたシステムに対し，要求する性質を論理式で記述

し，状態遷移系がこの論理式を満たすことを検査する手法である．したがってシステムの検査モデルさえ構築できれば，テストケースの検討をせずとも検査モデル上でシステムの振舞いはすべて網羅されるため，「正当でない状態に陥る」ことを想定した仕様とソースコードの振舞いの不一致の検証ができると考える．

9.1.9　本提案手法の適用性の限界

本研究が提案する検証の手法はメソッドをモデル化の単位としているため，もし仕様上で定義されている機能がメソッドとして定義されていない場合は，仕様とソースコードを紐づけるポイントが見つからないためモデル化ができない．通常の開発において，業務処理に関わる機能はあらかじめ開発プロジェクトの規約に則って定義され，メソッド化されているはずであるから，もしそのような箇所が見つかった場合は規約が守られていないということになり，コーディングの規約レベルの問題があることが判明する．その場合はソースコードレビューを行い，プログラムを修正後，再度検査を行う対応となる．

また，本研究の対象はシステムがもたらす業務機能の振舞いであるため，システムの状態の遷移が捉えられることが前提となる．システムの状態の遷移が捉えやすいため，本研究の適用対象は主にシステムの品質でいう機能性となる．効率性・操作性・信頼性等の非機能要件に関する性質については状態の遷移が捉えにくいため，検査モデルに表すことが難しく適していない．但しセキュリティに関しては一部，機密性（confidentiality）のアクセス制御のようにシステムの振舞いに関係する部分については適用可能と考えられ，2014 年度 IPA「ソフトウェア工学における先導的研究支援事業」において検証を実施している．

9.2 本研究のまとめ

業務システムの開発現場では不十分な要件定義や実装時のミス・誤解等により不具合がシステムに含まれる．そして綿密にテストケースの作成を行っても不具合が発見できない場合は多々発生する．テスト以外でシステムが仕様を満たしているかを判断する基準は，レビューの実施回数であったり，不具合発見率であったりするがシステムが仕様を満たしていることを客観的に証明する手段としては弱い．こうした不具合は大きな手戻りを発生させたり，障害対応への大きな工数を発生させたりして開発プロジェクトの進行を阻害する可能性が高い．

そうした不具合が発見できないのは，複雑な条件の組合せや同期・タイミングの問題で発生するため，検査を行う開発者がテストケース作成や不具合対応時に不具合を想定できないことが大きな原因といえる．

これはシステムの振舞いに関係していると考えられる．多くの開発者は業務の手続きの連携を理解することは得意であるが，自分たちが構築するシステムがどのように振舞うべきであるかを考えることは得意ではない．そのため不具合の原因が単純なコーディングミスや業務手続きの連携以外にあると想定が困難になる．

モデル検査はシステムの振る舞いを確認するには優れた手法であり，妥当性・実現可能性・整合性をもれなく網羅的に検査するには有効であるため，これを利用してシステムの振舞いに関する不具合を発見することができると考えて，本研究の検証手法を提案した．

提案した検証手法は，一般的な開発者でもモデル検査技術を用いてシステムに想定外の振舞いがないかをソースコードを対象として検証するソースコード検証手法である．またソースコードの代わりに要件定義段階のUMLモデルを検査対象とするUML検証手法についても提案した．

また開発現場でモデル検査を利用するために2.3節で記述した問題がありこれらについても対応した．

- 状態爆発への対応
  メソッド単位で抽象化したソースコードの段階的なモデル化により対応した．
- ソースコードのモデル化への対応
  検査支援ツールによる検査モデルの自動作成により対応した．
- 仕様のモデル化への対応
  仕様を拡張したデシジョンテーブルで整理してモデル化することにより対応した．
- 検査を行う非専門家の開発者への対応
  直接モデル検査器をさわらなくても検査できる検査支援ツールの導入により対応を行った．
  Java ソースコード：Source2UPPAAL(eclipse プラグイン)
  UML モデル：UML2UPPAAL(astah*プラグイン)

- 検査結果（反例）の解析への対応
  反例が持つデータ項目を選択してグラフ化する対応と結果が異なる２つの反例の状態遷移の分岐ポイントを探って比較することにより不具合原因のある箇所を特定する対応を行った．

