Vol.2011-SE-174 No.1
2011/11/1

メタモデルを用いたソースコードからのモデル自動抽出手法の提案

市井誠[†]　明神智之[†]　小川秀人[†]

モデル駆動型開発（MDE）は，ソフトウェアの品質や開発効率を向上させることが知られている．モデルを主たる開発成果物としない開発現場においては，リバースエンジニアリングによるソースコードからのモデル抽出により，MDE 技術の恩恵を受けることができる．しかし，モデルはその用途によって異なる観点や抽象度で作成する必要があり，既存のリバースエンジニアリング手法では目的に応じたモデルを得ることは困難である．そこで，本研究では，モデルの抽象化を考慮したソースコードからのモデル自動抽出手法 Program Oriented Modeling（POM）を提案する．POM はメタモデルによるモデル定義と，モデル変換言語による段階的抽象化により，目的に応じたモデルを柔軟に抽出する．また，モデル検査向けの検証モデルを抽出するツール POM/MC を試作した．また，適用実験により，検査可能なモデルを抽出できることを示した．

A Metamodel-based Automated Method for Extracting Software Model from Source Code

Makoto Ichii,[†]　Tomoyuki Myojin[†]
and Hideto Ogawa[†]

Model driven engineering (MDE) improves software quality and productivity. Reverse engineering provides a way to acquire merits of the MDE technologies without committing to the MDE-style development by extracting software models from source code. However, the conventional reverse engineering methods cannot provide the software model that is suitably abstracted for the use case. Therefore, we propose a novel method for extracting software model from source code, named Program Oriented Modeling (POM). POM extracts the model suitable for the use case by stepwise abstraction with the model transformation language based on the metamodel technology. We have developed a verification model extractor for model checking, named POM/MC. Additionally, we have conducted a case study, where we have successfully extracted the model-checking-capable model from source code.

1. はじめに

ソフトウェアのモデル駆動型開発（Model Driven Engineering, 以降 MDE）は，ソフトウェアの品質や開発効率に大きく貢献すると言われている[1][2]. 曖昧になりがちな自然語の代わりに，意味論の定義されたモデルを用いて仕様や設計を記述することにより，開発現場でのコミュニケーションを円滑にする． また，厳密な定義を持つモデルを用いれば，自動的な検証やコード生成が可能となり，開発の省力化にも繋がる．

一方で，MDE の導入には様々な障壁が存在するため，ソースコード主体の開発を脱することの出来ない現場も多く存在する．例えば，既存ソフトウェアのソースコードに変更を加えていく形で開発を進める，世代型開発と呼ばれる開発スタイルの場合，MDE への移行のためには既存のソースコード資産をモデルへと書き換える必要があり，大きな初期コストを要する．

リバースエンジニアリングによるモデル抽出は，ソースコードベースの開発現場において，MDE の恩恵を得るための一手段である．多くの UML モデリングツールは，ソースコードからクラス図等のモデルを抽出する機能を提供する．Fujaba[3]はソフトウェア理解支援を目的とし，Java ソースコードからの UML モデルの抽出およびデザインパターンの適用箇所の特定をおこなう．

しかし，抽出されたモデルは，利用目的に適した抽象度である必要がある一方，既存のリバースエンジニアリング手法は，その手法の（想定する用途の）抽象度でしかモデルを抽出できない. 例えば, UML モデリングツールにより抽出されるクラス図は，実装構造と同じ粒度であり，全体的な構造の把握には向かない．

そこで，本稿では，モデルの抽象化を考慮した，ソースコードからのモデル抽出手法 Program Oriented Modeling（POM）を提案する．提案手法では，利用者による柔軟なモデルの抽象化のため，メタモデルを用いたモデル変換により，抽象化の操作を利用者がルール化して与える手段を提供する．

また， C 言語ソースコードから，Promela モデルを抽出するツール POM/MC を構築した．また，適用実験により，ソースコードからモデル検査可能なモデルを抽出できることを示した．

以降，2 節でソフトウェア開発におけるモデル利用について述べた上で，3 節にて提案手法である POM について述べる．続いて，4 節にて POM の検証モデル向け実装 POM/MC について述べ，5 節にて適用実験をおこなう．6 節にて考察をおこない 7 節にて関連研究を述べる．最後に 8 節でまとめと今後の課題について述べる．

[†]株式会社日立製作所　横浜研究所
Hitachi, Ltd., Yokohama Research Laboratory

2. ソフトウェア開発におけるモデル

ソフトウェア工学の文脈において「モデル」はさまざまな意味に用いられる．本稿では，「あるプログラム（ソフトウェア）に関する情報を，特定の観点で切り出し，特定の抽象度で表現したもの」と定義する．

ソフトウェア開発においては，開発タスクに応じて，さまざまな観点からモデルが作成される．例えば，UML のクラス図はオブジェクト指向プログラムのクラス構造を表現するモデルであり，状態マシン図はソフトウェア（もしくはその一部）の振舞いを，状態とその遷移を中心として表現するモデルである[4].

