モデル検査環境とプログラミング環境のラウンドトリップに向けて

田辺 誠*

Towards a Seamless Round-Trip Development Environment between Model Checking and Programming

Makoto TANABE*

Key words: Model Checking, Model Driven Architecture, UML, UPPAAL, JML

1 はじめに

モデル検査は、システムの実装に先立って抽象的な設計図（モデル）を作成し、その妥当性を網羅的かつ厳密に検証する手法であり、ハードウェア設計やプロトコル設計の分野で近年大きく成功をおさめている。しかし、モデルとプログラムとの間には抽象度の隔たりがあり、モデルからプログラムを自動的に作成したり、プログラムからモデルを抽出したりすることは難しい。このため、開発現場にはなかなか手軽に受け入れられないのが現状である。

一方、モデル駆動開発とは、特定のプログラミング環境に依存せず、UMLなどの技法を使ってシステムの機能をモデル化する開発スタイルである。

モデル検査とモデル駆動開発はどちらも抽象的なモデルを扱う点で共通しているが、モデルの抽象度が異なる。前者はより抽象度が高いため、妥当性の網羅的な検証は可能であるが、プログラムの自動生成は困難である。逆に、後者はプログラムの自動生成は行ないやすいが、妥当性の検証は困難である。両者の長所を併せ持つ開発手法の確立が求められている。

本論文では、モデル検査技術とモデル駆動開発手法とを融合することを目指し、モデル検査で扱うモデルと、モデル駆動開発で扱うモデルとの間のラウンドトリップ環境を提案する。

本論文の構成は以下の通りである。まず、第2節では、既存の手法であるJava Modeling LanguageによるJavaプログラムの仕様記述法を紹介する。第3節では、モデル検査ツールUPPAALによって作成されたモデリングデータからJava Modeling Languageへの変換システムを先行研究として紹介し、このシステムによる設計手法の課題を述べる。この課題を解決するための手法として、モデル検査環境とモデル駆動開発環境（UML記述環境）を提案し（第4節）、結論を述べる（第5節）。

2 Java Modeling Language (JML) による仕様記述

本節では、Java Modeling Language (JML) によるJavaプログラムの仕様記述法を紹介し、その長所と短所についての考察を行なう。

(2005年11月24日受理)

* 宇部工業高等専門学校制御情報工学科

2.1 JML

Java Modeling Language (JML) は、DBC (Design by Contract: 契約による設計) パラダイムに基づき Java プログラムの仕様記述を行なうための言語である。契約による設計とは、プログラム設計時に、プログラムの呼び出し側が満たすべき条件とプログラムの提供側が満たすべき条件を両者間の「契約」として明示的に記述することにより、両者の責任分担を明確にするものである。例えば、平方根を越えない最小の整数を求める関数 `mySquare` のコーディングを行なうに先立ち、

- 関数の利用者は、引数として非負の整数値を与えなければならない (関数呼び出し側の責任)
- 非負の整数値が与えられた場合、「返り値$^2 \leq$ 引数 $<$ 返り値$^2+1$」でなければならない (関数提供側の責任)

のような「契約」をかわすことにより、関数の提供側は引数が負値であった場合の例外処理を行なう必要が無いことを明確にすることができる。

JML は、JAVA のメソッド呼び出し側とメソッド提供側の契約をプログラムのコメント内に記述する言語である。平方根を計算するメソッド `mySqrt` の JML 記述例を示す。

```
//@ requires x >= 0;
//@ ensures \result * \result <= y
//@     && y < (\result+1)*(\result+1);
protected static int mySqrt(double x) {
 // 平方根の計算をして、変数 r に代入
 return r;
}
```

一行目の「`requires x >= 0`」は、メソッド呼び出し側が `mySqrt` の引数を非負の値にする責任があることを示し、二行目の「`ensures...`」は、メソッド提供側が `mySqrt` の返り値を適切な値にする必要があることを示す。

これらの記述の先頭には「`//@`」がつけられているため JAVA コンパイラではコメントとして扱われるが、JML のコンパイラはこれらの記述を解釈し、契約の履行状態を実行時にチェックするためのコードを挿入する。

