卒 業 論 文

UPPAALを用いた自動運転車の
群制御アルゴリズムのモデル化と検証

Modeling and verification
of autonomous-vehicle group control algorithms using UPPAAL

指導教員　中村 正樹 准教授

富山県立大学工学部 電子・情報工学科

学籍番号：1515024

氏名　佐原 優衣

提出年月　平成31年（2019年）2月

目 次

第1章　はじめに

1.1　背景

近年，自動運転技術が急速に発達している。自動運転は，搭載される技術によってレベル1からレベル5までに分けられており，現在，日本国内では，運転者支援を主としたレベル2までが市販車に採用されている。今後，高速道路や，限定地域での特定条件下での完全自動運転を行うレベル4の車両の普及が目指されている。

完全自動運転の普及の環境の一例として，アラブ首長国連邦において再生可能エネルギーを利用し，二酸化炭素を排出しないゼロカーボンを目指すマスダールシティプロジェクトが2006年に始まった。マスダールはアラブ首長国連邦の一つアブダビ首長国の首都アブダビの近郊で図1.1の様な人口約5万人，面積約6.5km^{2}の人工都市として計画されている。

図 1.1: マスダール・シティの完成イメージ [1]

このプロジェクトでは道路交通は自動運転車のみで構成される予定である。住民が任意の時刻に自動運転車に乗降し都市空間内を移動することを想定している

[1]出典：Masdar 社

ため，大量の車両の配備が必要となる。道路上の車両密度が高くなるため，渋滞やデッドロックが発生することが想定される。したがって，個々の車両だけではなく，自動運転車群が効率的に走行するアルゴリズムが必要となる。

本研究では群制御アルゴリズムが安全性に関わる衝突回避やデッドロック回避，効率性に関わる時間制約などの性質を満たすかどうかを検証する手法を提案する。

1.2 目的

自動運転車の群制御アルゴリズムを形式的に記述し，モデル検査を用いて，性質を検証する。モデル検査は，システム上で起こり得る状態を網羅的に調べることにより設計の誤りを発見する自動検証手法の一種である。モデル検査手法は，図 1.2 に示すように，システムの振る舞いの設計，および検証したい性質をそれぞれモデル化し，ツールを用いて，システムが性質を満たしているかを調べる。本研究では，時間オートマトンによる時間制約検証が行えるモデル検査ツール UPPAAL [6] を採用する。

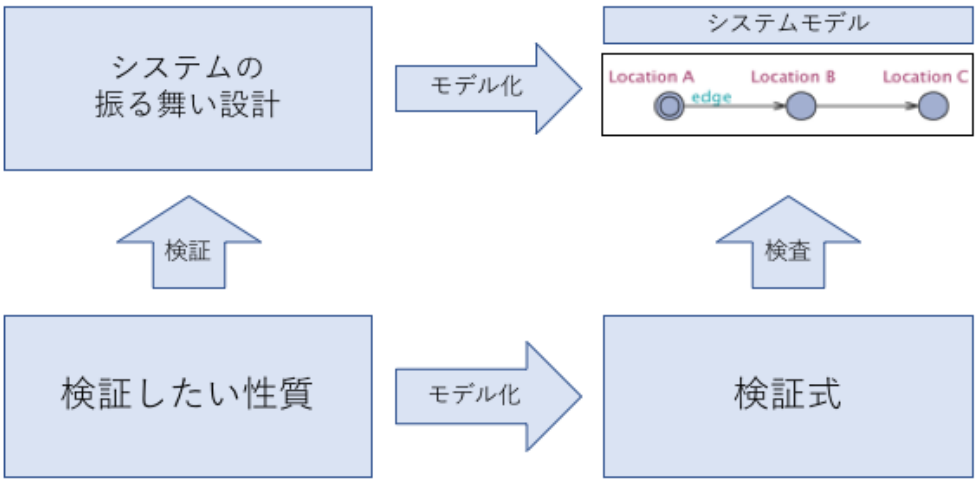

図 1.2: モデル検査による形式的な検証

1.3 論文の構成

本論文は，2 章で本研究で用いるモデル検査の概説と使用するモデル検査ツールの概説，3 章で単一の交差点における車両のモデル化と検証，4 章で検証にかかる時間や検証の質，5 章でまとめを述べるといった構成である。

第2章　モデル検査

本章では本研究で用いるモデル検査技術 [1] と UPPAAL の概説する。

2.1　モデル検査

モデル検査は，システム上で起こり得る状態を網羅的に調べることにより設計の誤りを発見する自動検証手法の一種である。モデル検査手法は，システムの振る舞いの設計，および検証したい性質をそれぞれモデル化し，ツールを用いて，システムが性質を満たしているかを調べる。

モデル検査において，システムの動作を表現するシステムモデルを作成する必要がある。ソフトウェア開発のどの段階でモデル検査を活用したいか，もしくは，何をどの程度検証したいかによって，どのような情報をもとにどのようにシステムモデルを作成するかが変わってくる。専用のシステムモデルを入力とするモデル検査を設計モデル検査，ソースプログラムを入力とするモデル検査をプログラムモデル検査と呼ぶ。これらのモデル検査がソフトウェア開発の流れの中での活用例を図 2.1 に示す。図 2.1 にはソフトウェアの品質向上のために行われる手順である設計レビュー，コードレビュー，およびテストも挙げた。設計モデル検査は設計レビューを，プログラムモデル検査はコードレビューをそれぞれ補完する位置付けである。

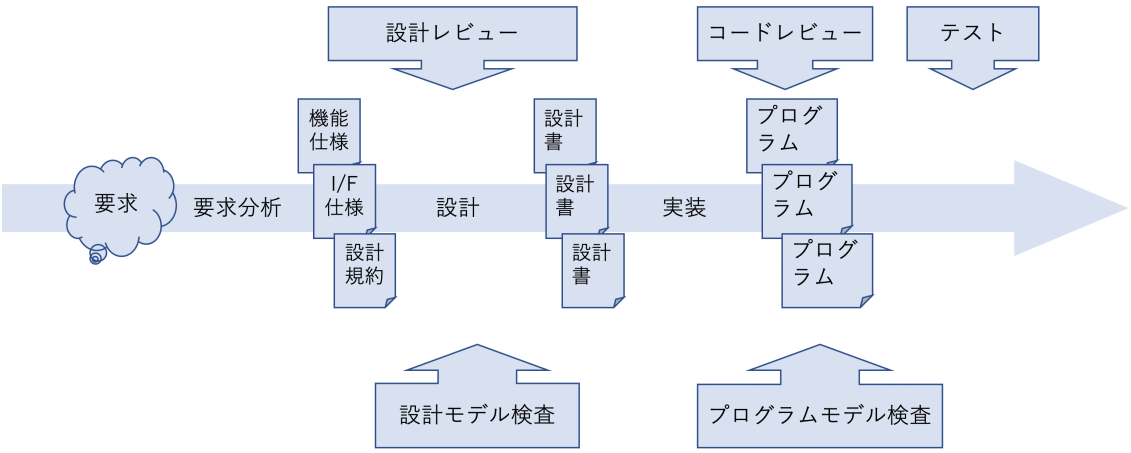

図 2.1: ソフトウェア開発プロセス

次に，いくつかのモデル検査ツールを特徴と共に例示する。代表的なモデル検査ツールには，処理が高速で大規模なモデルを扱える NuSMV [2]，並列処理やマ

ルチスレッドの設計モデルを扱える SPIN [4]，GUI ベースの入力による時間制約を扱える UPPAAL などがある。

2.1.1 NuSMV

NuSMV は状態遷移図からのモデル化に向いており，また，各状態が満たす論理式などを用いて記号的に検査を行う，莫大な状態数を持つ系に対して検査が可能なシンボリックモデル検査ツールである (図 2.2)。

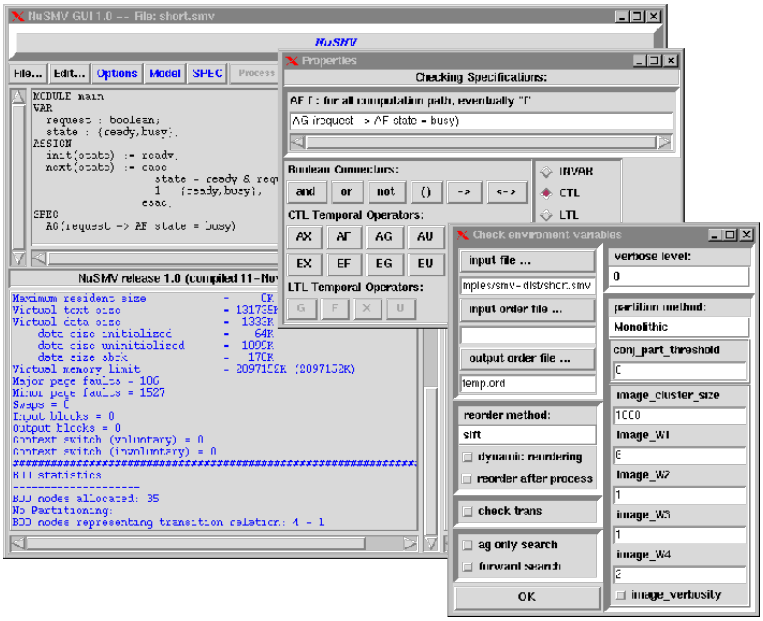

図 2.2: NuSMV のツール画面 [3]

2.1.2 SPIN

SPIN は Promela という専用の言語を用いて検査対象をモデル化する。SPIN の最大の特徴はモデル検査そのものを実施せず，対象を固有の C 言語ソースを生成する。メモリ使用量を削減し，性能向上と共に，モデルに使用者が自由に C 言語コードを追加できる利点がある (図 2.3)。

2.1.3 UPPAAL

UPPAAL の最大の特徴である時間が扱えることの利点を記述する。イベントの発生時刻や処理時間，これらの時間的なズレの三点について任意に設定できるため，時間が扱えないモデル検査と違い，応答時間などの時間制約を検証対象にす

```
// a small example spin model
 // Peterson's solution to the mutual exclusion problem (1981)

bool turn, flag[2];         // the shared variables, booleans
byte ncrit;                 // nr of procs in critical section

active [2] proctype user() // two processes
{
    assert(_pid == 0 || _pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);

    ncrit++;
    assert(ncrit == 1);     // critical section
    ncrit--;

    flag[_pid] = 0;
    goto again
}
// analysis:
// $ spin -run peterson.pml
```