一般的な MDE においては，抽象度の高いモデルを詳細化していくことで開発を進める[1][2]. それぞれの開発フェーズにて検討･検証すべき情報に集中することができ，開発の手戻りを極小化することが可能となる． また，抽象度の異なるモデル間もしくは観点の異なるモデル間でトレーサビリティを取ることで，設計と実装との乖離を防ぐこともできる．

一方で，リバースエンジニアリングによりソースコードからモデルを抽出することで，ソースコード主体の開発現場においても MDE 技術の恩恵を得ることができる．例えば，解析したソースコードをクラス図として表現することで，ソフトウェア構造の理解を助けることができる．また，形式記法を用いたモデルを抽出することで，ソフトウェアの形式検証が可能となる．

しかしリバースエンジニアリングツールにより抽出されたモデルは，目的のタスクに対して抽象度が合わないおそれがある．理解や分析が目的の場合，抽象度が高すぎるモデルは情報量が不足し，抽象度が低すぎるモデルは大規模・複雑となりソースコードそのままの方が役立つこともある．抽出したモデルを人間ではなく計算機が解釈する検証用途であっても，複雑すぎるモデルは，計算量が計算機の能力を超えるおそれがある．

次節以降，ソースコードから，目的に応じた観点のモデルを，必要な抽象度で抽出するための手法について議論する．

3. モデル抽出手法 Program Oriented Modeling の提案

本稿では，ソースコードからモデルを自動抽出する手法 Program Oriented Modeling（POM）を提案する．POM の概念を図 1 に示す．POM は，モデル抽出の際の観点や抽象度を柔軟性に構成するため，モデル抽出の枠組み（Modeling Infrastructure）と，具体的な抽出方法（Modeling Configuration）を切り分けた構成をもつ．モデル抽出の枠組みに，モデル定義などの基礎的なモデル構成を組み合わせることで，検証や理解支援といったユースケースごとのモデル抽出アプリケーションが構築される．個々のアプリケーションは，抽象化方法などのソフトウェアのドメインもしくは利用者の目的に応じたモデル構成を入力とし，ソースコードからモデルを抽出する．

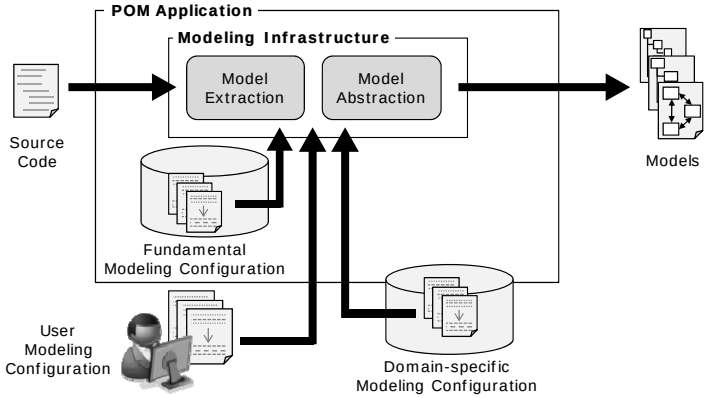

図 1 POM によるモデル抽出

3.1 モデル抽出の方針

モデル抽出の基本的なアイデアは，ソースコードから，その実装に忠実に情報を抽出した「モデル」を，モデル変換技術を利用し，目的の観点・抽象度のモデルへと変換することである．モデルの観点および抽象度の与え方を次に示す．

(1) メタモデルによる観点の定義
POM はモデル抽出において実装モデル，抽象プログラムモデル，目的モデルの 3 種類の中間モデルを用いる．それぞれ，ソースコードと等価な意味論を持つ実装レベルのモデル, 抽象化を実現するための抽象的な情報を保持するモデル，最終的なモデル出力に必要な情報を保持するモデルを意味する．中間モデルそれぞれは，MOF[5]によりメタモデルを定義し，モデル変換言語で記述する変換ルールによりモデル間の変換を実現する．中間モデルのメタモデルおよび対応する変換ルールの差し替えにより，異なる観点でのモデル抽出を実現する．

(2) モデル変換ルールによる段階的抽象化
抽象プログラムモデルを，複数のモデル変換ルールにより繰り返し操作することにより，意味論の変換や詳細の捨象などの抽象化を実現する．ルールの記述には，モデル変換言語 QVT[6]を用いる．

3.2 POM のアーキテクチャ

POM のアーキテクチャを図 2 に示す．POM は大きく分けて以下の 3 つのサブシステムから構成される．

(1) ソースコード解析部（Parse）：実装レベルのモデル構築

(2) モデル変換部（Intermediate Model Transformation）：メタモデルによる観点の定義とルールによるモデル変換の実現
(3) モデル出力部（Target Model Generation）：目的形式でのモデル出力

