C-006

実行可能UMLにおけるアサーションベースの動的検証
Assertion-Based Dynamic Verification on Executable UML

菅井 雅仁† 松本 倫子† 吉田 紀彦†
Masahito Sugai Noriko Matsumoto Norihiko Yoshida

1. はじめに

近年，組込みシステムの分野では，システムが大規模化・複雑化してきている．このため，検証の重要性が増すと同時に検証の工数が増加してきており，検証の効率化が課題となっている．このような背景に加えて，検証の効率化が図れることやシステムの信頼性を確保できるという理由から，アサーションベース検証 [1] が注目されている．これは，検証対象が満たすべき性質（プロパティ）をあらかじめ宣言し，その宣言（アサーション）に検証対象が違反しないかどうかを検証するものである．

一方で，設計においてはUMLモデルを実行することができる実行可能UML[2] が用いられ始めている．これはモデルの動的検証が行えるという利点が背景にある．しかし近い将来，実行可能UMLにおいても検証の効率化が課題になると考えられる．

実行可能UMLにおける検証の効率化を図る方法としては，前述のアサーションベース検証を導入するという手法が考えられる．実行可能UMLについてのアサーションベース検証としては，既に静的検証の研究事例 [3] があるが，動的検証の研究事例はまだ存在しない．そこで，本研究では実行可能UMLにおけるアサーションベースの動的検証の実現に取り組んだ．具体的には，実行可能UMLにおけるアサーションの記述方式を定め，実行可能UMLにおいてアサーションベースの動的検証を実現する方法を考案し，実現できたことを実験により確かめた．

2. 実行可能UML

実行可能UMLとは，UMLダイアグラムを用いて作成されたモデルを実行できるようにしたUMLの拡張言語であり，ゆえに実行可能UMLを用いて設計したシステムは，動的検証が可能である．

実行可能UMLでシステムのモデルを作成する場合，システムをドメインに分類する．ここでドメインとは，ある特徴的な規則と方針に従い振舞うクラスから形成される問題領域を指す．

モデルの実行に必要なダイアグラムはクラス図とステートチャート図である．ステートチャート図の各状態は，アクション群により構成されるプロシージャを持つ．各アクションは，データへのアクセスやループといった計算処理の基本単位である．状態遷移は，アクションにより発生させることができるイベントをクラスのインスタンスが受け取ることで起こる．

アクションはアクション言語を用いて記述される．現在，標準となるアクション言語は存在しないが，その意味論はOMGにより策定されたUML Action Semantics[4] に準拠している．

本研究では，実行可能UMLツールとしてiUML[5] を使用する．iUMLではアクション言語としてASL（Action Specification Language）[6] を採用している．

3. アサーションベース検証

アサーションはプロパティ記述言語を用いて記述され，アサーションにより宣言されるプロパティは，一般的に古典論理や時相論理に基づく記述となる．プロパティ記述言語としては以下のようなものがある．

- PSL（Property Specification Language）[7]
- OVL（Open Verification Library）
- e Temporal Language
- OVA（OpenVera Assertion）
- SVA（SystemVerilog Assertion）

本研究では最も標準的なプロパティ記述言語であるPSLを基にアサーションの記述方法を定義する．

アサーションベース検証では，プロパティが満たされなかった場合，エラーとしてその情報を出力させることができる．そのため，誤り箇所を特定しやすくなり，検証の効率化が図れる．

アサーションベース検証は静的検証（モデルチェッキング）と動的検証に大別される．前者は，到達可能な全ての状態でプロパティが満たされるかどうかをチェックするため，検証に漏れが発生しないという利点があるが，到達可能な状態数が爆発し検証不能となることがある．一方で，後者はシミュレーションによりプロパティが満たされるかどうかをチェックするため，検証不能となることはないが，検証に漏れが発生することがある．

4. 関連研究

Xieら [3] は，アサーションベース検証による実行可能UMLの検証を試みているが，実行可能UMLにおけるアサーションベースの静的検証の実現を目標としている点で本研究とは異なる．Xieらは，独自のプロパティ記述言語（xPSL）を定義し，xPSLによってアサーションを記述している．このxPSLは，実行可能UMLで用いる変数，イベント，状態などを扱えるようにした，PSLの拡張言語である．また，既存のモデルチェッカーを用いるために，実行可能UMLで作成したモデルとアサーションをモデルチェッカーが扱える形に変換し，変換後のモデルとアサーションをモデルチェッカーに入力してアサーションベースの静的検証を行う．