図 2.3: SPIN のソースコード [5]

ることが可能である。また，イベントの発生と特定の処理の開始を簡単に記述できる (図 2.4)。具体例として，ビジネスプロセスの時間と資源に関する性質のモデル化と検証の適用例などがある [9]。

2.2 モデル検査ツール UPPAAL

本節は，UPPAAL について概説する。UPPAAL は作成したシステムモデルをの入力を GUI ベースにより定義している。このため，作成したシステムモデルが直感的に把握しやすい。入力したシステムモデルに対して，GUI ベースでシミュレーション実行とステップ実行が可能である。シミュレーション画面では，各プロセスの現在状態と変数の値，状態遷移図とメッセージシーケンスが表示される。

モデル検査による検証を利用する際は，検証したい性質を検証式で入力する。検証はすべての可能性のある実行パスに対して網羅的に検査を行う。検証結果は，入力した検証式に対して成否が示される。性質に反した場合は，反するまでの実行履歴が反例として示される。反例の表示はシミュレーション画面で行われ，ステップ単位でトレースすることで，各プロセスの状態や変数値の変化を確認可能である。また時間制約を含む検証に関して,「最短時間で違反状態に到達する反例の出力」という機能を持つ。通常出力されるのは任意の一種類ではあるが，特に初期状態から検証したい性質に反するまでの経過時間が最短となる反例を出力する機能である。検証したい性質として,「仕事が完了することがない」という条件を与えることにより，仕事が完了する手順が反例となるが，仕事が完了する手順の中で最短時間のものを出力することになる。

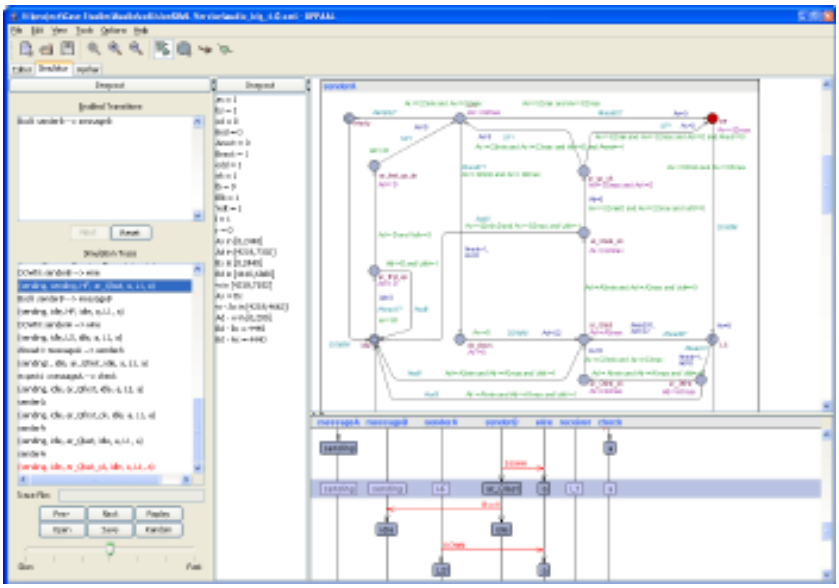

図 2.4: UPPAAL のシミュレーション画面 [7]

2.2.1 時間オートマトンの作成

本節では，例題を用いて UPPAAL を UPPAAL の各機能を紹介する。4 台の車が出発地点から到着地点まで，それぞれ違う方向から一つの交差点に進入し，4 台の車両全てが通過し終わるまでの間にデッドロックが起きないかを検証する。なお，この交差点には右折レーン，信号がないものとする。北から南に向かう車 ns，南から北にぬける車 sn，東から西にぬける ew，西から東にぬける we，の 4 台の挙動をモデル化し，検証を行う。

UPPAAL はシステムモデルを GUI 言語で作成する。UPPAAL では本例題における，車一台一台をエージェントと呼び，一台の車における出発地点から目的地まで通過するという一連の動作のことをプロセスと呼ぶ。同一の挙動を示すプロセスの定義を個別に行わなくてもすむように，テンプレートという概念を用いている。そのため，システムモデルの作成では，まずプロセスの挙動を定義したテンプレートの定義から行う。図 2.5 では現在選択しているものが青で囲われている。現在はエディタで，テンプレートが選択されている状態である。テンプレートの名前はエディタタブのすぐ下で定義できる。本例題では CarCouse とする。名前の隣にパラメータが入力部分がある。パラメータは参照渡しを行うことができる。同じテンプレート CarCouse のインスタンスとなる複数のプロセスで，プロセスごとにアクセスしたい変数が違う時に使う。

次は，変数定義を行う。テンプレートの一つ上の ” 宣言 ” で大域情報を入力する (図 2.6)。この編集ペインでは，大域定数，大域変数，大域時間変数，チャネルの情報を宣言できる。本例題では 2 つの大域変数と，大域時間変数を宣言している。テンプレートの一つ下層の宣言が各テンプレートに対して局所的な変数等の

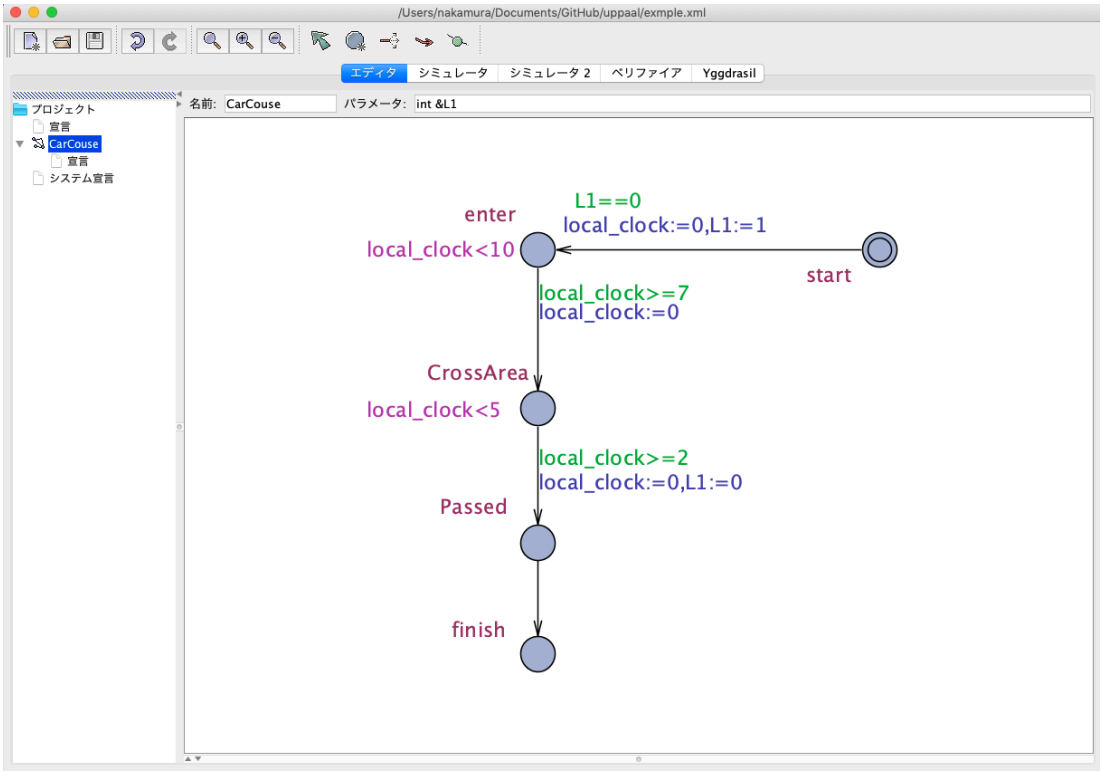

図 2.5: システムモデルの入力部分

宣言が可能である (図 2.7)。本例題では，各プロセスの経過時間，すなわち，車両が交差点を通過するのにかかる時間を記述するために，プロセスごとの時間変数 local_clock を宣言する。そして，テンプレート CarCouse を用いて，プロセスのインスタンスを宣言する。”システム宣言”のシステム定義編集ペインで各プロセス ns,sn,ew,we を図 2.8 のように定義する。最後に検証対象とするプロセスのインスタンスを列挙して，システムモデルの定義が完了である。