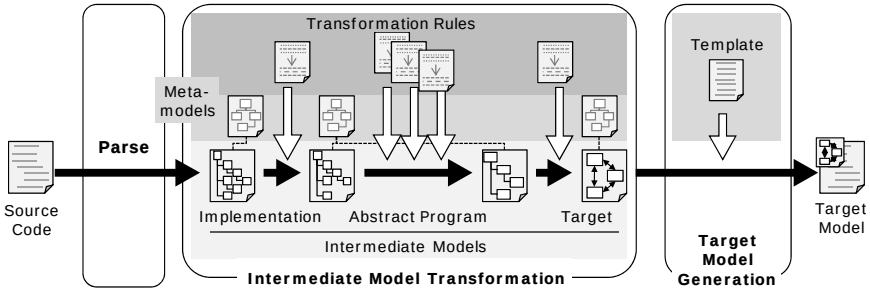

図 2 POM のアーキテクチャ

上記それぞれのサブシステムの詳細を，モデル抽出の流れに従い説明する．

(1) ソースコード解析・実装モデル構築
パーサーがソースコードに対する字句解析および意味解析をおこない，実装モデルを構築する．
実装モデル（Implementation Model）は，名前の参照情報を含む抽象構文木（AST）で構成される．実装モデルは MOF によるメタモデル定義をもつ．

(2) 抽象プログラムモデルへの変換
モデル変換ルール（Transformation Rule）により，実装モデルを，抽象プログラムモデル（Abstract Program Model）へと変換する．意味論を保持するように，要素間のマッピングをおこなう．
抽象プログラムモデルは，実装モデルの意味論と，後述する目的モデル（Target Model）の意味論と，抽象化支援の為の要素を含むように構成される．実装モデルの意味論を包含する一方で，抽象化の変換ルール記述を容易にするため，単純化された構文要素をもつ．

(3) 抽象化
モデル変換ルールによる抽象プログラムモデルの変換（操作）をおこない，下記 2 種類の変換を実現する．
  (a) 意味論の変換: 目的モデルの意味論のうち，実装モデルに含まれないもの，言い換えると，実装モデル上で直接表現されていない情報を抽出し，目的モデル表現へと置き換える．例えば，プロセス間のメッセージ通信を，プログラム中で呼び出す関数名やその引数のパターンから認識し，意味論を置き換える変換が挙げられる．
  (b) 情報の捨象: 目的に対して不要な情報をモデルから除去する．例えば，関数呼出階層を基準に下層部分を簡略化する変換や，特定機能に関する処理を削除する変換が挙げられる．

(4) 目的モデルへの変換
モデル変換ルールにより，抽象化の完了した抽象プログラムモデルを，目的モデル（Target Model）へと変換する．実装モデルから抽象プログラムモデルへの変換と同様，意味論を保持するように，要素間のマッピングをおこなう．
目的モデルは，抽出したいモデル（モデルコード）の表現に十分な意味論を持つように定義される．後述する書出しテンプレートが出力形式に関する責務を持つため，実装モデルと異なり，必ずしも完全な構文情報を持つ必要は無い．一方で，元のソースコードとの対応関係など，目的モデルに対して追加の意味論を持たせることもある．元のソースコードとの対応関係を持たせた場合は，書出し時にコメントを用いて示すことで，抽出したモデルの理解性を向上させることができる．

(5) モデルコード書出し
M2T[7]のテンプレート記述（Template）を用いて，中間モデルである目的モデルを，目的の形式で書き出す．形式記述とグラフ表現のように，異なる形式で出力する場合であっても，モデル抽出の観点が同一であれば，テンプレートのみを用意し，メタモデルや変換ルールなどの変換構成は再利用できる．

3.3 POM アプリケーションの構築

ここまで述べてきた POM のアーキテクチャおよび変換の流れは，抽出するモデルの観点や抽象度に関わらない共通的な枠組みである．意味論の変換や捨象といった抽象化の方法は，抽出するモデルに合わせて設計し，メタモデルやモデル変換ルール述で表現しなければならない．

まず，抽出するモデルの観点に従い，目的モデルのメタモデル設計が必要となる．加えて，実装モデルと目的モデルの意味論の変換のため，抽象プログラムモデルのメタモデル設計と，抽象プログラムモデル・目的モデル間のモデル変換ルールと，抽象プログラム上での意味論変換のモデル変換ルールが必要となる．

また，モデル抽出の目的に従い，抽象プログラムモデル上での抽象化（捨象）のためのモデル変換ルールと，必要に応じて，モデルコード書出の MT2 テンプレートの記述が必要となる．

次節では，モデル検査を目的とした，検証モデル抽出を例として，具体的なメタモデル設計や変換記述を含む POM アプリケーションのシステム実装について述べる．

4. POM/MC の構築

本節では，具体的な POM のアプリケーションとして，モデル検査向けの検証モデルを抽出するシステム POM/MC の構築について述べる．以下，準備としてモデル検査の概要およびその課題を述べた上で，POM/MC の実現について述べる．

4.1 モデル検査

4.1.1 モデル検査の概要

モデル検査とは，検証対象のシステムを仕様記述言語により記述し，その状態空間を網羅的に探索することで，与えた性質（Property）を満たさない実行パスが存在しないかどうかを検証する技術である．モデル検査ツール SPIN[8]は，Promela 言語で記述されたモデルに対して，デッドロックの有無や終了状態が不正でないかどうかを検査する safety property の検査や, LTL 式により与えた時相論理式による検査をおこなう．また，与えた性質を満たさない動作系列が存在するという検査結果が出た場合には，その動作系列を反例として出力する．

