

2020 年度

信州大学大学院総合理工学研究科 修士学位論文

テンプレートに基づいた UML 上位設計に対する  
整合性検査と自動コード生成

専攻名：工学専攻

分野名：電子情報システム工学分野

学籍番号：19W2099A

氏名：畠瀬尚之

2021 年 3 月

# 目次

1	はじめに	1
1.1	研究目的と背景 . . . . .	1
1.2	本論文の構成 . . . . .	1
2	上流工程と形式手法の事例	2
2.1	上流工程 . . . . .	2
2.2	形式手法 . . . . .	2
2.3	Java Servlet プロトタイプ半自動生成系 . . . . .	3
3	提案手法	5
3.1	テンプレートベース UML 上位設計の整合性検査と自動コード生成 . . . . .	5
3.2	提案手法を踏まえた生成系全体フロー . . . . .	5
4	テンプレートを用いた上位設計の作成	6
4.1	テンプレート構造を用いる有用性 . . . . .	6
4.2	テンプレート構造の挿入支援 . . . . .	6
4.3	実装と UI . . . . .	8
5	上位設計に対する整合性検査	11
5.1	整合性検査の概要 . . . . .	11
5.2	クラス図とアクティビティ図間の要素一致検査 . . . . .	13
5.3	記述規則に基づいた文脈自由文法によるスライス解析 . . . . .	13
5.4	図上へのユーザフィードバック . . . . .	16
6	スライス解析を用いた自動コード生成	18
6.1	提案手法の自動コード生成プロセス . . . . .	18
6.2	提案手法による自動コード生成 . . . . .	20
7	評価	26
7.1	挿入支援機能 . . . . .	26
7.2	整合性検査器 . . . . .	33
7.3	自動コード生成 . . . . .	34
8	まとめと今後の課題	35
8.1	まとめ . . . . .	35
8.2	今度の課題 . . . . .	35
	参考文献	37
	謝辞	38

# 1 はじめに

## 1.1 研究目的と背景

プログラム開発の上流工程において、自然言語による曖昧さは人的なエラーを引き起こす原因となる。これを回避する有用な手段として形式手法がある。作成された上位設計を用いて早期に妥当性確認を行うことで、開発の早い段階でのエラーを取り除き、手戻りや修正コストが削減できる。従来研究ではイベント予約システムのVDM++を用いた仕様記述や、その妥当性を確認するためのJava Servletスケルトンコード半自動生成器[1]が開発されているが、これは必ずしも上位設計間で整合性が取れていることを保証しない。上位設計間で整合性が取れていなければ、プロトタイピングの成果も誤ったものとなる。これらの課題に対して、本研究では、UML上位設計に基づく自動コード生成を対象とした整合性検査を行い、上位設計での正しさを確認した上でプロトタイピング法を提案する。

UML図はオブジェクト指向の分析や設計のために、モデル図の記法が統一されたモデリング言語である。UML図はプログラミングの知識がなくとも図によるフローの理解が可能であり、また用途により図を使い分けることによってより正確に記述することができるが、図による表現能力の違いから仕様を記述するには向かない場合もある。本研究はスケルトンコード半自動生成器をベースとしたものであり、提案手法で対象とする上位設計はクラス図とアクティビティ図とする。

提案手法によるUML上位設計の整合性検査を行うにあたって、用いる上位設計をテンプレートに基づいて作成することにより、記述の厳格化、不整合の混入抑制を図り、より信頼性の高いプロトタイピングを目指す。そのため、(1)頻出である制御構文・例外処理の構造パターンを用いたアクティビティ図作成支援、(2)クラス図とアクティビティ図間の名前ベースでの要素一致検査とアクティビティ図に対するスライス解析による構造検査、これらに伴う検査結果の図上ユーザフィードバック、(3)テンプレート作成支援・アクティビティ図構造検査情報を用いたテンプレートベースのスケルトンコード生成、の三項が提案手法の核心である。これらは、astah\*[4]上のプラグインとして実装した。(1), (2)に対して、提案手法の記述規則に則った上で、プラグインを用いた場合とすべて手作業で行った場合での作業工程数や不整合混入数といった観点からの定量的比較、(3)に対しては既存研究であるスケルトンコード半自動生成器と生成能力の比較をし、提案手法の評価を行う。

## 1.2 本論文の構成

第2章では上流工程と形式手法、および既存研究であるJava Servletスケルトンコード半自動生成器について述べる。第3章では従来研究に提案手法を踏まえた本研究の全体像の説明を行う。第4章ではテンプレートを用いた上位設計の作成とその実装について述べる。第5章では上位設計に対する整合性検査について述べる。第6章ではスライス解析を用いた自動コード生成について述べる。第7章では提案手法の評価として行った既存手法との比較実験について述べる。第8章では本研究の結果についてまとめ、今後の課題を述べる。

## 2 上流工程と形式手法の事例

### 2.1 上流工程

ウォータフォール・モデルは古くから利用されている開発プロセスである。これを各テストが何に対するテストであるかを表現したものがVモデル(図1)である。Vモデルの左側は開発工程を、右側はテストの流れを表しており、各テストはそれぞれ同じ高さの開発を検証するものである。ウォータフォール・モデルに代表されるソフトウェア開発プロセスは、一般に要求分析、設計、実装、テストから成り、このうち要求分析、設計は上流工程と呼ばれる。この上流工程である仕様や設計の欠陥が見逃され下流工程に持ち越されるとその修正のため前工程への手戻りが発生することになり、ソフトウェア開発コストが高くなる。しかし、従来のソフトウェア開発において仕様や設計は、自然言語による曖昧さを含み不整合の混入に繋がる。その結果、ソフトウェア開発における欠陥の多くは要求分析、設計の部分で発生している。これらの問題を防ぎ、品質・信頼性の高いシステム開発を行う手法の一つとして形式手法がある。

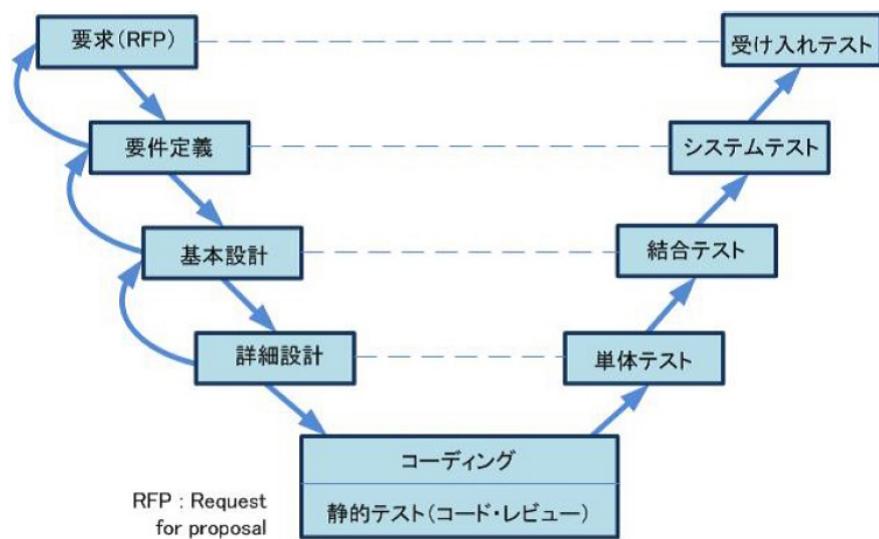


図1 V モデル

### 2.2 形式手法

形式手法とは、厳密な文法と意味論を持つ言語(形式仕様記述言語)を用いて設計対象の注目すべき側面の仕様を人による解釈違いや曖昧性を排除した上で記述し、品質の高いソフトウェアを効率よく開発するための手法である。また、形式手法はツールによる機械的な仕様の分析・検証処理を行うことができ、従来の下流工程で発見・修正されていた欠陥を上流工程の段階で抑えることが可能となる。そのため、上流工程における検証コストは増大するが下流工程での欠陥修正コストが必要なくなるため、結果としてトータルコストが抑えられる。これらが形式手法を用いる主なメリットである。

## 2.3 Java Servlet プロトタイプ半自動生成系

### 2.3.1 上流工程における妥当性確認の全体フロー

当研究室の従来研究として、ある上流工程における妥当性確認の全体フローを図2に示す[1]。本稿では、VDMJWebサービスを用いてVDM++モデル記述を行う人間を『開発者』、開発者に対しシステムの構築を要求する人間を『ユーザ』と呼称する。ユーザによる仕様の妥当性確認は、早期の内に行われるべきである。しかしコマンドライン・インターフェースでの妥当性確認は、VDMインタプリタの知識、及びVDMで構築されたモデルの知識が必要となり、ユーザが実施するには現実的ではない部分が多い。そのためユーザによる妥当性確認の際は、GUIで構築されたクライアントアプリケーション(Java Servletで構築されることを想定した)を記述することが望ましい。しかし、GUIで構築されたクライアントアプリケーションの記述は妥当性確認を容易にする一方、コード記述のためにコストが増大するという欠点がある。そこで仕様記述の際に作成されるクラス図やアクティビティ図から、Java ServletやHTML,JSPを半自動生成しコスト削減を図ろうというのがJava Servletスケルトンコード生成器である。生成されたコードはJava向けに構築されているVDMJCCというWebAPIを用いてVDMJWebサービスと通信を行う。

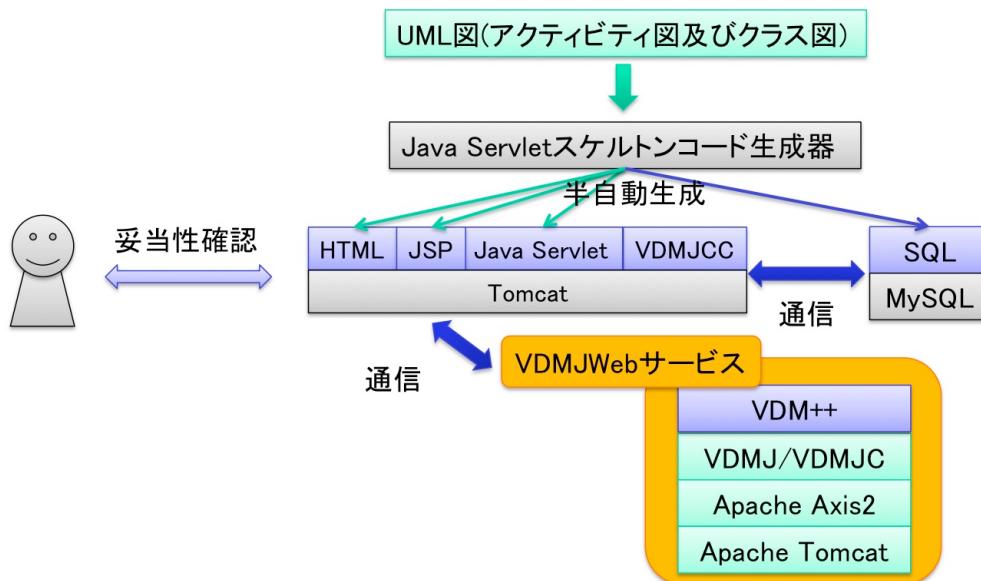


図2 Java Servlet スケルトンコード半自動生成系

### 2.3.2 VDMJWeb サービス

VDM++にはVDMToolsやVDMJといった、仕様実行のための援用ツールが存在する。このうちオープンソースであるVDMJをWebサービスとしてApache Tomcatサーバー上に配置し、クライアントアプリケーションとSOAP通信を行うことで、分散環境下でVDM++仕様のモデル実行を行うこと目的としたVDMJWebサービスが開発されている[2]。VDMJWebサービスの全体の流れを図3に示す。VDMJのインタプリタによる処理結果は、VDMJCCを用いたJava ServletによってWebブラウザに表示され、その結果を見てユーザは妥当性確認を行う。

