2008－SE－159 (26)
2008／3／18

UPPAALによるモデル検査適用ガイドラインの作成

青木翼† 長谷川哲夫† 宮本博暢† 渡邊竜明†
†株式会社 東芝

モデル検査によりソフトウェア設計の品質を向上させることが注目されている．一方，モデル検査を開発に適用するノウハウはまだ不足している．本論文ではモデル検査をソフトウェア開発に適用させるためのガイドラインについて報告をする．このガイドラインではモデル検査を実施する目的と効果，モデル検査が適用できる箇所，モデルや検証式を作成する手順，モデルや検証式の記述テクニックの４点について説明を行っている．

Guideline for model checking in UPPAAL

Mamoru Aoki† Tetsuo Hasegawa†
Hironobu Miyamoto† Tatsuaki Watanabe†
†Toshiba corporation

While improvement of the quality of software design using model checking is required, the current know-how pertaining to the application of model checking is insufficient. This paper describes the guideline for model checking in UPPAAL, which consists of 4 sections explaining the objectives and outcomes of model checking, target documentation, process, techniques of models and verification expressions building.

1. はじめに

近年のソフトウェア開発では大規模化が進むことで，レビューや試験の実施コストが大きくなり，十分な品質の確保も困難になってきている．この問題に対してモデル検査などの形式手法によりソフトウェアの設計段階から品質を確保することが注目されている．形式手法のためのツールは無料のものを含めて複数提供されており，特にモデル検査は開発に適用するための敷居が比較的低いと考えられている．

一方でモデル検査を開発に適用するにはまだ課題も残っている．主なものを以下に列挙する．

- モデル検査でどんな検証ができて，どんな効果が得られるのか分からない
- ツールがあってもモデル検査を実施するノウハウがない

そこで開発現場でモデル検査を適用することを目的としてガイドラインを作成した．以下の章ではこのガイドラインについての説明をする．

2. ガイドラインの概要

2.1 ガイドライン作成の方針

このガイドラインはモデル検査によるソフトウェア検証の経験から得られたノウハウを文書化することを目指した．経験の元としたのは複数の実事例ならびに仮想事例である．仮想事例は，実事例かどうかに大きくは依存しない部分の経験を得ること，ならびにガイドライン内での具体例とすることを意図して実施した．文書化されたノウハウが開発現場に受け入れられることを確認するために，開発者によるレビューも実施した．またガイドラインを適用する開発分野についても特に限定はせずに，なるべく多くの開発で使用できるように心がけた．

ガイドラインではモデル検査ツールとしてUPPAAL[1]を前提とした．UPPAALとはUPPSALA大学とAALBORG大学によって開発されたモデル検査ツールであり，GUIベースで操作できることと，時間オートマトンが扱えることが特徴である．特にGUIを持つことでユーザへの敷居が低くなると期待できる．

2.2 ガイドラインで説明したこと

ガイドラインで説明をしたのは以下の事柄についてである．

1. モデル検査を実施する目的と効果

モデル検査によって得られる効果，ならびにモデル検査に適した開発を明らかにすることで，これを実施する目的を明確にした．

2. モデル検査によって検証できること

開発時に作成される様々な設計成果物の中でも，どこがモデル検査の対象となるのかを明確にした．そのためにモデル検査が前提とする設計プロセスと設計成果物についても説明をしている．併せて陥り易い間違いについても触れてある．

3. モデルや検証式を作成する手順

まずモデルの構成について説明した後，モデルや検証式を作成するための手順を，事例を交えて解説した．

4. モデルや検証式の記述テクニック

モデルや検証式を記述する上で役立つ方法や誤りやすい事についてまとめた．書かれた内容はUPPAALに依存しているが，他のツールについても同じ考え方が当てはまるものもある．

