UPPAAL2k: Small Tutorial

日本語版

Ver.1.0

1 イントロダクション

このドキュメントはUPPAAL と検証を始める人のためのチュートリアルです。形式手法の知識があまりない人であってもこのチュートリアルを実施した後には実際に UPPAAL を使うことができるようになります。

セクション2では UPPAAL を説明し、セクション3はチュートリアルとなっています。

2 UPPAAL

UPPAAL はリアルタイムシステムの妥当性検査（グラフィカルなシミュレーションによって）と検証（自動的なモデル検査によって）を行なうツールボックスで、主にグラフィカル・ユーザインタフェースとモデルチェッカーエンジンの2つの機能から構成されています。 ユーザインタフェースは Java で実装されており、ユーザーワークステーションの上で実行されます。 実行にはJava 1.2以上を必要とします。 エンジン部分はデフォルトでユーザインタフェースと同じコンピュータの上で実行されますが、サーバー上で実行する事もできます。

時間オートマトンを用いてシステムをモデル化するとは、ミュレートし、その上で特性を検証する事という意味です。時間オートマトンとは時間を持つ有限の状態マシンです。 システムはロケーションで構成されているプロセッサーのネットワークから成り立っています。 ロケーション間の遷移はシステムの振る舞いを定義しています。 シミュレーションステップではインタラクティブに、システムが思い通りに機能するかを調べていきます。 また到達可能性をチェックすることができるので、ある状態へ到達可能かどうかがわかります。これはモデル検査と呼ばれ、基本的にシステムの全ての起こりうる動的な振る舞いを網羅的に調査します。

より正確に述べるとエンジンはシンボリック技術を使ったオンザフライ検証で制約システムの検証問題を減少します。[YPD94, LPY95].

verifierでは簡単な不変式と到達可能性を検査します。他の特性はtesting automata [JLS96] でデバッグ情報と共に複雑なシステムでチェックできます[LPY97]。

3 UPPAAL を学ぶ

UPPAAL は時間オートマトン（クロックのある有限状態マシン）がベースです。クロックとはUPPAALで時間を扱うための方法です。時間は連続で、クロックは時間の進行を測定します。これによってクロックの値をテストしたりクロックをリセットする事ができます。時間はシステム全体に対して同じペースでグローバルに進行します。

UPPAALではシステムは並行プロセスから成り立ち、各プロセスは1つのオートマトンとしてモデル化されます。オートマトンはロケーションのセットを持っています。遷移はロケーションを変えるために使用します。ガードと同期によって遷移をいつ発火するかをコントロールします。ガードは遷移できるようになるための条件で変数とクロックによって定義します。UPPAALの同期メカニズムはハンドシェイキング同期です。ハンドシェイキング同期とは2つのプロセスが同時に遷移を行なう。1つはa!を持っているであろう一方はa?をもっている。a は同期チャンネルです。遷移が起きるときアクションが可能になります：アクションとは変数の割り当てかクロックのリセットです。

以下の例ではこの同期を詳しく説明しています。

3.1 概要

UPPAALのメインウィンドウ(図 1) はメニューとタブから構成されています。

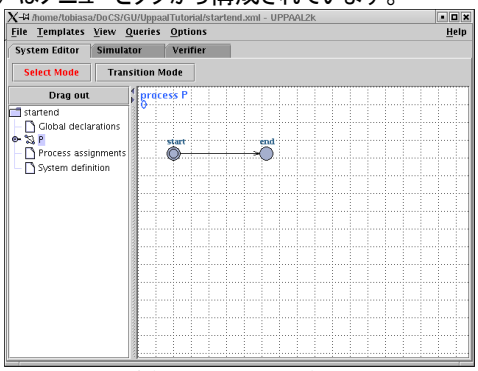

図 1: UPPAALの画面.

