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

自動運転技術は発達し続けている。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??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。

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。

しかし，効率的に制御された自動運転車が安全かどうかアルゴリズムから一目では判断しきれない。そこで本研究では群制御アルゴリズムが欲しい性質を持っている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。

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。

[1]出典：Masdar 社

図 1.1: マスダール・シティの完成イメージ [1]

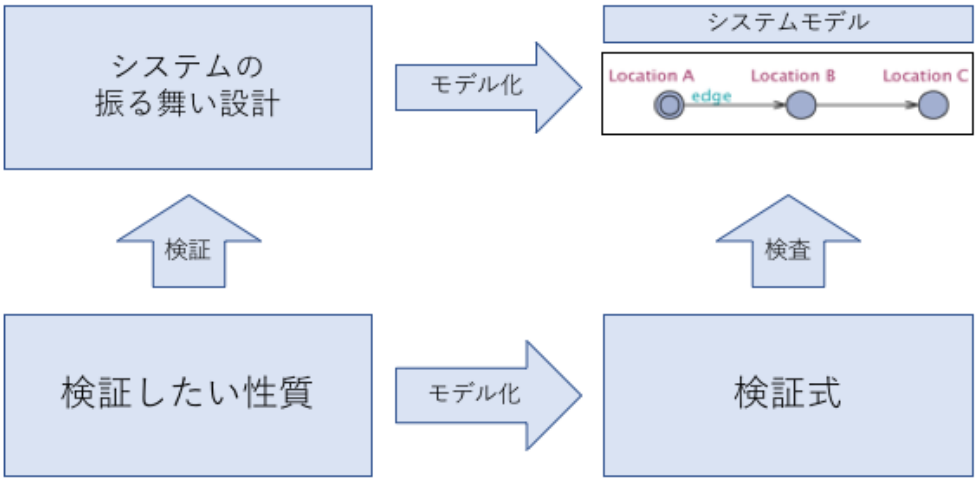

図 1.2: モデル検査による形式的な検証

第2章　モデル検査

本章では文献 [?] からモデル検査と UPPAAL [?] の概説を行う。

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??に示す。図??にソフトウェアの品質向上のために行われる手順も挙げた。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。

