Symmetry Reduction and Compositional Verification on Timed Automata

by

Hoang Linh Nguyen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

Abstract

This thesis is about techniques for the analysis of concurrent and real-time systems.

As the first contribution, we describe a technique that incorporates automatic symmetry detection and symmetry reduction in the analysis of systems modeled by timed automata. First, our approach detects structural symmetries arising from process templates of real-time systems, requiring no additional input from the user. Then, the technique involves finding all variables of type *process identifier* and computing a set of generators that forms a group of automorphisms. Our technique is fully automatic, and not restricted to fully symmetric systems.

The second contribution of this thesis is that we combine elements of compositional proof, abstraction and local symmetry to decide whether a safety property holds for every process instance in a parameterized family of real-time process networks. Analysis is performed on a small cut-off network; that is, a small instance whose compositional proof generalizes to the entire parametric family. Our results show that verification is decidable in time polynomial in the state space of the "cut-off" instance. Then we apply these ideas to analyze Fischer's protocol, CSMA/CD protocol and Train-Bridge protocol.

Acknowledgements

First of all I want to thank my supervisor, Prof Richard Trefler. This thesis would never be completed without his guiding and support. I have learnt a lot about formal method and verification during two years we have worked together and I hope that one day I will be a good researcher as he is. I would like to thank all current and former members of the PAT group at National University of Singapore (NUS), especially Dr Khanh Nguyen and Prof Sun Jun. I am also grateful to my colleague, Joe Scott, for fruitful discussions.

Also I would like to thank everyone at David R. Cheriton School of Computer Science for making the department such an enjoyable environment.

Table of Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Background and Motivation . . . 1

1.2 Summary of Contributions . . . 3

1.3 Thesis Outline and Overview . . . 3

1.4 Related Work . . . 4

1.4.1 Partial-order Reduction . . . 4

1.4.2 Symmetry Reduction . . . 4

1.4.3 Symbolic Model-checking . . . 5

1.4.4 Approximation Methods . . . 5

1.4.5 Efficient Representation of Clock Constraints . . . 5

1.4.6 Symmetry Detection . . . 5

1.5 Publication from The Thesis . . . 6

2 Background **7**

2.1 Timed Automata . . . 7

2.1.1 A Brief Introduction to Timed Automata . . . 7

2.1.2 Semantics of Timed Automata . . . 9

2.2 Zone and Difference Bound Matrices 14

2.2.1 Canonical DBMs 14

2.2.2 Operations on DBMs 15

2.3 Basics of Model Checking in Timed Automata 17

2.4 System Modeling 18

2.5 CSMA / CD Protocol 20

3 Symmetry Reduction in Timed Automata **23**

3.1 A Theory of Symmetry 23

3.2 Symmetry Reduction 25

3.3 Representative Function 27

3.4 Fischer's Protocol 29

3.5 Extraction of Automorphisms 30

3.5.1 State Swap 30

3.5.2 Action of State Swap on Process Template 32

3.5.3 Automorphisms 33

3.5.4 Group of Automorphisms 33

4 Symmetry Reduction Package in PAT **35**

4.1 Dining Philosopher 35

4.2 Symmetry Detection In The Specification 37

4.2.1 Step 1: Detection of *pid* Variables 37

4.2.2 Step 2: Compute a Set of Possible Permutations 39

4.2.3 Step 3: Compute a Set of Generators 40

4.3 Generate The Symmetry-reduced State Space 42

5 Parameterized Compositional Model Checking in Timed Automata 44

5.1 Preliminaries 45

5.1.1 Process 45

5.1.2 Internal State 45

5.1.3 Neighborhood 46

5.1.4 Local State 46

5.1.5 Inductive Invariant 46

5.1.6 Interleaved Composition of Processes 46

5.1.7 Compositional Invariants 47

5.1.8 Parameterized Compositional Invariants 47

5.1.9 Compositional Cutoff 47

5.2 Local Symmetry 48

5.3 Parameterized Network Families 49

6 Verification for Fischer's Protocol 51

6.1 Modeling Process Behavior 51

6.2 Modeling Fischer's Protocol 52

6.3 Verification Properties 52

6.4 Symmetry Detection 53

6.5 Symmetry Reduction and Experimental Results 54

6.6 Compositional Verification 55

7 Verification for CSMA / CD Protocol 57

7.1 Model For CSMA/CD Protocol 57

7.1.1 Modeling Sender Behavior 60

7.1.2 Modeling Bus Behavior 60

7.1.3 Modeling CSMA/CD Protocol 62

7.2 Verification 62

7.3 Symmetry Detection 62
7.4 Experimental Results 63
7.5 Compositional Verification 64

8 Verification for Train-Bridge Protocol **66**
8.1 Model For Train-Bridge Protocol 66
8.1.1 Modeling Train Behavior 67
8.1.2 Modeling Bridge Behavior 68
8.1.3 Modeling Train-Bridge Protocol 68
8.2 Verification Properties 69
8.3 Symmetry Detection 70
8.4 Experimental Results 71
8.5 Compositional Verification 72