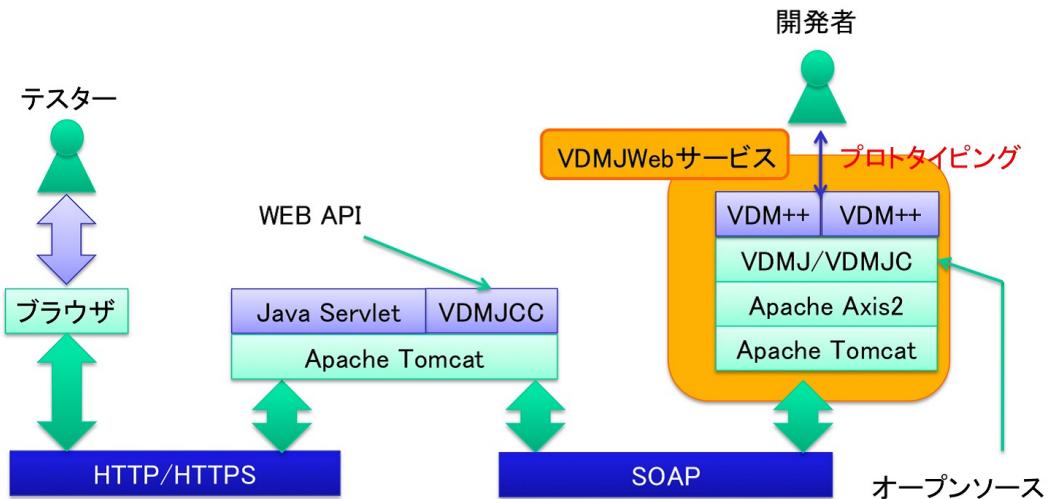


図 3 VDMJWeb サービス全体の流れ

### 2.3.3 UML 図と VDM++ コードの再利用による高速プロトタイピング手法

当研究室で研究された Java Servlet スケルトンコード生成器の問題点として、VDM++ の仕様記述や、アクティビティ図からの自動生成されない部分のコード記述、アクティビティ図の記述コストなどがある。こういった問題点の削除のため、従来のプロトタイプに対し再利用可能な部分を選定し、新たに作成する量を抑える手法 [3] が考案されている。具体的な例として、予約システムモデルを出発点として飛行機予約システム、列車予約システムといった類似システムモデルに対して適用が可能である。

### 3 提案手法

#### 3.1 テンプレートベース UML 上位設計の整合性検査と自動コード生成

従来研究である早期な妥当性確認を目的とした Java Servlet スケルトンコード半自動生成器 [1] は、上位設計間で整合性が取れていることを保証するものではないため、上位設計間で整合性が取れていなければプロトタイピングの成果も誤ったものとなる。これらの課題に対して、テンプレートベースの UML 上位設計に対する整合性検査を行い、上位設計での正しさを確認した上のプロトタイピング法を提案する。

#### 3.2 提案手法を踏まえた生成系全体フロー

本研究での提案手法を踏まえた Java Servlet スケルトンコード半自動生成系の全体フローを図 4 に示す。Java Servlet を用いた妥当性確認ツールは従来研究通り、提案手法は生成に用いる上位設計に対してのものである。そのため、上位設計として用い、整合性検査を行う UML 図は従来研究に則り、クラス図とアクティビティ図とする。同様に生成言語は Java 言語を対象とする。

提案手法を大別したとき、(1) 頻出である制御構文・例外処理の構造パターンを用いたアクティビティ図作成支援、(2) クラス図とアクティビティ図間の名前ベースでの要素一致検査とアクティビティ図に対するスライス解析による構造検査、これらに伴う検査結果の図上ユーザフィードバック、(3) テンプレート作成支援・アクティビティ図構造検査情報を用いたテンプレートベースのスケルトンコード生成、の三項に分けられる。これらは完全に独立したものではなく、(2) の整合性検査結果を用いて、(1) や (3) での構造判定を行ったり、(1) のテンプレート構造作成の際に印付けをし、(2) での処理の緩和を行う。

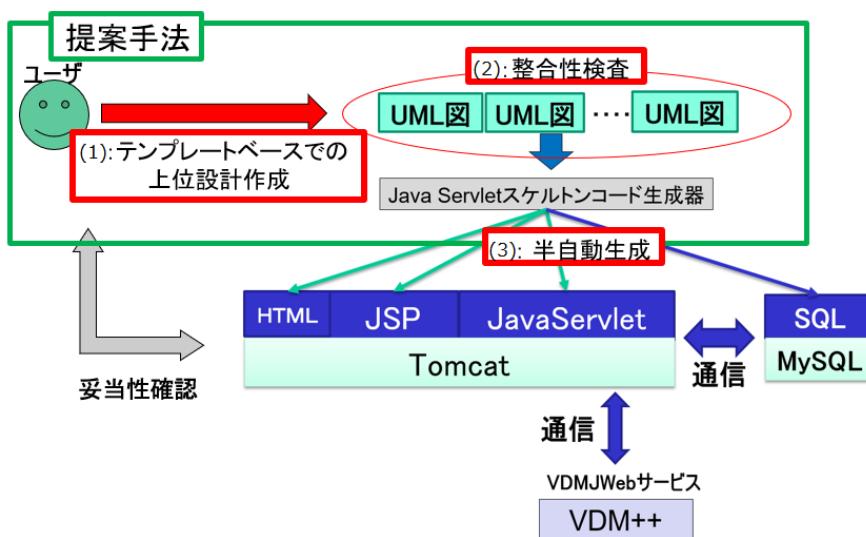


図 4 提案手法を踏まえた生成系全体フロー

## 4 テンプレートを用いた上位設計の作成

### 4.1 テンプレート構造を用いる有用性

提案手法による整合性検査は、検査を行うにあたり曖昧さの排除といった目的から、複数のUML図記述規則をユーザに強いている。これにより、UML図記述のコストの増大や不整合混入といった問題が考えられるため、テンプレートをベースとしたUML図作成支援を行うことでこれらの問題の緩和を図る。

### 4.2 テンプレート構造の挿入支援

#### 4.2.1 挿入可能構造

システムの実行系列を記述する際、頻出構造と考えられる制御構文・例外処理をテンプレート構造としてアクティビティ図上に作成し、それらをアクティビティ図内の任意のフローへ挿入することで不整合混入抑制を図る。テンプレート構造として作成された構造はif文、switch-case文、for-while文、while文、do-while文、try-catch文、ラベル付きbreak文の7構造となる。図5にif文、図6にfor-while文のテンプレート構造を記述したアクティビティ図の例を示す。テンプレート構造は図5、6のように提案手法により定義された最小のノードのみで構成される。また、これらテンプレート構造の挿入方法は一つのフローに対し差し込む挿入と、二つのフローを選択しそのフロー間を挟み込む挿入の二種類がある。

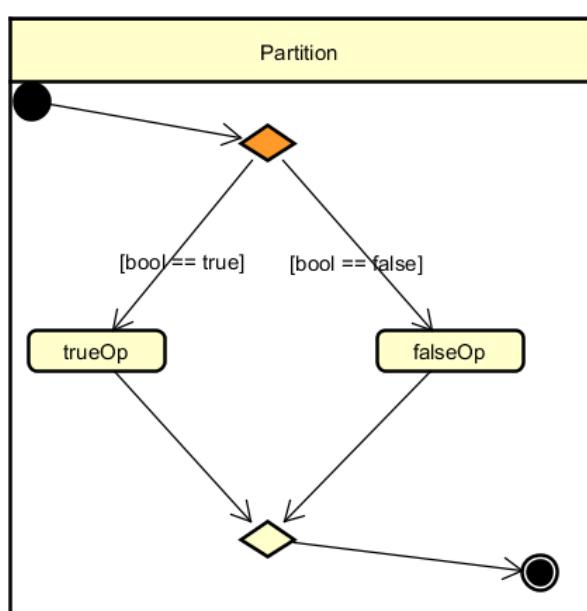


図5 if文のテンプレート構造の例

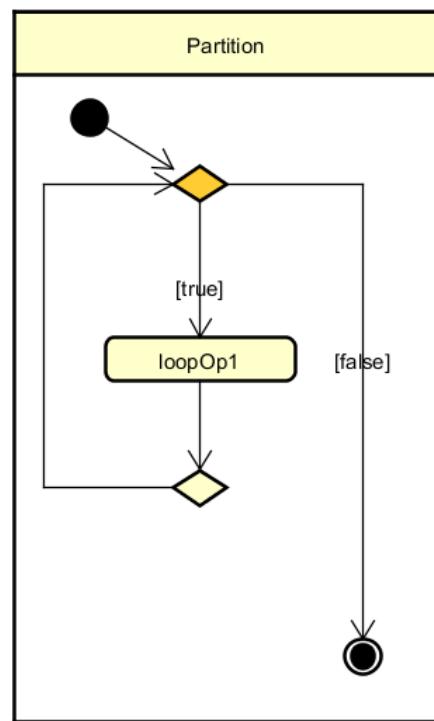


図6 for-while文のテンプレート構造の例

#### 4.2.2 差し込み型挿入

フローを一つ選択し、そのフロー間に指定したテンプレート構造の挿入を行う。この挿入方で挿入可能な構造は提案手法で定義した計7つのテンプレート構造となる。差し込み型挿入の例を図7に示す。図7(A-0)の選択されたフローに対しtry-catch文のテンプレート構造を差し込み型で挿入したものが図7(A-1)である。

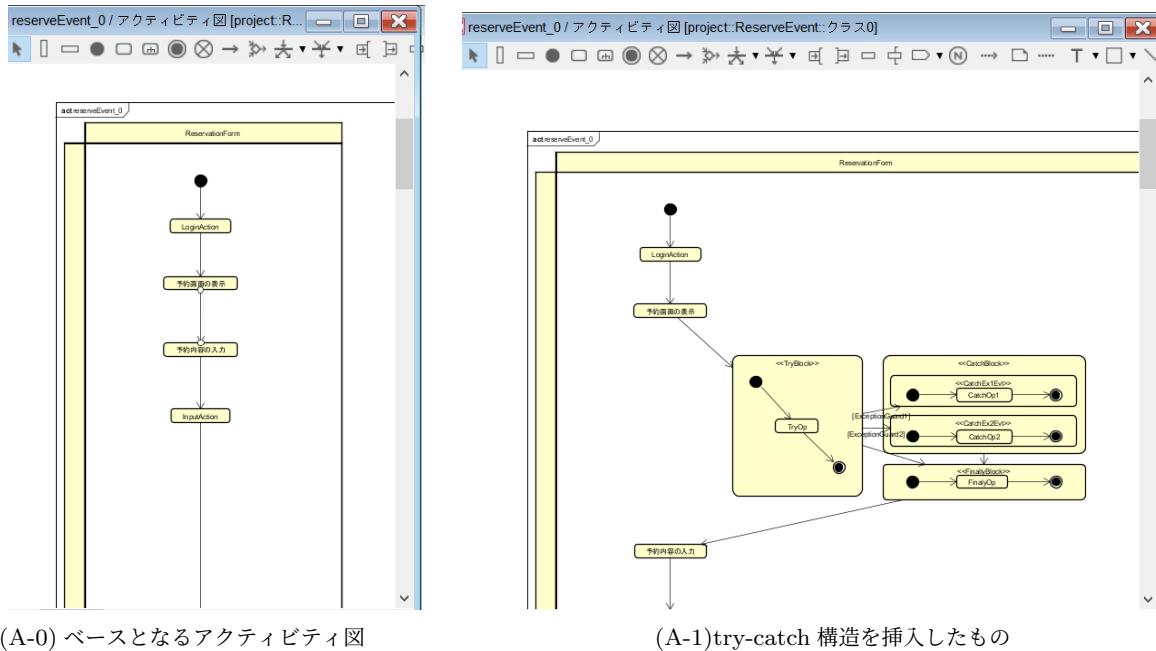


図7 差し込み型挿入によりtry-catch文のテンプレート構造を挿入した例

#### 4.2.3 挿込み型挿入

仕様の変更などから、すでに記述された構造を挿込み込むような挿入を行いたいとき、差し込み型挿入では、挿入後のテンプレート構造内部の入出力フローを接続しなおす必要があり、このときの人力の操作では、図の整合性を崩すような操作が行われることが考えられる。そのため、フローを二つ選択し、そのフロー間を挿込み込むように指定したテンプレート構造の挿入方法を作成する。このとき、図5のif文の構造のように内包するスライスが複数ある場合、挿入操作が複雑化することを鑑み、図6のような内包するスライスが単一であり、フロー間を挿込み込むのに適当であるとして、for/do-while文、ラベル付きbreak文のみを挿込み型挿入で挿入可能な構造とする。差し込み型挿入の例を図8に示す。図8(B-0)で選択された二つのフローに対しdo-while文のテンプレート構造が選択されたフロー間を内包するように挿入されているのが分かる。