なおツール自体の使い方についてはガイドラインの目的にそぐわないために説明はしていない．

3. モデル検査でできること

3.1 モデル検査を実施する目的と効果

3.1.1 モデル検査の効果

まずモデル検査の効果としては以下の4つにまとめられると考えている．

- 設計時にレビューと相補的な検証をする

設計の誤りが試験工程まで残ってしまうと，修正のために手戻りの大きなコストが生じてしまう．そこで設計の品質を向上させるためにレビューやモデル検査を使用する．レビューでは設計者が設計で意図したことを含めた確認を，対してモデル検査では設計の論理的な整合性の確認をすることができ，両者は相補的に使い分けるものである．

- 動作条件の組合せや非決定的な振舞いを網羅的に探索する

動作条件の組合せが多くなったり，非決定的な振舞いをする場合には，例外なく正常動作することの確認が困難になる．そのような場合にはモデル検査により網羅的な探索を行うことができる．

- 設計資料の一部として利用

設計を行うための資料として，あるいは設計成果物として検証モデルを利用することが可能である．

- 開発者に段階的な設計プロセスを意識させる

モデル検査では段階的に設計作業を行うことが重要になる．そこでモデル検査を実施することで，開発者に対しても設計プロセスを意識させる副次的な効果もある．

3.1.2 モデル検査に適したシステム

モデル検査の対象として特に適しているシステムがある．いずれもレビューや試験では確認が難しい，以下のようなものである．

- コンカレントシステム

マルチスレッドやマルチタスクなどの並行動作をするシステム．

- タイミングによって動作が変化するシステム

非同期イベント処理や通信プロトコルなどタイミングによって動作が変化するシステム．

- 条件の組合せが多い動作に対する検証

異常系処理のように条件の組合せによって考慮すべきケースが多くなってしまうシステム．

3.2 モデル検査によって検証できること

3.2.1 設計プロセスと設計成果物について

設計プロセスは組織やプロジェクトごとに異なるものであるが，ここでは以降の説明に必要な範囲で設計プロセスと設計成果物についての用語の定義や考え方を説明する．まず設計プロセスと設計成果物についての用語定義を表1に，設計プロセスと設計成果物の関係を図1に示した．

表1：設計プロセスと設計成果物の用語定義

用語	定義
要求	顧客や開発者がシステムに対して意図していること
仕様	システムが守るべき性質や規則（入力，出力，事前条件，事後条件，不変条件）
仕様書	仕様を記述した文書
仕様化作業	要求に基づいて仕様書を作成する作業
設計	仕様の実現方法
設計書	設計を記述した文書
設計作業	仕様に基づいて設計書を作成する作業

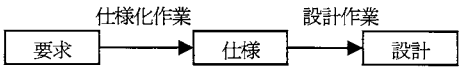

図1：設計プロセスと設計成果物の関係

そして設計プロセスとは，仕様化作業と設計作業を繰り返しながら詳細化を進めてゆくものである．設計プロセスの例を図2に示した．ここで重要なのは仕様と設計が区別されることである．これはソフトウェアの持つ性質とその構造を区別することであり，これを意識することがモデル検査では重要になる．