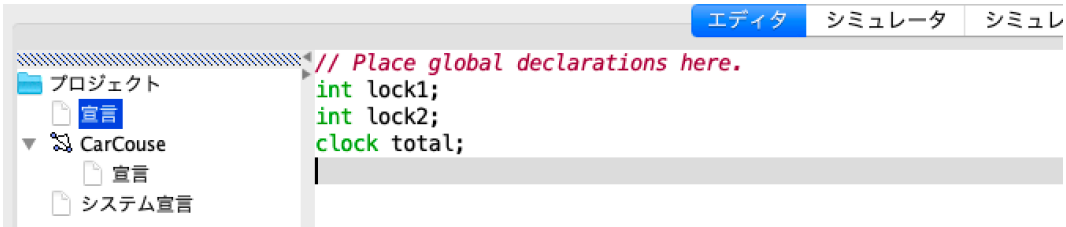

図 2.6: 大域情報編集ペイン

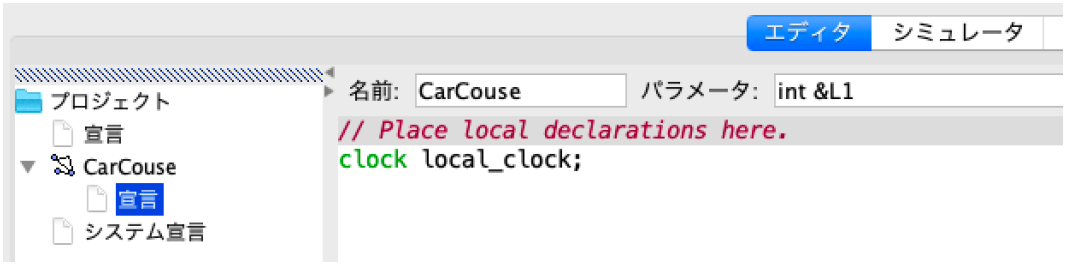

図 2.7: 局所的情報編集ペイン

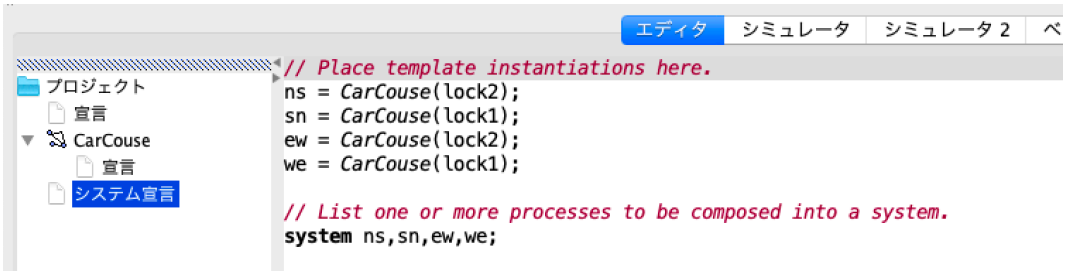

図 2.8: システム定義編集ペイン

2.2.2 シミュレーション

シミュレーションタブから，シミュレーション機能表示に切り替わる (図 2.9)。左上部は，現在の状態から次に遷移可能なプロセス名が表示される。そこから一つ選択すると，右上部の選択されたプロセスの時間オートマトンの遷移が実行される。遷移が実行されると，左下部では，遷移が記録されシミュレーショントレースとしてみれる。各プロセスの状態が (start,start,start,start) のように表示され，その次に遷移したプロセス名が表示される。その後，実行可能な遷移を繰り返すことで，ステップ実行が可能である。シミュレーショントレースから任意の状態から再生なども可能である。図中央部は現在状態で取り得る各変数値の範囲が表示されている。右下部は，シミュレーショントレースをシーケンスダイアグラム表現したものが表示される。

2.2.3 モデル検査

ベリファイアタブから，モデル検査機能表示に切り替わる (図 2.10)。入力した検証式を一覧で表示してあり，その下のクエリで検証式の編集を行う。検証したい性質を検証式を適切に書かなくてはいけない。検証方法には以下の 3 つの基本特性がある。

- 安全特性：すべての実行例で常に危険な動作をしないこと
- 活性特性：すべての実行例でいつかは要求された動作をすること

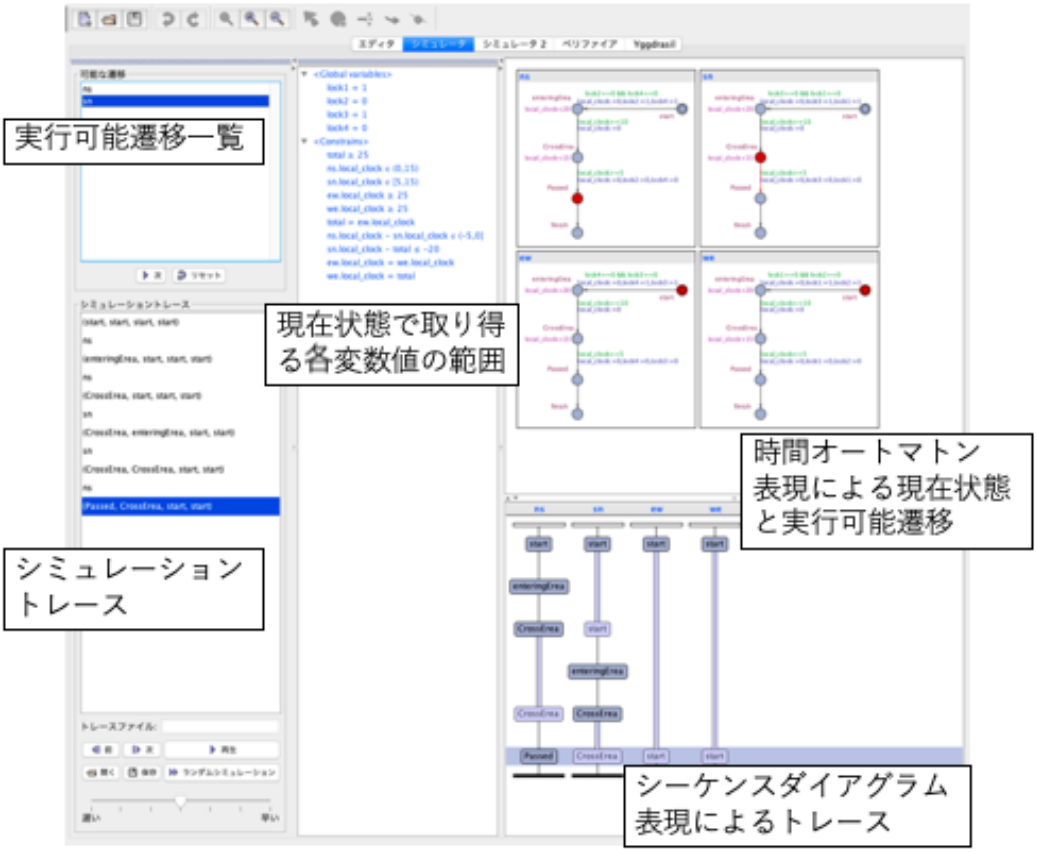

図 2.9: シミュレーション機能

- 到達可能性：ある実行例でいつかは要求された動作をすること

UPPAAL の検証式は時間付計算木論理の部分論理であるので，基本特性は次のように記述する。

- 安全特性：`A[] P` (すべての実行例で常に特性 P が成り立つ)
- 活性特性：`A<> P` (すべての実行例でいつかは特性 P が成り立つ)
- 到達可能性：`E<> P` (ある実行例でいつかは特性 P が成り立つ)

図 2.10: モデル検査機能

第3章　群制御アルゴリズムのモデル化と検証

本章では，UPPAAL を用いて交差点を通過する 1 台の自動運転車の挙動をモデル化する。交差点は 2 車線対面通行で右折用レーンはなく，信号もない交差点とする。3.1 節では時間は扱わず，任意の方向へ進む車両をモデル化する。3.2 節では時間に関する条件を用いることで，交差点通過時間を記述する。3.3 節では，全ての車両が交差点を通過するのに掛かる最小時間を検証する。

3.1　時間制約のない進行方向を固定しない交差点

本節では，経路選択をして交差点を通過する車両のモデルを作成する。信号のない交差点を無秩序に通過すると衝突などが起きる可能性がある。したがって今回は交差点に使用権というものを設定し，交差点を通過するにはこの使用権が取得できた車両が通過できることとする。

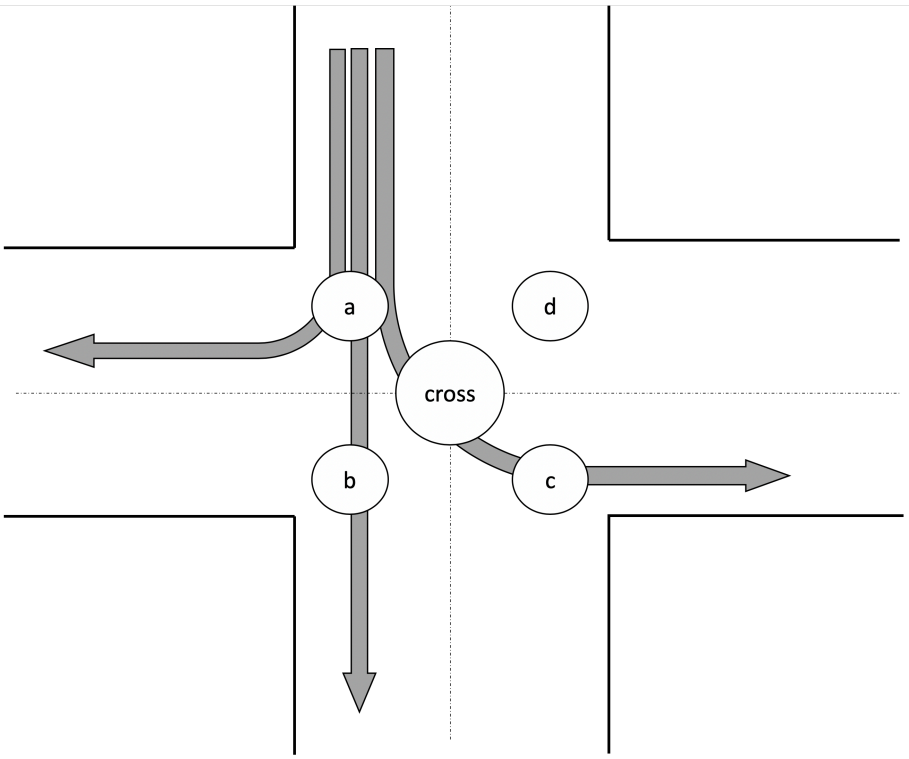

図 3.1: 交差点における使用権の鍵の組み合わせモデル

交差点に 5 つの鍵を設定し，その組み合わせで交差点を通過可能とする。図 3.1 は 5 つの鍵を用いた交差点の車両の進行モデルである。図上からの進入に対して，直進，右折，左折の選択肢があり，直進は鍵 a,b を，右折は鍵 a のみを，左折は鍵 a,c,cross をそれぞれ取得する。該当の鍵を取得可能な時，その進行方向へ進入可能となる。図 3.1 を基にして，1 台の車両の交差点通過を表すオートマトンを作成する。