5. アサーションの記述方式

5.1 構文

BNFで表したアサーションの構文を図1に示す．構文はPSLを参考にして定義した．書体が太字になっている単語は予約語や演算子である．“Condition”は，“Component”の“Event_Specification”を除けば，ASLのif

†埼玉大学 Saitama University

```
Assert_Statement ::= assert Property
Property ::=  Condition
            | ( Property )
            | Property -> Property
            | Property <-> Property
            | always Property
            | eventually! Property
Condition ::=
    Component
  | Component Binary_Logical_Operateor Component
  | ! Component
Component ::=  Instance_Handle.Attribute
             | Local_Variable
             | Constant
             | countof { Instance_Handle_Set }
             | countof Class
             | Event_Specification
```

図 1: アサーションの構文

文の条件文の構文に従うものとした．また，イベントの指定に用いる “Event_Specification” は ASL の構文に従うものとした [8]．

5.2 記述方法

アサーションの記述場所は，プロパティが古典論理式か時相論理式かにより異なる．プロパティが古典論理式のアサーションは，ステートチャート図の状態のプロシージャの中に 1 行のコメントとして記述するものとした．これは，アサーションが記述されている場所に到達した時点で，そのアサーションに違反しないかをチェックするということを意味する．コメントとしたのは，本来の処理とアサーションを区別しやすくし，アサーション挿入による本来の処理の可読性悪化を抑制するためである．以下に，プロパティが古典論理式であるアサーションの記述例を示す．なお，“#” は ASL において行コメントを表す．

- # assert this.val1 > 0 & this.val2 = 0
- # assert !x -> y

プロパティが時相論理式のアサーションは，“class_クラス略称.ast”，“domain.ast” というファイルに記述するものとした．前者は 1 つのクラスに関わるアサーションを記述し，後者は 1 つのドメインに属する複数のクラスに関わるアサーションを記述するものである．なお，この 2 種類のファイルには，アサーション以外のアクションを記述することも許す．これは，アサーション中のプロパティの記述で用いる “Instance_Handle” や “Instance_Handle_Set” にどのようなインスタンスを保持するかを指定できるようにするためである．以下に，プロパティが時相論理式であるアサーションの記述例を示す．

- assert always (this.flag1 | !this.flag2)
- assert eventually! this.val1 = 0

6. 実装方法

6.1 基本方針

本研究では，モデルの実行中にプロパティが満たされるかをチェックし，満たされない場合はエラーメッセージを出力するという機能を実装するために，条件判定を用いた．すなわち，アサーションから条件判定を生成し，生成した条件判定を適切な場所に挿入することで実装を行った．生成する条件判定，及び条件判定を挿入する場所については次小節以降で説明する．

条件判定により実装が行えることを確かめるために，iUML 上で試験的な実装を試みた．iUML では，実行ファイルを生成するために Write 処理と Build 処理を行う．Write 処理を行うと，実行ファイル生成に必要なファイルが生成される．以下では，この Write 処理により生成されるファイルを「Write ファイル」と呼ぶことにする．Write ファイルには，クラスに関する情報が記述されたファイル，状態のプロシージャが ASL で記述されているファイル（al ファイル）などがある．Write 処理が終わると Build 処理を行う．Build 処理では，Write ファイルを基にプログラミング言語 C のコードが記述されたファイル（以下，C ファイルと呼ぶ）を生成し，その C ファイルから実行ファイルを生成する．そこで，本研究の実装では以下の処理を行う．

1. al ファイルや ast ファイルからアサーションを読み込む．
2. 読み込んだアサーションから ASL の構文に基づく条件判定を生成する．
3. 生成した条件判定を al ファイルに挿入する．
4. Build 処理を行い実行ファイルを得る．

上記の手順 1〜3 を行うために，オブジェクト指向スクリプト言語 Ruby を用いて処理系を作り使用した．処理系は，Build 処理に対してプリコンパイルを行うものである．

6.2 古典論理

古典論理プロパティは，アサーションが記述されている場所に処理が到達したときに満たされるかどうかをチェックすればよい．したがって，条件判定を挿入する場所は，アサーションが記述されている行の次の行とした．