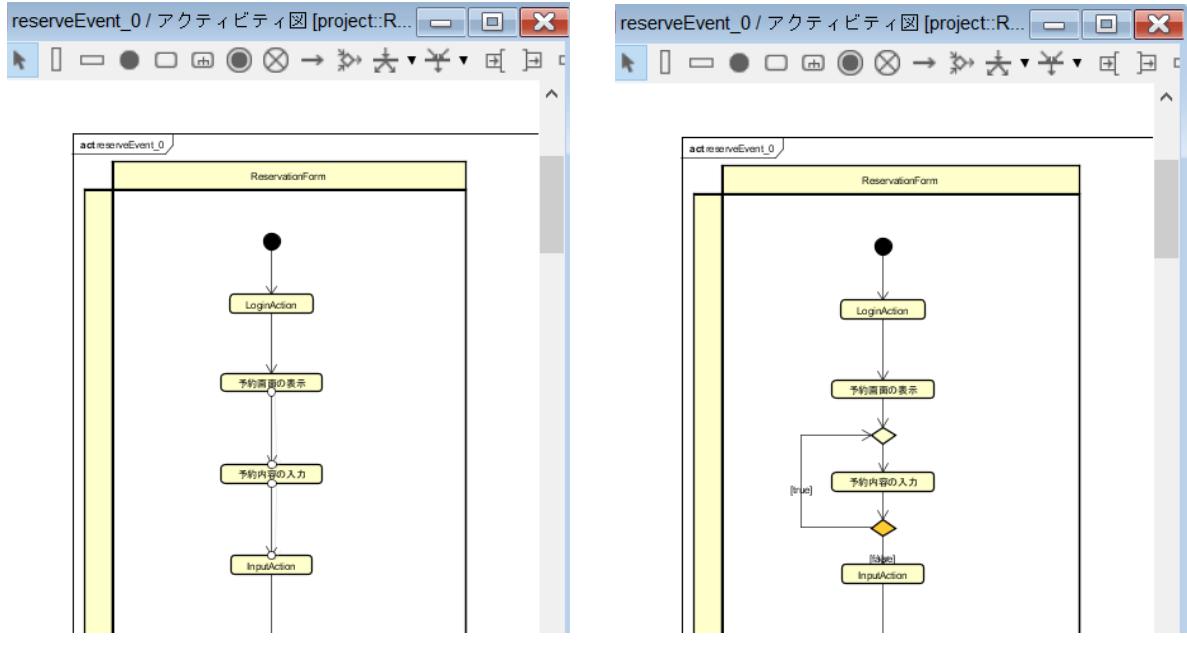


図 8 挿込み型挿入により do-while 文のテンプレート構造を挿入した例

#### 4.2.4 スライス解析を用いた挿入判定

構造挿入は、挿入箇所によっては各制御構文・例外処理を表現するために設けた記述規則を逸脱する可能性がある。そのため、スライス解析を行い挿入可能箇所の明示化や不正な挿入の検知を図る。各種テンプレート構造とアクションノードの計8構造でスライス分けされる。テンプレート構造のスライスはそれ単体では制御構造でしかないため、内部にもまたスライスを持つ形となる。これらは全てワークフロー型であり、スライスへの入出力は共に一つずつである。このとき、構造挿入可能フローはスライスへの入出力のみであり、これら以外のフローに挿入した場合、構造記述規則の逸脱やスライス内に新たに入出力を持たせるような入れ子構造を作るため不整合な構造となる。スライス解析については5.3節にて詳しく述べる。

### 4.3 実装と UI

テンプレート構造挿入支援機能および挿入箇所判定機能支援機能を astah\*professional[4] にプラグインとして実装した。また、挿入支援機能を使った際、新たな不整合を発生させてしまわないように、挿入可能箇所の図上への可視化、不整合となる挿入の検知・抑制を行う。

#### 4.3.1 テンプレート構造挿入支援機能の実装

テンプレート構造の挿入を行う際、差し込み型挿入では一つ、挟み込み型では二つのフローを選択状態にし、挿入したいテンプレート構造の記述されたアクティビティ図を開いた上で、プラグインによる拡張タブ上の Inset ボタンを押すことで挿入される。挿入時には、ノードが重ならないように元のノードの位置を自動調整し挿入を行う。

#### 4.3.2 テンプレートファイルの判定

テンプレート挿入を行うにあたり、図5、図6のようなテンプレート構造の記述されたアクティビティ図をユーザに提供する。提供するアクティビティ図には印付けがされており、これを目印にして挿入を行う。テンプレート構造のアクティビティ図の開始ノードに、キーが「Initial」、値が「1」、終了ノードにはキーが「Final」、値が「1」のタグが付与されている。挿入を行う際、このタグ付きの開始ノードと選択されたフローの接続元、タグ付きの終了ノードと選択されたフローの接続先をそれぞれ繋げ直す形で挿入を実現させている。そのため、開始・終了ノードに正しくタグが付与されているアクティビティ図であれば、提案手法により提供されるテンプレート構造でなくとも差し込み型の挿入を行うことが可能である。ただし、提案手法によるテンプレート構造は後述の構造検査に対応したものであるが、タグを付与したことにより独自に作成した新規テンプレート構造を保証するものではない。

#### 4.3.3 挿入可能箇所の可視化

プラグインによる拡張タブ上のViewボタンを押すことで、スライス解析結果を用いた図上への挿入可能箇所の可視化を行う。図9に挿入可能箇所の可視化の例を示す。デフォルトの黒色に対し、挿入可能フローは緑色でハイライトされる。この状態でフローを一つ選択すると、選択フローとの挟み込み型挿入で選択可能候補となるフローが赤色でハイライトされる。図9の例ではdo-while文の内部処理の入力フローを選択している。このとき、赤色でハイライトされたフロー以外を選択し構造挿入を行おうとすると、スライスをまたぐような入れ子構造ができてしまうため、不整合抑制のため挿入支援機能を使った構造挿入は行えない。

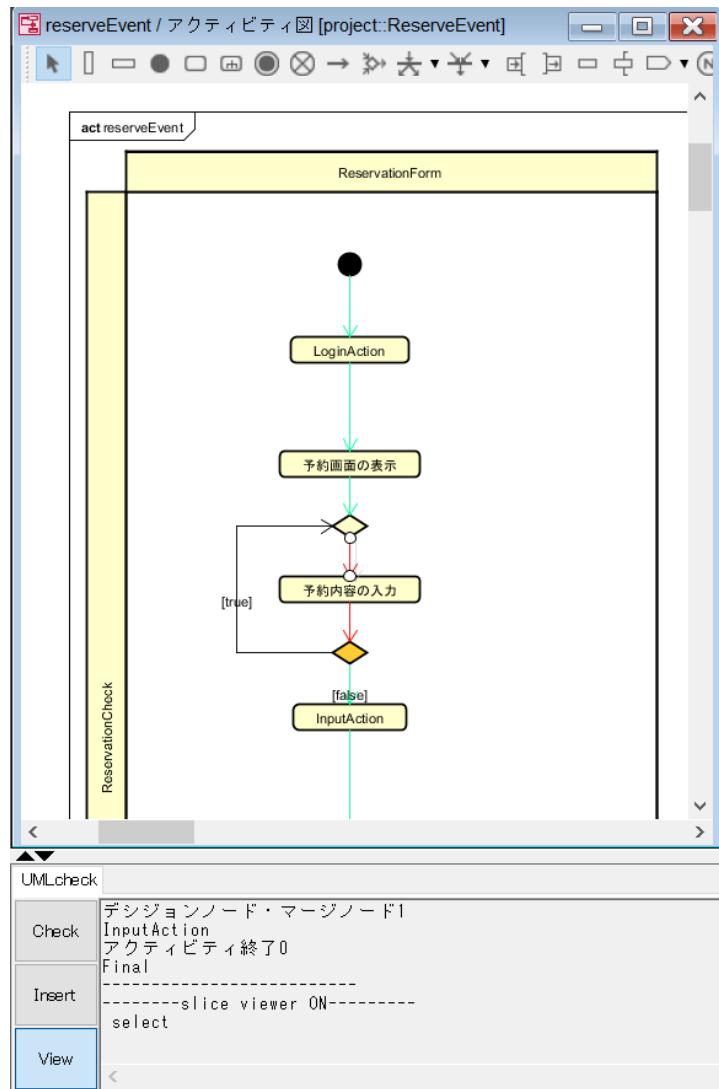


図 9 挿入可能箇所の可視化の例

## 5 上位設計に対する整合性検査

### 5.1 整合性検査の概要

提案手法は自動コード生成を目的としたものであるため、行う整合性検査は生成に用いる要素を主とする。対象とするのはクラス図とアクティビティ図の二つであり、クラス図をより上位の設計と位置付けた上で検査を行う。検査内容としては、クラス図とアクティビティ図間の名前ベースでの要素一致検査と、アクティビティ図に対するスライス解析による構造検査である。アクティビティ図には制御内容なども記述することとなるが、提案手法では意味論検査などは行わず、あくまで記述された構造に対する静的な文法検査にとどまる。

#### 5.1.1 UML 図記述ガイドラインの策定

UML 図を astah\*professional で記述する際、構造解析・コード生成を行うための曖昧さの排除といった理由で一定の記述規則を設ける。クラス図の記述規則は以下のようである。

- 属性の記述は VDM++ でのインスタンス宣言と同じように記述する
- 操作の引数・返し値も属性の記述と同様に行う
- プロジェクト内に最低一つのクラスを持つこと

また、アクティビティ図の非構造的な記述規則は以下のようである。

- アクティビティは最低一つのパーティション、開始ノード、終了ノードを持つこと
- 水平パーティションをクラス、垂直パーティションをメソッドとすること

#### 5.1.2 提案手法におけるアクティビティ図構成要素

提案手法で用いるアクティビティ図の各要素について述べる。

**■パーティション** パーティションは一連のアクションを実行する際の工程を分けるために用いられる。提案手法では水平パーティションをクラスレベル、垂直レベルをメソッドレベルでの分割として用いる



図 10 パーティション

■開始ノード・終了ノード 開始ノードと終了ノードはそれぞれのアクティビティの開始と終了を表す。したがって開始ノードには入力フローは存在せず、同様に終了ノードに出力フローは存在しない。



図 11 開始ノード



図 12 終了ノード

■アクションノード アクションノードは一つの動作(処理)を表し、アクティビティの最小単位である。



図 13 アクションノード

■ディシジョンノード・マージノード ディシジョンノードは条件分岐を表す。ディシジョンノードの分岐条件は出力フローにガード条件として記述する。ディシジョンノードは条件が合致した一つの出力フローの先へしか処理が進行せず、マージノードは到達した処理から順次出力フローの先へ処理が進行する。

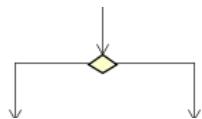


図 14 ディシジョンノード

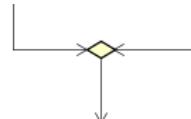


図 15 マージノード

■ノート ノートには説明や注釈などをつけたい場合に使用する。



図 16 ノート

■コネクタ コネクタはハイパーリンクによってフローの繋がりを表す。通常の制御フローでも表現できるが、アクティビティの規模が大きくなり構造が複雑化すると理解することが困難になるため、視覚的にわかりやすく表現するために使用される。