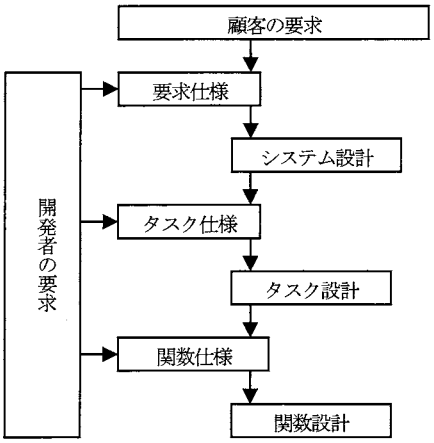

図 2：設計プロセスの例

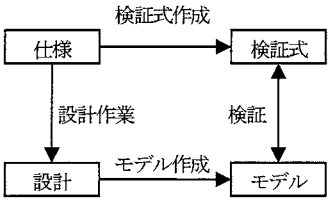

図 3：モデルと検証式の作り方

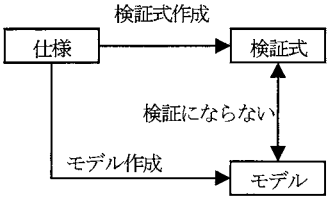

図 4：検証できない場合

3.2.2 モデル検査と設計プロセスの関係

ソフトウェアの検証には少なくとも検証されるもの（検証対象）と正しいかどうかの基準（検証項目）が必要である．そこで検証のためには設計成果物から２つの文書を選び出す必要がある．モデル検査では検証対象となる設計を決めてモデルを作成し，検証項目となる仕様を決めて検証式を作成する．この検証を行うことで設計が仕様通りに作成されていることの確認が可能である（図 3）．

またモデルと検証式が異なる文書から作成されていることは重要な点である．もし同一の仕様からモデルと検証式を作成してしまうと意味のある検証にならない（図 4）．これは検証したいことを明確にしないままモデルを作ったときに起きやすい間違いである．モデルは作成できたのに「うまく検証ができない」と感じる時には，この間違いをしている可能性がある．

4. モデル検査実施のための手順とノウハウ

4.1 モデルや検査式を作成する手順

4.1.1 モデルの構成

モデルを作成する手順を説明する前に，モデルの構成についての説明をする．モデル検査では検証対象となる部分の他に，検証対象と関わりを持つ部分も同時にモデル化する必要がある．前者を検証対象部分，後者を環境と呼ぶ．環境のモデルは検証対象部分の使われ方を規定するものであり，例えばシステムを操作するユーザなどである．この２つを合せて初めて一つの動作可能なモデルができる（図 5）．

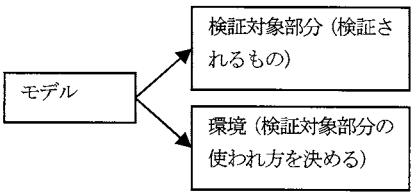

図 5：モデルの構成

環境のモデルが持つ役割は以下の５つにまとめられる（常に５つとも必要なわけではない）．

- 検証対象にイベントを送る

環境のモデルの中でこの役割をもつ部分を特にドライバと呼ぶことにする．モデル全体の動作シナリオを開始・進行させる働きがある．

- 検証対象からイベントを受ける

この役割を持つ部分はモデル全体の動作シナリオを進行させる働きがある．

- 通信経路を表現する

通信を行うモデルに対しては通信経路のエラー，トポロジー，遅延，非同期性などを表現するモデルを作成する場合がある．

- 状態遷移に制約を与える

これはモデルの状態遷移可能な箇所を意図的に遷移させないことである．特定の機能や動作シナリオだけを検証したい場合，状態爆発を防ぎたい場合などに使われる．この制約をモデルに与える方法には２つある．１つはイベントの発生する順番・種類・タイミングを

制限することであり，もう１つはGuard条件を使用することである．

- 検証式を書きやすくする

検証式だけでは書きにくい検証項目を記述するために，モデルの振舞いを観測するためのモデルを作成することがある．

4.1.2 検証プロセスの概要

モデル検査による検証プロセスはおよそ以下のような作業を順番に行うことである．

1. 検証内容の明確化
2. モデル化対象と検証項目の明確化
3. モデルの作成
4. 検証式の作成
5. 検証の実施
6. 状態爆発の解消
7. 反例解析

あらかじめ何を検証したいのかを明確にすることが特に重要である．なお検証の実施はツールが行うのでガイドライン中ではその詳細までは説明していない．また検証プロセスを説明するために簡単な通信プロトコルの仮想事例を利用している．