2.2 JML による大域的仕様記述の問題点

JML による契約の記述は、個々のメソッドの入出力関係を規定するものであり、プログラムの実行全体から見ると局所的な性質である。一方、メソッドの呼び出し順や、プログラム実行中の不変量などの大域的な性質も表現できる (後述するように、JML によるこれらの大域的な性質の記述は可読性・保守性の点で欠点があると筆者は考える)。

例として、JavaCard 上のデータ送受信プロトコル APDU (Application Protocol Data Unit) の仕様記述を考える。APDU プロトコルによるメソッド呼び出しは図 1 の状態遷移図に沿った順序で行なわれる。大域的な状態を 7 つ持ち、各

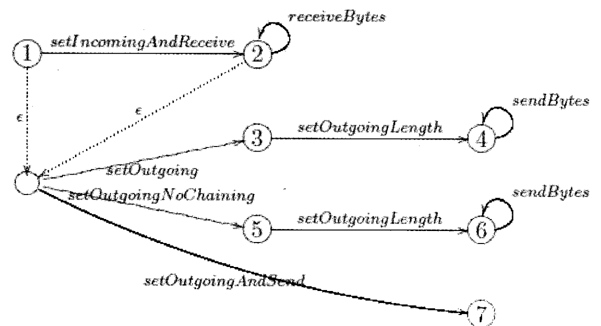


図 1: APDU プロトコルのメソッド呼び出し順

状態において実行可能なメソッドが定まっている。関連研究 [1] ではこのプロトコルの仕様記述を JML を用いて行なっている。例えば、メソッド `setIncomingAndReceive` は以下のように記述されている。

```
//@ public model int _APDU_state;
/*@ public_behavior
  @   requires _APDU_state == 1 && ...:
  @   ensures _APDU_state == 2 && ...;
  @*/
public short setIncomingAndReceive();
```

大域的な状態を表す JML の変数「`_APDU_state`」を用いて、

- `setIncomingAndReceive` の呼び出し時には状態「1」でなければならない。
- `setIncomingAndReceive` の終了後には状態

「2」となる。

ことを記述したものである。

ここで、図1の状態遷移図を基にして手作業でJMLを記述する場合、いくつかの問題点が生じる。JML記述時に状態変数の値を書き間違えた場合(例えば上例で「`requires _APDU_state == i`」と書いてしまい、かつ変数iが存在しているためコンパイルエラーが出ない場合など)、これが原因となるエラーのデバッグを行なうことは困難である（可読性に欠ける）。また、状態遷移図を変更した場合のJML記述の保守も困難である（保守性に欠ける）。

状態遷移図からJML記述を自動生成することができれば、大域的な性質を直接JMLで記述する必要がなくなり、可読性や保守性を上げることができる。そこで、本研究では、Uppaalで記述された状態遷移図からJML記述を自動生成するツールを整備した。次章に詳細を述べる。

3　モデル検査によるモデリングデータからJMLへの自動変換

前節で述べたJMLの弱点を補うため、[5]ではモデル検査UPPAALによるモデリングデータからJML記述(JAVAプログラムのスケルトン）への自動変換システムを提案した。本節ではこのシステムについて説明する。システム構成図を図2に示す。

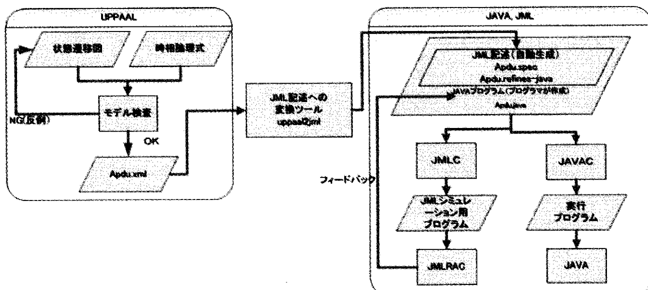
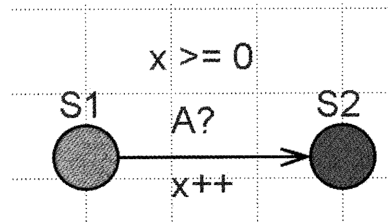