図 17 コネクタ

## 5.2 クラス図とアクティビティ図間の要素一致検査

クラス図とアクティビティ図に対し、記述された要素の一致検査を行う。提案手法では、アクティビティ図の水平パーティションにクラス名、垂直パーティションにメソッド名とし、パーティション区切りで囲まれた範囲にメソッドのロジックを記述していく。そのため、クラス名とメソッド名をクラス図上にも記述されているかの検査を行う。クラス図をより上位の設計と位置づけるため、クラス図に記述されているがアクティビティ図にはない要素は許容する。

## 5.3 記述規則に基づいた文脈自由文法によるスライス解析

### 5.3.1 各構造の記述規則

提案手法によりサポートされる制御構文・例外処理は if 文, switch-case 文, for-while 文, while 文, do-while 文, try-catch 文, ラベル付き break 文, 特定条件下におけるコネクタ遷移の 8 構造とアクションであるが、構造的には if 文と switch-case 文, for-while 文と while 文は同一構造とみなせる。よって構造検査においては、if 文と switch-case 文といった分岐構造, for-while 文と while 文といった前判定ループ構造, 後判定ループ構造である do-while 文, try-catch 文による例外処理, ラベル付き break 文, 特定条件下におけるコネクタ遷移の 6 つに分類される。それぞれの記述規則は以下のようである。

#### 分岐構造

入力 1, 出力 2 以上のディシジョンノードが分岐を表し、分岐と同数の入力を持ち、出力が 1 であるマージノードとの対応により記述される。このとき、それぞれの分岐先は他の分岐先とは互いに素となる必要がある。

#### 前判定ループ構造

入力が 2, 出力が 2 のディシジョンノードが前判定部分を表す。出力の内、ループ内処理へと接続されるフローのガードに「true」を、ループ外に向かうフローのガードに「false」を記述すること。ループの戻りは、入力 1, 出力 1 のマージノードによって表し、この出力は判定部分であるディシジョンノードへと他のノードを挟まず直接接続される。

#### 後判定ループ構造

入力が 2, 出力が 1 のディシジョンノードがループの開始を表す。ループの戻りは、入力 1, 出力 2 のマージノードによって表し、ループの開始へと接続されるフローのガードに「true」を、ループ外に向かうフローのガードに「false」を記述すること。この出力はループの開始であるディシジョンノードへと他のノードを挟まず直接接続される。

## 例外処理

try-catch 文による例外処理を表現する上で、ステートチャート図を用いた階層表現による例外処理モデリング [6] を参考に行う。これをアクティビティ図上で表現をするため、ノード間に依存関係を作成することで疑似階層化させる。階層表現となるノードは、TryBlock, CatchBlock と内包される複数の Catch 処理層、FinallyBlock とし、これらすべての階層が記述される必要がある。依存関係は各階層がその内部の処理を表現するための開始・終了ノードに依存する形で結び、この間のノードも一意に関連付けられることとする。図 18 にアクティビティ図上での依存関係による疑似階層化を用いた try-catch モデリングを示す。

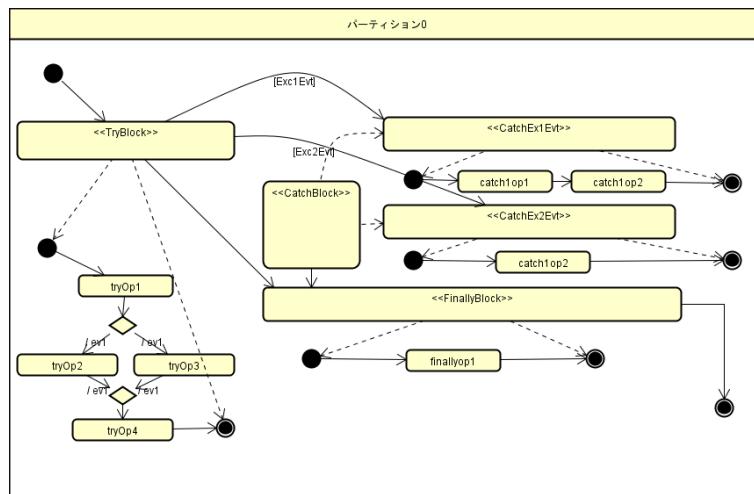


図 18 依存関係による疑似階層化を用いた try-catch モデリング

## ラベル付き break 文

ラベル付き break 文は、ラベルにより括られる区間の開始と終了を、入力 1, 出力 1 のディシジョンノードで表し、これとラベル名を記述したコネクタを依存線により結ぶことで表現する。依存関係は、ラベル名であるコネクタが依存先となるよう作成する。

## アクション

入力 1, 出力 1 のアクションノードで表す。try-catch 文の各階層の記述にもアクションノードは用いるが、例外的な使用として区別する。

## コネクタ遷移

コネクタによる遷移はプログラムコード的にはラベル遷移に当たるが、どのような箇所にも遷移可能であるというのは、フロー探査やコード出力といった観点から現実的ではない。そのため、ループ構造下での break/continue 文、ラベル下での break のみとする。記述規則としては、break, continue またはラベル名の記述されたコネクタであること、入力は 1 のみで出力を持たないこと。また、フローを途切れさせすことのないよう分岐構造下での記述となる。このとき、コネクタの記述された分岐が合流ノードと接続しないこととなるが、分岐数が合流数とコネクタにより途切れたフロー数の合計と一致すれば良いこととする。

### 5.3.2 文脈自由文法による解析

5.1.2 節で示した各ノード・フローを記号とみなし、開始ノードから終了ノードまでを文脈自由文法として解釈することで、提案手法により定義した構造となっているかの検査を行う。分岐構造やループ構造は、ディシジョン/マージノードを用いて表現されるが、これらノードの入出力フロー数から構造判定を行う。例外処理やラベル付き break など、記述規則による印付けにより構造判定を行っているものはアクションと同様考え、文脈自由文法解析の対象にはしない。各ノード・フローを記号として表したものを見ると図 19 に示す。使用されるノードをベースに、それに対する入出力フローの数で記号分けを行っている。ベースとなるノード  $N$  に対する入力フローを  $N_c$ 、出力フローを  $N_g$  とし、複数本ある場合は数字を付け表現している。

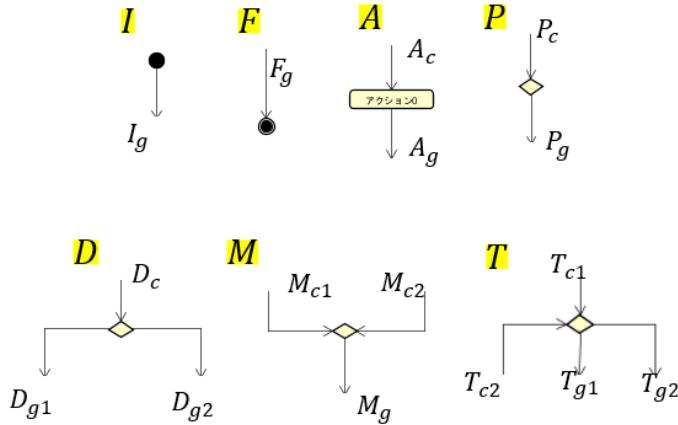


図 19 各ノード・フローを記号として表したもの

### 5.3.3 記述定義

開始ノード  $I$  から終了ノード  $F$  までのフローと接続するノード、ノードから接続するフロー、といったように接続順に記述を行い、1 単位の終わりには「;」を付ける。フローは小括弧で囲み、フロー同士の接続の場合、一つの小括弧で括り間には「,」を付ける。入力  $N_c$ 、出力  $N_g$  を持つノード  $N$  を記述したとき、 $(N_c)N(N_c);$  のようになる。

分岐構造を表現する場合、分岐は「<」、合流は「>」を用いて記述を行う。分岐・合流で囲った内部でそれぞれの枝の文脈を記述していく。図 19 での分岐  $D$ 、合流  $M$  の接続を記述したとき、 $(Dg)D;<(Dg1, Mc1), (Dg2, Mc2)>; M(Mg);$  となる。

ループ構造の表現をする場合、戻りとなるフローの入力が先に出現する。つまり入力の予約のような形となる。また以後出現するこの入力と接続されるフローを通常のフローと区別を付ける必要がある。そのため、予約された入力を  $N_{cin}$  とし、これを中括弧で囲み表現する。このフローへ向かう出力フローは  $N_{cout}$  とする。

図 19 を用いた図 20、図 21 のような制御構文の記述は以下のようになる。このとき、スライス  $S$  は各制御構文・例外処理とアクションノードの一塊を指す。また、アクション  $A$ 、スライス  $S$  は入出力が 1 ずつであるため、フローの記述は省略する。

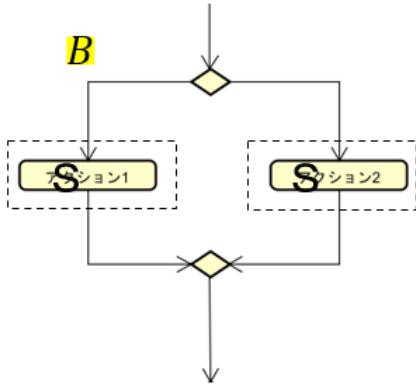


図 20 分岐構造:B

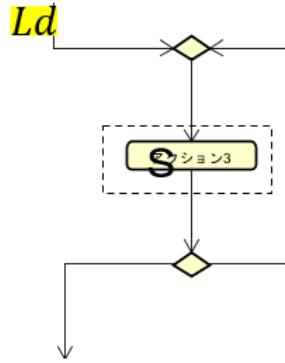


図 21 判定繰り返し構造:Ld

$$B \leftarrow D; < S^+, S^+ >; M; \\ \leftarrow (Dc)D; < (Dg1); S^+; (Mc1), (Dg2); S^+; (Mc2) >; M(Mg)$$

$$Ld \leftarrow M; S^+; D; < (Dg1), M >; \\ \leftarrow (Mc1)\{(Mc2in)\}M(Mg); S^+; (Dc)D; < (Dg1), (Dg2, Mc2out) > (Dg1);$$

このように、各制御構文・例外処理を定義することで、スライスをまたぐような構造記述や、フローの途切れといったものは受理できない記号列を作るため、不整合な記述と判断できる。

#### 5.4 図上へのユーザフィードバック

検査結果をユーザにフィードバックし、記述のサポートを行う。フィードバックとして、不整合内容に対応したエラー文の例を表5に示す。不整合の内容により Critical と Warning に分類される。Critical はコード生成不可能となるような重大なエラーであり、Warning はコード生成に問題をきたすレベルではないがエラーであることを示す。

表1 不整合内容に対するユーザフィードバック内容の例

評価値	不整合内容	ノート内容
1.Critical	11 図内にクラスを一つも持たない	Unable to find the Class in this ClassDiagram
	21 アクティビティ図の要素がクラス図に無い	Unable to find the [element] in ClassDiagrams
	31 ノードが途中で途切れている	Flow is broken at [node]
	41 分岐・合流/ループの開始と戻りの不一致	Unable to match any node with [node]
2.Warning	11 クラス図の要素がアクティビティ図に無い	Unable to find the [element] in ActivityDiagrams
	31 開始から終了の間で出現しないノード	[node] does not appear until FinalNode