適用事例を使って提案手法の有効性の評価を行った．

１つ目の事例は，実際の発生した不具合を基に作成したサンプルプログラムに適用したものである．対象は販売システムの受注処理の１機能である与信チェックである．サンプルプログラムからソースコードモデルを作成，仕様からデシジョンテーブルを作成し，そこから仕様モデルを作成した．ソースコードモデルと仕様モデルを結合して作成した検査モデルを検査し不具合の原因を特定した．今回サンプルプログラムで検証をおこなったが，不具合の本質はソースコードの大小とは無関係であるため，ソースコードをモデル化する作業量の問題はあるとしても実際のソースコードに適用しても提案手法で不具合を発見できると考える．

２つ目の事例は，実際に提示された仕様に基づき作成されたソースコードに適用したものである．対象は芝浦工業大学のプログラミング演習で使われた課題の長方形エディタプログラムである．ソースコードからソースコードモデルを作成，仕様は一度整理してからデシジョンテーブルにまとめて仕様モデルを作成した．ソースコードモデルと仕様モデルを結合して作成した検査モデルを検査し不具合の原因を特定した．学生が作ったソースコードを検査対象としたことにより，不具合の内容自体は長方形エディタに類するものになるが，ソースコードに不具合が含まれる過程は実際の開発現場での場合と同じである．従って開発現場で発生する不特定の不具合についても，この事例と同じ過程，ソースコードへの機能の挿入ミスや条件式の記述ミス，で不具合が含まれるならば，不具合原因の特定ができると考える．

３つ目の事例は，要件定義段階の UML モデルに適用したものである．芝浦工業大学で稼働中の LMS である LUMINOUS[70]の BBS 機能についての UML モデルを対象として検査を行った．調査する性質はセキュリティ要件の中でもアプリケーションのロジックに深くかかわるアクセス制御である．結果として初期設定アクションの記述漏れが発見できた，他のセキュリティ要件については問題ないことが確認できた．システムの振舞いが表わせていればソースコードでなくても適用可能であり，上流工程における要件定義等の検証にも本提案手法が適用できることが確認できた．

以上のように本論文では，本研究が提案する手法を用いて一般的な開発者でも通常のテストとは違うモデル検査を用いる観点からの検査により，通常テストを補完して未検出の不具合を発見してシステム開発における品質の向上に貢献できることを示した．

9.3 今後の課題

本研究の提案の大きな特徴は拡張したデシジョンテーブルを作成することにより，仕様モデルを作成でき，その仕様の可否を自動生成可能な検査式で安全性を用いて確認できる点である．仕様モデルは手動で作成しているため，モデルの精度が検査者のスキルに左右されるので，今後ツール化を検討し，精度の向上を目指す．また仕様モデルを作成する時の仕様の解析，デシジョンテーブルの作成，仕様モデルへの変換については現状では自動化されていない部分であり，手順を整理してツールに反映していきたい．

検査結果（反例）の解析についてはまだ充分検討しきれていない部分がある，直感的に理解できる表示についてさらに検討したい．また，正常な状態を表す反例と異常な状態を表す反例を比較することにより反例の理解を助ける手法を提案したが，現状追加ポイントの設定を手作業で行っている．作業をもっと効率的にする方法の検討が必要であり，今後自動化を目指す．

状態爆発への対処は本研究では主に段階的なモデル化により対応するが,分散型のモデル検査を行う研究[58]もあり，今後そちらの方面も検討したい．

既存のシステムを別の環境へマイグレーションする場合，通常のテストでは言語が違うため変換コードの正しさは，テストを実施した範囲でしか確認できないが,本研究の提案手法を用いればマイグレーションによるシステムの再構築時のシステムの振舞いの保証が可能になると考える．多言語対応とともにマイグレーションへの対応についても今後検討して効率的な検査ができるように目指したい．

モデル検査技術の知識を殆ど持たない一般的な開発者を支援するためのツールとして eclipse プラグインで実装した Source2UPPAAL を利用した．これは Java 言語を対象にソースコードを制御フローベースに UPPAAL モデルへ変換するツールである．社内等でリサーチすると既に仕様書がない COBOL の仕様を確認するための要望があることもわかっており，今後は Java 固有の処理と他の言語の相違点などを整理した上で広く使ってもらえるように対応言語を増やしていきたい．