図 2 にアサーションとアサーションから生成するコードの例を示す．「挿入コード」の 2〜6 行目が条件判定本体である．条件判定の条件文は基本的にプロパティの論理否定をとったものとし，条件に合致した場合の処理ではエラーメッセージを出力する．ただし，図 2 のようにプロパティにイベントの指定がある場合は，イベントの発生を示すフラグによりイベントの指定の部分を置き換える．このフラグは，モデルの初期化処理で FALSE に初期化し，そのイベントを発生させるアクションの直後で TRUE にし，条件判定の直後で FALSE にするようにする．また，ASL では論理含意 “->” や論理同値 “<->”

アサーション

```
1  # assert SC1:event1 & x
```

⇓ コード生成

挿入コード

```
1  ef = find-only EventFlag
2  if !(ef.flag1 & x) then
3  $INLINE
4  fprintf(stderr, "Assertion error:
   \"SC1:event1 & x\" failed
   at domain: %s, file: %s, line: 1\n",
   CGEN_DOMAINNAME, CGEN_SOURCENAME);
5  $ENDINLINE
6  endif
7  ef.flag1 = FALSE
```

図 2: 挿入コードの生成例

といった演算子はサポートしていないので，“x -> y” や “x <-> y” のようなプロパティは，一度それぞれ等価な論理式 “!x | y”，“(x & y) | (!x & !y)” に変換する．なお，ASL において “!”，“|”，“&” はそれぞれ論理否定，論理和，論理積を表す．

ASL では標準出力などに出力するアクションは用意されていない．しかし，ASL では記述したコードがそのまま C ファイルに挿入されるインライン文が用意されているため，エラーメッセージはインライン文において C 言語の fprintf 関数を使用し出力するようにした．なお，図 2 の fprintf 関数の引数 CGEN_DOMAINNAME, CGEN_SOURCENAME は，生成された C 言語のコードにおいてそれぞれドメインの略称，プロシージャが記述されているファイル名を示すようになっており，そのファイル名からは，どのクラスのどの状態のプロシージャであるかが分かるようになっている．したがって，エラーメッセージからプロパティが満たされなかったドメイン，クラス，状態が判断できる．

6.3 時相論理

本論文の段階では，古典論理式に時相論理演算子が 1 つだけ付加された時相論理プロパティのチェックの実装方法についてのみ考案できているので，それについて説明する．

6.3.1 always

本小節では，「ある古典論理式が常に成立する」という時相論理プロパティのチェックの実装方法について説明する．図 3 に ast ファイルに記述されているアサーションとそのアサーションから生成するコードの例を示す．

ある古典論理式が常に成立することをチェックするには，その古典論理式が全ての状態で成立することをチェックする必要がある．また，古典論理式がある状態で成立しなかったら，それ以降のチェックは必要なくなる．そこで，TRUE で初期化したフラグを用意し，フラグを用いた条件判定を全ての状態に挿入する．条件判定の条件文はフラグと論理否定をとった古典論理式との論理積とし，条件に合致した場合の処理ではフラグを FALSE にする．

なお，図 3 の「全ての状態に挿入するコード」は，ast ファイルがある 1 つのクラスに関するアサーションが記述されているファイルである場合，そのクラスの全ての状態に挿入する．一方 ast ファイルが，ある 1 つのドメインに属する複数のクラスに関わるアサーションを記述するファイルである場合，そのドメインに属する全クラスの全ての状態に挿入する．

ast ファイル（class_SC.ast）

```
1  assert always this.val != 0
```

⇓ コード生成

全ての状態に挿入するコード

```
1  af = find-only AssertionFlag
2  if af.ok1 & !(this.val != 0) then
3  $INLINE
4  fprintf(stderr, "Assertion error:
   \"always this.val != 0\"
   (domain: SD, file: class_SC.ast, line: 1)
   failed at domain: %s, file: %s\n",
   CGEN_DOMAINNAME, CGEN_SOURCENAME);
5  $ENDINLINE
6  af.ok1 = FALSE
7  endif
```

図 3: always から生成するコードの例

6.3.2 eventually!