これら不整合内容をノートとして図上に不整合要素と紐づけた形で生成を行う。ノートを用いた図上ユーザフィードバック例を図??に示す。これによりユーザは不整合要素に修

正を加え、再度検査を行うことで徐々に整合性のとれたモデルに近づけていくことを想定している。

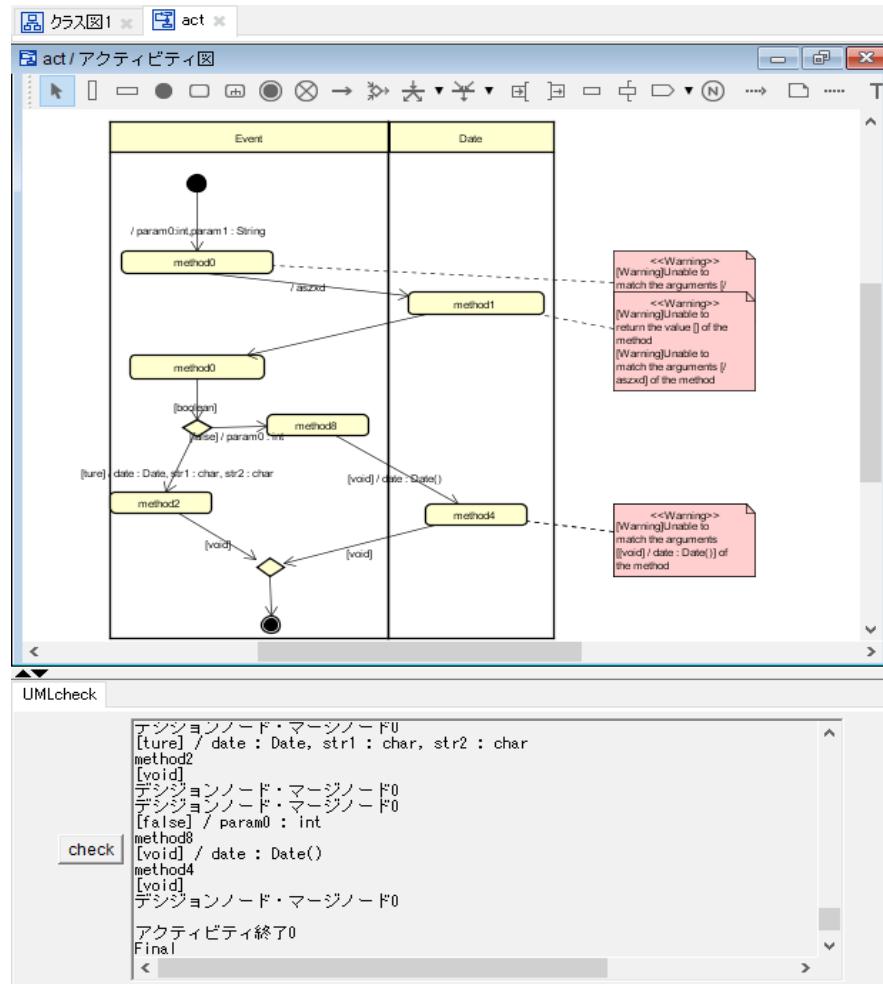


図 22 ノートを用いた図上ユーザフィードバック例

## 6 スライス解析を用いた自動コード生成

### 6.1 提案手法の自動コード生成プロセス

#### 6.1.1 従来研究との生成プロセスの違い

従来研究である Java Servlet スケルトンコード半自動生成器の作成された当時, astah\* プラグイン作成のための API は充実しておらず, ツールを用いて対象とする UML 図を XML 形式のファイルとして出力し, これを解析することでコード生成を行っていた.

本研究では astah\* API を用いて 4,5 章の提案手法を実装している. コード生成を行う際, これらによる要素の作成・解析情報を利用するため, コード生成器も同様に astah\* API を用いてのプラグインという形で実装を行う. そのため, 従来研究から生成プロセスが大きく変わることとなるため, オープンソースの astah\* プラグインであるモデル駆動開発 m2t プラグイン [5] を参考に実装を行う.

#### 6.1.2 モデル駆動開発 m2t プラグイン

m2t プラグインはオープンソースの astah\* プラグインであり, モデル駆動開発手法 MDD を astah\* 上で実現したものである. Groovy の Simple Template Engine を用いて, クラス図, ステートマシン図からコード生成を行う. UML 図上の要素を groovy のテンプレートに渡し生成を行うため, 用いるテンプレートを言語, 対象ドメイン別を作成することで様々なコードを生成することが可能である. また, 各要素のステレオタイプに応じたテンプレートを選択し生成を行う. ステレオタイプ別生成パターンは以下の三種類である.

**Default** ステレオタイプの無いクラスを対象にコード生成を行う

**Stereotype** ステレオタイプごとにテンプレートを指定

**Global** クラスごとでなく, プロジェクトに共通のコード生成を行う

#### 6.1.3 m2t プラグインによるコード生成

以下に m2t プラグインを用いたコード生成の例を示す. 使用する図 23, 図 24 の UML 図はオープンインターフェースによりユーザがコントロール可能な趣味用ロボット「iRobot Create2」の基本的な動きである, 待機, 前進, 旋回の状態遷移を表したものである. Listing1 に得られたコードを示す. 「iRobot Create2」向け C++ コード生成テンプレートを用いてステートマシンに記述された三状態のコードを生成している.

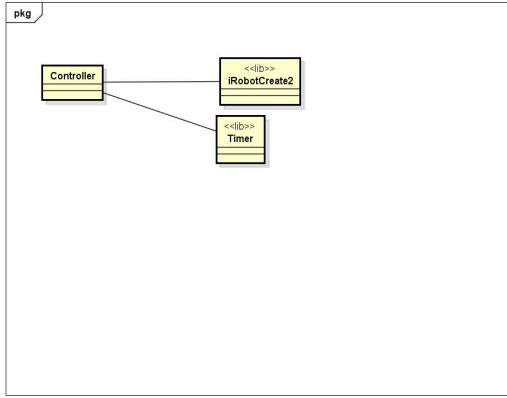


図 23 生成に用いるクラス図

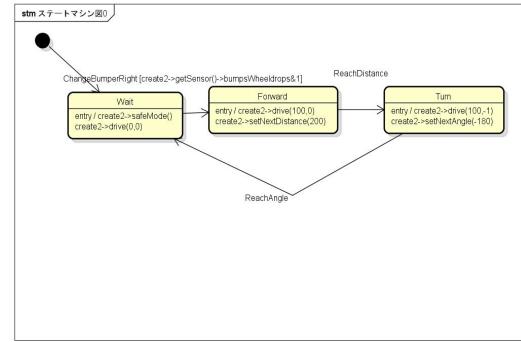


図 24 生成に用いるステートマシン図

Listing 1 :m2t プラグインにより生成された C++ コード

```

#include "Controller.h"
Controller::Controller(){
    controller_state=Wait;
    create2 = iRobotCreate2::getInstance();
    timer = Timer::getInstance();
    create2->safeMode();      // initial entry action
    create2->drive(0,0);      // initial entry action
}
Controller::~Controller(){}

void Controller::transition(Event event){
    switch(controller_state){
        case Controller::Wait:
            if((create2->getSensor()->bumpsWheeldrops&1)
                && ChangeBumperRight==event){
                controller_state = Controller::Forward;
                create2->drive(100,0); // entry action
                create2->setNextDistance(200); // entry action
            }
            break;
        case Controller::Forward:
            if(ReachDistance==event){
                controller_state = Controller::Turn;
                create2->drive(100,-1); // entry action
                create2->setNextAngle(-180); // entry action
            }
            break;
        case Controller::Turn:
            if(ReachAngle==event){
                controller_state = Controller::Wait;
                create2->safeMode(); // entry action
                create2->drive(0,0); // entry action
            }
            break;
        default:
            break;
    }
}
\label{ccode}

```

#### 6.1.4 m2t プラグインに対する改修点

提案手法によるコード生成を実現するため, m2t プラグインのコード生成部に改修を加える. m2t プラグインはクラス図とステートマシン図からコードを作成するが, これをアクティビティ図にも対応させる必要がある. スライス解析情報を用いるための処理, 提案手法に沿った生成用の groovy テンプレートファイルの作成を行う. 生成テンプレートの選択や UI, クラス図要素の取得といった処理は m2t プラグインのものを用いる. これらを踏まえた上で提案手法による自動コード生成プロセスを図 25 に示す. 図内の提案手法部として囲まれて部分が m2t プラグインに対して追加・改修を行った部分となる.

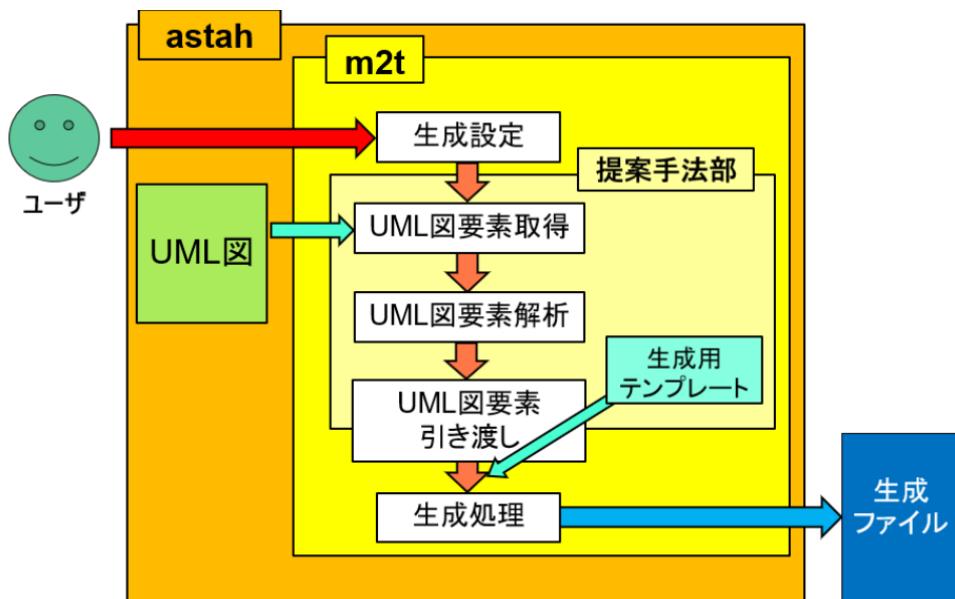


図 25 提案手法による自動コード生成プロセス

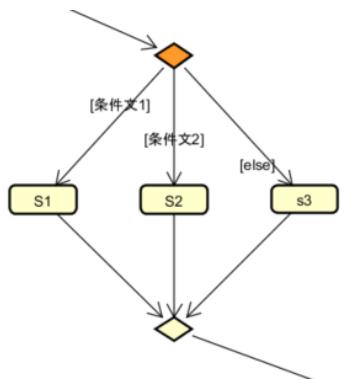
## 6.2 提案手法による自動コード生成

#### 6.2.1 アクティビティ図要素と生成コードとの対応

アクティビティ図要素からどのコードを生成するかの判断は、5.3節による解析結果を用いる。スライス解析により、各制御構文・例外処理の記述規則を満たしているかチェックを行う際、分岐構造、前判定ループ構造といったスライスの分類情報を出現ノードと紐づけしたリストを作成する。コード生成の際、このリストを生成用のgroovyテンプレートに引き渡し、スライスの分類、ノードの種類に応じて処理が行われる。各構造と生成コードとの対応を述べていく。

### 6.2.2 if 文

if文の生成には分岐を示す各フローのガードに記述されたものをそのまま条件文として出力する。1本目のフローは「if(条件文)」として、2本目以降は「else if(条件文)」のように生成される。ただし、ガードに「else」と記述されていた場合、「else」として生成される。



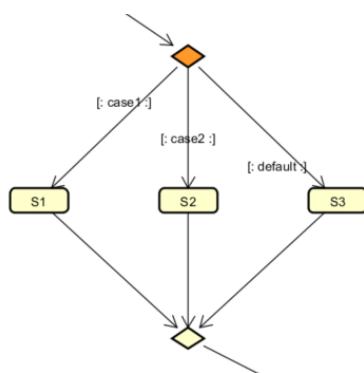
Listing 2 :図 26 より生成される if 文

```
if( 条件文1 ){
    //S1
}
else if( 条件文2 ){
    //S2
}
else{
    //S3
}
```

図 26 生成元となる if 文モデル例

### 6.2.3 switch-case 文

構造的には if 文と同様であるため、生成時の区別を付けるため、分岐を表すディシジョンノードにキーが「switch」、値が「1」のタグを付ける。スライス解析時はこれを目印に switch 文と分類する。switch 文の式は分岐を表すディシジョンノードの定義部に記述する。switch 文の各定数部は、if 文の条件文記述との差別化のため「:」で囲み記述を行う。ガードに「: default :」と記述されていた場合、「default」として生成される。また、分岐の終端となる合流ノードとの接続に合わせ、各 case の break 文が生成される。