3.1.1 時間オートマトンの作成

図 3.2 は初期状態 Start から交差点に進入し，通過後 Start に戻ることで，上下左右の全ての方向から直進，右折，左折を非決定的に選択することを繰り返す 1 台の車両のオートマトンである。Start から交差点進入前状態 rightOver に遷移したとき，車両は，直進，左折，右折の 3 つの選択肢がある。直進するときは，交差点通過中状態 a4 への遷移可能条件として大域変数 a==0 かつ,b==0 すなわち使用されていないとき，遷移時に a=1 と b=1 として使用権を取得し，b4 への遷移時に a=0 として使用権の一部を解放する。そして，b4 から交差点通過後状態 UnderF への遷移時に b=0 としてもう 1 つの使用権も解放する。右折するときは，通過中状態 a1 への遷移可能条件を a==0 とし，遷移時に a=0 と更新し，使用権を取得する。通過後状態 RightF への遷移時に a=0 として使用権を解放し Start に戻る。左折するときは，通過中状態 a3 への遷移条件を a==0 かつ，c==0 かつ，cross==0 とし，遷移時に 3 変数を 1 に更新して使用権を取得する。通過中状態 c3 への遷移時に a=0 として鍵 a を解放し，c3 から通過後状態 LeftF への遷移時に使用権を解放し，Start に戻る。以上の遷移を交差点に進入する 4 方向についてすべて記述する。

3.1.2 シミュレーション

前節で作成した時間オートマトンのインスタンスを 2 つ宣言し，シミュレーションを行う。図 3.4 では，Car1 は leftOver から右折して交差点を通過中状態で，Car2 は rightOver で交差点進入前状態である。このとき大域変数は図 3.5 となっている。したがって，Car2 は直進右左折それも選択可能となっている (図 3.6)。

3.1.3 モデル検査

UPPAAL では，どれだけ時間が経過しても，いずれのプロセスも遷移できないことをデッドロックという。本モデルでデッドロックが起きないか検証する。本例題におけるデッドロックは同じ鍵を同時に使うことや，Start に戻ってこれないこ

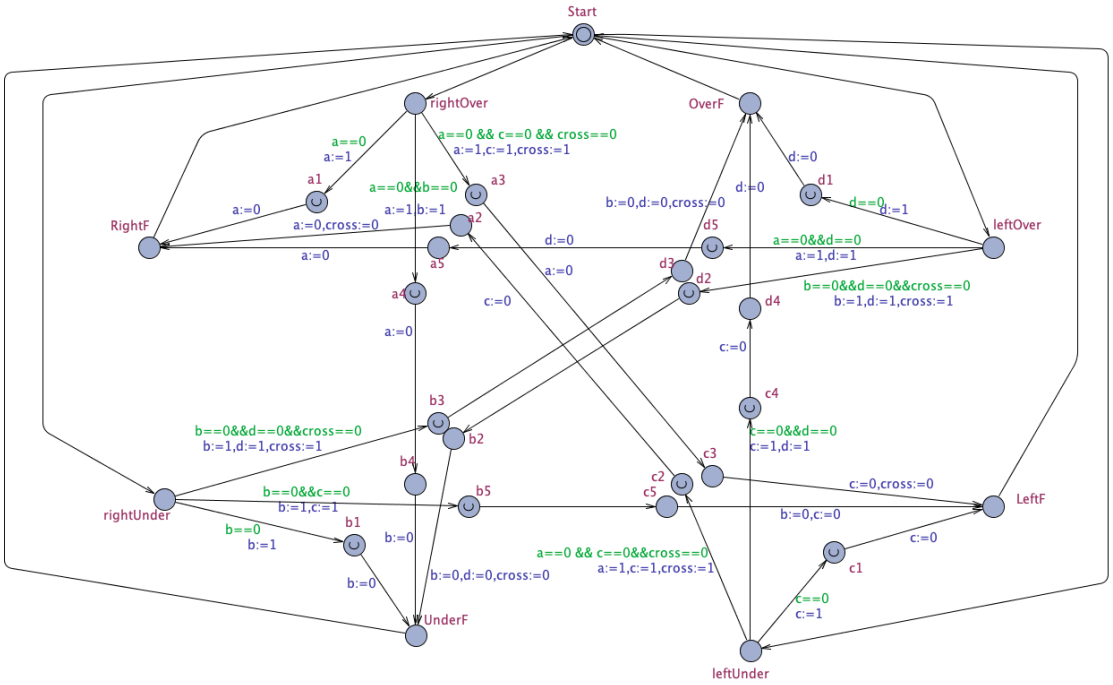

図 3.2: 進行方向を固定しない交差点の時間オートマトン

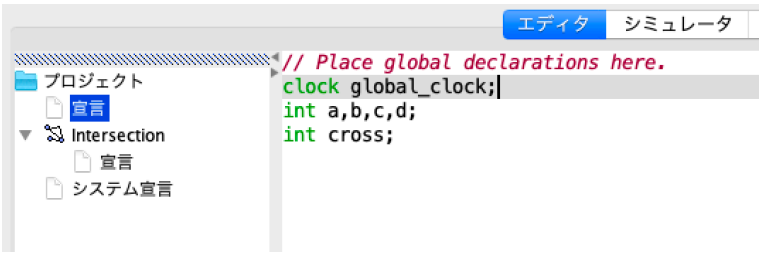

図 3.3: 時間制約のない交差点の時間オートマトンの大域情報

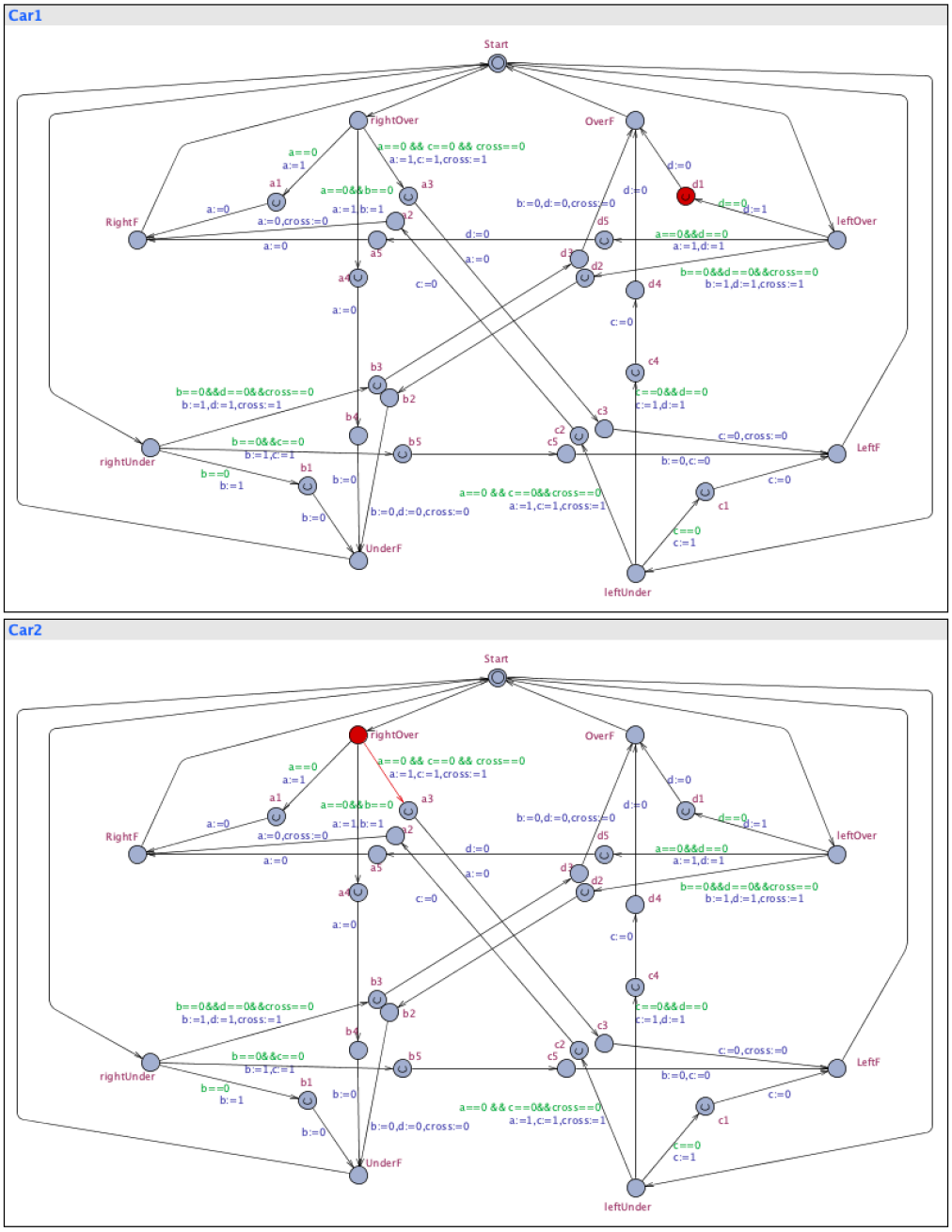

図 3.4: 時間制約のない交差点の時間オートマトン

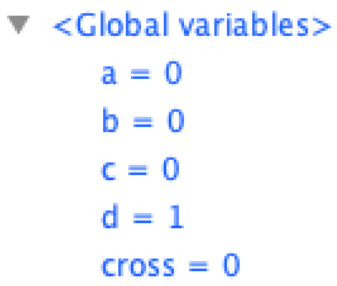

図 3.5: 時間制約のない交差点の時間オートマトンの大域変数値

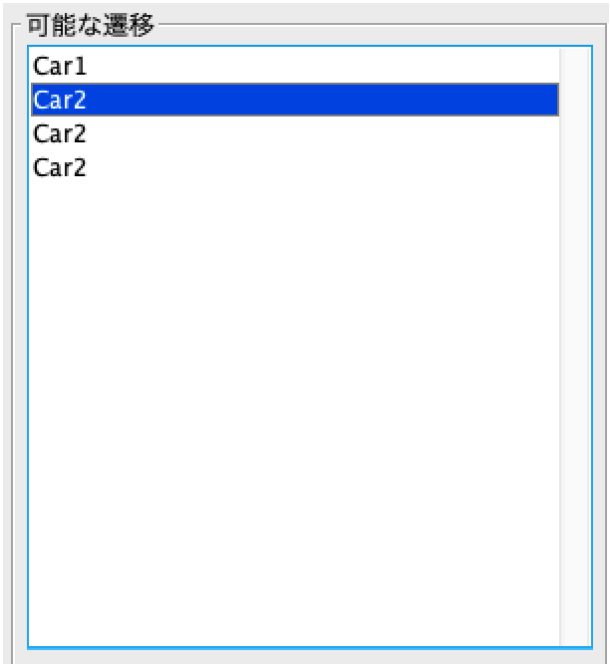

図 3.6: 時間制約のない交差点の時間オートマトンの実行可能遷移一覧

とを指す。すべての実行列で常にデッドロックにならないことを表す検証式は次のように記述される：

```
A[] not deadlock
```

検証結果は図 3.7 のように示される。

プロセスの車両インスタンスを 1 から 6 まで 1 ずつ増やしながらデッドロック検証を行った。表 3.1 は車両の台数に対する検証にかかった時間である。図 3.8 はそれをグラフ化したものである。