本小節では，「ある古典論理式がいつかは成立する」という時相論理プロパティのチェックの実装方法について説明する．図 4 に ast ファイルに記述されているアサーションとそのアサーションから生成する 2 つのコードの例を示す．

ある古典論理式がいつかは成立することをチェックするには，その古典論理式が成立するかどうかを全ての状態でチェックしなければならない．そして，実行開始から状態遷移の最後の状態のプロシージャが終わるまでの間に古典論理式が一度も成立しなかったら，エラーメッセージを出力しなければならない．そこで，FALSE で初期化したフラグを用意し，全ての状態と状態遷移の最後の状態にフラグを用いた条件判定を挿入する．全ての状態に挿入する条件判定は，条件文を論理否定をとったフラグと古典論理式の論理積とし，条件に合致した場合の処理ではフラグを TRUE にする．状態遷移の最後の状態に挿入する条件判定は，条件文を論理否定をとったフラグとし，条件に合致した場合の処理ではエラーメッセージを出力する．

ast ファイル（class_SC.ast）

```
1  assert eventually! this.val = 1
```

⇓ コード生成

全ての状態に挿入するコード

```
1  af = find-only AssertionFlag
2  if !af.ok1 & (this.val = 1) then
3  af.ok1 = TRUE
4  endif
```

最後の状態に挿入するコード

```
1  af = find-only AssertionFlag
2  if !af.ok1 then
3  $INLINE
4  fprintf(stderr, "Assertion error:
   \"eventually! this.val = 1\"
   (domain: SD, file: class_SC.ast, line: 1)
   failed at domain: %s, file: %s\n",
   CGEN_DOMAINNAME, CGEN_SOURCENAME);
5  $ENDINLINE
6  endif
```

図 4: eventually!から生成するコードの例

7. 例題実験

実装が正しいかどうかを確かめるために，iUML で作成した自動販売機のモデルにアサーションを付加して実験を行った．図 5 に自動販売機のモデルのクラス図，図 6 に Drink_Slot クラスのステートチャート図を示す．このステートチャート図では，以下のアサーションに対して違反を起こさせるために，状態 **2**，**3** で図に示すように処理の一部を意図的にコメントアウトしている．

a. # assert this.purchasability_lamp

b. assert always (this.stock = 0 -> DS5:sellout)

c. assert eventually! this.purchasability_lamp

アサーション a は Drink_Slot クラスの状態 **3** のプロシージャの最初に挿入したもので，アサーション **b**，**c** は Drink_Slot クラスの ast ファイル（class_DS.ast）に書き込んだものである．