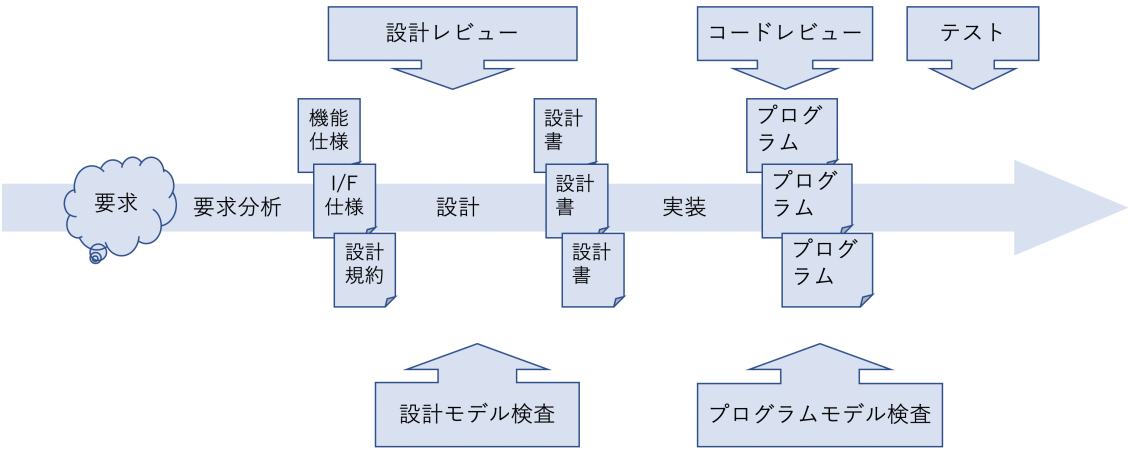

図 2.1: ソフトウェア開発プロセス

2.1.1 様々なモデル検査ツール

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

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??)。

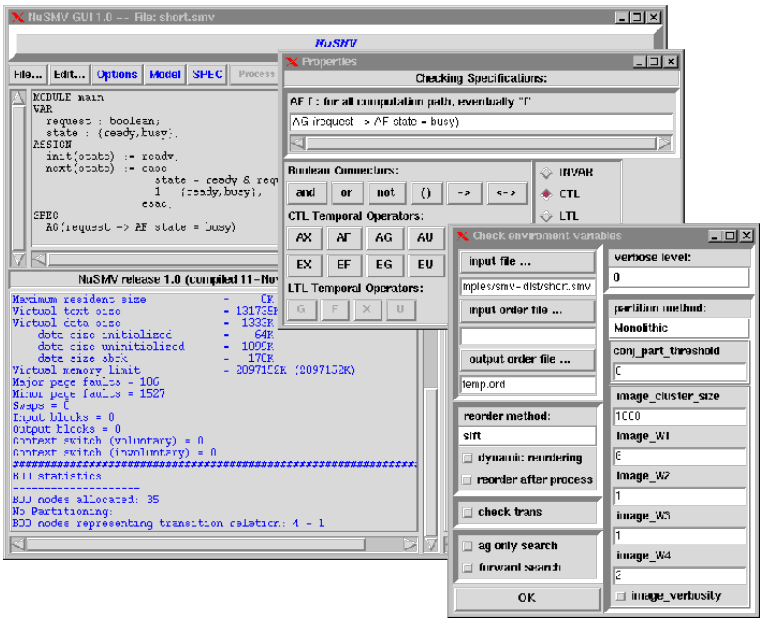

図 2.2: NuSMV のツール画面 [?]