4.1.2 不具合解析への応用

我々は，モデル検査をソフトウェア不具合解析への応用を進めている[9]．モデル検査実施の流れを図 3 に示す．不具合が発生したソフトウェアに対し，そのソフトウェアから検証モデル（Promela）を作成し，「その不具合現象が起きないこと」を検証する性質として，SPIN によるモデル検査を行う．この方法により，不具合現象に至る動作系列を検査の反例として得ることが出来る．また，不具合対策を施したソフトウェアに対しても同様にモデル検査を行うことで, 不具合対策が正しく完了したかどうか, 言い換えると，類似不具合の見逃しが存在しないかどうかを検証できる．

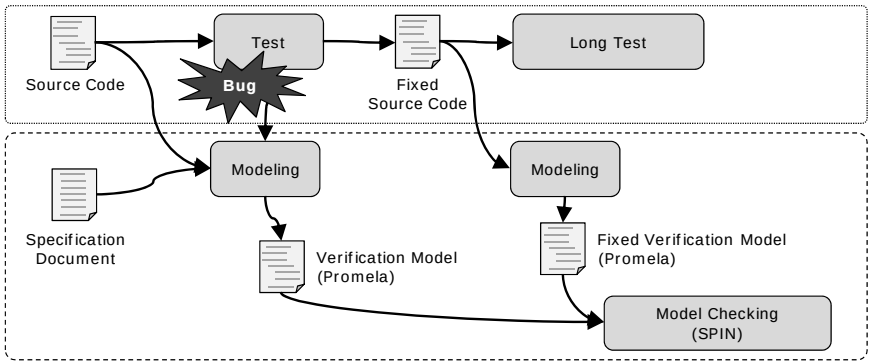

図 3 不具合解析へのモデル検査の応用

[9]では，複数の不具合対策方針をそれぞれモデル化し，検証することで，不具合修正漏れ（類似不具合の発生）を防げる不具合対策を選択でき，モデル検査が製品の品質向上に有効であることが述べられている．

4.1.3 検証モデル作成における課題

一方で, [10]では，次のような，モデル構築に関する課題について述べられている．

(1) 検証モデル設計の難しさ
モデル作成に必要な情報を得るのに，仕様書だけでは詳細が不足する一方，ソースコードだけでは振舞いの解釈が出来ないため，複数の情報源を横断的に活用する必要があった．言い換えると，ソースコードに記された振舞い情報を，ドメイン知識を活用して解釈・抽象化する必要がある．
また，H/W などシステムの外部環境のモデル化や，異常系の扱いなどは，検査の観点に従い検討を行う必要がある．

(2) 状態爆発への対応
システム全体をモデル化すると状態爆発が生じ検査不能となるため，システム分割や構造の縮退を行うことで, モデル規模を抑制する必要がある. 抽象化は，健全な抽象化のみでは不十分であり，振る舞いに影響を与えるアドホックな抽象化[11]が必須である．これらの作業において，注目するシステムの振舞いを誤って変更してしまうと，不正な検査結果が出力されることになる．

この事例報告より，検証モデルの作成においても，抽象化の問題が生じることが分かる．つまり，プログラミング言語とモデル言語との構文や意味論のマッピングを取るだけの単純なリバースエンジニアリングでは，詳細な振舞いの追跡や単純な誤り修正のコストは削減できても，ドメイン知識に基づくモデル設計や，状態爆発対策のための抽象化には対応できない．

4.2 検証モデル抽出のための POM アプリケーション POM/MC

このようなモデル検査の課題に対応できるように，POM の枠組みを用いて検証モデル抽出を実現する．基本的な方針としては，モデル検査向けの検証モデルという観点からメタモデル設計をおこない，外部環境のモデル化や状態爆発対策のための抽象化を，モデル変換ルールにより実現する．

以下，メタモデルおよびモデル変換ルール，書出しテンプレートの構成を，モデル抽出の流れに従い説明する．なお，ここでは，[9]の事例に合わせ，C 言語ソースコードからの Promela モデルの抽出について議論する．

(1) 実装モデルから抽象プログラムモデルへの変換
3.2 節で述べたように，意味論的に等価な変換を行う．

(2) 抽象化/意味論変換
C 言語がライブラリ関数によって実現する，マルチプロセス（スレッド）やメッセージ通信など並行動作に関する機構を，Promela は言語レベルで備えている．

そのため，特定のパターンにマッチする手続き（関数呼出し等）を，Promela の機構に対応付けることで，意味論の変換をおこなう．このとき，異なる C 言語ライブラリへの対応や，他の抽象化変換との連携を容易にするため，proctype 宣言や run 文といった，Promela の構文に直接変換するのではなく，”プロセス生成”や”メッセージ通信”を意味する，抽象プログラムモデルへ上の要素へと変換する．