開発者がモデル検査技術の知識を持たなくても要件定義段階において要件定義の不整合の早期発見ができるツールとして，astah* プラグインで実装した UML2UPPAAL を利用した．これを用いて網羅的な検査を実施できることがわかった．しかし，コレクションとその要素に関する振舞い，オブジェクトとその属性であるオブジェクトの連動等の定義方法は検討中である．要求分析モデルを段階的に形式化することの見通しはあるが，定義方式が複雑にならないように検討する必要がある．

現状，本研究の提案手法の実際の案件への適用は未実施であるが，弊社社内において研究内容を発表し議論する中で，社内業務システム等でシステムの振る舞いに問題がある案件を探し，部分的にでも適用可能ならば検証を行う計画である．また芝浦工業大学が採択された 2014 年度 IPA「ソフトウェア工学における先導的研究支援事業」においても本学の授業支援システムに対して本研究の提案手法を利用して提案手法の検証を行っている．

謝辞

本研究を学位論文としてまとめるにあたり多くの方にご支援とご教授を頂きました.

本論文をまとめるにあたり，５年間にわたり修士課程・博士課程におきまして非常に丁寧かつ熱心なご指導を賜った芝浦工業大学大学院工学研究科 松浦 佐江子 教授（情報科学博士）に心から感謝いたします. 松浦 教授のご指導なくして本論文の完成はありませんでした.

芝浦工業大学大学院工学研究科 長谷川 浩志 教授（工学博士）,新津 善弘 教授（工学博士)，野田 夏子 准教授（情報科学博士)，東京学芸大学大学院教育学研究科 櫨山 淳雄 教授（工学博士）におかれましては大変お忙しい中，論文審査をお引き受け頂くとともに本論文に対する貴重なご意見を頂きました. 深く感謝いたします.

本研究の幅を広げる多くの示唆を頂いた信州大学 大学院理工学系研究科 小形 真平 助教(工学博士)に深く感謝いたします.

本研究にご協力頂いたソフトウェア工学研究室（松浦研究室）の諸氏に感謝いたします. 特に研究結果をまとめるためにご尽力頂いた奥田 博隆 氏，野呂 惇 氏に深く感謝いたします.

本研究の実証に欠かせない検査支援ツールの開発にご尽力頂いた株式会社ボイスリサーチ 谷沢 智史 氏，西村 一彦 氏に深く感謝いたします.

職場において様々なご支援を頂いた日本ユニシス株式会社 総合技術研究所 羽田 昭裕 所長，今道 正博 室長，村上 利秀 氏，山田 勉 氏に感謝いたします.

最後に温かい励ましをいつも送り続けてくれた家族に感謝の意を表して謝辞とたします.

参考文献

[1] 機能要件の合意形成ガイド(ver.1.0) ～「発注者ビューガイドライン ver.1.0」改訂版～ 分冊２ システム振舞い編,http://www.ipa.go.jp/files/000004505.pdf,2014

[2] UPPAAL, http://www.uppaal.com/, 2014.

[3] B.Beizer: Software Testing Techniques, Computer Press, 1990.

[4] Marc Eisenstadt ,Tales of Debugging from The Front Lines Marc Eisenstadt Paper Submitted to Empirical Studies of Programmers V,1993.

[5] Ko, A.J, Myers, B.A, Coblenz, M.J, Aung, H.H. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. Software Engineering, IEEE Transactions on , vol.32, no.12, pp.971-987,2006.

[6] 形式手法の実践ポータル, http://formal.mri.co.jp/db/fmtool/,2014.

[7] SPIN, http://spinroot.com/spin/whatispin.html/, 2014.

[8] NuSMV, http:// http://nusmv.fbk.eu/ ,2014.

[9] Pathfinder,http://javapathfinder.sourceforge.net /,2014.

[10] Markosian, L.Z.; Mansouri-Samani, M.; Mehlitz, P.C.; Pressburger, T., Program Model Checking Using Design-for-Verification: NASA Flight Software Case Study, 2007 IEEE Aerospace Conference, vol.1, no.9, pp.3-10, 2007.