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??)。

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。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 [?]。

```
// 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 [?]

2.2 モデル検査ツール UPPAAL

本節は，UPPAAL について概説する。UPPAAL はシステムモデルの入力を 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。

検証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緑か赤で示される。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。

2.2.1 UPPAAL の操作

本節では，例題を用いて UPPAAL を使用する。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，通過するとき 4 台の車が全て通過し終わるのにかかる最小時間を検証する。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，一台の車の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??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。

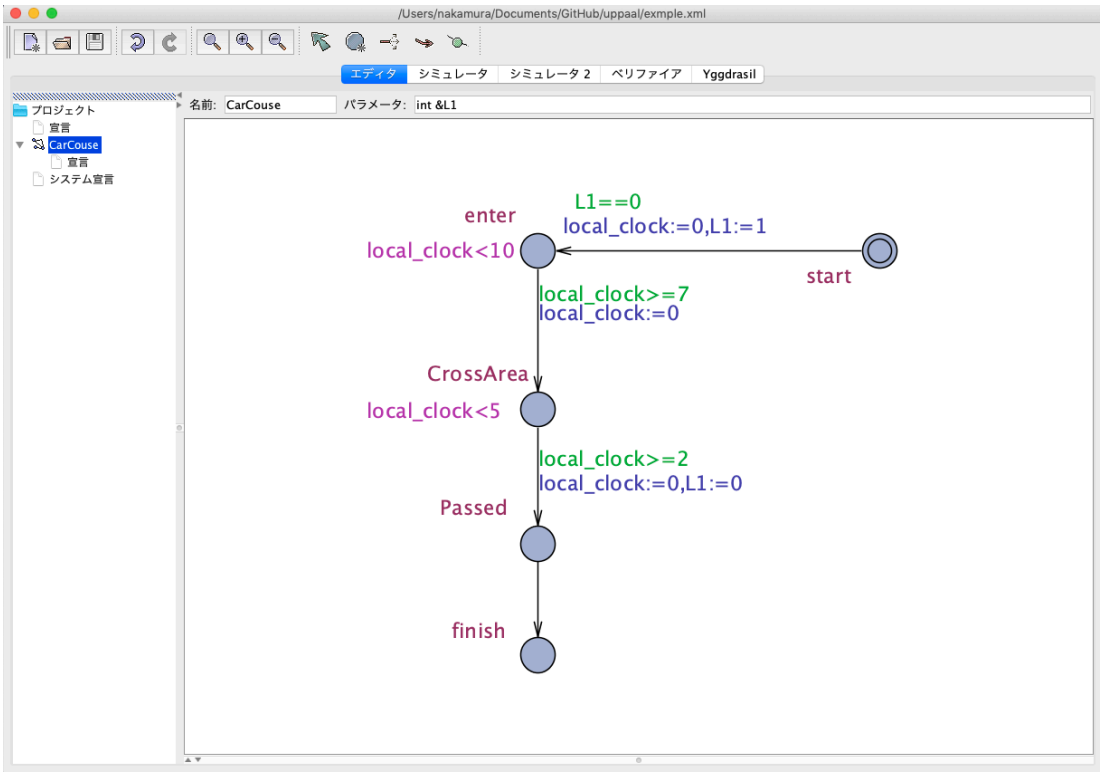

図 2.4: システムモデルの入力部分

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??)。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テンプレート CarCouse の局所的な情報を宣言で