4.1.3 検証内容の明確化

まず検証内容を明確化するために以下の３点について順番に考える．

1. 現在の問題は何であるか？

現在，品質面でのリスクとなっているものが何であるかについて考える．不安があるから検証をしたいはずなので，まずはその不安を言葉で表現してみる．

2. 検証の目的は何か？

品質を確認するためには「どの機能が」「どのように」動作すればよいのかを明らかにする（ここから検証式が作られる）．

3. どのような確認をするのか？

検証の目的を果たすためには「どのソフトウェアが」「どのような場合に」「何をする」ことを確認できれば良いのかを明らかにする（ここからモデルが作られる）．

以上の３点に沿って，あらかじめ検証したい事を決めておく．3.2.2で説明したように検証対象と検証項目の組が何であるかが明らかになればよい．

検証内容が曖昧なままでは検証するコストが大きくなり，あるいは状態爆発により検証自体ができなくなる恐れがある．特に詳細設計や実装に近いものを検証するときには複雑なモデルを簡略化するためにモデル化の対象を絞ることが重要である．

またモデル検査では仕様書や設計書に記述された範囲内でのみ検証ができるので，検証対象が設計済みであることにも注意を要する．設計書に書かれていないものを検証しようとしても無駄である．

さらにモデル検査は与えられた検証式に対して網羅的に検証するのであって，検証式自体が十分網羅的に作られていることは検証者が保証する必要がある．

4.1.4 モデル化対象と検証項目の明確化

検証のためのモデル化対象と検証項目をより明確にしてゆく．そのための手順について以下の８項目にまとめたので順番に説明をしてゆく．

1. 検証のシナリオを考える

検証のシナリオとはモデルの振舞いと結果のことである．これを見つけ出す方法には２通りがある．

１つ目は，モデルの上で「何が」「どうすると」「どんな場合には」「どうなって欲しい／欲しくない」をそれぞれ考えてゆく．例えば「ユーザが」「スタートボタンを押すと」「ログオン中ならば」「メニューが表示される」のように記述する．「どんな場合には」という部分はシナリオによっては無くても良い．

２つ目は，ソフトウェアの機能や振舞いについての記述を仕様書から見つけ出すことである．特にソフトウェアの機能はユースケース図として，振舞いはシーケンス図として書かれることが多い．そこで仕様書に書かれた機能や振舞いを確認する．シナリオとして適切なものが見つかれば同様に「何が」「どうすると」「どんな場合には」「どうなって欲しい／欲しくない」という形で記述する．

2. モデル化の対象を具体化する

先に述べたようにモデルには検証対象の他に環境の部分も必要である．そこで環境を含めたモデル化の対象を明らかにする．検証対象と関係のあるソフトウェアのモジュールやタスク，ハードウェア，ユーザ，通信経路などが考えられる．特にシナリオに登場するものは漏れなく探し出す．

3. 検証の粒度について確認する

これまでに考えてきたことを見直して，目的どおりの検証が出来るかどうかを確認する．主な確認すべき点としては以下のようなものがある．

- 検証内容が特定されているか
- モデルとシナリオの整合性は取れているか
- シナリオによって検証の目的が果たせるか

モデル化対象の一部が冗長すぎてモデルにそぐわないと感じた場合には，抽象度の異なる要素がモデルに入っている可能性がある．その時には同じ役割を持つ

部分をまとめて抽象化するする方法を検討してみる．例えば通信経路のモデルについて，到着したかどうかだけを考えて遅延時間は無視する，などである．

もし検証内容，モデル，シナリオの間で整合性が取れていなければ，今までの作業を再度やりなおしてみる．

4. モデル構成要素と設計書の対応を明らかにする

モデルの構成要素について，どの設計書と対応するものかを確認する．この対応関係からモデル構成要素の役割や性質を知ることができる．モデルはソフトウェアとは限らないが，直接的な対応が取れないものについては，システム内での役割を明らかにしておく．

5. 検証対象の洗練

検証対象のモデルをどのようなものにするかを決定する．検証される機能については既に考えてあるが，モデルを作成するコストや状態爆発まで考慮をしてモデルの作成方法を検討する．