メニューについては統合ヘルプで説明されています。ヘルプではGUIについて詳細に説明していますのでこのチュートリアルではツールの使い方にフォーカスします。3つのタブからはUPPAALの3つのコンポーネントである**editor**, **simulator**, **verifier** へアクセスすることができます。

図 1では**editor** を表示しています。完全なシステムを作成するためにインスタンス化されるプロセスの(C++風の)テンプレートを定義することができます。システムがしばしば非常によく似たプロセスを複数使用している時にはテンプレートを定義すると有用です。制御構造 (つまりロケーションとエッジ) は同じで、定数か変数かが異なるだけです。そのためにテンプレートはパラメータとしてシンボリック変数と定数を保持しています。テンプレートは同様にローカル変数とクロックを持っています。

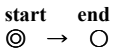

図 2: オートメーション.

それではまず始めに UPPAAL で右から 3 つめのアイコン“Add Location” をクリックしてから描画エリアをクリックするとロケーションが追加されますこれを 2 回繰り返します。

右から 4 つめのアイコン“Select” をクリックしてからロケーションをダブルクリックします。
これらのロケーションをクリックして名前を「start」と「end」に変更します。

右から 2 つめのアイコン“Add Transition” をクリックして“start location”をクリックし、次に “end location” をクリックします。

右から 4 つめのアイコン“Select” をクリックしてから“start location”を右リックして“Initial”を選択します。小さい円が中に表示されます。図 2 で表示されているようになり、最初のオートマトンが準備できました。
Simulator タブをクリックしてシミュレートを開始します。yes ボタンをクリックすると実行する準備が整います。

.

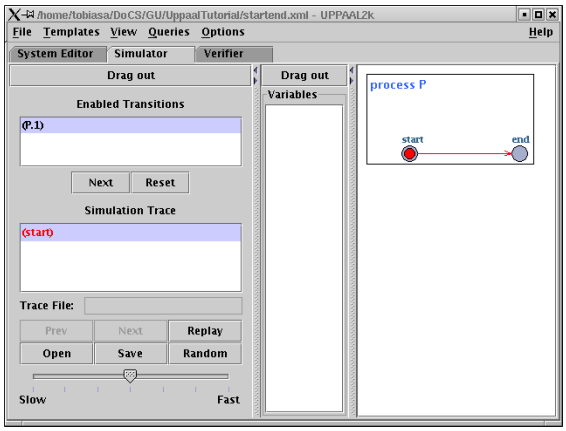

図 3: グラフィカルなシミュレータ

図 3は**Simulator** ビューです。左側にはコントロール部分があり、上部で遷移、下部でトレースを再生／保存／ロードを選択できます。ウィンドウの中央には変数が表示され、右側にはシステム自身が表示されます。

この小さなシステムをシミュレートするために左側の上部にあるリストから実行可能な遷移を選びます。この例ではまだ1つしか選べません。**Next**をクリックします。右側のプロセスビューが変化し（現在の赤い点が移動します）“simulation trace”の表示が増えます。

これでシステムをシミュレートし、検証しました。**Verifier**タブをクリックします。**Verifier**ビューは図 4のように表示されます。上部でクエリーを指定することができます。下部ではモデル検査エンジンとのコミュニケーションを記録します。

クエリーフィールドに***E<>P.end*** と入力します。これはUPPAALの表記法で時相論理式である。***P.end***は「プロセスPがend状態になる」と解釈します。 ∃は「エグジスト」と読み、「存在する」という意味です。◇ は「そのうち」もしくは「最終的に」と解釈します。よって ∃◇ ***P.end***は「そのうちプロセスPがend状態になる場合がありうる」と解釈されます。 **Model Check** をクリックして検証します。overview の黒丸はこの時相論理式が成立すると緑色に変化します。