きる (図??)。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??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。

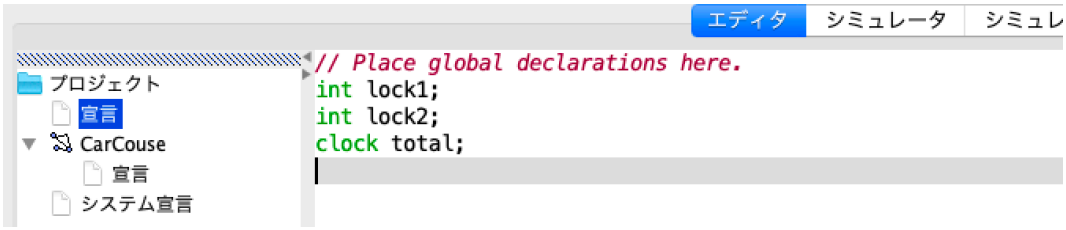

図 2.5: 大域情報編集ペイン

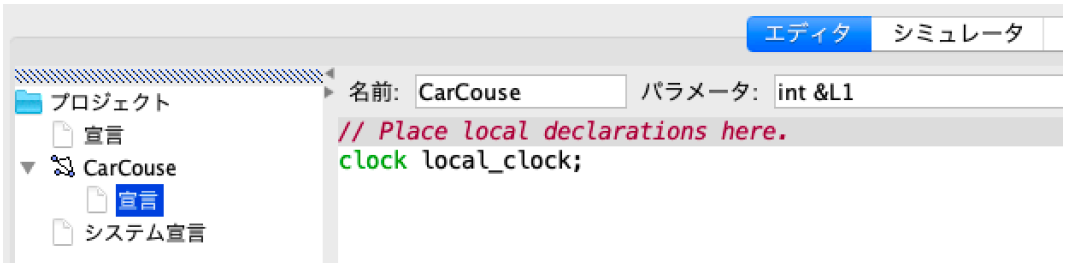

図 2.6: 局所的情報編集ペイン

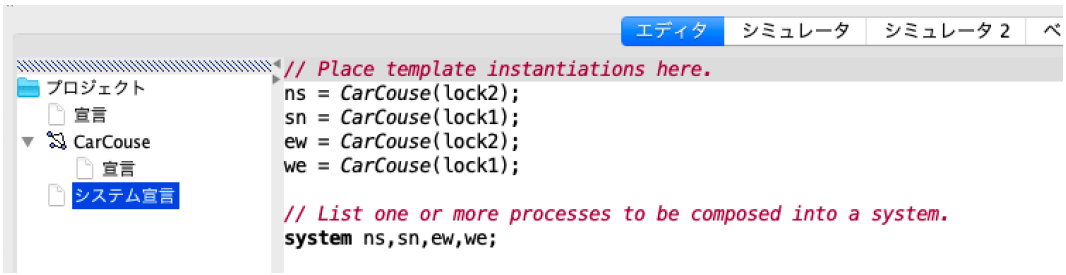

図 2.7: システム定義編集ペイン

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??)。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。

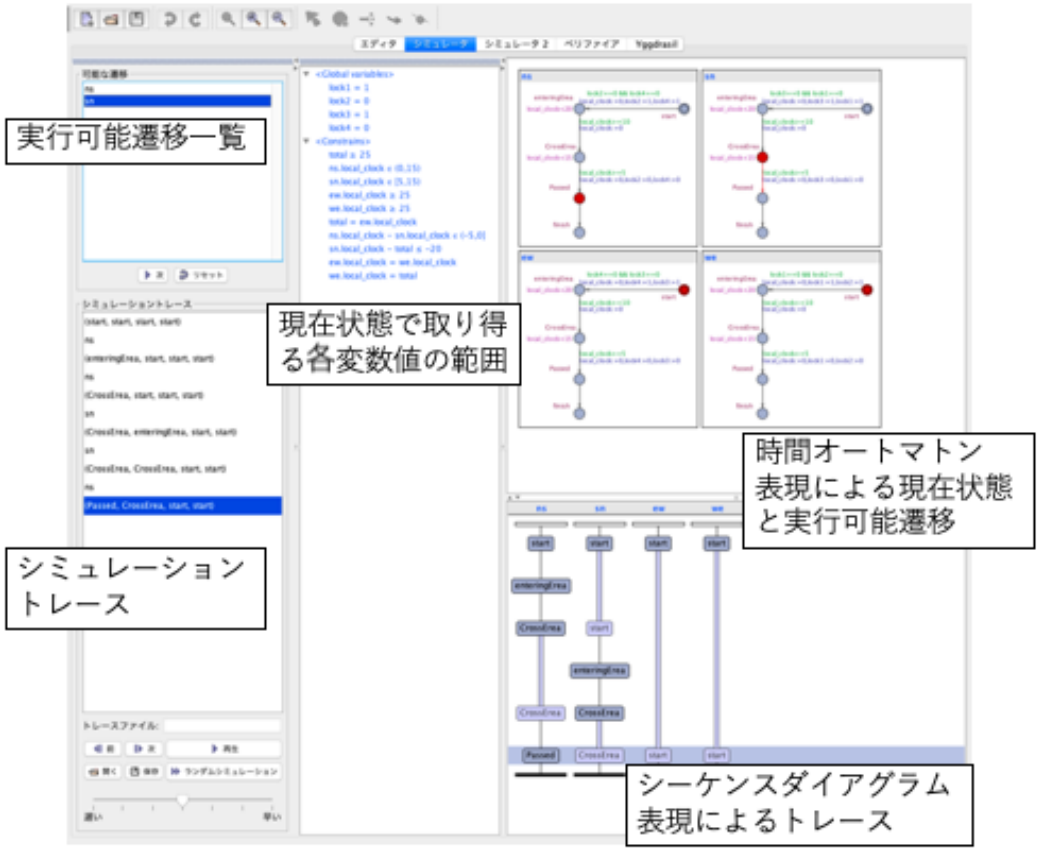

図 2.8: シミュレーション機能

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??)。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。


エディタ
シミュレータ
シミュレータ 2
ベリファイア
Yggdrasil
概要
E<> (sn.CrossErea && ns.CrossErea)
A[] not(sn.CrossErea && ew.CrossErea)
A[] not deadlock
検査
追加
削除
注釈表示
検証式一覧表示
クエリ
E<> (sn.CrossErea && ns.CrossErea)
検証式編集エリア
注釈
コメント編集エリア
状況
ローカルサーバへの直接接続が確立されました
(Academic) UPPAAL version 4.1.19 (rev. 5648), October 2014 -- server.
検証状況表示エリア