Listing 3 :図 27 より生成される switch-case 文

```
switch(num){
    case case1 :
        //S1
        break;
    case case2 :
        //S2
        break;
    default :
        //S3
        break;
}
```

図 27 生成元となる switch-case 文モデル例

### 6.2.4 do-while 文

ループ条件の記述は判定部分であるマージノードの定義部に記述を行う。

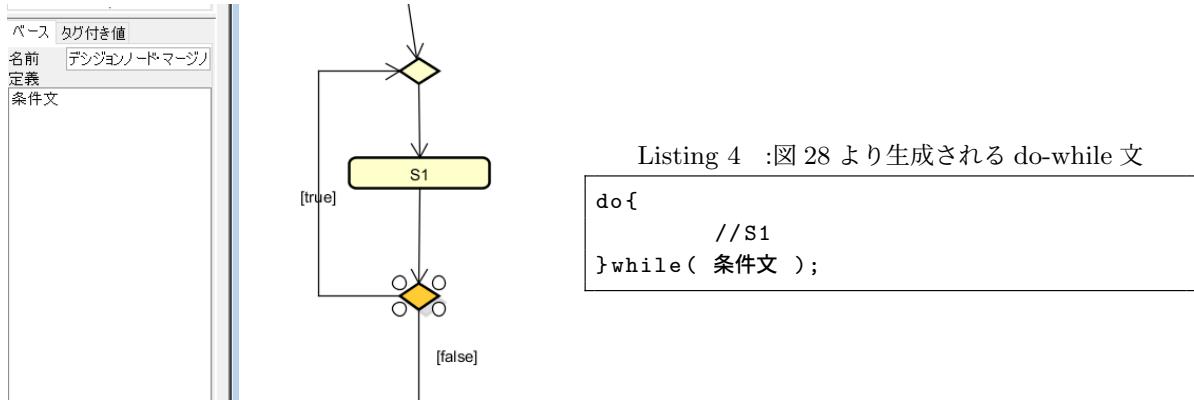


図 28 生成元となる do-while 文モデル例

### 6.2.5 for-while 文

do-while 文と同様に、ループ条件の記述は判定部分であるディシジョンノードの定義部に記述を行う。



図 29 生成元となる for-while 文モデル例

### 6.2.6 while 文

if 文と switch-case 文の区別と同様、for-while 文との区別のため、条件判定部を表すディシジョンノードにキーが「while」、値が「1」のタグを付ける。それ以外は for-while 文の生成と大きな違いはない。図 29 のモデルに while 文識別のためのタグ付き値を付与した場合、Listing6 のようなコードが得られる。

Listing 6 :生成される while 文

```
while( 条件文 ){
    //S1
}
```

### 6.2.7 try-catch 文

try-catch 文の生成は、try 部、catch 部、finally 部に分けられる。これらは階層となる TryBlock, CatchBlock, FinallyBlock との依存関係から求められる。拾う例外については TryBlock から各 Catch 処理層への遷移フローのガードに記述を行う。

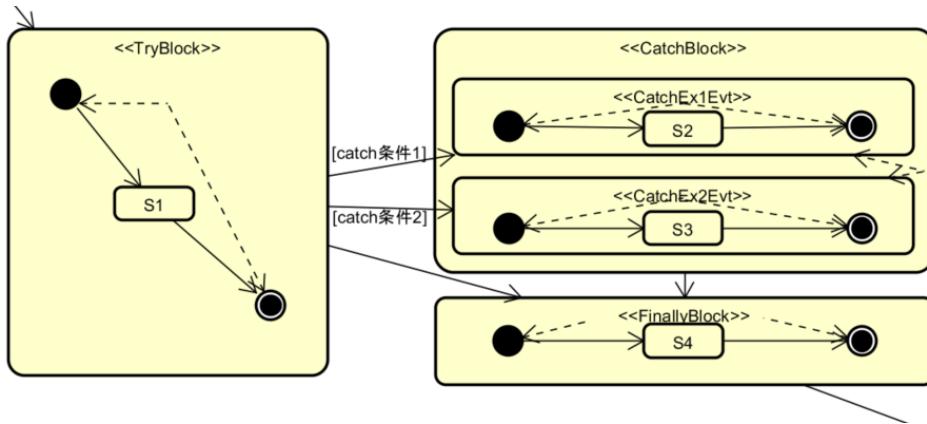


図 30 生成元となる try-catch 文モデル例

Listing 7 :図 30 より生成される try-catch 文

```
try{
    //S1
}
catch(条件catch1){
    //S2
}
catch(条件catch1){
    //S3
}
finally{
    //S4
}
```

### 6.2.8 アクション

アクションノードはノード名をコメントアウト文として生成を行う。

### 6.2.9 ラベル付き break 文

コネクタと依存線で結ばれたディシジョンノードを目印に、コネクタ名として記述された名前をラベル名として生成する。

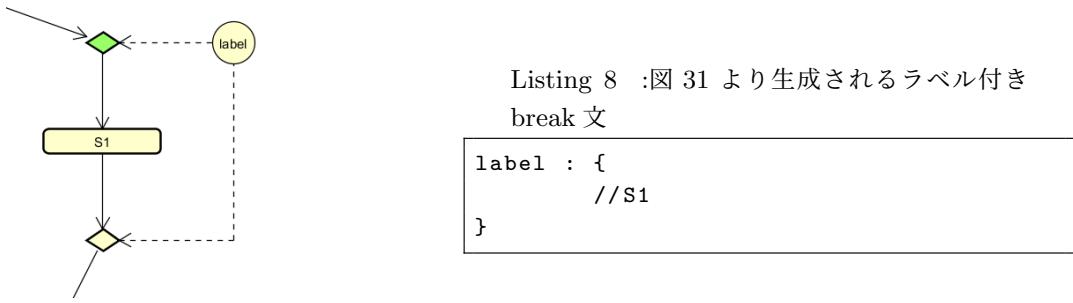


図 31 生成元となるラベル付き break 文モデル例

### 6.2.10 コネクタ

コネクタの処理は条件下によって異なる。ラベル付き break 下で出現し、コネクタ名がラベル名であった場合、ラベル付き break の遷移として扱う。そうでない場合は break/continue によるループ抜け遷移とする。図 32 は一つ上の do-while 文を抜けるための単純な break 文のモデル例である。



図 32 生成元となる break 文モデル例

### 6.2.11 生成例

以下に生成例を示す。図 33 から生成された Java コードのアクティビティ図を元に生成された部分を抜き出したものが Listing10 である。

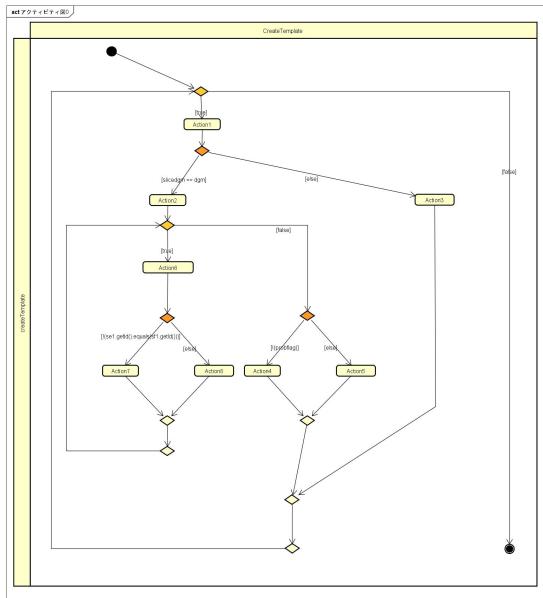


図 33 生成に用いるアクティビティ図

Listing 10 :図 33 より生成されるロジック部のコード

```
for(IActivityDiagram slicedgdm : UMLcheck.slicematchact.keySet()){

    //Action1
    if(slicedgdm == dgm){
        //Action2
        for(IFlow sf1 : sfm.keySet()){
            //Action6
            if(!(sel.getId().equals(sf1.getId()))){
                //Action7
            }
            else{
                //Action8
            }
        }
        if(!(probflag)){
            //Action4
        }
        else{
            //Action5
        }
    }
    else{
        //Action3
    }
}
```

## 7 評価

提案手法により作成した挿入支援器、構造検査器、自動コード生成器の評価実験を行っていく。

### 7.1 挿入支援機能

#### 7.1.1 評価実験の概要

提案手法による挿入支援機能の評価として、アクティビティ図の作成時に、全て手動で操作した場合と挿入支援機能を用いた場合との比較を行う。フローチャートと図を元にアクティビティ図を作成する。作成は筆者が行い、試行回数は各一回ずつとする。評価する項目は以下のようである。

##### クリック数

マウスの総クリック数。

##### 所要時間

作成に要した時間。作成開始時から図を完成させ構造検査により不整合が検出されなくなるまでの時間とする。

##### 要素追加操作数

astah\*上へ追加した要素の総数。要素の追加には手動追加と挿入支援機能による追加がある。支援器ありの要素追加数は挿入支援機能による追加数と手動追加数の合計となる。手動追加数の計測はプロジェクトが変更されたイベントをリスナで取得し行う。

##### 要素選択数

astah\*上で選択した要素の数。主にノードやフローの名前を記述する際に用いる。挿入支援器による要素の挿入を行う際もフローを選択することになる。計測は astah\*API である図の選択状態が変更されたイベントをリスナで取得し行う。

##### 不整合数

作成後に構造解析器を用いて、検出された不整合の数。提案手法により抑制を狙っている上位設計作成時の不整合混入にあたる。

#### 7.1.2 case1

以下の図 34 のフローチャート図を元にアクティビティ図を作成する。この図には、分岐構造、前判定ループ構造、後判定ループ構造、例外処理が使用されている。総要素数は 192 と規模は大きくなく、構造も複雑ではない。

実験結果を表 2 に、作成したアクティビティ図を図 35 に示す。

表 2 case1 の結果

	クリック数	所要時間 (分)	要素追加操作数	要素選択数	不整合数
支援器なし	284	36	202	110	0
支援器あり	302	23	24(手動 4)	112	0