このドキュメントの残りでは例を通して UPPAAL のキーポイントを体験していきます。

3.2 排他制御アルゴリズム

ここではプログラム／アルゴリズムからモデルをどのように引き出し、関連する特性をチェックしたかを理解するためにPetersonの排他制御アルゴリズムを学びます。

2つのプロセスのアルゴリズムはC言語では次の通りになります。

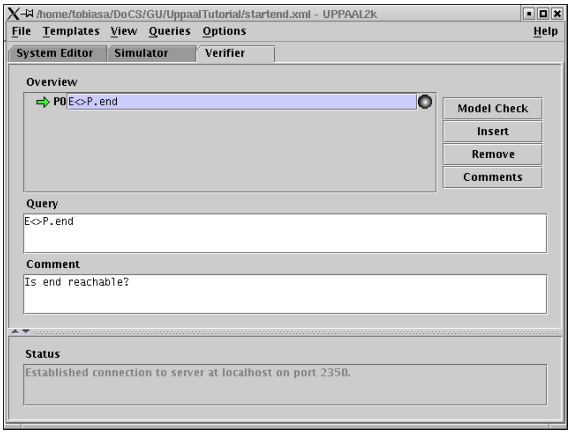

図 4: verifier ビュー

Process 1	Process 2

req1=1;
turn=2;
while(turn!=1 && req2!=0);
//critical section
job1();
req1=0;
	req2=1;
	turn=1;
	while(turn!=2 && req1!=0);
	//critical section
	job2();
	req2=0;

対応するオートマタを構成します。 プロトコルが対称であることに注意し、モデルを単純化するために UPPAAL のテンプレートを使います。 最初にUPPAALをメニューから New systemを選択してリセットします。前の例は削除します。デフォルトテンプレートPをダブルクリックして名前を mutex に変更してください。

ここではクリティカルなセクションにおける実際の処理（job1,job2）には関心がないのでそのままにしておきます。プロトコルは直接Pettersonアルゴリムから導かれた4つの状態（idle,want,wait,CS）を持っています。状態をgotoラベル風に示したものが以下のようになります。

```
Process 1
idle:
   req1=1;
want:
   turn=2;
wait:
   while(turn!=1 && req2!=0);
CS:
//critical section
   job1();
//and return to idle req1=0;
```

図 5.のようにオートマタ を描いてください。

テンプレート名の横にある Parametersのテキストボックスでテンプレートパラメータを定義することができます。

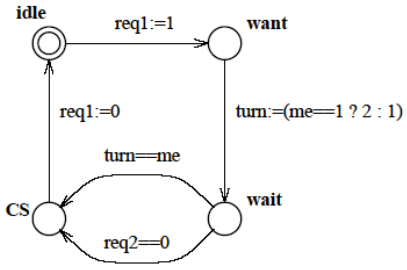

図 5: Mutex テンプレート

. **int[0,1] req1,req2; const me**と入力して、0と1に制限されたinteger型，つまりboolean型である3つの変数を定義します。 最後のパラメータは定数です。

描画から推測できるように、mutex型の2つのインスタンス**P1:=mutex(req1,req2,1);** と **P2:=mutex(req2,req1,2);** のインスタンス化をします。どのように**turn:=(me==1 ? 2 : 1)** （C言語のような式）で評価します。まず2つのインスタンスを作るために、プロジェクトツリーでProcess assignmentラベルを開き、上記の宣言（**P1:=mutex(req1,req2,1);** と **P2:=mutex(req2,req1,2);**）を入力します。

まだ変数の宣言をしなくてはいけません。**Global declarations**ラベルをクリックして**int[0,1] req1,req2;** と **int[1,2] turn;** を宣言します。次にシステムを定義します。**System definition** をクリックして**system P1,P2;**. を定義します。

さてテンプレートを定義し、そのインスタンスを作り、システムでインスタンス化を使って適切な変数を宣言しました。気付いたように、宣言された変数はグローバルです！これは共通の**turn**のために使われます。名前宣言のスコープは最初にローカルで次にグローバルです。テンプレートのパラメータとグローバル変数の名前からこのことに気付きます。モデルの中の名前はわざとそれが異なった場所でどのように働くかを示すように同じものが選ばれます。

Simulatorタブをクリックして、そして2つのオートマタがどのようにインスタンス化されたか調べてください。特に対称な2つのオートマタの名前に注意してください。インタラクティブに遷移を選択していくことによって、システムをシミュレートすることができます。同時に両方のプロセスでクリティカルセクションに到達しようとしてください・・・できないようであれば到達できません。それを確認するには**Verifier**を使います。