A[] not deadlock
Verification/kernel/ellapsed time used: 0.004s / 0.003s / 0.006s.
Resident/virtual memory usage peaks: 5,472KB / 4,340,352KB.
属性は満たされませんでした
A[] not(sn.CrossErea && ew.CrossErea)
Verification/kernel/ellapsed time used: 0.003s / 0.003s / 0.006s.
Resident/virtual memory usage peaks: 5,516KB / 4,340,352KB.
属性は満たされました
A[] not(sn.CrossErea && ew.CrossErea)
Verification/kernel/ellapsed time used: 0.001s / 0s / 0s.
Resident/virtual memory usage peaks: 5,524KB / 4,340,352KB.
属性は満たされました
E<> (sn.CrossErea && ns.CrossErea)
Verification/kernel/ellapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 5,524KB / 4,340,352KB.
属性は満たされました

図 2.9: 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 節ではは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，初期状態 Start で発生した車両は任意の交差点進入前状態に遷移する。最初の遷移も知らない。

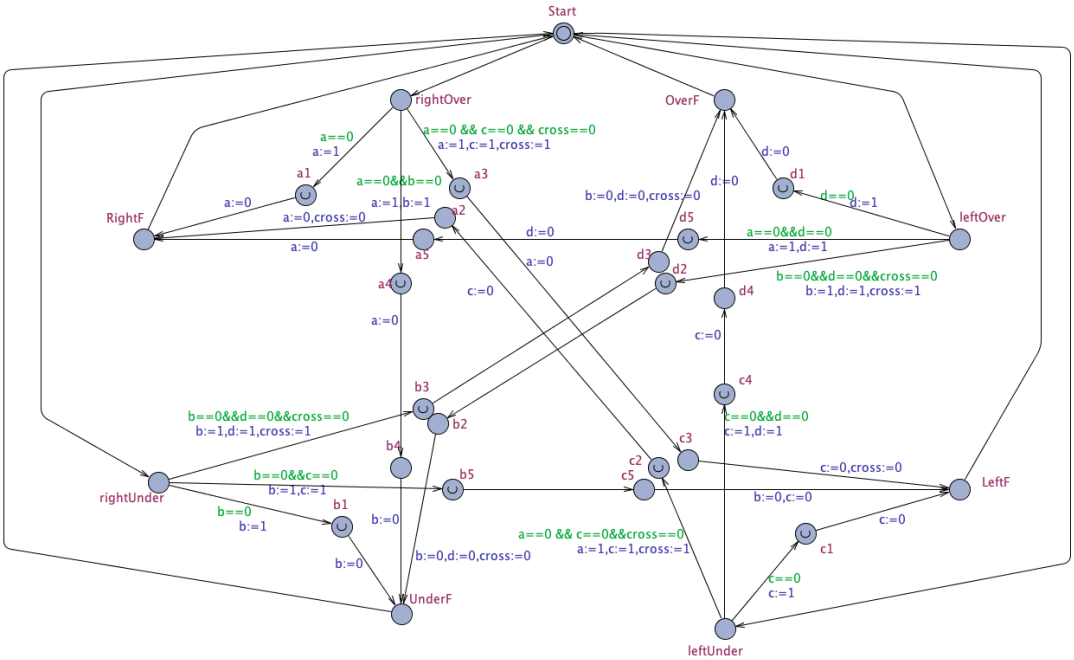

図 3.1: 進行方向を固定しない交差点のモデル

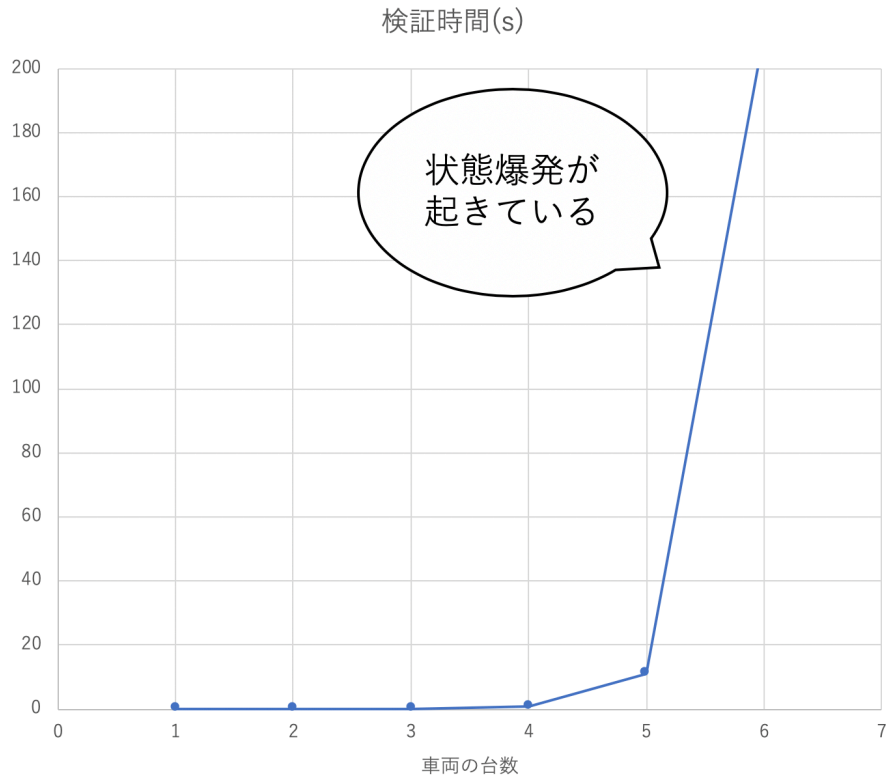

図 3.2: 車両の台数と検証にかかる時間の関係