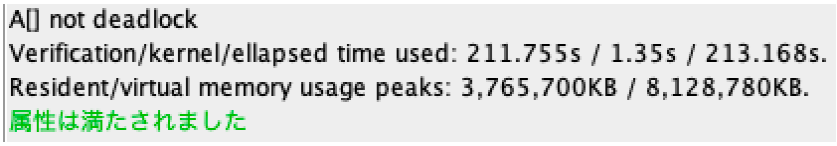

図 3.7: 進行方向を固定しない交差点のデッドロック検証結果

表 3.1: 車両の台数とデッドロック検証にかかる時間

車両の台数	検証時間 (s)
1	0.00
2	0.004
3	0.068
4	0.808
5	10.873
6	211.755

3.2 時間制約を用いる進行方向を固定する交差点

本節では，交差点に進入する車両の進行方向を固定して，時間オートマトン [8] を作成する。車両 1 台の挙動は，交差点進入前，交差点通過中，交差点通過後の 3 つの状態で記述する。交差点進入前に交差点の使用権を取得し，通過後に使用権を解放する。遷移可能条件と状態不変条件に時間に関する条件を与えることによって，通過にかかる時間や使用権を何秒前に取得しなければならないかを記述できる。

3.2.1 一方通行の 2 車線で構成される交差点モデル

一方通行の交差点で互いの進行方向が交わる時，交差点を図 3.9 のようにモデル化する。cross は交差点の使用権を表現しており，使用権を取得できた車両が交差点を通過するモデルである。

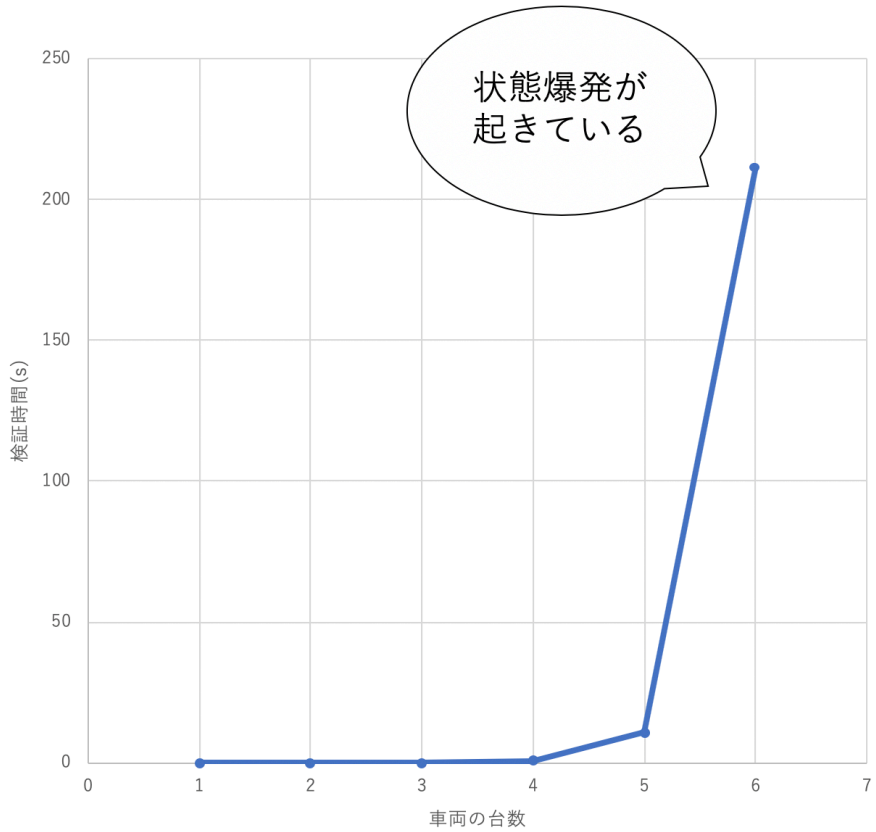

図 3.8: 車両の台数と検証にかかる時間の関係

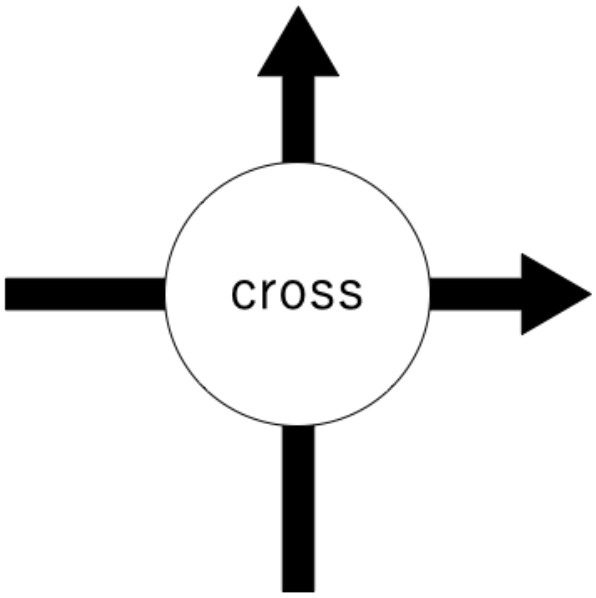

図 3.9: 交差点の使用権

時間オートンマトンの作成

交差点に直進する車両 2 台の進行方向が互いに交差するとき，衝突回避するためには交差点に同時に進入するのは 1 台までにする。交差点進入時に使用権を取得する車両の時間オートマトンを作成する (図 3.10)。respawn は車両の初期状態かつ，任意の時刻に車両が発生する状態である。BeforeEnter は車両の交差点進入前の状態である。crossArea は交差点通過中の状態である。Passed は交差点通過後の状態である。respawn から BeforeEnter への遷移可能条件の cross==0 は交差点の使用権が未獲得であることを示しており，遷移時に cross==1 と更新して交差点使用権を獲得する。同時にタイマーである local_clock を 0 にして，BeforeEnter の時間を測定する。BeforeEnter から crossArea の遷移でも同様にし，交差点手前の約 5 秒から 10 秒前までに使用権を獲得し，交差点通過に 2 秒から 5 秒弱かかるということが記述されている。

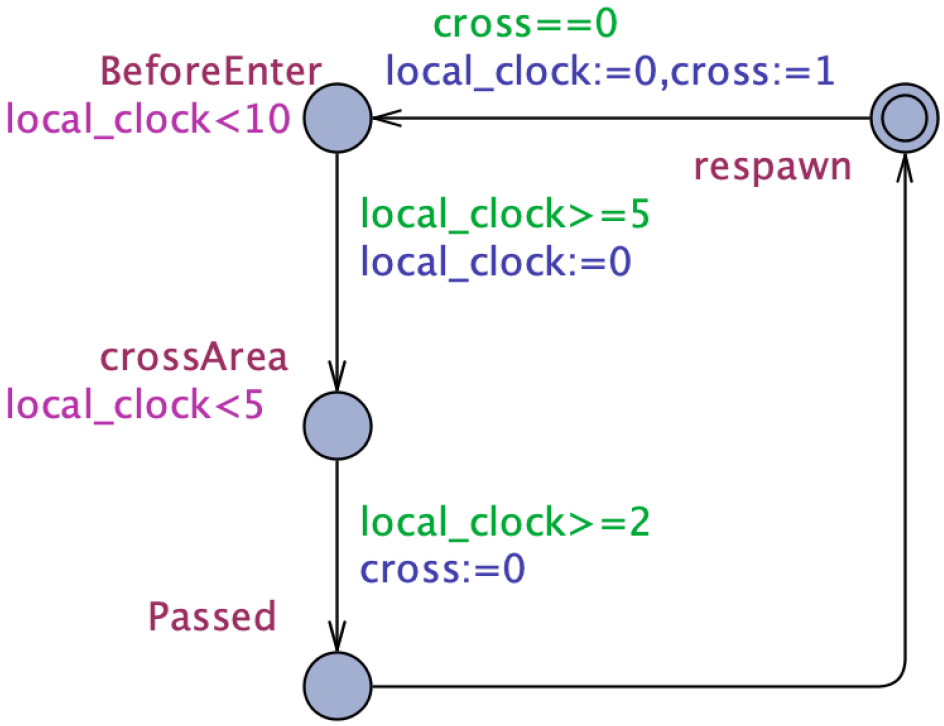

図 3.10: 交差点進入時に使用権を取得する車両 1 台の時間オートマトン

シミュレーション

図 3.11 は，前節で定義したテンプレートから生成された 2 つのインスタンスが合成された時間オートマトンに対するシミュレーションのスクリーンショットである。左のプロセス vertical を縦方向に直進してるとし，右のプロセス horizon を横

方向に直進しているとする。現在状態は vertical が交差点の使用権を獲得し，進入前状態で，horizon が交差点通過後の状態である。

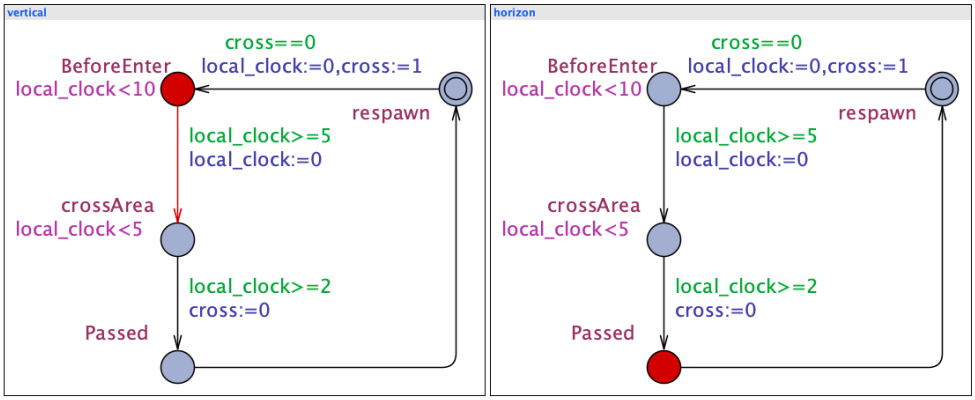

図 3.11: 垂直に交差点に進入する車両の時間オートマトンの合成

モデル検査

このモデルでデッドロック検証を行った結果を図 3.12 に示す。

```
A[] not deadlock
Verification/kernel/ellapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 4,928KB / 4,317,824KB.
属性は満たされました
```