[11] Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C.,Robby, and Zheng, H., Bandera: extracting fnite-state models from Java source code, Proc. the 22nd Int'l Conf. on Software. Eng. (ICSE 2000), pp.439–448, 2000.

[12] Beyer, D. ; Henzinger, T.A. ; Jhala, R. ; Majumdar, R.," An Eclipse Plug-in for Model Checking",Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on Digital Object Identifier: 10.1109/WPC.2004.1311069 Publication Year: 2004 , Page(s): 251 – 255.

[13] Lei Wang ; Qiang Zhang ; Pengchao Zhao ," Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking"Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on Digital Object Identifier: 10.1109/SCAM.2008.24 Publication Year: 2008 , Page(s): 165 – 173.

[14] Wang, L., Zhang, Q., Zhao, P., "Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking," Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on Digital Object Identifier: 10.1109/SCAM.2008.24, pp. 165–173, 2008.

[15] Classen, A; Heymans, P.; Schobbens, P.; Legay, A, "Symbolic model checking of software product lines," Software Engineering (ICSE), 2011 33rd International Conference on , vol., no., pp.321,330, 21-28 May 2011.
[16] de Angelis, F.; Polini, A; De Angelis, G., "A Counter-Example Testing Approach for Orchestrated Services," Software Testing, Verification and Validation (ICST), 2010 Third International Conference on , vol., no., pp.373,382, 6-10 April 2010.
[17] Dury, A., Boroday, S., Petrenko, A., Lotz, V., "Formal Verification of Business Workflows and Role Based Access Control Systems, Emerging Security Information, Systems, and Technologies," SecureWare 2007.
[18] Roy, S., Sajeev, A., Bihary, S., Ranjan, A., "An Empirical Study of Error Patterns in Industrial Business Process Models," Services Computing, IEEE Transactions on Volume PP, Issue 99, Digital Object Identifier: 10.1109/TSC.2013.10, p. 1, 2013.
[19] Chen Yan; Wu Dan, "A Scenario Driven Approach for Security Policy Testing Based on Model Checking," Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on , vol., no., pp.1,4, 19-20 Dec. 2009.
[20] Yanhua Du; Wending Zhang; Wei Tan, "Pattern-Based Model Checking for Dynamic Analysis of Workflow Processes with Temporal Constraints," Signal-Image Technology & Internet-Based Systems (SITIS), 2013 International Conference on , vol., no., pp.225,232, 2-5 Dec. 2013.
[21] Lindsay, P.A; Winter, K.; Yatapanage, N., "Safety Assessment Using Behavior Trees and Model Checking," Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on , vol., no., pp.181,190, 13-18 Sept. 2010.
[22] Ngui, J.; Strooper, P.; Wildman, L.; Wojcicki, M., "Comparing the Cost-Effectiveness of Statically Analysing and Model Checking Concurrent Java Components for Deadlocks," Software Engineering Conference, 2007. ASWEC 2007. 18th Australian , vol., no., pp.223-232, 10-13 April 2007.
[23] JAXA - 人工衛星の姿勢制御ソフトウェア, http://formal.mri.co.jp/db/region/domestic/jaxa--.html,2014.
[24] NASDA IV&V, http://www.nasa.gov/centers/ivv/ppt/172865main_NASDA.ppt, 2014.
[25] Hao Zheng; Yingying Zhang, "Local State Space Analysis Leads to Better Partial Order Reduction," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.33, no.6, pp.839,852, June 2014.
[25] M.B. Dwyer, G.S.Avrunin, and J.C. Corbett:"Patterns in property specifications for finite-state verification,"Proceedings of the 1999 International Conference on Software Engineering(ICSE),pp.411--420, 1999, ACM.