```
A[] not deadlock
Verification/kernel/ellapsed time used: 211.755s / 1.35s / 213.168s.
Resident/virtual memory usage peaks: 3,765,700KB / 8,128,780KB.
属性は満たされました
```

図 3.3: 進行方向を固定しない交差点のモデルにおける 6 台の車両の検証結果

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
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 [?] 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 垂直に交差点に進入する 2 車両のモデル

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??)。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記述した。

図??は時間オートマトン表現で進行方向が垂直に交わる 2 台の車両の現在状態である。便宜的に左のプロセス vertical を南北方向に直進してるとし，右のプロセス horizon を東西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。

このモデルでデッドロック検証を行うと 0.001 秒未満の時間で検証できた。s

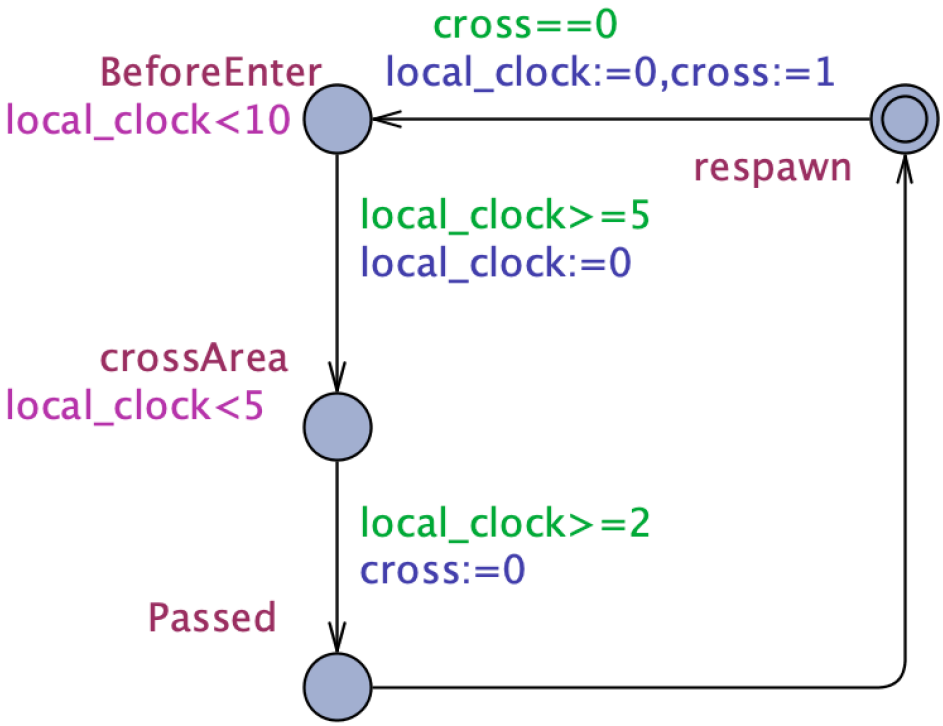