Verifierタブをクリックして**Insert** ボタンをクリックします。次に**Query**テキストに排他制御特性を記入します：**A[] not (P1.CS and P2.CS)**。 **Model Check**ボタンを押して実行できます。緑色のボタンが点灯して特性が検証されたことを意味します。赤色に点灯していれば検証されなかったことになります。**A[]not (P1.CS and P2.CS)**は**not (P1.CS and P2.CS)** が常にtrueとなることをチェックするセーフティ特性です。別の特性のタイプで**E<>**は到達可能性特性のために使用されます。例えば新しい**E<> P1.CS**を挿入してプロセスP1がクリティカルセクションに到達したかどうかをチェックできます。

もしシステムが正しくなければUPPAALは**diagnostic trace**を返します。最初は結果をfaultyにするために例えばガードを**req2==0** から **req2==1**へ変更します。次に**Options**メニューで**Diagnostic Trace**をチェックし、排他制御特性を選択し、**Model Check**ボタンをクリックします。するとこの特性は満たされず、トレースを保存するかをたずねるウィンドウが表示されるのでそこで**yes**を選択するとシミュレータに戻ります。**Replay** を押下し、トレースができます。

これでモデル化し、シミュレートし、簡単な排他制御特性の検証ができました。独立したディレクトリ以下のデモフォルダーにいくつかの簡単な例があります。例えばfile fischerにはまた別の排他制御プロトコルを含んでいます。

3.3 UPPAAL での時間

このサブセクションではUPPAALの時間の概念をわかりやすく説明します。

UPPAALでの時間は連続した時間です。技術的にはそれはリージョンとして実装されます。したがって状態はシンボリックです。これはある状態の時間の値は絶対値ではなく相対値です [AD94] 。UPPAALで時間がどのように扱われるかを理解するために簡単な例を勉強していきましょう。違いを見るためにobserverを使用します。通常observerは影響を与えずにシステムを観察し、イベントを検出するアドオンオートマタです。このケースではリセットされたクロック（x:=0）はobserverに委譲され、遷移ループで直接リセットされるので、実際にはオリジナルのふるまいは変わりません。

図 6 はobserverの最初のモデルです。時間はクロックを通して使われます。例ではxはクロックxとして Global declarationsラベルでグローバル宣言されています。チャンネルはobserver と同期をとるために使用されています。この例ではチャンネル同期はreset! と reset? 間のハンドシェイクです。つまりこの例ではクロックは2つの時間ユニットの後にリセットされることになります。Observer はこれを検出して実際にリセットを行ないます。

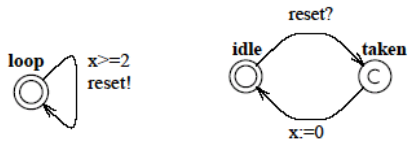

図 6: observer を用いた例.

モデルを描いて オートマタを P1 とObs と命名してシステムで定義してください。Observer の状態がcommit型であることに注意してください。もしシステムをシミュレートすると多くはわかりません。見たものを解釈することができるようになるために、クエリーを使ってシステムを徐々に修正していきます。システムのふるまいで期待される事は図 7 に描かれています。グローバル変数宣言セクションで chan reset; を使ってチャンネルを宣言してください。

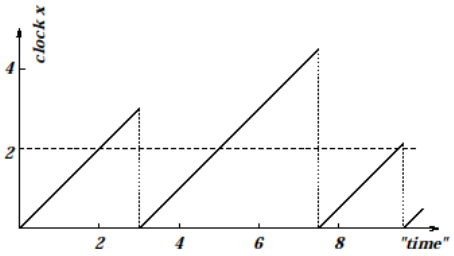

図 7: 時間の振る舞い: this is one possible run.