6. 環境の決定

検証を行うために必要な環境について決定する．4.1.1 で述べた環境の持つ役割を参考にすると良い．特にドライバはモデル全体をシナリオ通りに動作させるために重要である．検証対象部分に対してドライバが網羅的なイベント列を送るか，あるいは特定のイベント列だけを送るかによって，モデル全体の振舞いを変えることができる．

7. 検証対象が満たす性質から検証項目を決定

検証対象が満たす性質を仕様書から探し出して検証項目を決定する．例えば「変数ｘは常に１００以下である」のような性質を見つけ出せればよい．これはアサーションのような働きをする．

8. 検証のシナリオから検証項目を決定

検証のシナリオから検証項目を決定する．検証シナリオの「結果」に注目して，望ましい結果が成立すること，あるいは望ましくない結果が成立しないことを検証項目とする．

4.1.5 モデルの作成

以上の作業によりモデル化の対象が明らかにされたらモデルの作成を開始する．ガイドラインではモデルの作成方法についても解説をしている．しかし UPPAAL に依存した話であり，特別なノウハウもないのでここでは説明しない．

4.1.6 検証式の作成

検証式はその目的によって以下の３種類に分類することができる．

- モデル自体の正しさを検証する

モデルを作成しただけでは，本当にモデルが正しく作られたか分からない．そこで単純な検証式をいくつも書くことでモデル自体の正しさを確認する作業が必要になる．

- 検証対象部が保つべき性質を検証する

検証対象部が保つべき性質を見つけ出して検証式とする．これはモデルに対するアサーションを記述していくことに近い．

- 検証シナリオの結果を検証する

最後に検証シナリオの結果を確認するための検証式を書くことでモデル全体の動作を検証する．

モデル検査で使用する検証項目は時相論理式で記述する必要がある．しかし時相論理式に慣れていない開発者も多いと思われる．そこで典型的な検証項目を UPPAAL の検証式で記述したものを表 2 に整理した．なお表中の※は直接 UPPAAL の検証式を書くことが出来ないので変数を利用するものである．この表によって記述したい内容から検証式の形を導き出すことが可能になり，また検証式で何が書けるかの理解を助けることにもなる．この表を作成するためには過去に作成したモデル，ならびに検証式の使用頻度の研究[2]を参考にした．

検証式を書くときの注意点としては「予想通りの動作ならば結果が真になる検証式」を書くことである．もし「予想と異なる結果が出たときに真になる検証式」を書くと反例が出なくなり，後述する反例解析に差し支えるためである．

4.1.7 状態爆発の解消

モデルが複雑すぎる場合には検証が現実的な時間内で終わらなくなってしまう．UPPAAL で状態爆発が発生した時には，メモリ消費量が増加し続けるばかりで検証結果がいつまでも出なくなる．状態爆発が発生した場合には，少しずつモデルや検証式の見直しをしながら状態爆発が解消されるまで検証を繰り返すことになる．

状態を減らすために，もし一時的な値を持つ変数があれば使用しない間は常に同じ値(0 などで良い)を持たせておく．さらにクロック変数を使用する箇所が適切であるかも見直しをする（4.2.5 参照）．

次に検証シナリオの簡略化を検討する．検証の意味を失わない範囲でより簡単なシナリオへ変更してみる．特にドライバをより単純なものにする，繰り返し数を削減してみる，などについて考えてみる．