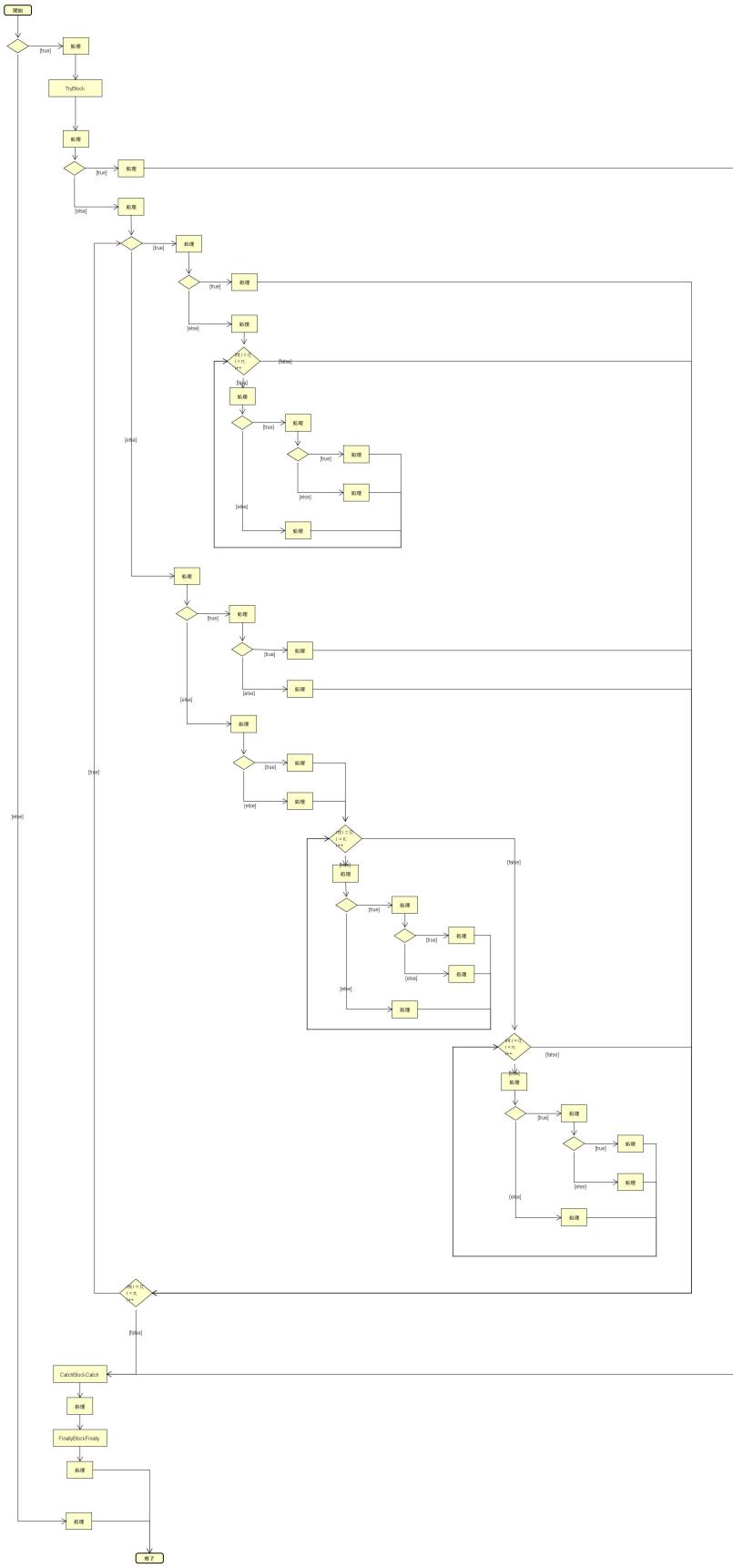


図 34 case1:ベースとなるフローチャート図

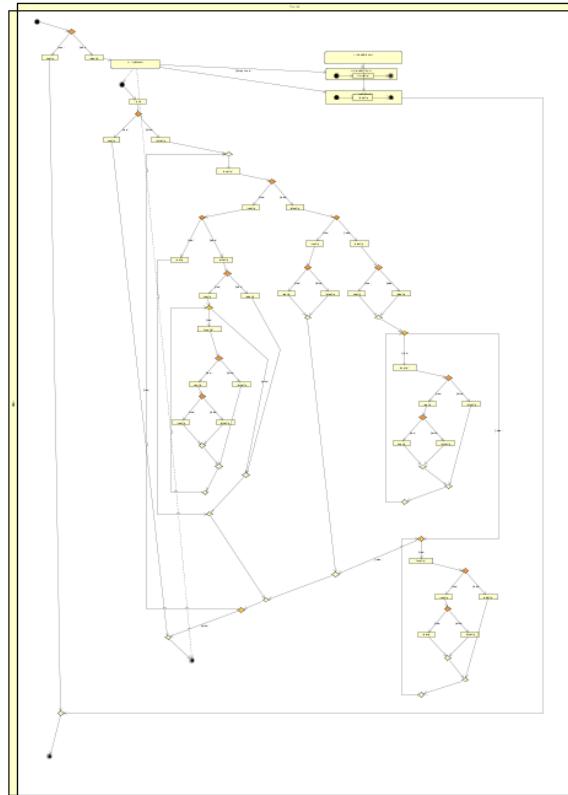


図 35 case1:作成したアクティビティ図

### 7.1.3 case2

以下の図 36 のフローチャート図を元にアクティビティ図を作成する。出現する構造は分岐構造、後判定ループ構造、コネクタ遷移、例外処理が使用されている。総要素数は 336 であり、後判定ループ構造が多く含まれている。また、徐々にネストが深くなっていくような構造である。

実験結果を表 3 に、作成したアクティビティ図を図 37 に示す。

表 3 case2 の結果

	クリック数	所要時間(分)	要素追加操作数	要素選択数	不整合数
支援器なし	1863	72	343	257	3
支援器あり	1106	45	32(手動 12)	232	0

図 36 case2:ベースとなるフローチャート図

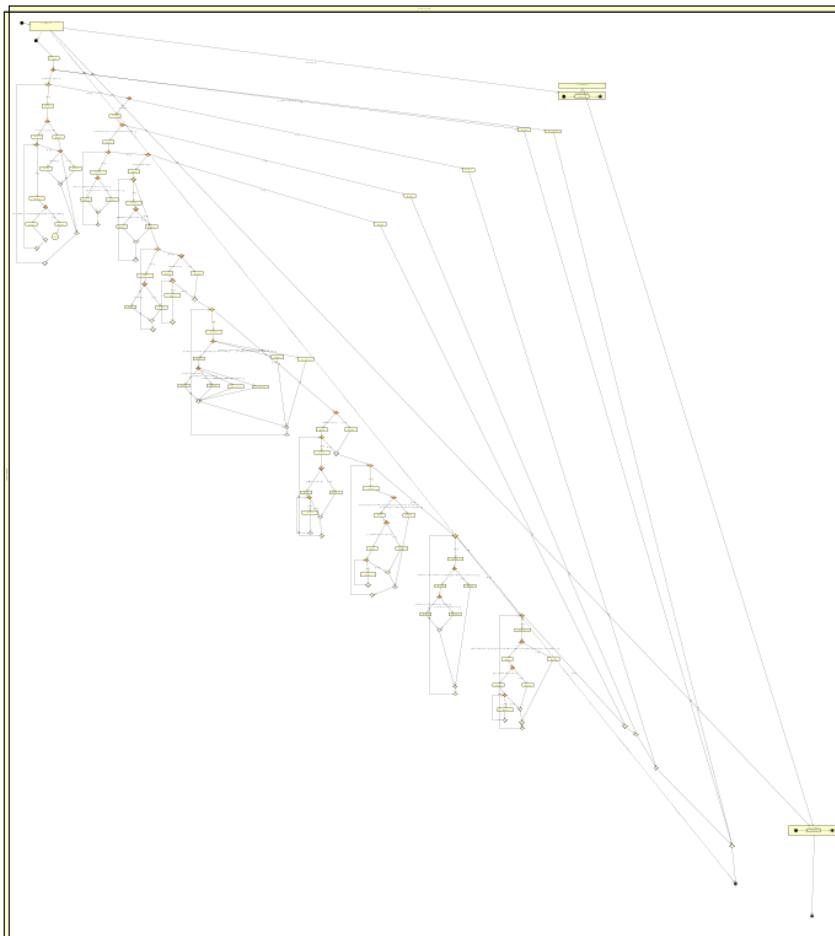


図 37 case2:作成したアクティビティ図

#### 7.1.4 case3

以下の図 38 のフローチャート図を元にアクティビティ図を作成する。出現する構造は分岐構造、後判定ループ構造が使用されている。総要素数は 468 であり、分岐構造が多く含まれている。分岐の枝それぞれのネストはあまり深くないが、分岐先でさらに分岐していく横に広がっていくような構造となっている。

実験結果を表 4 に、作成したアクティビティ図を図 39 に示す。

表 4 case3 の結果

	クリック数	所要時間(分)	要素追加操作数	要素選択数	不整合数
支援器なし	2761	93	482	401	5
支援器あり	1560	54	50(手動 13)	353	0

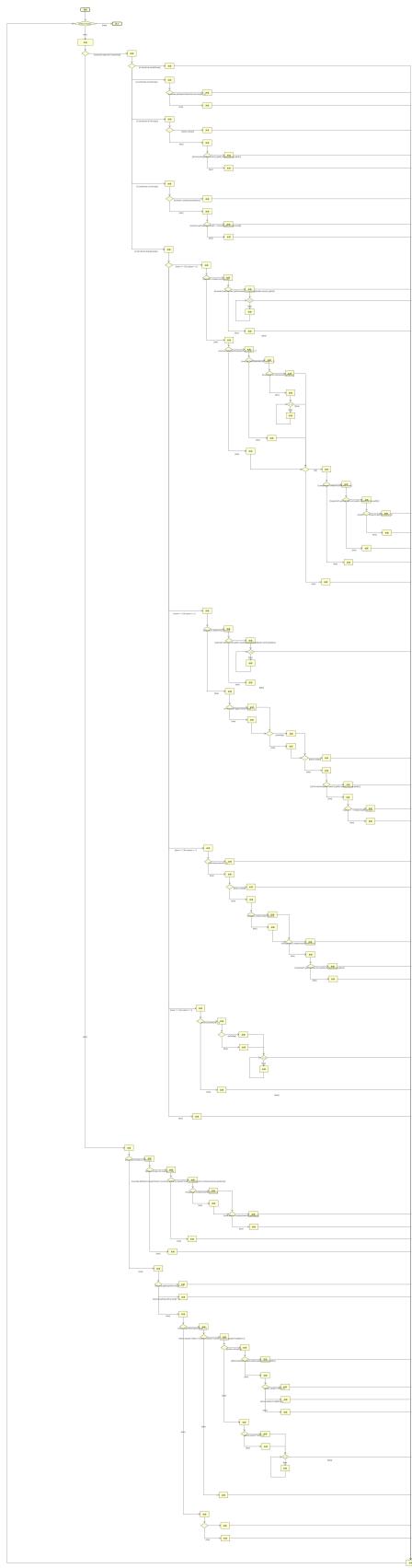


図 38 case3:ベースとなるフローチャート図

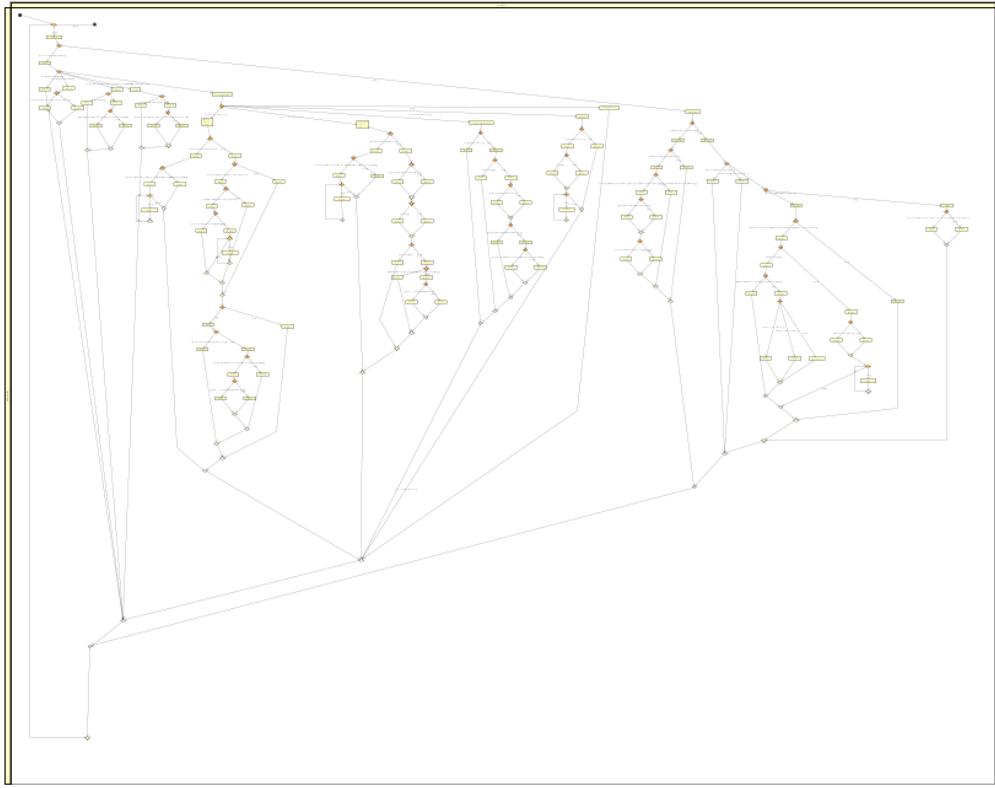


図 39 case3:作成したアクティビティ図

### 7.1.5 実験の考察と評価

case1 では支援器なしとありで共通し、不整合数は 0 であった。より複雑な構造となる case2, case3 においては、支援器なしでは不整合が混入していたのに対し、支援器ありでは case1 同様 0 となった。挿入支援器は使用時に構造検査を行い、挿入可能箇所を判断するため、整合性を維持したまま要素の追加を行えるためであると考える。