[26] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil: "PROPEL: an approach supporting property elucidation, "Proceedings of the 24th International Conference on Software Engineering (ICSE '02),pp.11-21, 2002, ACM.
[27] L. Di Guglielmo, F. Fummi, N. Orlandi, and G. Pravadelli:"DDPSL: An easy way of defining properties", ,2010 IEEE International Conference on Computer Design (ICCD),pp.468--473, 2010.
[28] Salamah Salamah, Ann Q. Gates, Steve Roach, and Matthew Engskow:"Towards support for software model checking: improving the efficiency of formal specifications,"Advances in Software Engineering, Vol. 2011, Jan. 2011.
[29] Clarke, E.,Jha, S.,Yuan Lu ,Veith, H.,"Tree-like counterexamples in model checking" ,Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on ,pp.19 - 29 ,2002.
[30] Cong Tian,Zhenhua Duan ,"Detecting Spurious Counterexamples Efficiently in Abstract Model Checking",Software Engineering (ICSE), 2013 35th International Conference on ,pp.202-211,2013.
[31] Lerda, F.; Kapinski, J.; Maka, H.; Clarke, E.M.; Krogh, B.H., "Model checking in-the-loop: Finding counterexamples by systematic simulation," American Control Conference, 2008 , vol., no., pp.2734,2740, 11-13 June 2008.
[32] Chan, W.; Anderson, R.J.; Beame, P.; Burns, S.; Modugno, F.; Notkin, D.; Reese, J.D., "Model checking large software specifications," Software Engineering, IEEE Transactions on , vol.24, no.7, pp.498,520, Jul 1998.
[33] P. Bose, “Automated translation of UML models of architectures for verification and simulation using SPIN,” Proc. of the ASE, pp.102-109, Fairfax, VA, 1999.
[34]L. Jing, et al. “Model Checking UML Activity Diagrams with SPIN,” Proc. of the CiSE 2009, pp.1-4, Qingdao, China, 2009.
[35] Pathfinder, http:// javapathfinder.sourceforge.net/, 2014.
[36] Eclipse, http://www.eclipse.org/,2014.
[37] ASTParser,
http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html,2014.
[38] 金田重郎，井田明男，酒井孝真，英語 7 文型と関数従属性に基づくクラス図の理解，電子情報通信学会技術研究報告 KBSE，113(475)，pp.31-36，2014.
[39] 金田重郎，世古龍郎，「認知文法に基づくオブジェクト指向の理解」，電子情報通信学会技術研究報告，知能ソフトウェア工学，111(396)，pp.61-66，2012 .
[40] 青木翼,長谷川哲夫,宮本博暢,渡邊竜明,UPPAAL によるモデル検査適用ガイドラインの作成,情報処理学会.ソフトウェア工学研究会報告,VOL.2008,NO.29,pp203,210,2008.

[41] 三菱総合研究所・経済産業省 (2011). フォーマルメソッド導入ガイダンス. http://formal.mri.co.jp/,2014.
[42] 田辺良則, 高井利憲, 高橋孝一, 抽象化を用いた検証ツールの調査,http://cent.xii.jp/tanabe.yoshinori/03/12/08abstTools.pdf,2014.
[43] Pajic, M.; Zhihao Jiang; Insup Lee; Sokolsky, O.; Mangharam, R., "From Verification to Implementation: A Model Translation Tool and a Pacemaker Case Study," Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th , vol., no., pp.173-184, 16-19 April 2012.
[44] Konrad, S.; Campbell, L.A.; Cheng, B. H C, "Automated analysis of timing information in UML diagrams," Automated Software Engineering, 2004. Proceedings. 19th International Conference on , vol., no., pp.350-357, 20-24 Sept. 2004
[45] Thompson, S.; Brat, G., "Verification of C++ Flight Software with the MCP Model Checker," Aerospace Conference, 2008 IEEE , vol., no., pp.1-9, 1-8 March 2008.
[46] Xiaoli Gong; Jie Ma; Qingcheng Li; Jin Zhang, "Automatic Model Building and Verification of Embedded Software with UPPAAL," Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on , vol., no., pp.1118-1124, 16-18 Nov. 2011.
[47] Corbett, J.C.; Dwyer, M.B.; Hatcliff, J., "Bandera: a source-level interface for model checking Java programs," Software Engineering, 2000. Proceedings of the 2000 International Conference on , vol., no., pp.762-765, 2000.
[48] Tudose, C.; Opria, R., "A Method for Testing Software Systems Based on State Design Pattern Using Symbolic Execution," Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on , vol., no., pp.113-117, 13-18 Sept. 2010.
[49] Watahiki, K.; Ishikawa, F.; Hiraishi, K., "Formal verification of business processes with temporal and resource constraints," Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on , vol., no., pp.1173-1180, 9-12 Oct. 2011.
[50] VDMTools, http://www.vdmtools.jp/,2014.
[51] del Mar Gallardo, M.; Merino, P.; Sanan, D., "Model Checking C Programs with Dynamic Memory Allocation," Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International , vol., no., pp.219-226, July 28 2008-Aug. 1 2008.
[52] Bolong Zeng; Li Tan, "Test criteria for model-checking-assisted test case generation: A computational study," Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on , vol., no., pp.600-607, 8-10 Aug. 2012.