(3) 抽象化/外部環境のモデル化
外部環境のモデル化は，呼出元のモデルであるドライバの構築と，呼出し先のモデルであるスタブの構築の 2 種類に分類される．
(a) ドライバ: フレームワークなどのソフトウェアの枠組みを提供する部分は，ソースコードから変換すると過剰に複雑なモデルとなりやすく，また，関数ポインタなどの低レベルの機構の利用により，そもそも変換できないことも多い．検査対象のソースコード部位の入り口をエントリポイントとして指定し，フレームワークの設計に従ってエントリポイントを呼び出すモデルを構築するような変換ルールを用意する．例えば，周期起動呼出や，イベント駆動といった必要最低限まで抽象化したドライバを変換ルールにより構築し，元のソースコードのモデルは，エントリポイントから先のみを利用する．
(b) スタブ: ライブラリ関数により提供される機能や，ハードウェア操作の関数呼出しや変数アクセスなどは，ソースコード中に含まれない，もしくは，変換して利用するには過剰に複雑であるため，不具合再現に必要な動作を与えなければならない．そのため，特定の関数呼出や変数アクセスを，対応するライブラリ関数の機能やハードウェア動作を抽象化したモデルへと置き換える．

(4) 抽象化/情報の捨象（状態爆発対策）
不具合動作に関係のない箇所は，単純化された振舞いに置き換える．外界モデルと同様に関数などのまとまりでスタブ化する方法や，不用処理を削除する方法などにより，振舞いを単純化する．

(5) 抽象プログラムモデルから目的モデル（検証モデル）への変換
3.2 節で述べた通り，意味的に等価な変換をおこなう．また，関数など C 言語に存在して Promela に存在しない機構の対応付けをおこなう．また，検証モデル中の各要素に対し，元の実装モデルとの対応付け，すなわち抽出元のソースコードとの対応付けを持たせることで，抽出されたモデルとソースコードとを対応付け可能とする．

(6) Promela コードの書出し
検証モデルから M2T テンプレートにより Promela コードを書き出す. このとき，Promela コードの各行に対し，元のソースコードの位置をコメントで示すことで，モデル検査の結果を元のソースコードに対応付けやすくする．

4.3 POM/MC の実装

試作した POM/MC のシステム構成を図 4 に示す.

4.3.1 基本構成（Modeling Infrastructure）

C 言語のパーサーとして，LLVM/Clang[a]を，解析中の内部 AST を XML として出力する変更を加えて利用する．

モデル変換の基盤として，Eclipse Modeling Framework (EMF)[12]を中心としたモデル変換環境である，Eclipse Modeling Tools (EMT)[13]を用いる．メタモデルは EMF の ecore モデルにより記述し，変換には，QVT/Operational Mappings の EMF 実装である Operational QVT プラグイン[b]を用いる．モデルコードの書出しには，MOF/M2T の実装である Acceleo プラグイン[c]を用いる．

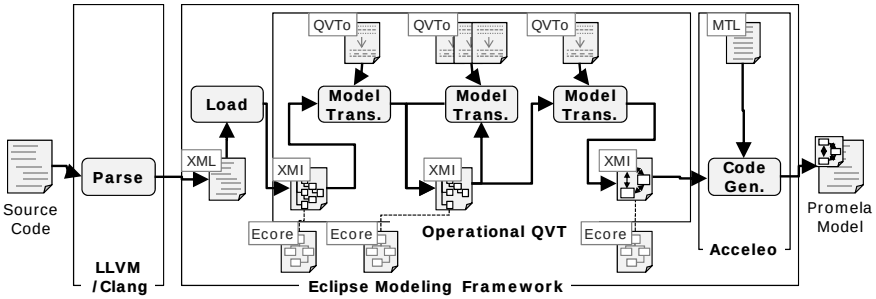

図 4 POM/MC のシステム構成

4.3.2 メタモデル設計とモデル変換ルール（Modeling Configuration）

実装モデルは C99 仕様を，検証モデルおよび M2T テンプレートは SPIN6 の Promela を，それぞれ表現出来る様に構築する．抽象プログラムモデルは，これらのモデルに基づき，4.2 節で述べた通り，意味論変換と状態爆発対策の抽象化をモデル変換ルールとして記述できるように構成する．

実装モデルから抽象プログラムモデルへのモデル間変換では，条件分岐と繰り返しの表現を統一する．そのほかは一対一の構文変換をおこなう．また，抽象プログラムモデルから検証モデルの変換では，構文変換の他，データ型の変換などの単純な意味

a http://llvm.org, 2011/9
b http://www.eclipse.org/m2m/, 2011/9
c http://www.acceleo.org/, 2011/9

論の変換をおこなう．

本試作において作成した，抽象化のモデル変換ルールのうち，代表的なものを表 1 に示す．モデル抽出の時には，利用者が，これらから必要なものを選択すると共に，必要に応じて対象ソースコードに対応したモデル変換ルールを記述する．