図 3.4: 交差点進入時に使用権を取得する車両 1 台の時間オートマトン

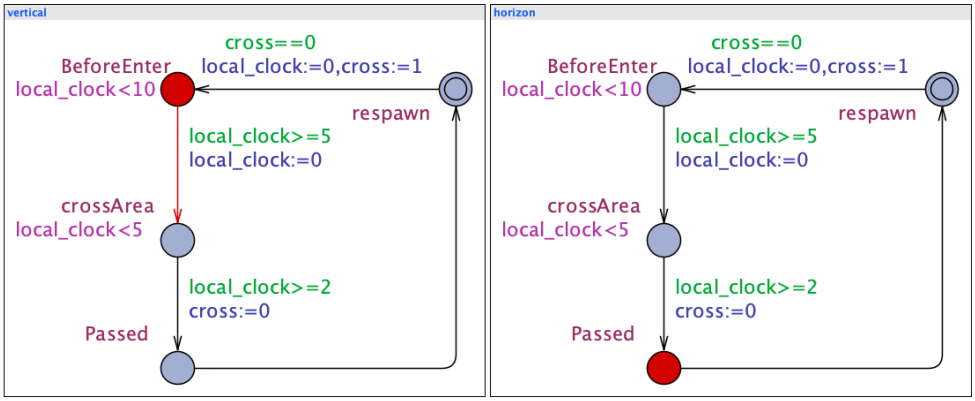

図 3.5: 垂直に交差点に進入する車両の時間オートマトンの合成

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??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。

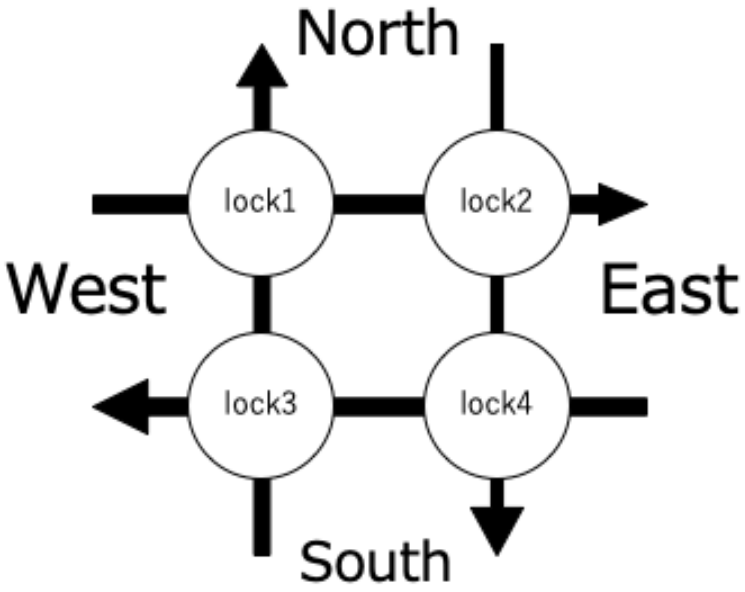

図 3.6: 交差点の使用権を管理する 4 つの鍵の組み合わせ

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??)。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。

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，図??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。

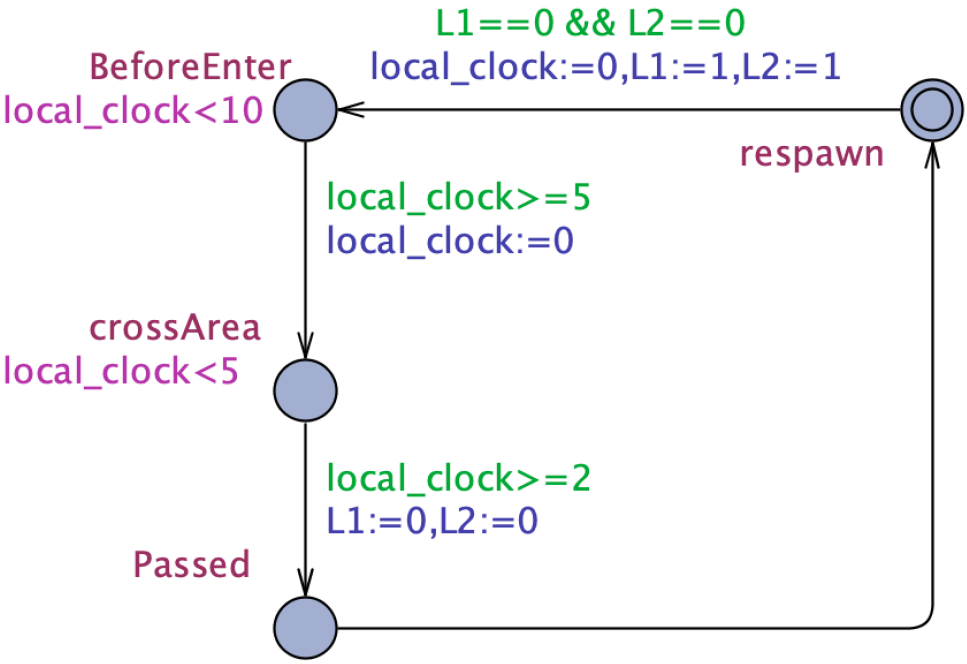

図 3.7: 使用権を 4 つの鍵で管理する交差点に直進する車両の時間オートマトン

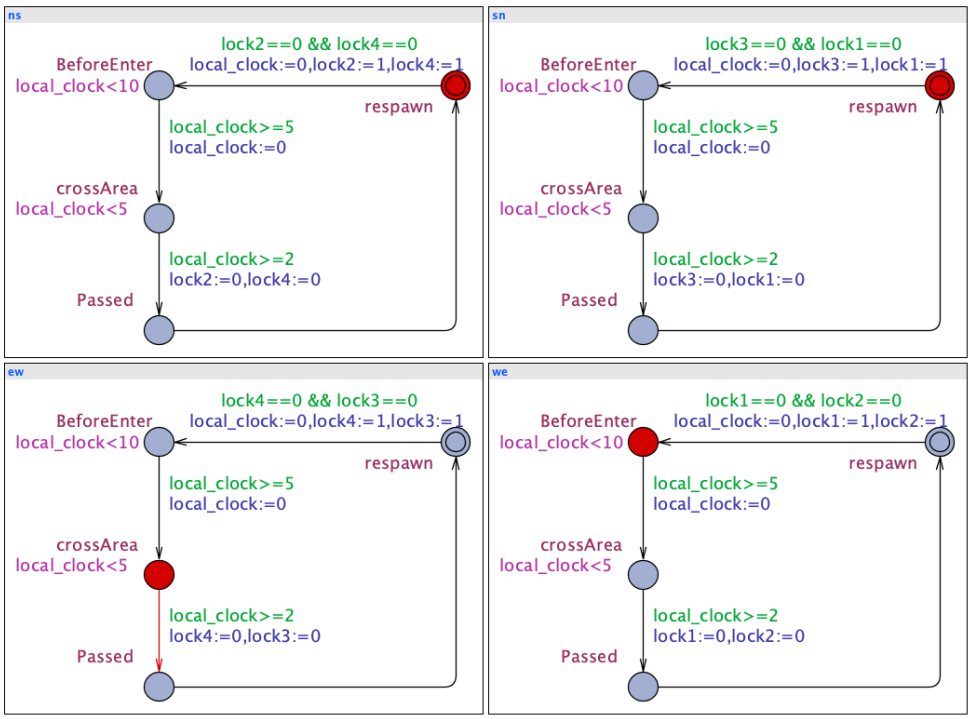

図 3.8: 交差点に直進する車両の時間オートマトンの合成