[53] Klaus Havelund and Thomas Pressburger: Model Checking Java Programs Using Java PathFinder, International Journal on Software Tools for Technology Transfer STTT, Volume: 2, Issue: 4, Publisher: Springer, Pages: 366-381, 2000.
[54] Shanbhag, V.K., "Deadlock-Detection in Java-Library Using Static-Analysis," Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific , vol., no., pp.361-368, 3-5 Dec. 2008.
[55] K. Lano, H. Haughton: Specification in B: An Introduction Using the B Toolkit, Imperial College Press, 1996.
[56] M. Achenbach and K. Ostermann, ”Engineering Abstractions in Model Checking and Testing”, Source Code Analysis and Manipulation, Proc. of SCAM ’09., pp.137-146,2009.
[57] astah*, http://www.change-vision.com/,2014.
[58] Verstoep, K.; Bal, H.E.; Barnat, J.; Brim, L., "Efficient large-scale model checking," Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on , vol., no., pp.1-12, 23-29 May 2009.
[59] DIVINE, http://divine.fi.muni.cz/index.html,2014.
[60] Common Criteria, ” CC/CEM v3.1 Release4 ”, http://www.commoncriteriaportal.org/cc/,2014.
[61] 矢竹健朗，青木利晃，片山卓也，コラボレーションに基づくオブジェクト指向モデルの検証，コンピュータソフトウェア，Vol.22, No.1, pp.58-76, 2005.
[62] El Maarabani, M., Cavalli, A., Iksoon Hwang; Zaidi, F. ,” Verification of InteroperabilitySecurity Policies by Model Checking”, High-Assurance Systems Engineering (HASE),pp.376-381,2011.
[63] Jianli Ma, Dongfang Zhang, Guoai Xu; Yixian Yang,” Model Checking Based Security Policy Verification and Validation”, Intelligent Systems and Applications (ISA), pp.1-4,2010.
[64] Armando, A., Carbone, R., Compagna, L.,” LTL Model Checking for Security Protocols”,Computer Security Foundations Symposium, pp.385-396,2007.
[65] N. Trcka, Wil M. Aalst, and N. Sidorova ., “Data-Flow Anti-Patterns: Discovering Dataflow Errors in Workflows,” Proc. Of the CAiSE 2009, pp.425-439., 2009.
[66] Rik Eshuis, Symbolic model checking of UML activity diagrams, ACM Transactions on Software Engineering and Methodology, Volume 15 Issue 1,pp.1-38, 2006.
[67] S. Ogata, and S. Matsuura, “A UML-based Requirements Analysis with Automatic Prototype System Generation,” Communication of SIWN, Vol.3, pp.166-172, Jun. 2008.