表 1 モデル変換ルールの例

記述のタイプ	名称	説明
意味論変換	Pthreads	POSIX スレッドライブラリ（Pthreads）で提供される，スレッド生成や排他制御を，Promela のプロセスやガードの機構へ変換する．
	非決定的選択文	rand()関数を用いて，ランダムな分岐を実現する条件文を，Promela の非決定的選択の機構へ変換する
	式のネスト展開	C 言語では式である代入や関数呼出などは，Promela では文であるため，a=(b=c) → b=c; a=b のように分解する．
	多次元配列の分解	Promela の配列は 1 次元のみのため，多次元配列は，配列を要素としてもつ構造体の配列へと置き換える．
情報の捨象	型宣言の最適化	整数型で宣言された変数および関数について，代入関係から最大値および最小値を得る．実際の宣言よりも値域の小さな型で十分ならば，宣言を置き換える．
	未使用宣言の削除	ドライバから再帰的に利用される可能性のある宣言（変数・関数・型）を求め，利用されない宣言を削除する．外界モデル構築後に適用する．
	関数のスタブ化	パラメータとして関数名を指定することで，容易にスタブへと置き換える．現在は，下記 4 種類のスタブに対応している．(1) 何もしない関数 (2) 定数値を返す関数 (3) 指定範囲の乱数を返す関数 (4) 別途記述したモデルコードを呼び出す関数

5. 適用実験

POM/MC を用い，C 言語ソースコードから，モデル検査可能な Promela モデルを抽出できることを示す．変換対象のプログラムは次の 2 種類である（表 2）．

(1) 哲学者問題
相互排他の例題として知られる「食事する哲学者問題」のプログラムである．哲学者が円状に並び，哲学者の間に 1 本ずつフォークがある状態において，哲学者は，左右のフォークを取り，食事をし，フォークを置く，という作業を繰り返す．ここでは，左右のフォークは，ランダムなタイミングで，ランダムな方を取る動きとする．
結果として，全員が同じ手のフォークを取り，反対側のフォークが開くのを待つ，というデッドロック状態に陥ることを検出する．
C 言語実装にあたり，Pthreads を用いる．哲学者 1 人に対して 1 スレッドを割り当て，フォークの取得を mutex のロック，フォークの解放を mutex のアンロックで表現する．

(2) 座席予約
簡単な座席予約システムの振舞いを模擬したプログラムである．複数台のクライアントが，サーバーに対し，空席問い合わせ・予約実行・発券の一連の手続きを行う．空席問い合わせで対象席が空席でなければ選びなおす．
空席問い合わせと予約実行がアトミック実行では無いために，複数のクライアントに対して同じ席が空席であると応答してしまい，同じ席に対する予約が重複するという問題を検出する．
C 言語実装にあたり，哲学者問題と同様，Pthreads を用いる．クライアント 1 台に対して 1 スレッドを割り当てる．メッセージ通信は利用せず，サーバー機能を提供する関数を直接呼び出す．また，モデル検査時には捨象されるべき，printf によるデバッグコードを含む．

表 2 モデル抽出の概要

名称	ソースコード行数	抽象化構成	Promela 行数	検査内容
哲学者問題	約 100 行	Pthreads，各種構文変換（式のネスト展開など），非決定的選択	約 300 行	Safety Property
座席予約	約 200 行	Pthreads，各種構文変換（式のネスト展開など），未使用宣言の削除，関数のスタブ化	約 400 行	assertion

5.1.1 抽象化の構成

それぞれのソースコードからのモデル抽出にあたり，用いた抽象化を，表 2 の抽象化構成列に列挙する．以下，哲学者問題における抽象化「Pthreads」によるモデル変換を，モデル変換ルールの働きの例として説明する．

哲学者問題のソースコードのうち，抽象化「Pthreads」により影響を受ける部分を抜粋して図 5 の C Source Code の枠内に示す．また，抽象プログラムモデルへと変換されたものを，オブジェクト図で表現したものを，Abstract Program Model の枠内に示す．抽象化「Pthreads」は，関数呼出し（FunctionCall）のうち，参照名（保持する Identifier の name 属性）が ‘pthread_create’ であるものを探し，その参照先（FunctionReference）

を，プロセス生成関数参照（ProcessRunningFunctionReference）に置き換える。また，そのプロセス生成関数参照の属性に, 対象関数として第 3 引数の PhiloThread の宣言 ，プロセスの引数として第 4 引数の philo_id[i]を持たせる。同時に，PhiloThread の関数定義（FunctionDecl）を，プロセス関数定義（ProcessFunctionDecl）で置き換える．最終的に生成される Promela コードを Promela Model Code の枠内に示す．プロセス生成関数参照は run 文，プロセス関数定義は，proctype 宣言として書き出される．