このふるまいの特性を示すために以下を試してみてください。

- A[] Obs.taken imply x>=2 : クロック値（curve 参照）のすべてのフォールダウンが2の上にある。
  このクエリーの意味：全ての状態に対してロケーションObs.takenであればx>=2である。
- E<> Obs.idle and x>3 : これは待ち期間に対して、30,000といった値を試すことができ、同じ結果を得る。
  この質問の意味：x>3 で Obsのロケーションがアイドルの状態に到達できるかどうか。

図8に示されるように状態ループになるように不変式を加えてください。

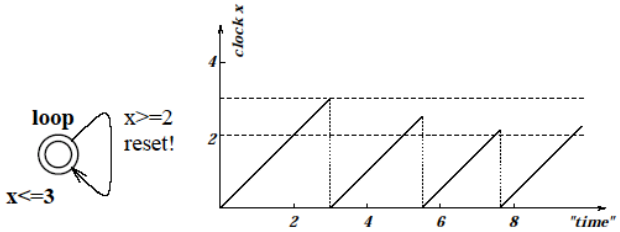

図 8:不変式を追加：新しい振る舞い.

不変式はプログレス条件です：システムは状態の3つ以上の時間ユニットで留まることができませんので遷移をとられなければならず、この例ではクロックがリセットされます。
違いを見るために特性を試してください：

・ A[] Obs.taken imply (x>=2 and x<=3) インターバル2と3の間でその遷移が常に起こることを示す。
・E<> Obs.idle and x>2 : インターバル2と3に遷移を起こす可能性がある。
・A[] Obs.idle imply x<=3 : 上限が守られることを示す。

前の性質 E<> Obs.idle and x>3 は成り立たない。

不変式を消してガードを x>=2,x<=3 へ変更してください。これは同じように見えますが違います！システムはプログレス状態を持っておらず、ガードに新しい条件が加わっています。図9では新しいシステムを表示しています。

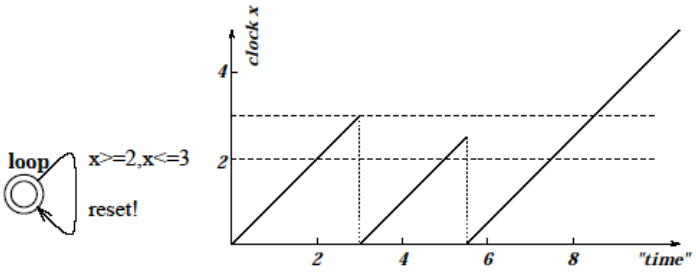

図 9: 不変式なし、ガード条件あり：新しい振る舞い

システムが以前と同じ遷移をするように見えますが、今のシステムはデッドロックがあります。システムは3時間ユニットのあとに遷移しなければ動かなくなってしまいます。何が起こるか見てみてください。
同じ特性をもう一度試してみると、最後の特性は今は成り立ちません。実際には以下の特性でデッドロックが起こります。:A[] x>3 imply not Obs.taken, では3時間ユニットのあとに遷移しません。

3.4 Urgent/Committed ロケーション

さて UPPAAL の異なるロケーションの種類を見ましょう。すでに前の例でcommit型を見ました。

UPPAAL では不変式(the x<=3) のある、またはないnormalロケーションとurgent ロケーション committed ロケーションといった3つの異なるロケーションがあります。図１０で示されるオートマタを描いてください。ローカルにクロックを定義してください。オートマタのサブツリーを開き、テンプレートの下にDeclarationsラベルがあります。clock x をクリックして定義してください。.

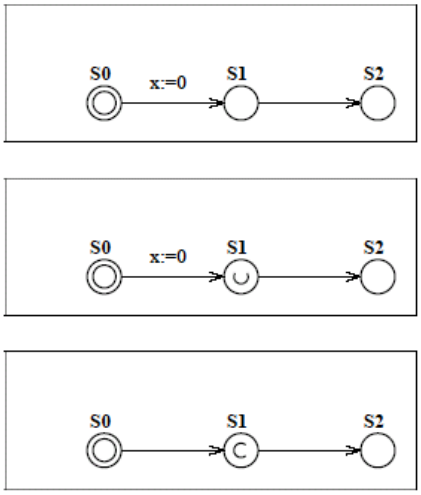

図 10: normal, urgent, commit 状態のオートマタ

それぞれオートマタをP0 、 P1 と P2と名前を付けてください。状態に「U」と付けられているのは緊急（urgent）を意味し、「C」と付けられているのはcommitted であることを意味します。シミュレートでcommit 状態のときに唯一可能な遷移は常にcommit 状態から外に向かう遷移であることを試してください。commit 状態はすぐにその状態から出なくてはいけません。normal状態とurgent状態の違いを見るために特性を検査してみてください。:

- E<> P0.S1 and P0.x>0 : S1で待機できる。
- A[] P1.S1 imply P1.x==0 : S1で待機できない。

urgent 状態では時間が進まないが、シミュレータでnormal状態との切り替えができます。

3.5 特性の検証

上記の例で我々は数回 Verifier を使いましたのでVerifierをより完全に扱う事ができます。まとめると、Verifier で使える質問は:

- E<> p: やがてp が成立するパスがある
- A[] p: 全てのパスでp は常に成立する
- E[] p: 常にp が成立するパスがある
- A<> p: 全てのパスでp がやがて成立する

・ p --> q: p が成立するときはq はいつか成り立つ

ここでp とqは(P1.cs and x<3) の形の状態式です。クエリー言語の文法はオンラインヘルプで参照できます。デッドロックをチェックするのに有用な特別な書式A[] not deadlock があります。.

3.6 モデリングのトリック

UPPAAL は遷移可能で遅延なく同期をとらなければならない時にurgentチャンネルを提供します。これら遷移でのクロック条件は使えません。urgentチャンネルによってurgent 遷移をプログラムすることは可能です。それは変数をもったガード条件、すなわち変数上でのビジー待ちのことです。1つの遷移read!でループ状態となっているダミープロセスを使ってください！ urgent 遷移は例えば x>0 read? となります。

チャンネルでパラメータを渡せませんが共有変数で簡単にプログラムすることができます。変数xをグローバルに定義し、それを使ってwrite、readをします。read! x:=3 とread? y:=x;は正しく動くかはわかりませんが、commit状態を使ってread?からcommit状態にしてy:=x;.とします。

ブロードキャストコミュニケーションがない：同期はペアのみです。ブロードキャストを得るために一連のcommit状態を使います。典型的な手順は以下の通りです： go1! commit go2! commit go3! そして3つのオートマタgo1?と go2?と go3?を対応させます。この他にもいくつかの手順があります。

整数の配列は便利です。int a[3];のように宣言すると0から2までのインデックスをもつ配列となります。インデックスは典型的に変数int[0,2] i;とすることができます。

モデルを扱いやすくしておくために、若干のポイントに注意を払わなければなりません：

・ クロックの数が複雑さに重要な影響を与えます。
・ committed locations は際立って状態空間を減らすことができますが、適切な状態を取り去ることができるかもしれないので、この特徴について気をつけなくてはいけません。
・ 変数の数は重要な役割を演じます。例えばintegerは-32000 から 32000を超える範囲は取り扱えません。特にinteger では値が範囲を超えると無限ループとなるため、避けてください。

参照

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Proc. of the 7th International Conference on Formal Description Techniques, 1994.
[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In Proc. of Fundamentals of Computation Theory, volume 965 of Lecture Notes in Computer Science, pages 62{88, August 1995.
[JLS96] H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In Proc. of 2nd International Workshop on the SPIN Verification System, pages 1{20, August 1996.
[LPY97] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller: an Industrial Case Study using UPPAAL. In preparation, 1997.
[AD94] R. Alur and D. Dill. A Theory for Timed Automata In Theoretical Computer Science, volume 125, pages 183{235, 1994.

履歴

March 2001 First version by Alexandre David.
28 Apr 2001 Corrections by Alexandre David. Bug in a requirement, added: chan declaration, bug in declarations: int[0,1] req1,req2, turn; turn is int, not int[0,1]!

17 Dec 2001 Updates by Alexandre David. Added how to mark initial states (because the new UPPAAL does not make the first state initial by default anymore).
16 0ct 2002 Updates by Tobias Amnell. Changed screen-shoots to recent version (3.2.11), added verification walk-through in start-end example, added section on query language plus text updates on several places.

16 Sep 2005 Translated by T.Fujikura & K.Hirano　Ver 1.0