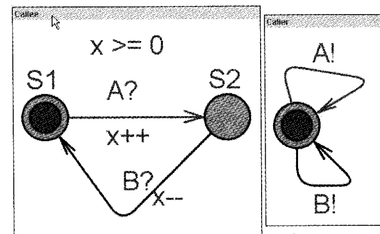
図2: UPPAALからJMLへの変換システム

本システムを用いた仕様記述手順は以下の通りである。

ステップ1. UPPAALを用いた大域的なシステム設計　メソッドの呼び出し順などの大域的な設計を、Uppaalの状態遷移図を用いて行なう。例えば、メソッドAの呼び出し時に要求する条件が「大域的な状態が$S1$であり、かつxが非負であること」であり、呼び出し終了後に保証する条件が「大域的な状態が$S2$であり、かつxの値が1つ増えていること」であるようなメソッドAを以下の状態遷移で記述する。



ステップ2. UPPAALを用いた検証　作成された状態遷移図に対する検証を行なう。例えば、呼び出し時に要求する条件が「大域的な状態が$S2$であること」であり、呼び出し終了後に保証する条件が「大域的な状態が$S1$であり、かつxの値が1つ減ってくること」であるようなメソッドBを付け加え、初期状態が$S1$であるとする(callee)。このとき、メソッドの呼び出し側のプロセス(caller)も含めた全体の状態遷移図は以下のようになる。



これらの条件のもとでxの初期値を1としてプログラムを実行すると、xの値は1と2の値しか取らない。論理式`A[](x == 1 || x == 2)`をUPPAALでモデル検査することによりこの性質が成立することを検証することができる。

ステップ3. upp2jml: UPPAALの状態遷移図からJML記述への変換　本研究では、UPPAALの状態遷移図からJML記述を自動生成するツール`upp2jml`を構築した。これを用いてJML記述への変換を行なう。メソッドAはこのツールにより以下のJML記述に変換される。

```
//@ public model int _state;
/*@ public_behavior
  @    requires _state == S1 && x >= 0:
  @    ensures _state == S2
  @              && x == \old(x)+1;
  @*/
public int A(int x) {
}
```

変換ツール upp2jml は, JavaCC[4] を用いて JAVA で実装した。このツールは

java upp2jml ファイル名 クラス名

のように使用する。ファイル名には Uppaal の状態遷移図の保存ファイル (XML 形式) を、クラス名にはメソッドの属するクラス名をそれぞれ指定すると、JML のファイルが出力される。

ステップ **4.** メソッドの実装 メソッドの実装を行なう。実装と JML 記述との整合性のチェックは、JML コンパイラ (jmlc) および JML 実行時チェッカ (jmlrac) を用いて行なう。jmlrac によってエラーが発見されると、その責任がメソッドの呼び出し側にあるか実装側にあるかが明確に示されるため、デバッグが容易である。

3.1 本システムの問題点

本手法をタンクの制御システム [3] に適用し、状態遷移図を用いたシステム設計に本手法が有効であることを確認した。しかしながら現時点ではいくつかの制約がある。

事後条件の記述 UPPAAL の遷移では、遷移の前提となる条件（メソッド呼び出し側に要求する条件）は記述できるが、遷移後の状況（メソッド実装側が保証する条件）に関しては明示的な代入しか許されず、条件式で抽象的に記述することがでない。UPPAAL の方が抽象度が上位であるにもかかわらず、事後条件に関してはより具体化したものを書く必要がある。これは、UPPAAL の記述力からくる問題である。

並列システムの記述 現行の JML にはマルチスレッドの性質を記述する枠組みがないため、UPPAAL の売りの一つである、並列システムの動作検証の結果を生かすことができない。これに関しては、JML をマルチスレッドの記述に拡張する [2] などの研究を参考にして対処したいと考える。

4 モデル検査環境とプログラミング環境とのラウンドトリップに向けて

前節では、モデル検査によるモデリングデータから JML への自動変換システム ([5]) を紹介した。これを用いると、プログラムの限定された性質については、モデル検査によって検証を行い、正しさの保証されたモデルよりプログラムの一部分を自動作成することができる。しかし、モデル検査環境とプログラミング環境の抽象度に大きな隔たりがあるため、検証できる性質がかなり限られる。