図 3.12: 一方通行の 2 車線で構成される交差点のデッドロック検証結果

3.2.2 東西南北それぞれから交差点に直進する時間オートマトン

直進して交差点を通過する車両の進行方向に対して，他の車両の進行方向が垂直であったり，平行であったりする時の交差点の使用権の獲得方法を考える。前例では，cross が交差点の使用権そのものでそれの有無で進入を決定していたが，本例では，進行方向が平行となる場合は同時に進入可能としたい。したがって交差点の使用権を図 3.13 に示すように 4 つの鍵の組み合わせで管理したい。例えば，西から東へ進行する車両は lock1 と lock2 を取得する。北から南へ進行する車両は lock2 と lock4 を取得しようとするが，既に lock2 が取得されているためこの車両は交差点へ進入できない。一方で前者に対して，東から西へ進行する車両は lock3 と lock4 を取得できるため，交差点へ進入する。

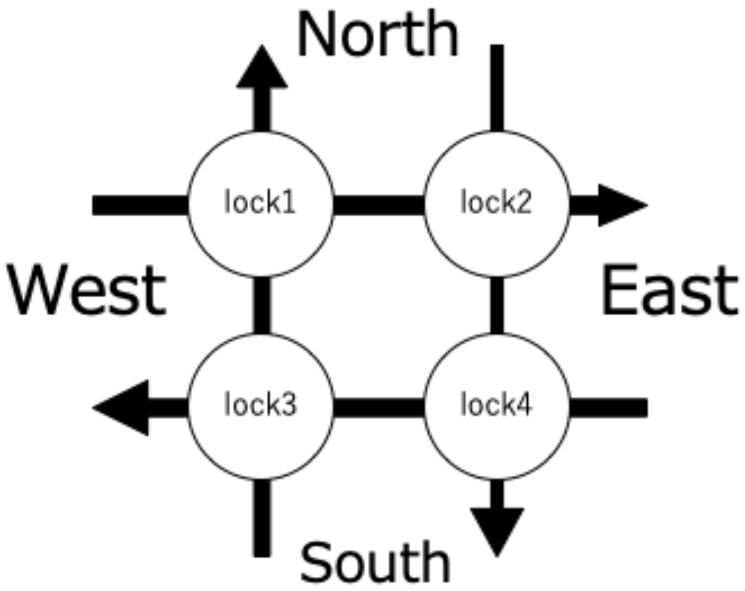

図 3.13: 交差点の使用権を管理する 4 つの鍵の組み合わせ

時間オートマトンの作成

使用権を 4 つの鍵で管理する交差点に直進する車両の時間オートマトンを作成する (図 3.14)。遷移可能条件をパラメータ (L1,L2) で記述し，それぞれが取得する鍵に紐付けている。初期状態 respawn から交差点進入前状態 BeforeEnter へ遷移時にふたつの鍵 L1 と L2 を 1 に更新して使用権を取得する。前例と同様にこのオートマトンも直進のみのため，タイマーである local_clock は同内容を記述している。交差点通過中状態 crossArea から交差点通過後状態 Passed への遷移時に鍵の解除として L1==0 と L2==0 と記述した。

シミュレーション

北から南へ直進する車両を sn，南から北へ直進する車両を ns，東から西へ直進する車両を ew，西から東へ直進する車両を we として，インスタンスの宣言をした (図 3.16)。図 3.15 では，ew が交差点を交差点通過中で，lock3 と lock4 が取得されている。lock1 と lock3 を取得できた we が交差点進入前状態である。

モデル検査

デッドロック検証を行った結果を図 3.17 に示す。

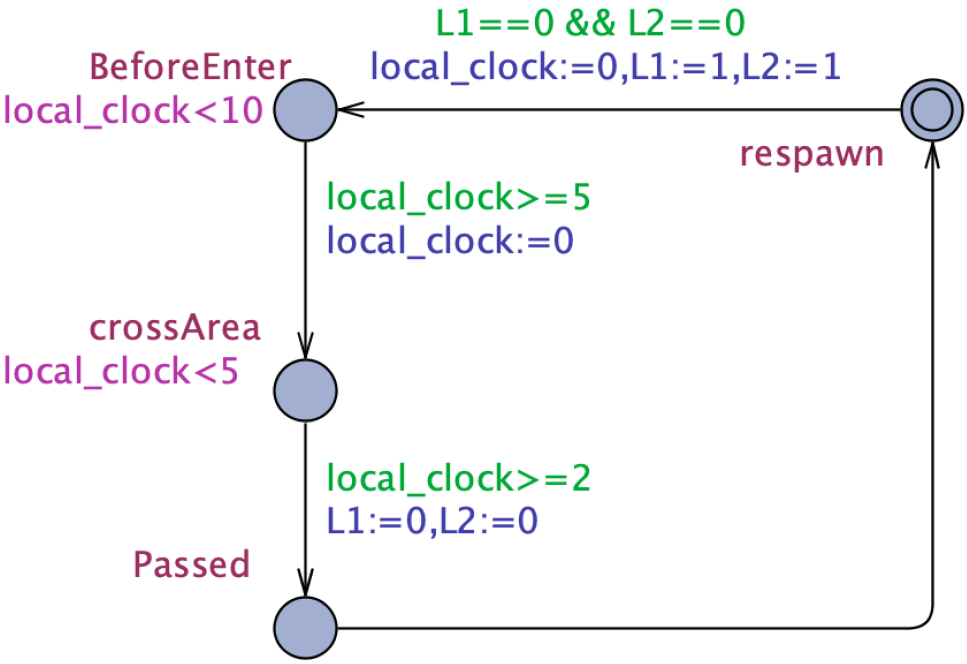

図 3.14: 使用権を 4 つの鍵で管理する交差点に直進する車両の時間オートマトン

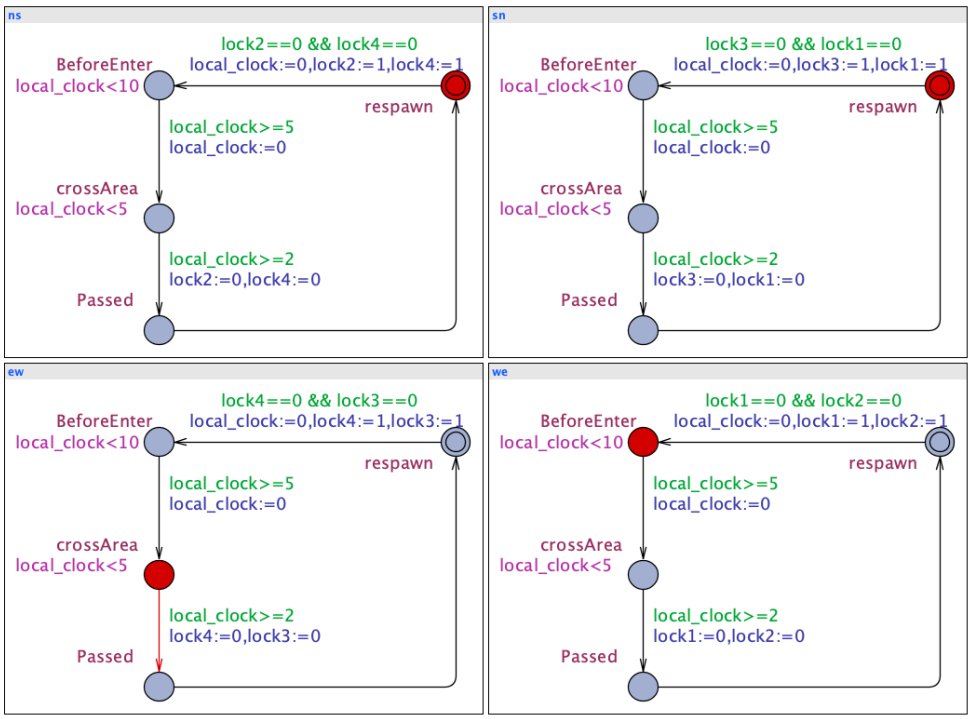

図 3.15: 交差点に直進する車両の時間オートマトンの合成

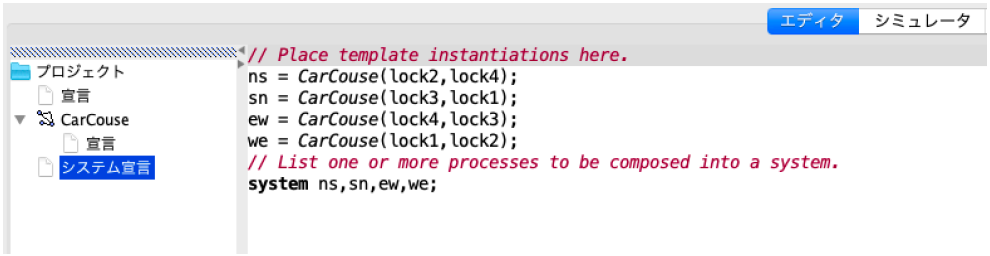

図 3.16: 4 車両のシステム宣言

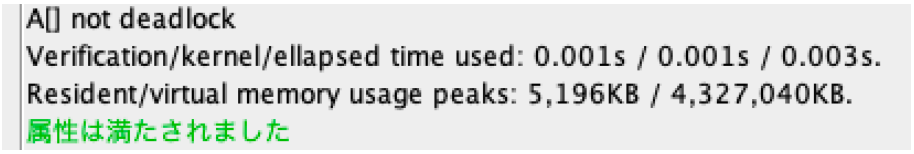

図 3.17: 4 車線で構成される交差点のデッドロック検証結果

3.2.3 交差点を直進・左折・右折する車両の時間オートマトン

交差点を直進する車両と左折する車両と右折する車両を考慮したモデルを考える。前述までとは違い，右折する場合もあるため，4 つの鍵の組み合わせでは右折する車両同士での衝突が起こる可能性がある。したがって前述の 4 つの鍵に加えて，5 つ目の鍵を右折用の鍵として cross を用意したモデルを考える。このモデルで同時通行可能な組み合わせの 2 つの例を図 3.18 と図 3.19 に示す。