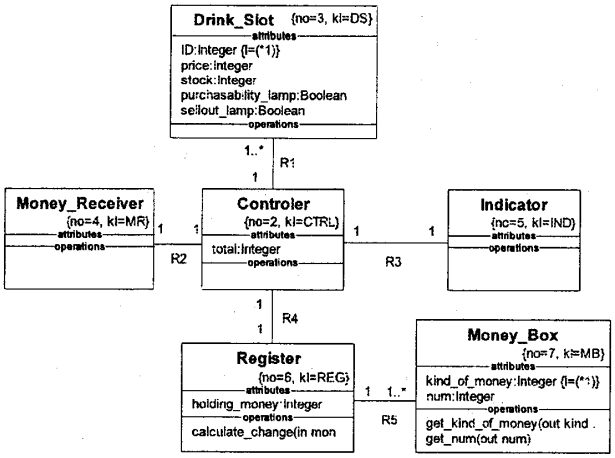

図 5: クラス図

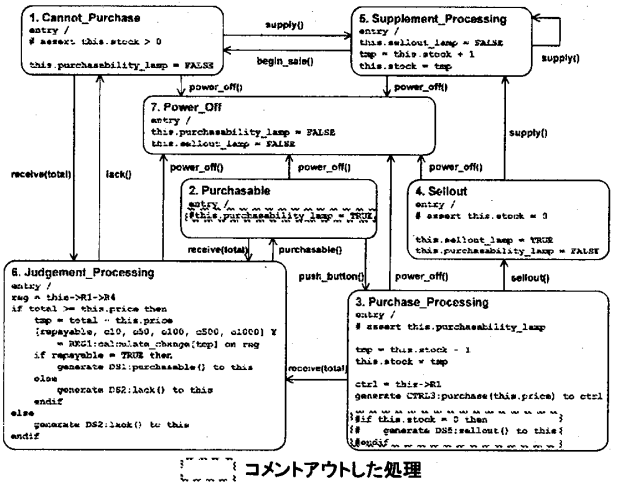

図 6: Drink_Slot クラスのステートチャート図

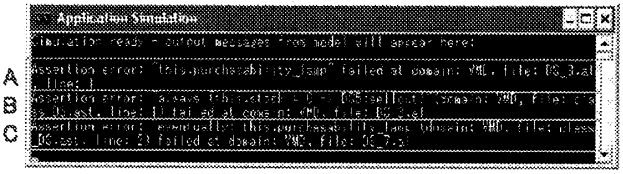

図 7: 実行結果

Drink_Slot クラスのインスタンスが状態 **1**, **6**, **2**, **3** **7** の順で状態遷移を起こすように実行した結果は図 7 のようになった．図 7 のエラーメッセージ A，B，C はそれぞれ上記のアサーション **a**，**b**，**c** に対して違反が起こったことを示している．

次に，状態 **2** でコメントアウトした処理をコメントアウトせずに実行したところ，エラーメッセージ A，C は出力されなかった．これはアサーション **a**，**c** のプロパティが満たされたことを示すが，確かに状態 **2** でコメントアウトせずに実行することでアサーション **a**，**c** のプロパティは満たされる．また，状態 **3** でコメントアウトした処理をコメントアウトせずに実行したところ，エラーメッセージ B は出力されなかった．これはアサーション **b** のプロパティが満たされたことを示すが，確かに状態 **3** でコメントアウトせずに実行することでアサーション **b** のプロパティは満たされる．

以上より，正しい実装であると確認できた．

8. 議論

iUML 以外の実行可能 UML ツールへのアサーションベースの動的検証の適用可能性について述べる．本研究における実装は，アサーションから ASL の条件判定を生成し，生成した条件判定を状態のプロシージャが ASL で記述されたファイルに挿入することで行った．一方で，iUML では実行ファイルの生成過程で C ファイルも生成されるため，我々の手法以外にも，C ファイルに C 言語の条件判定を挿入するという実装方法も考えられる．つまり，実行ファイルの生成過程で何らかの言語のコードが記述されたファイルが生成されれば，その言語に従う条件判定を生成されるファイルに挿入することで実装が可能であると言える．iUML 以外の実行可能 UML ツールでも，実行ファイルの生成過程で何らかの言語のコードが記述されたファイルが生成されると考えられるため，生成されるファイルにその言語に従う条件判定を挿入することで実装が可能であると考えられる．

次に，本研究で定義したアサーションと PSL，xPSL との差異について述べる．本研究で定義したアサーションは，実行可能 UML，特に iUML を対象としている．一方，PSL はハードウェア記述言語（HDL）を対象としており，様々な時相論理演算子が使用できる．xPSL は PSL を拡張して実行可能 UML で用いる変数やイベントを指定できるようにしたものであり，HDL と実行可能 UML の両方を対象としている．

9. まとめ

本研究では，実行可能 UML におけるアサーションベースの動的検証の実現に向け，アサーションの記述方式を定め，プロパティが満たされるかどうかのチェック方法を考案し，チェックが正しく行えることを例題実験により確認した．

今後の課題としては，より複雑な時相論理プロパティに向けて拡張していくことが挙げられる．

参考文献

[1] Harry D. Foster, Adam C. Krolnik, David J. Lacey, （監訳：東野輝夫, 岡野浩三, 中田明夫）, “アサーションベース設計 原書 2 版”, 丸善株式会社, 2004

[2] Stephen J. Mellor, Marc J. Balcer, （監訳：二上貴夫, 長瀬嘉秀）, “Executable UML MDA モデル駆動型アーキテクチャの基礎”, 株式会社翔泳社, 2003

[3] Fei Xie, Huaiyu Liu, “Unified Property Specification for Hardware/Software Co-Verification”, Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1, pp. 483-490, July 2007

[4] “UML Action Semantics”, http://www.omg.org/cgi-bin/doc?ptc/02-01-09

[5] “Intelligent UML”, http://www.kc.com/products/iuml.php

[6] Ian Wilkie, Adrian King, Mike Clarke, et al., “UML ASL Reference Guide”, http://www.kc.com/download/index.php

[7] Harry Foster, Erich Marschner, et al., “Property Specification Language Reference Manual Version 1.1”, http://www.eda.org/ieee-1850/

[8] 菅井雅仁, “実行可能 UML におけるアサーションベースの動的検証”, 埼玉大学卒業論文, 2008