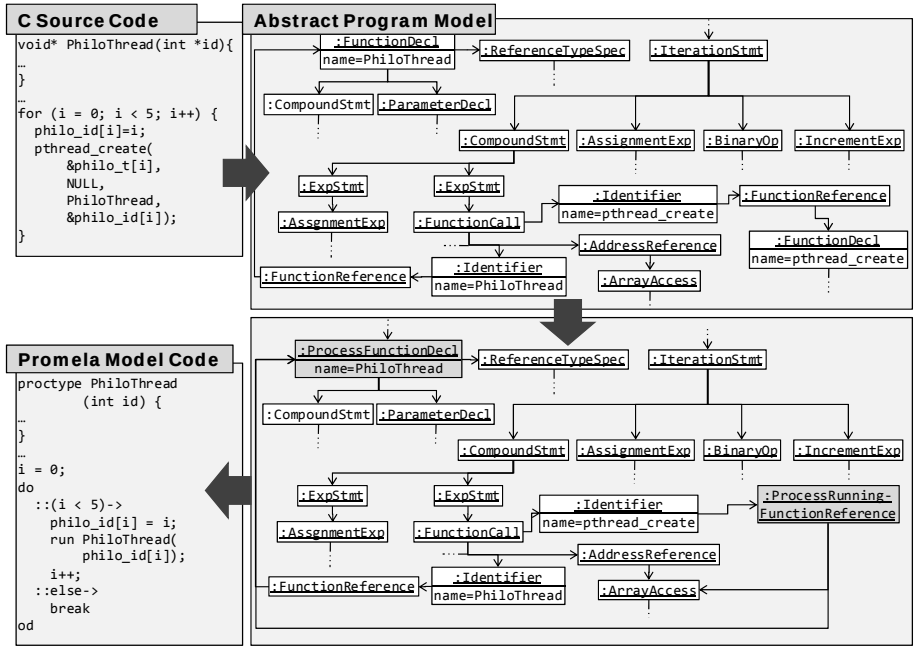

図 6 変換ルールによる抽象化（意味論変換）の例

5.1.2 結果

POM/MC によって抽出された Promela の行数を，表 2 の Promela 行数列に示す．それぞれの Promela コードは，SPIN6.1 を用いて表 2 の検査内容列に示す検査を行い，不具合の検出が可能であることを確認した．

6. 考察

適用実験で用いた C 言語ソースコードは，実製品ソースコードと比較して小規模であるが，リバースエンジニアリングによるモデル抽出をおこなう上で重要な要素を含む．例えば，それぞれのソースコードは Pthreads を用いたスレッド生成や排他制御を含んでおり，宣言と使用箇所など複数のプログラム要素を関連づけながら，Promela の対応する言語要素へと変換する必要がある．POM はこのような変換を QVT 記述によるモデル変換ルールにより実現できており, POM の意味論変換の枠組みの柔軟性を示せた．一方で，本適用実験では，状態爆発対策のアドホックな抽象化はおこなっていない．実製品ソースコードを用いた適用実験により，状態爆発への対応能力の検証を進めることを検討している．

抽象化の記述に関して，抽象プログラムモデルは AST ベースであるため，関数単位での置き換えといった構造ベースの抽象化は比較的容易に記述できる．さらに，本試作にて，いくつかの単純な変換に関してはライブラリ化しており，ほとんど QVT を記述すること無く変換できる．一方で，入力値の値域制限に基づいて関連箇所の分岐を減らすような, 振舞いベースの抽象化を直接的に記述する方法は提供されていない．振舞いベースの抽象化方法を容易に記述可能な，メタモデル設計および QVT ライブラリの構築を進めることで，状態爆発対策を効率的に行えると考えられる．

POM/MC の試作において，本稿の紙面の制約上省略したものも含め，様々な構文および意味論の変換を実装した．一方で，ポインタの解釈など，C 言語ソースコードの”意味”の解析は本質的に困難であり, ケースバイケースでの対応が必要となる. しかし，ポインタのような一般的な解釈が困難な要素も，多くの場合はフレームワーク利用のようなパターンやイディオムであり，ドメイン等を限定すれば，変換ルールによる対応が可能であると考えられる．このような意味論の変換パターンを調査し，洗練された変換ルールを蓄積することで，モデル抽出における障害を減らせると考えられる．

Promela は手続き言語に類似した文法であるため， もとの C 言語ソースコードの構造を維持しながら，比較的素直に変換出来た．しかし SMV [14]など状態遷移を直接記述するタイプの検証モデルや, UML の状態マシン図などの振舞いモデルの抽出を実現するためには，データフローや制御フローなど，現在の POM/MC で取得していない情報を用いた意味論変換が必要となる．これらのモデル抽出に対応するためには，より高度な解析の利用も視野に入れた設計検討が必要であると考えられる．

UML のクラス図や関数コールツリーなどの構造モデルの抽出は，現状の POM のアーキテクチャ上で容易に実現可能である．しかし，理解支援など，どのレベルの情報を，どのように見せるかという可視化方法が課題となる．複数種類のモデルを関連付け方法などを含めた，具体的なユースケースに基づく，より大きな枠組みでのツール構築を検討している．

7. 関連研究

UML ドローツールなど MDE ツールのうち高機能なものは，ソースコードからのリバースエンジニアリング機能をもつものがある．しかし，このようなツールにより抽出されるモデルは，実装に忠実な構造モデルのみであることが多く，利用者による抽象化の制御は提供されていない．