また、クリック数、所要時間、要素追加操作の項目において、支援器ありの方がより少ない操作で作成可能であると示された。例えば、分岐数 2 の if 文を挿入する際、分岐のディシジョン、各枝のアクションノード、合流のマージノード、これらを結ぶフローの計 10 要素を作成する必要がある。挿入支援器を用いた場合、フローの選択、挿入テンプレートの選択、挿入ボタン押下の 3 クリックで作成することができるため、要素追加操作数では特に優位性が見られる。

一方、要素選択数ではあまり優位性が見られなかった。これは、手動の場合、要素の作成と同時に要素名を記入できるのに対し、支援器を用いた場合は挿入操作後に各要素を選択し、編集を行う必要があるためである。また、支援器によるレイアウト調整機能がユーザの想定を満たしておらず、その調整のための操作が発生したことの一因であると考える。

これらより、挿入支援機能の狙いである上位設計作成時の不整合混入抑制、アクティビティ図記述コストの低減としての働きは確認できた。

## 7.2 整合性検査器

### 7.2.1 評価実験の概要と結果

提案手法による自動コード生成の評価として、UML 図に手動で不整合を混入させ、これらを検出できるかの確認を行う。提案手法による構造検査は開始ノードから終了ノードへとフロー探査を行っていくものであり、探査中に不整合な構造と検出された場合、その後出現する構造が正しい構造か確認ができないため、フロー探査はそこで終了する。そのため、一度の構造検査では図内全ての不整合は検出できない。評価実験では、検査を行い、都度不整合を修正しながら不整合が検出されなくなるまで行うものとし、検出不整合数はこの合計となる。また、意味論解析を行うものではないため、挿入する不整合は構造的なものであり、意味論的な不整合は対象外とする。実験では図 37 のアクティビティ図に対し要素の追加・変更を適宜行う。実験番号 2 での挿入不整合例を表 5 に示す。

表 5 実験番号 3 での挿入不整合例

挿入 不整合番号	挿入内容
e1	水平パーティションがない
e2	クラス図に一致するクラス名がない
e3	TryBlock のステレオタイプが記述されていない
e4	TryBlock の出力フローの不足
e5	スライスをまたぎ接続されるフロー
e6	後判定ループ構造の遷移条件の不足
e7	ループ下にない break 文
e8	分岐構造における合流ノードがない

実験結果は表 6 の通りとなった。

表 6 実験番号 3 での挿入不整合例

実験番号	挿入不整合数	検出不整合数
1	4	4
2	4	4
3	8	9
4	8	8
5	12	14
6	12	13
7	16	17
8	16	16

### 7.2.2 実験の考察と評価

最終的に不整合が検出されなくなる間に挿入された不整合が発見されなかつたことはなかったが、実験番号 3, 5, 6, 7 では、挿入された不整合より多くの不整合が検出される結果となつた。これは不整合を挿入する際、余分な出力フローが足したとき、これを入力とするノードにも記述規則の逸脱が発生する場合があるからである。

また、複数の不整合、記述規則の逸脱が重なることにより、他の構造の記述規則を満たしてしまい、不整合要素の誤判定が見られた。例として、実験 2 の e3 である TryBlock のステレオタイプが記述されていないという不整合に、e4 の TryBlock からの出力が足りないという不整合が重なつた場合、TryBlock 判定を受けず、アクションの記述規則である入力 1、出力 1 を満たすこととなる。この場合、CatchBlock などは記述されているため、そこに遷移して初めて、try-catch 文の記述規則を逸脱していることが検出される。

これらより、検査・修正を行っていく上で挿入した不整合を漏らすということはなかつたが、不整合要素を正確に把握した検出については完全ではないという結果となつた。

## 7.3 自動コード生成

提案手法による自動コード生成の評価として、既存研究である Java Servlet スケルトンコード半自動生成器との生成能力の機能の比較を行う。

### 7.3.1 既存研究との比較

提案手法による生成能力は 6 章の通りである。Java Servlet スケルトンコード半自動生成器によるコード生成は対象ドメインを定めた特化型であり、具体的には、予約システムを対象とし、VDMJWeb サービスを用いた妥当性確認のための UI, Java Servlet, VDM++ のコード生成を目的としている。これら 3 層のやり取りを主とするため、シグナルノードや各ノードの定義部への記述から、画面の切り替え、入力イベント、データ通信といった通信アクションがサポートされている。一方、ロジックを記述する Java Servlet 層では、if-else 文、switch 文といった分岐構造のみを生成可能な制御構文としている。

### 7.3.2 考察と評価

提案手法によるコード生成はテンプレートを用いるため、テンプレートの拡張により生成言語、内容を変えることが可能である。しかし、現状の提案手法が有する生成能力は、Java Servlet スケルトンコード半自動生成器による予約システムのプロトタイプ生成には不十分である。

## 8まとめと今後の課題

### 8.1 まとめ

本論文では、上位設計での正しさを確認した上でプロトタイピングを行うため、テンプレートベースのUML上位設計に対する作成支援、アクティビティ図に対する制御構文・例外処理の構造検査、スライス解析を用いた自動コード生成手法を提案した。

従来研究では用いられなかったastah\*APIを使用し、一つのプラグインとしてこれらを実装した。このとき、UML図の設計を図として感覚的に捉えることのできる特性を生かし、視覚的な挿入支援、検査結果の図上フィードバックを行い、上位設計作成時の不整合混入抑制、記述コストの緩和を試みた。

各機能ごとに実験を行い提案手法の評価を行った。挿入支援器については、レイアウト調整の不十分さなどの課題はあったが、この手法の狙いである不整合混入抑制、記述コストの緩和としての優位性が確認できた。構造検査器については、実験の範囲内では不整合の漏れは起こさなかったが、複数不整合の併発による誤判定が懸念される。自動コード生成器は既存研究に比べ、ロジック層の構造的な記述については記述可能な制御構文・例外処理の観点では優位と言える。しかし、通信に関する記述や特定ドメイン向けの生成といった点に関してはサポートできておらず、本研究が目指す、早期な妥当性確認のための生成系としては不十分である。

提案手法全体としては、従来研究では人力の作業であったUML図作成や構造検査を、astah\*APIを用いて機械的に行うことを可能とし、構造的な正しさを保った上で、テンプレートに対応したコードを得ることができるという結果となった。

### 8.2 今度の課題

#### 8.2.1 整合性検査の発展

提案手法による整合性検査は、最低限のクラス図とアクティビティ図の要素一致検査と、アクティビティ図の各制御構文・例外処理に対する構造検査のみである。これらの検査を現在はプラグインにより作成されたボタンを押すことで実行しているが、図要素の変更を監視し、適当なタイミングで解析を行うことが出来れば、上位設計の作成支援、不整合混入抑制の働きとなる。また、生成コードの信頼性を高めるために、分岐やループ構造に記述する条件文に使用する変数の記述規則を設け意味論解析を行うことが望ましい。意味論解析とスライス解析と合わせることで、ある変数を変更した際に影響が及ぶスライス、スライスに変更を加えた際に影響の及ぶ変数が判断可能となり、バックグラウンドでの構造検査タイミングや、検査の必要な範囲を絞ることができ、仕様の変更・追加の際の不整合混入抑制が期待できる。

#### 8.2.2 テンプレートの拡充

提案手法による自動コード生成の現状では、生成言語はJavaであり、生成能力は基本的な制御構文・例外処理のみと、元となったJava Servletスケルトンコード生成器の目的でもあった早期な妥当性確認を行うための生成器としては不十分である。提案手法による

上位設計の作成から自動コード生成までの流れは、サポートするテンプレート構造に一对一で対応する生成用テンプレートを用いる形であるため、テンプレートの拡充を行うことで、パーティションごとにテンプレートを使い分けることによる3層データモデルの作成・コード生成といった、生成能力の強化が可能である。このとき、テンプレートの拡大に伴い、それに対応する構造検査の追加を行う必要がある。

### 8.2.3 大規模な上位設計への対応

提案手法により想定しているアクティビティ図の規模は1アクティビティのみであるが、大規模なシステムフローを1アクティビティに全て記述を行うのは、図の可視性や編集の行いやすさといった点から現実的ではない。これに対し、振る舞い呼び出しアクションによる陰仕様記述といった解決方法が考えられる。振る舞い呼び出しアクションはアクティビティ図を内包するノードであり、入出力フローに事前・事後条件を記述することで陰仕様を表現することができる。このとき、抽象度の違いや図間の接続性といった問題がメソッド単位での畳み込みやワークフロー型での記述といった制限を設ける必要がある。

## 参考文献

- [1] 村林慧, 多田圭佑, 和崎克己 : VDMJ と Apache Axis2 を用いた上流工程におけるモデル実行環境の構築, FIT2014 講演論文集(第 13 回情報学技術フォーラム), (B-108), pp.155-158, 2014
- [2] 村林慧, 和崎克己 : 分散オブジェクト基盤を用いた VDM 向けモデル実行環境と管理機能の構築, 信州大学大学院工学系研究科修士論文, 2015
- [3] 森島耕, 和崎克己 : テストコード半自動生成を用いたプロトタイプ検査と VDM++ 妥当性確認のコスト削減, FIT2016 (第 15 回情報科学技術フォーラム) 講演論文集, (A-010), pp.103-106, 2016
- [4] 株式会社 チェンジビジョン : astah\*professional, <http://astah.changevision.com/ja/product/astah-professional.html>
- [5] 細合晋太郎 : モデル駆動開発 m2t プラグイン, <https://github.com/s-hosoai/astahm2t>.
- [6] G.Pinter, I.Majzik : Modeling and Analysis of Exception Handling by Using UML Statecharts, LNCS 3409, pp.58–67, 2004
- [7] 佐原伸 : ~ソフトウェアトラブルを予防する~ 形式手法の技術講座, ソフト・リサーチ・センター, 2008
- [8] Scott W. Ambler 著, 越智典子訳 : オブジェクト開発の神髄 : UML2.0 を使ったアジャイルモデル開発のすべて, 日経 BP 社, 2005
- [9] 多田圭佑, 村林慧, 和崎克己 : VDM++ を用いたラピッドプロトタイピング向けの Java スケルトンコード生成器, 平成 26 年度電気関係学会東海支部連合大会講演論文集, (p1-1), 1page(DC-ROM), 2014
- [10] 村林慧, 多田圭佑, 和崎克己 : VDMJWeb サービスを用いた妥当性確認とワーカースペース機能の実現, 平成 26 年電子情報通信学会信越支部大会講演論文集, (1B-1), 5, 2014
- [11] 多田圭佑, 村林慧, 和崎克己 : UML 上位設計を用いた Java スケルトンコード生成器によるラピッドプロトタイピング, 平成 26 年度電子情報通信学会信越支部大会講演論文集, (1B-2), 6, 2014
- [12] 森島耕, 和崎克己 : UML と VDM++ コードを利用した妥当性確認とテストコード半自動生成, 平成 27 年電子情報通信学会 信州大学 Student Branch 論文発表会 講演論文集, 3, 2015
- [13] 石川冬樹 : VDM++ による形式仕様記述, 近代科学社, 2011
- [14] ジョン・フィッツジェラルド, 他 : VDM++ によるオブジェクト指向システムの高品質設計と検証, 翔泳社, 2010

## 謝辞

本論文を執筆するにあたって、熱心にご指導いただいた信州大学工学部和崎克己教授には、終始熱心な御指導、御教示を賜った。心より感謝申し上げる。