時間オートマトンの作成

このモデルから時間オートマトンを作成する (図 3.20)。鍵 cross は 3 つ目のパラメータ use の値と照らし合わせる。パラメータ use は右折用の鍵を使用するかどうかを車両の固定情報として保持している。初期状態 respawn から交差点進入前状態 BeforeEnter への遷移可能条件をパラメータ L1，L2，右折用鍵 cross が取得可能であることとした。また，左折時は，パラメータは 2 つあるが，取得する鍵は 1 つである。そのため，南から東へ右折する車両と，東から南へ左折する車両と，西から北へ左折する車両の 3 台が同時に交差点に進入可能となっている。

シミュレーション

北から南へ直進する車両を sn，南から北へ直進する車両を ns，東から西へ直進する車両を ew，西から東へ直進する車両を we，北から東へ左折する車両を ne，南から西へ左折する車両を sw，東から南へ左折する車両を es，西から北へ左折する車両を wn，北から西へ右折する車両を sw，南から東へ右折する車両を se，東から北へ右折する車両を en，西から南へ右折する車両を ws として，インスタンスの

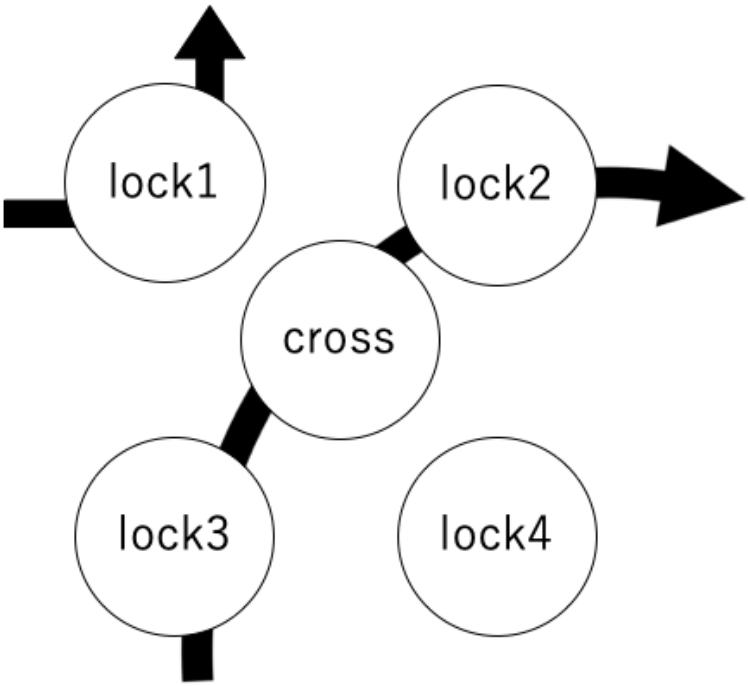

図 3.18: 使用権の取得例：右折と左折

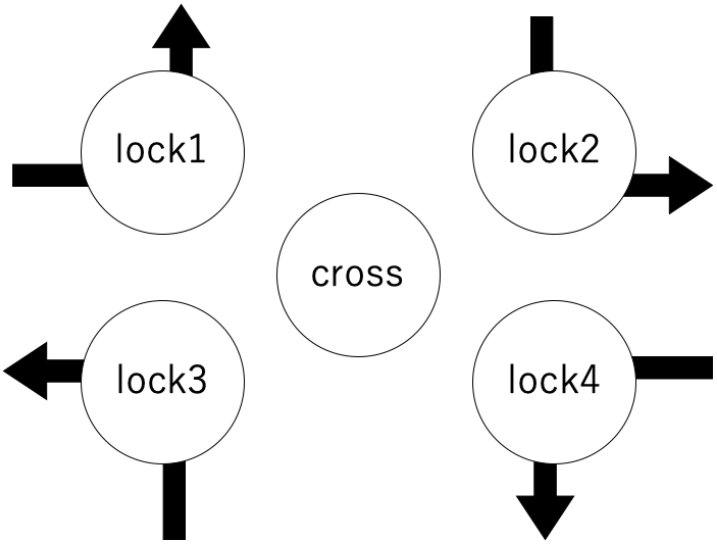

図 3.19: 使用権の取得例：4 台の左折

宣言をする (図 3.22)。上記の 3 台の車両が交差点に進入する例が図 3.21 のように，右折する se が通過中状態 crossArea の時，左折する es も状態 crossArea で，もう一つの wn が状態 BeforeEnter となっている。

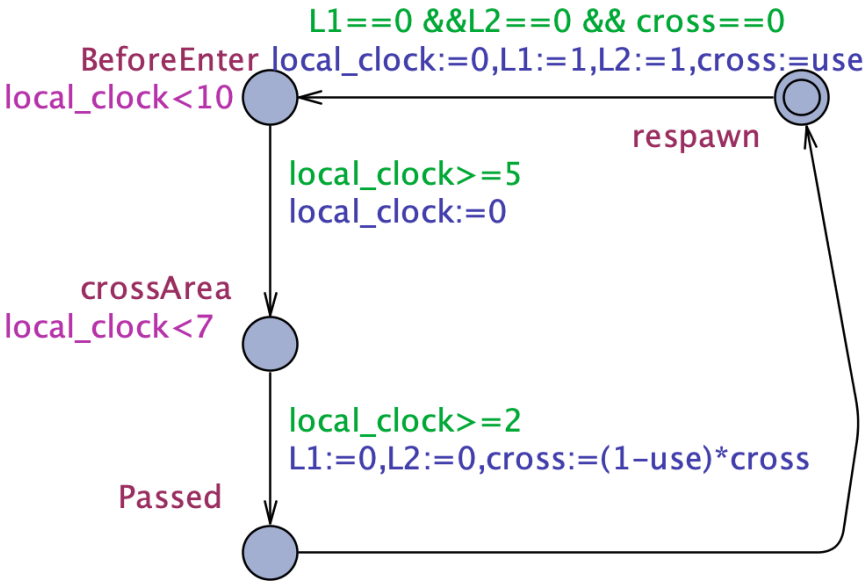

図 3.20: 交差点を通過する車両の時間オートマトン

モデル検査

デッドロック検証を行った結果を図 3.23 に示す。

3.3 交差点の通過にかかる最小時間の検証

本節では，車両全てが交差点を 1 回通過するのにかかる最小時間について検証を行う。交差点の使用権の取得方法は前節と同仕様の 5 つ鍵によって管理する。1 回だけなので循環するオートマトンではなく一方通行的なオートマトンを作成する (図 3.24)。初期状態 start から交差点進入前状態 BeforeEnter への遷移可能条件に交差点の使用権の取得として，L1==0 かつ L2==0 かつ cross==0 を与える。状態 BeforeEnter から交差点通過中状態 crossArea への遷移可能条件と BeforeEnter の状態不変条件で交差点進入前 7 秒以上 10 秒未満の間で使用権を獲得しなければならないかを記述し，状態 crossArea から交差点通過後である終了状態 finish への遷移可能条件と crossArea の状態不変条件で交差点の通過にかかる時間を記述し，crossArea から finish への状態遷移時に使用権の解除を行う。

最小時間の検証を行う。車両 1 台の start から finish までプロセスにかかる最小時間は 7 秒である。車両は全部で 12 台のため単純に考えると 84 秒で終わるかと