そこで、モデル検査環境とプログラミング環境を UML ツールによって媒介する設計手法を図 3 に示す。

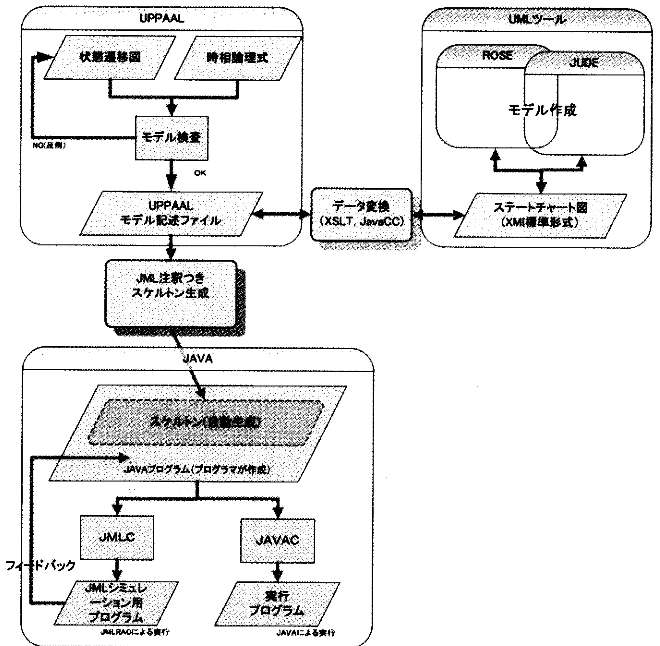


図 3: モデル検査とプログラミングとのラウンドトリップ環境

本手法を用いた設計手順を示す。

ステップ 1: モデル検査ツールを用いた仕様記述・検証　モデル検査ツール（図3ではUPPAAL）を用いて、抽象度の高いレベルでの設計（モデルの作成）を行う。作成されたモデルに対する仕様検証を行ない、このレベルでの設計に不整合がないことを確認する。

ステップ 2: UML への変換　ステップ1によって作成されたモデルをUMLに変換する。UMLにはXMIによる標準データ形式が定められているので、XML変換技術等を用い、この標準データ形式に変換する。

ステップ 3: UML の詳細化　ステップ2によって作成されたUMLデータを基に、UMLツールを用いてさらに詳細なUML記述を行なう。

ステップ 4: モデル検査環境と UML 記述環境のラウンドトリップ　UMLデータからモデル検査のモデルへの逆変換により、UMLツール上での仕様変更に対応する。これにより、モデル検査環境とUML記述環境を行き来しつつ設計を行なうこと（ラウンドトリップ設計）が可能となる。

ステップ 5: UML 記述環境とプログラミング環境のラウンドトリップ　UMLからプログラムスケルトンの作成機能や、プログラムからUML記述を抽出するリバースエンジニアリング機能が各種UMLツールによって提供されている。この機能を使うことにより、UML記述とプログラミング環境とを行き来しつつ設計を行なうこと（ラウンドトリップ開発）が可能となる。ステップ4と併用することにより、モデル検査環境とプログラミング環境とのラウンドトリップも可能となる。

5　おわりに

本論文では、UMLデータを媒介とした、モデル検査環境とプログラミング環境との融合手法を提案した。これは、先行研究[5]で提案した、モデル検査環境からプログラムスケルトンの自動作成技術の弱点を補うものである。現在、モデル検査のモデルデータとUMLデータとの間の相互変換システムを作成中である。

参考文献

[1] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[2] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In Andrew P. Black, editor, *ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, volume 3586, pages 551–576. Springer-Verlag, Berlin, July 2005.

[3] 結城浩. キーワードで学ぶオブジェクト指向プログラミング入門 – 状態遷移. *JAVA Developer*, (7), 2004.

[4] 五月女健治. *JavaCC* – **コンパイラ・コンパイラ** *for Java*. テクノプレス, 2003.

[5] 田辺 誠. モデル検査ツール uppaal による java 仕様記述支援. In **第二回システム検証の科学技術シンポジウム**, pages 177–181, 10月 2005.