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時間オートマトンを作成する (図??)。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，右折用の鍵 cross を用意する。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。本オートマトンでは，右折や左折が直進より時間がかかることとし，時間に関する条件は前述までのモデルより長くした。

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 とすると，上記の 3 台の車両が交差点に進入する例が図??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。

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??)。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。

検証したい性質の導出：

検証式

検証結果

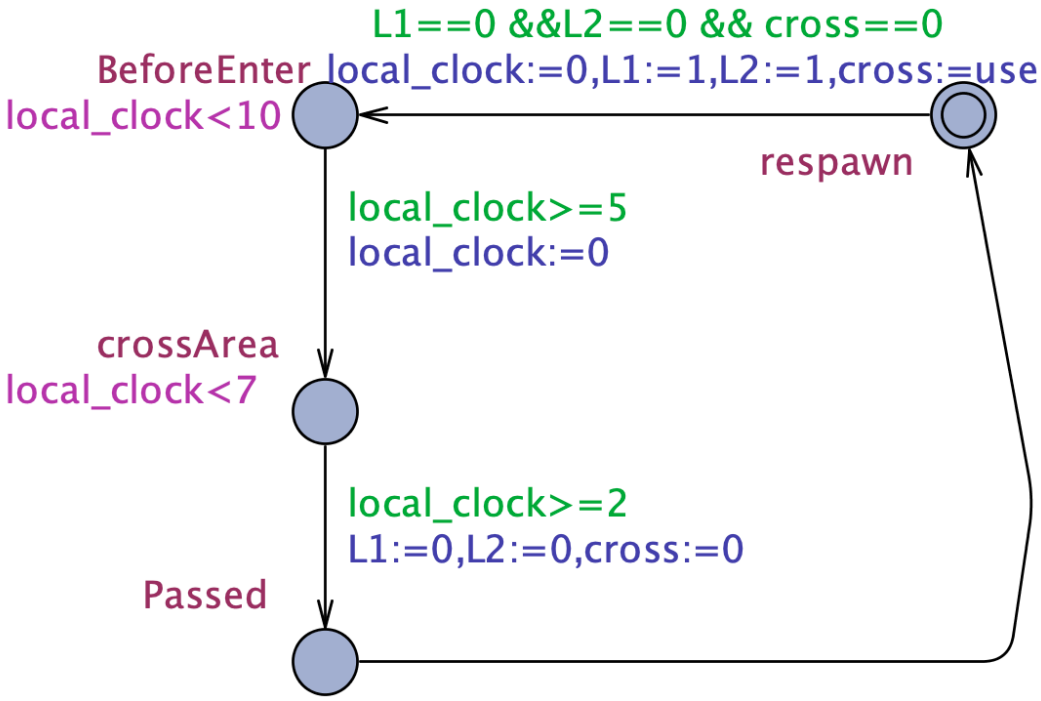

図 3.9: 交差点を通過する車両の時間オートマトン

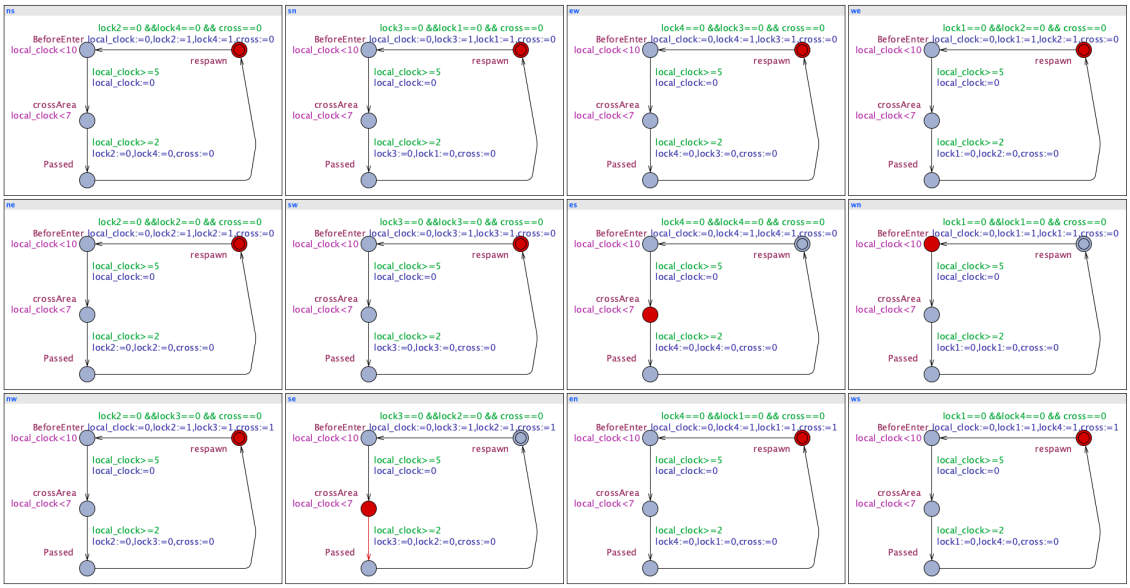

図 3.10: 交差点を通過する車両の時間オートマトンの合成

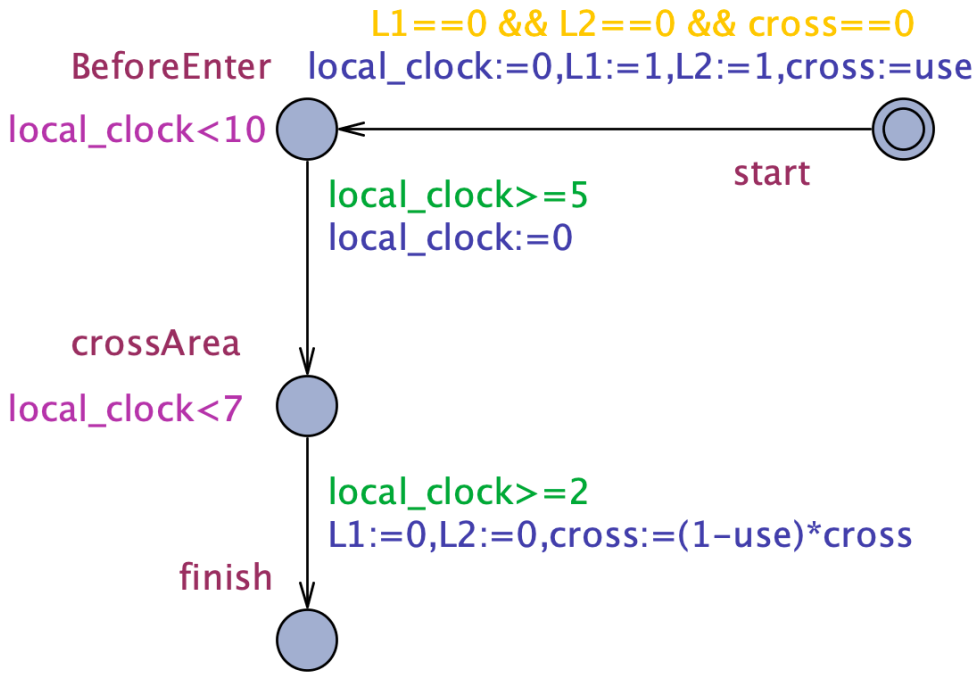

図 3.11: 交差点を 1 回通過する時間オートマトン

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.