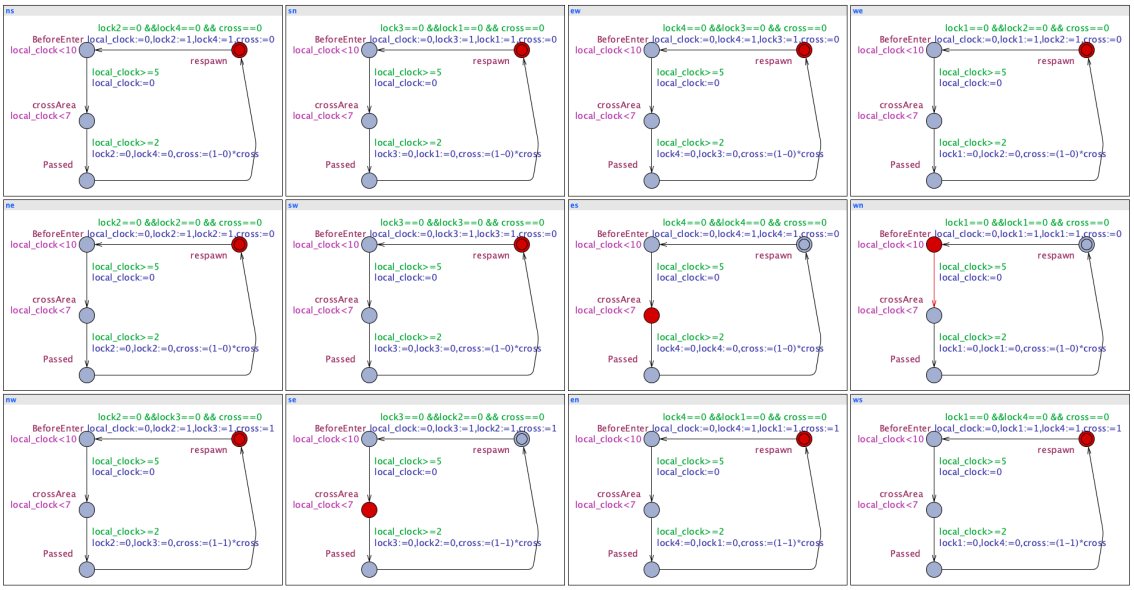

図 3.21: 交差点を通過する車両の時間オートマトンの合成

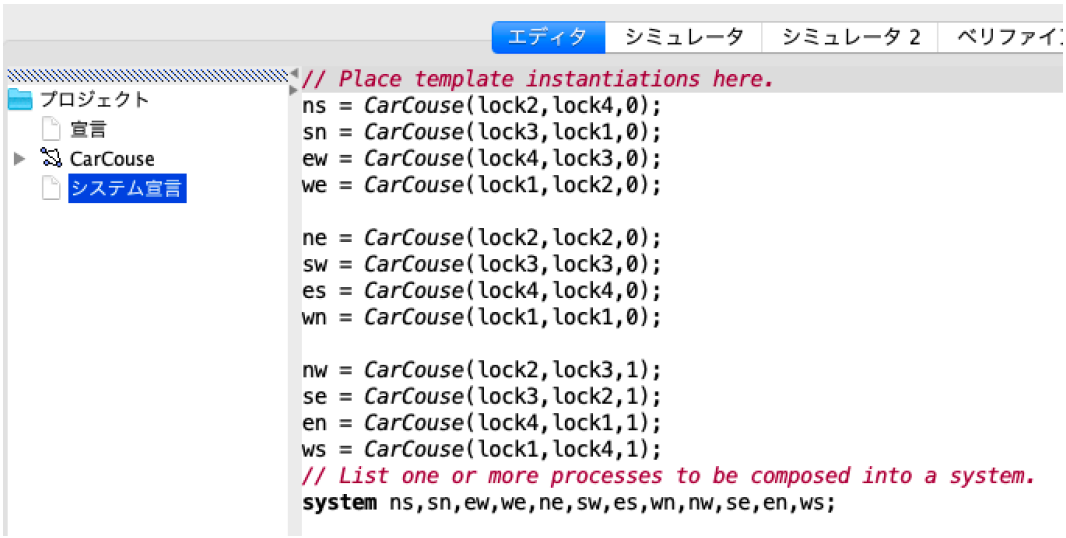

図 3.22: 12 車両のインスタンスの宣言

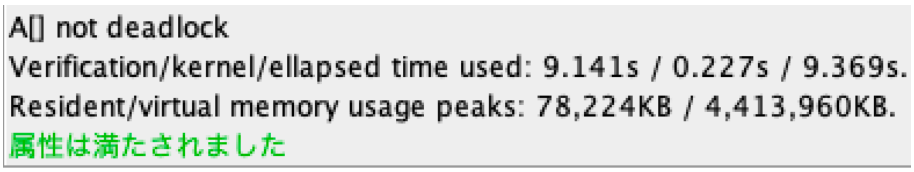

図 3.23: 12 車両で構成される交差点のデッドロック検証結果

考えられるが，同時通行可能なプロセスもあるため，もう少し少ないと考えられる。例えば図 3.19 のように 4 台同時通行もあり得る。直線が 2 台ずつ，左折が 4 台まとめて，右折が 1 台ずつと考えると，7 台分の時間に短縮されると考え，最小時間は 49 秒と推測し，検証を行う。まず，49 秒で本当に全過程が終わる可能性があるかどうかを次の検証式を用いて検査する。検証のために大域時間変数 gc を宣言する。

```
E<> (gc==42 and ns.finish and sn.finish and ew.finish
and we.finish and ne.finish and sw.finish and es.finish
and wn.finish and nw.finish
and se.finish and en.finish and ws.finish)

A[] (gc<42 imply not (ns.finish and sn.finish
and ew.finish and we.finish and ne.finish and sw.finish
and es.finish and wn.finish and nw.finish and se.finish
and en.finish and ws.finish))
```

その結果 42 秒が最小であることが求められた。以下の検証式が成り立つ。その時のルートは図 3.25 の手順だった。

```
E<> (gc==42 and ns.finish and sn.finish and ew.finish
and we.finish and ne.finish and sw.finish and es.finish
and wn.finish and nw.finish
and se.finish and en.finish and ws.finish)

A[] (gc<42 imply not (ns.finish and sn.finish
and ew.finish and we.finish and ne.finish and sw.finish
and es.finish and wn.finish and nw.finish and se.finish
and en.finish and ws.finish))
```

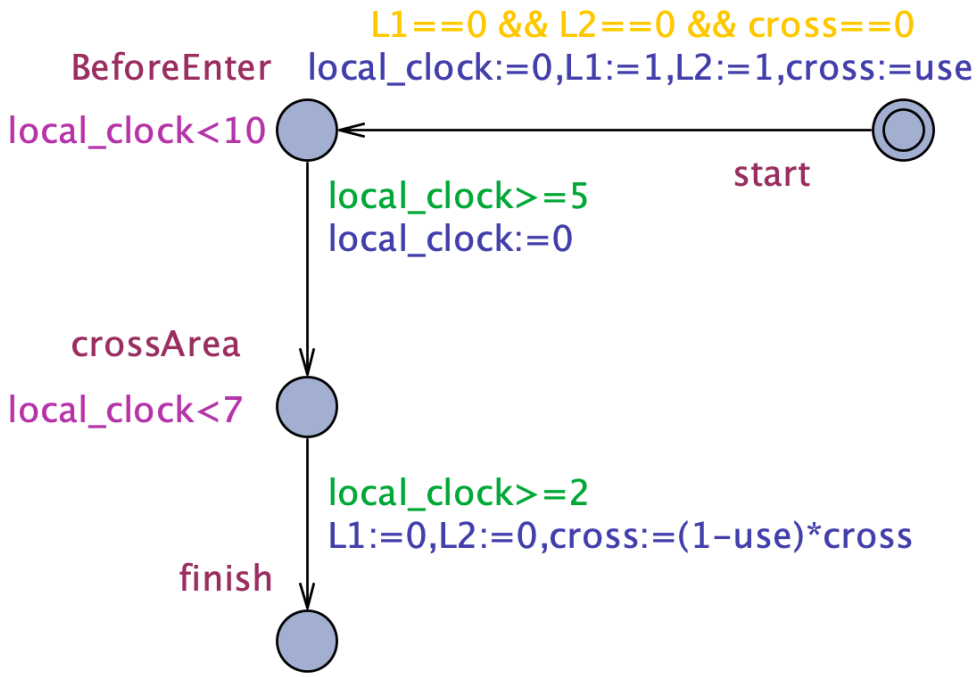

図 3.24: 交差点を 1 回通過する時間オートマトン

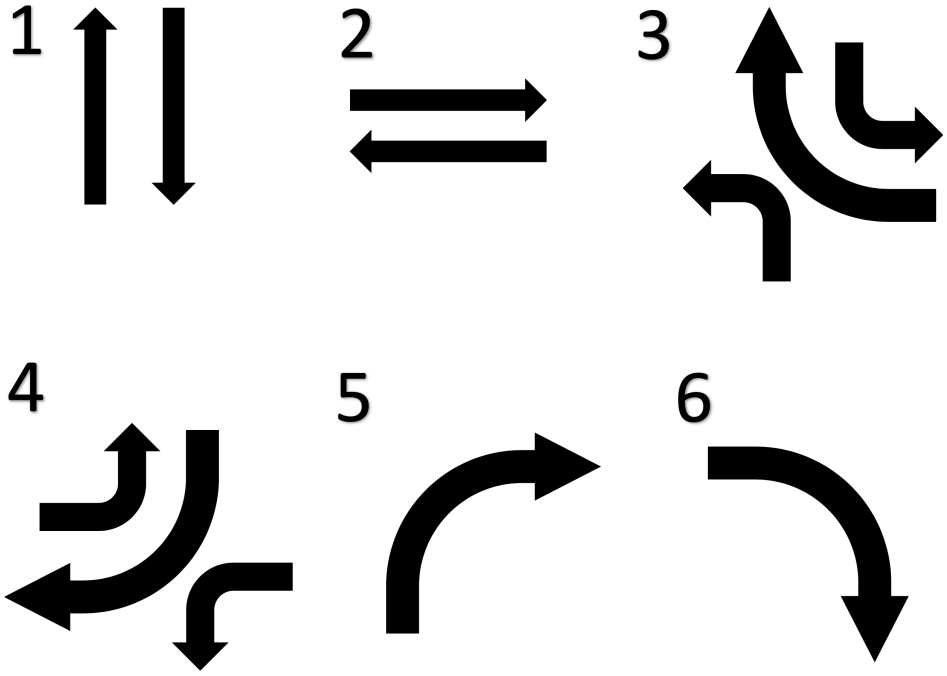

図 3.25: 最小時間のルート

第4章　おわりに

本研究では，UPPAALを用いた自動運転車群制御アルゴリズムのモデル化と検証の手法を提案した。単一の交差点においては車両の挙動をモデル化し，デッドロックや通過時間を検証することができた。複数の交差点から構成される都市空間のモデルを作成し検証することが今後の課題である。

謝 辞

本研究を進める上で，多くのご助言とご指摘をいただきました中村准教授，榊原准教授に心より感謝の意を表します。また，多くの助言と協力をいただきました中村研究室，榊原研究室の皆様にも深く感謝いたします。

参 考 文 献

[1] 長谷川哲夫, 田原康之, 磯部祥尚, UPPAAL による性能モデル検証ーリアルタイムシステムのモデル化と検証ー, 近代科学社, 2012.

[2] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella,NuSMV 2: An OpenSource Tool for Symbolic Model Checking,In Proceeding of International Conference on Computer-Aided Verification (CAV 2002), pp. 359-364, 2002.

[3] NuSMV home page, `http://nusmv.fbk.eu`

[4] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.

[5] Spin - Formal Verification, `http://spinroot.com`

[6] Uppaal in a Nutshell. Kim G. Larsen, Paul Pettersson and Wang Yi. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[7] UPPAAL, `http://www.uppaal.org`

[8] Timed Automata: Semantics, Algorithms and Tools, Johan Bengtsson and Wang Yi. In Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.

[9] 綿引健二, 石川冬樹, 平石邦彦, 時間, 資源の制約をもつビジネスプロセスの形式検証, 電子情報通信学会論文誌 D, 96(8):1878-1891, 2013.