それでも状態爆発が解消しなければ，検証の対象を

表 2：検証項目の分類と UPPAAL の検証式の形

検証項目の分類	UPPAAL の検証式の形	検証式の意味
デッドロックがない	A[] not deadlock	デッドロックがない
ライブロックがない	X --> Y E<> X	必ず処理中状態 X は終了して待機状態 Y に戻る
不変条件	A[] X	不変条件 X は常に成立する
事前条件	A[] X imply Y E<> X	条件 X 成立時には常に事前条件 Y が保たれる（X 成立中は Y も成立し続ける場合）
事前実行	A[] X imply Y_occurred E<> X	条件 X の前に必ず条件 Y が成立している（※変数 Y_occurred は初期値を false として定義され，条件 Y が成立した時点で true に変更する）
イベントと対応動作	X-->Y E<> X	イベント X があると対応動作 Y をする
制限付きイベントと対応動作	A[] X or OrderOfXYZ==false E<> X E<> Y	条件 X が成立している間に，イベント Y があると対応動作 Z をする（※変数 OrderOfXYZ は初期値を false として定義され，X が成立している間に Y が発生すれば true に，X が成立している間に Z が発生すると false に変更する）
処理の進行	X-->Y E<> X	処理 X の後には必ず処理 Y がある
実現可能	E<> X	ある条件 X が実現する可能性がある
実現不可能	A[] not X	ある条件 X は実現しない
必ず実現する	A<> X	いつか必ず条件 X が実現する
最長時間	A<> (X and t <= Y)	条件 X は必ず Y 秒以内に一度は成立する（t はクロック変数）
最短時間	E<> (X and t <= Z)	条件 X は最短 Z 秒で成立する可能性がある（t はクロック変数）
制限付き不変条件	A[] X imply Y E<> X	条件 X の間は常に条件 Y が成立する

より狭くしたモデルにすることも検討する．

4.1.8 反例解析

UPPAAL では"E<> X"など一部の検証式は偽になっても反例が出ない．検証式の意味を考えれば反例が出ないのも当然であるが，"E<> X"は頻繁に使われる式なので対策が必要になる．まず「条件 X」が成立するために必要な前提条件を探して，その前提条件が実際に成立しているかを確認する方法が有効である．例えば「条件 X」が成立するためには「条件 Y」と「条件 Z」が前提条件であれば"E<> Y"と"E<> Z"の検証も実行すればよい．もし条件 Y が成立しなければ，その付近をさらに詳細に解析してゆく．

反例が複雑すぎるために解析が困難な場合には以下の方法を使う．

- 反例が発生する最小のモデルを作成する
- モデルの特定箇所の振舞いを確認する検証式を追加する

前者は，モデルの Guard 条件を false にするなどにより遷移を少しずつ制約してゆき，なるべく小さなモデルで同じ反例が発生するものを作り出してゆく．

後者は，検証項目と関連がありそうな箇所の振舞いについての検証式を新たに追加することで，原因となっている箇所を狭めてゆく．

反例解析のための基本的な方針は問題の最小化であるが，もう一方で設計や検証の意味を理解していることも重要である．反例といえどもモデルは作成した通りに動作しているので，意味を知らなければ動作が不正かどうかの判断もつかないためである．また時に実現不可能と考えられる極端な反例が出てくることもある．この場合にはモデルないし検証式を修正することで反例をなくすこともある．

4.2 モデルや検証式の記述テクニック

UPPAALでモデルや検証式を記述するためのテクニックについていくつか整理した．この節ではUPPAALの用語を多く使用しているが，詳細についてはマニュアルを参照してほしい．

4.2.1 命名規則について

モデル作成前には混乱しない程度の命名規則を決めておくと良い．特に決まったものは無いので，コーディング規約に準じたものなどでよい．

4.2.2 ロケーションと変数について

UPPAALのモデルは状態を表現するのにロケーションと変数のいずれかを使用できる．一方，ロケーションと変数はそれぞれ一長一短であるため，モデル作成時にはそれらを理解して適切に使い分けを行う必要がある．ロケーションを分ける目安としては以下のようなものがある．

- チャネルによる通信を伴う遷移の前後を別ロケーションにする
- 状態遷移図が入れ子で記述されている場合の大きな状態を1つのロケーションにする
- クロックと関わる箇所の前後を別ロケーションにする
- 並列性を考慮する単位をロケーションにする