Fujaba [3] は，Java のソースコードを解析し，クラス図などの UML 図を抽出するツールである．Fujava は，クラス設計中でのデザインパターン適用箇所を検出することで，プログラム理解を支援する．このようなデザインパターン検出は，モデルの中のある要素集合に対して意味を与える操作であり，本稿で述べた意味論変換の抽象化の一種であると考えることができる．

Modex (FeaVer) [15] は，モデル検査向けむけに Promela モデルを C 言語ソースコードから抽出するツールである．Modex は，特定のフレームワーク上のソースコードから，特定の抽象度でモデル抽出を行う．一方で，POM/MC は，変換ルールによってフレームワークのモデリングや，抽象度の操作が可能であり，より柔軟なモデル抽出を行うことができる．

モデルの抽象化に関する研究として，Androutsopoulos らは，状態遷移モデルの抽象化手法を提案している [16]．この手法では，状態遷移モデルにおいて，特定の状態やイベントが使用されないときに不要となる状態を削除する．POM は，ソースコードからのモデル抽出と，抽象化の観点の指定，いいかえると，「どの状態やイベントが使用されないか」の指定のための手段を提供することが特長であるため，Androutsopoulos らの手法は POM と相補的な手法であるといえる．また，市井ら [17] は，ソフトウェア部品の再利用性を計測する Component Rank 法を用いて，クラス図のなかから重要なクラスのみを抽出する手法を提案している．市井らの手法は利用者が閾値を与える必要があり，一種のパッケージ化された抽象化であると捉えることができる．

8. おわりに

本稿では，ソースコードからのモデル自動抽出手法 POM を提案した．提案手法では，メタモデルを用いたモデル変換技術を応用することで，利用者が，抽象化の操作をモデル変換言語 QVT によるモデル変換ルールにより与える手段を提供する．

また，モデル検査向け検証モデルを抽出する POM/ MC を試作した．POM/ MC は，C 言語ソースコードから Promela モデルを抽出する．さらに，試作した POM/ MC を用いて，C 言語ソースコードから，モデル検査可能な Promela モデルを抽出出来ることを示した．

今後の課題として，まず，実際の製品ソースコードを用いた実ユースケースでのフィージビリティスタディが挙げられる．実装に忠実なモデル抽出では状態爆発がおきるような規模のソースコードから，適切な抽象化を行い検査可能なモデルを抽出できるかどうか検証する．また，理解支援などの検証以外のユースケースにおいても，POM のアプローチが有効であるかどうかを検証するため，検証モデル以外のモデル抽出を実現する POM アプリケーションを開発する．

参考文献

1) D. S. Frankel, “MDA モデル駆動アーキテクチャ”，エスアイビー・アクセス, 2003
2) S. J. Mellor and M. J. Balcer, “Executable UML”，翔泳社, 2003
3) M. von Detten, M. Meyer, D. Travkin, “Reverse Engineering with the Reclipse Tool Suite”, Proc. the 32nd ACM/IEEE International Conference on Software Engineering(ICSE 2010), pp. 299—300, 2010
4) G. Booch, J. Rumbaugh, I. Jacobson, “UML ユーザガイド第 2 版”，ピアソン・エデュケーション, 1999
5) The Object Management Group, "Meta Object Facility (MOF) Core Specification", formal/06-01-01, http://www.omg.org/spec/MOF/2.0/PDF, 2006
6) The Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification", formal/2008-04-03, http://www.omg.org/spec/QVT/1.0/PDF, 2008
7) The Object Management Group, "MOF Model to Text Transformation Language, v1.0", formal/2008-01-16, http://www.omg.org/spec/MOFM2T/1.0/PDF, 2008
8) Gerard J. Holzmann, "The SPIN Model Checker: Primer and Reference Manual", Addison-Wesley Professional, 2003
9) 中川, 明神, 小川, "組込みシステムへのモデル検査の適用", 第 73 回全国大会講演論文集, pp. 257−259, 2011
10) 中川, 小川, "組込みシステム向けモデル検査適用の課題", 組込みシステム技術に関するサマーワークショップ（SWEST13）,pp.94−95, 2011
11) 中島, “SPIN モデル検査”，近代科学社, 2008
12) D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, “EMF Eclipse Modeling Framework Second Edition”, Addison-Wesley, 2008
13) R. C. Gronback, “eclipse Modeling Project”, Addison-Wesley, 2009
14) J. R. Burch, E. M. Clarke, K. L. McMillan, "Symbolic Model Checking: 10^{20} states and beyond", Information and Computation, 98:142—170, 1992
15) G. J. Holzmann and M. H. Smith, "An Automated Verification Method for Distributed Systems Software Based on Model Extraction", IEEE Trans. Software Engineering, Vol. 28, No. 4, pp. 364—377, 2002
16) K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, Z. Li, “Model projection: simplifying models in response to restricting the environment”, Proc. the 33rd international conference on Software engineering (ICSE’11), pp. 291—300, 2011
17) 市井, 横森, 松下, 井上, "コンポーネントランクを用いたソフトウェアのクラス設計に関する分析手法の提案", 信学技報, SS2005-37, Vol.105, No.229, pp.25-30, 2005