[68] S. Ogata. and S. Matsuura, “A Method of Automatic Integration Test Case Generation from UML-based Scenario,” WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS, Issue 4, Vol.7, pp.598-607, Apr.2010.
[69] IPA (2010). 形式手法適用調査. http://www.ipa.go.jp/sec/softwareengineering/reports/20100729.html,2014.
[70] LUMINOUS, https://lmns.sayo.se.shibaura-it.ac.jp/,2014.
[71] OMG,” UNIFIED MODELING LANGUAGE”, http://www.uml.org/,2014.
[72] E. Choi, T. Kawamoto, and H. Watanabe, “Model Checking of Page Flow Specification”, Computer Software, Vol.22, No.3, 2005, pp.146-153. (in Japanese)
[73] Standish Chaos Report, http://blog.standishgroup.com/,2014.
[74] Eshuis, R.; Wieringa, R., "Tool support for verifying UML activity diagrams," Software Engineering, IEEE Transactions on , vol.30, no.7, pp.437-447, July 2004.
[75] IEEE Computer Society, IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830 (1998).
[76] 独立検証機関による形式手法を用いた第三者検証のコスト評価 実施報告書,http://www.ipa.go.jp/files/000026857.pdf,2014.
[77] Keqin Li, "Towards Security Vulnerability Detection by Source Code Model Checking," Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on , vol., no., pp.381-387, 6-10 April 2010.
[78] 長谷川哲夫, 田原康之, 磯部祥尚,UPPAAL による性能モデル検証―リアルタイムシステムのモデル化とその検証 (トップエスイー実践講座),近代科学社,2012.
[79] 吉岡信和, 青木利晃, 田原康之,SPIN による設計モデル検証―モデル検査の実践ソフトウェア検証 (トップエスイー実践講座),萩谷昌己監修 ,近代科学社,2008.
[80] 中島 震,モデル検査法のソフトウェアデザイン検証への応用, コンピュータ ソフトウェア, Vol. 23 , No. 2 , pp.72-86,2006.
[81] 中島 震, ソフトウェア工学の道具としての形式手法, コンピュータ ソフトウェア, NII Technical Report ISSN 1346-5597,2007.

研究業績

学術論文誌

[1] Aoki, Y., Matsuura, S., "Verifying Business Rules Using Model-Checking Techniques for Non-specialist in Model-Checking", Institute of Electronics, Information and. Communication Engineers (IEICE) TRANSACTIONS on Information and Systems , Volume E97-D No.5, pp.1097-1108,2014.

[2] 松浦佐江子,小形真平,青木善貴,矢沢智史,西村一彦,要件定義プロセスと保守プロセスにおけるモデル検査技術の開発現場への適用, Information-technology Promotion Agency (IPA) SEC journal37 JUL, pp8-15,2014.

国際会議（査読あり）

[1] Saeko Matsuura, Shinpei Ogata, Yoshitaka Aoki , “Practical Behavioral Inconsistency Detection between Source Code and Specification using Model Checking”, 25th International Symposium on Software Reliability Engineering, to appear.

[2] Yoshitaka Aoki, Saeko Matsuura, "Verifying Security Requirements using Model Checking Technique for UML-Based Requirements Specification", Requirements Engineering and Testing(RET) 2014,pp.18-25, 2014.

[3] Yoshitaka Aoki, Shinpei Ogata, Hirotaka Okuda, Saeko Matsuura "Data Lifecycle Verification Method for Requirements Specifications Using a Model Checking Technique", International Conference on Software Engineering Advances (ICSEA) 2013,pp.194-200 2013.

[4] Yoshitaka Aoki, Shinpei Ogata, Hirotaka Okuda, Saeko Matsuura, "Quality Improvement of Requirements Specification Using Model Checking Technique" The 14th International Conference on Enterprise Information Systems (ICEIS) ,2,pp.401-406,2012.

[5] Yoshitaka Aoki, Saeko Matsuura ,"Verification of Embedded System by a Method for Detecting Defects in Source Codes Using Model Checking", IEEE Symposium on Computers & Informatics(ISCI)2011 ,pp.530-535,2011.

[6] Yoshitaka Aoki, Saeko Matsuura ,” A Method for Detecting Unusual Defects in Enterprise System Using Model Checking Techniques”, The 10th World Scientific and Engineering Academy and Society (WSEAS) ,pp.165-171,2011.

[7] Yoshitaka Aoki, Saeko Matsuura ,"A Method for Detecting Defects in Source Codes

Using Model Checking Techniques", The 34th Computer, Software & Applications Conference (COMPSAC) ,pp.543-544,2010.

国内学会（査読あり）

[1] 青木善貴，松浦佐江子，モデル検査を利用した仕様とソースコード間におけるシステムの振舞いの不一致発見，ソフトウェアエンジニアリングシンポジウム 2014 論文集，pp. 212-213，2014.