4.2.3 Meta 変数と scalar 変数

UPPAALの変数にはmeta変数とscalar変数という特殊な変数が用意されている．meta変数とは，その値が異なっていても同じ状態と見なされる変数である．一時的な値の保存や，検証式を書きやすくするための変数に利用できる．scalar 変数とは UPPAAL 上で同一 template から複数のプロセスを生成する場合に対称性を考慮して状態を削減できる変数である．これらの変数を適切に利用することで状態爆発が起きにくくすることができる．

4.2.4 遷移可能と遷移する，の違い

Guard条件がfalseでないエッジは遷移可能であるが，遷移するとは限らない．このような遷移可能であることと遷移することの違いには注意が必要である．遷移可能なだけでは，永久に遷移しないことも可能なためである．遷移可能ならばいずれ必ず遷移する，というモデルを作成するためのもっとも簡単な方法はモデル全体に時間制約を与えることである（図6）．また遷移可能になったら即座に遷移する場合には，アージャント遷移用チャネル(urgent chan)を使えばよい．

図 6：時間制約を与えるだけのモデル

4.2.5 時間がクロック変数で表されるとは限らない

UPPAAL にはクロック変数があり，時間のカウントに使用することもできる．しかしモデルの時間がクロック変数で表されるとは限らない．発生するイベントの順番だけが重要な検証ではクロックなしに記述することができ，その方が状態爆発も起きにくくなる．クロック変数を使用するのは時間そのものに関する検証をしたい場合である．

4.2.6 GUARD 条件と UPDATE 処理の順番

チャネルを使用して状態遷移を同期させるときに，エッジに書かれたGuard条件の評価とUpdate処理が行われる順番は以下のように決まっている．

1. 送信側・受信側の Guard 条件の評価
2. 送信側の Update 処理
3. 受信側の Update 処理

この順番に時間の差はないが，モデルの振舞いには影響を与える．最も間違えやすいのは，送信側のUpdate処理の結果を受信側Guard条件で使用しようとして遷移不可能になることである．なお，送信側 Update は受信側 Update より先に実行されるため，チャネル通信と同時にプロセス間で値の受け渡しは可能である．

4.2.7 AND，IMPLY，"-->"の使い分けと注意点

検証式に使用する and と imply は意味が異なるが，複雑な検証式を書くときに誤りやすい．"X and Y"の意味は「X と Y が両方とも真」である．対して"X imply Y"は「X が真であれば Y も真であるが，X が偽であれば Y は真でも偽でも良い」である．よって"A[] X imply Y"という検証式が真になった場合には，X が一度も真にならない可能性があることに注意が必要である．「X が真になることがある」という確認まで行うためには検証式"E<> X"を別に実行すればよい．

また検証式"A[] X imply Y"と"X --> Y"も意味が異なる．"A[] X imply Y"の意味は「X が真であれば常に Y も真である」である．"X --> Y"の意味は「X が真である間以降に Y が真になる」であり，X と Y が同時に成

立しなくても良い．Xが一度も真にならない可能性がある点には同様に注意を要する．

4.2.8 初期化関数

モデル内に定義した変数の初期値は変数定義時に指定できるが，初期化処理が複雑な場合には初期化関数を定義してその中で行う方が簡単になる．初期化関数を作ったら，初期ロケーションをurgentにして最初の遷移で初期化関数を呼び出せばよい．ローカル変数の初期化関数はモデルごとに作る必要があることに注意すること．

5. おわりに

UPPAALによるモデル検査のノウハウをガイドラインとしてまとめることを行った．モデル検査実施のために必要になる情報を広く記述することができたと考えている．今後の課題は，適用事例を増やしてガイドラインへのフィードバックを得ることで，より実践的な内容としてゆくことと，モデル検査適用コストを下げられる補助ツールの検討をすることである．

参考文献

1) UPPAAL ホームページ http://www.uppaal.com/
2) Spec Patterns http://patterns.projects.cis.ksu.edu/