[2] 青木善貴，小形真平，野呂惇，松浦佐江子，モデル検査技術を用いたセキュリティ要求の検証 ,第 20 回 ソフトウェア工学の基礎ワークショップ FOSE，pp.209-214，2013.

[3] 青木善貴，小形真平,奥田博隆,松浦佐江子,要求分析における CRUD 観点のモデル検査技術の適用,第 19 回 ソフトウェア工学の基礎ワークショップ FOSE，pp.75-80，2012.

[4] 青木善貴，松浦佐江子（芝浦工大），ソースコード解析を利用したモデル検査に基づく欠陥抽出手法 ，第 17 回ソフトウェア工学の基礎ワークショップ FOSE，pp.95-100，2010.

国内学会（査読なし）

[1] 青木善貴，松浦佐江子，反例からの検査式自動生成による不具合原因特定支援，電子情報通信学会技術研究報告 KBSE 114(127) ，pp.87-92，2014.

[2] 青木善貴，松浦佐江子，企業内でのモデル検査のツールの利用と普及，電子情報通信学会技術研究報告 KBSE 113(66)，pp.47-52，2014.

[3] 青木善貴，松浦佐江子，モデル検査における反例解析容易化支援 ，電子情報通信学会技術研究報告 KBSE 113(475)， pp.1- 6，2014.

[4] 青木善貴，小形真平，松浦佐江子，UML 要求分析モデルへのモデル検査技術適用による実現可能性の検証 ，電子情報通信学会技術研究報告 KBSE 113(160)，pp.97-102 ，2013.

[5] 青木善貴，松浦佐江子，モデル検査技術の開発現場への適用， 電子情報通信学会技術研究報告 KBSE 113(160)，pp.91-96 ，2013.

[6] 青木善貴，松浦佐江子，開発現場を想定したモデル検査に基づくプログラムの不具合検証－検査支援ツールを用いた安定的な検査－，電子情報通信学会技術研究報告 KBSE 112(496)，pp.1-5,2013.

[7] 谷沢智史，西村一彦，青木善貴，小形新平，松浦佐江子，Source2UPPAAL: ソースコードの効率的な検証へ向けた開発者支援ツールの検討, 第 19 回ソフトウェア工学の基礎ワークショップ FOSE ，pp.241-242，2012.

[8] 小形真平，谷沢智史，西村一彦，青木善貴，奥田博隆，松浦佐江子，データライフサイ

クルの妥当性に着目したモデル検査ツールの自動利用法，電子情報通信学会技術研究報告 KBSE 112(119)，pp.119-114，2012.

[9] 青木善貴，小形真平，奥田博隆，松浦佐江子，UML 要求分析モデルにおける CRUD 観点のデータライフサイクルの妥当性検査手法，情報処理学会第 74 回全国大会講演論文集 ，5A-5，2012.

[10] 青木善貴，松浦佐江子，開発現場を想定したモデル検査に基づくプログラムの欠陥抽出手法，電子情報通信学会技術研究報告 KBSE ，111(396)，pp.43-48，2012.

[11] 青木善貴，松浦佐江子，モデル検査を用いたプログラムにおける再現性の低い潜在的欠陥の抽出手法　～ データベースロック問題の検証 ～，電子情報通信学会技術研究報告 KBSE，110(468)，pp.79-84，2011.

[12] 青木 善貴，松浦佐江子，モデル検査に基づくプログラム欠陥抽出作業支援ツールの開発と実践，情報処理学会創立 50 周年記念(第 72 回)全国大会，2P-3，2010.

[13] 青木善貴，松浦佐江子，ソースコード解析を利用したモデル検査に基づく欠陥抽出手法による組込みシステムの検証，第 9 回情報科学技術フォーラム，B-022，2010.

[14] 青木 善貴，松浦佐江子，ソースコード解析を利用したモデル検査に基づく欠陥抽出手法の提案，電子情報通信学会技術研究報告 KBSE, 109(307)，pp.79-84，2009.

[15] 青木善貴，松浦佐江子，ソースコードの解析を利用したモデル検査に基づく欠陥抽出手法の提案，第 8 回情報科学技術フォーラム，B-016，2009.