References **77**

List of Tables

6.1 Experimental Results for Fischer's Protocol 55

7.1 Experimental Results for CSMA/CD Protocol 64

8.1 Experimental Results for Train-Bridge Protocol 71

List of Figures

2.1 A Timed Automaton. 8
2.2 A Timed Safety Automaton. 9
2.3 Network of Timed Automata. [7] 10
2.4 Combined Automaton for The Network in Figure 2.3. [7] 10
2.5 Regions for a System with Two Clocks. [7] 11
2.6 A Timed Automaton and The Corresponding Zone Graph. [9] 12
2.7 A Timed Automaton with an Infinite Zone Graph. [9] 13
2.8 Normalized Zone Graph for The Automaton in Figure 2.7. [9] 13
2.9 Graph Interpretation of The Zone and Its Closed Form. [7] 15
2.10 All DBM operations applied to the same zone. [9] 16
2.11 CSP# Modeling Language. 19
2.12 Model for a Sender i [48] 21
2.13 Model for the Bus [48]. 22

3.1 The process template for Fischer's protocol 30

4.1 Dining Philosophers Example 36
4.2 The state space of the dining philosopher. 42

6.1 The process template for Fischer's protocol 52

7.1 An Overview of CSMA/CD Protocol 58

7.2 Components of CSMA/CD Protocol 59
7.3 Model for a Sender i [48] 60
7.4 Model for the Bus [48]. 61
7.5 Model for the BroadcastCD [48]. 61

8.1 Model for the Train 68
8.2 The template for the Bridge 69

Chapter 1

Introduction

1.1 Background and Motivation

Today's world runs on computers and in fact every aspect of our life involves computers in some form or fashion. Actually computers affect how we live, work and entertain ourselves. However, there are many reasons why computers fail to work as expected, which may lead to catastrophic consequences. If one server is suddenly down, it may cause an economical damage. But what happens if an aircraft computer system does not function properly? It even leads to loss of human lives. Thus, it is very important to verify such systems whether they have fulfilled specified requirements. This is so-called *System Verification*.

System Verification is a set of actions used to check the correctness of any element, such as a system, a system component, a service, a task, etc. Traditionally, *System Verification* has been accomplished by methods like reviewing of design documents, source code or empirical testing. However, those methods have two drawbacks: 1) it is time-consuming and 2) it only provides statistical measures of correctness. Many novel approaches have been proposed in the literature. They are mostly common in the sense that the verification is done by providing a formal proof on a mathematical rigorous model of a complex system. They are so-called *formal methods*.

Note that in formal methods, there is a tradeoff between the need for rigor and the ability to model all behaviors. Therefore, applying formal methods on real-life systems is considered too difficult. In contrast to formal methods, *model checking* is fully automatic. Given a model of a system, *model checking* checks whether this model meets a given specification without manual interaction.

Although model checking has an obvious advantage over formal methods, it must cope with the *state space explosion* - the state space can grow exponentially since the number of components in a system increases. For example, the number of valid solutions for the 9x9 Sudoku grid is roughly 6,670,903,752,021,072,936,960. It means that a model checker must visit at most all possible valid solutions to check for a certain property ϕ. Many concurrent systems, however, consist of many replicated processes and they clearly exhibit symmetry of the underlying state space. For those cases, the exploitation of structural symmetries can gain a considerable reduction in processing time and memory consumption, by a factorial magnitude. These techniques are so-called *Symmetry Reduction*.

Symmetry Reduction detects structural symmetries statically and then use these knowledge to construct a *smaller* quotient structure. Since a given property ϕ is invariant under the symmetry, checking over the quotient structure is sufficient for verifying an input system. Therefore, the use of quotient graph can speed up the model-checking process.

However, statically extracting symmetry information from a model is challenging. In fact, existing solutions require the user to manually specify symmetry to be exploited, either directly or by using additional keywords. This approach has two clear drawbacks: 1) It is error-prone and 2) a modeler must have in-depth background knowledge of symmetry reduction. Moreover, they are also limited to fully-symmetric systems whose components are identical up to permutation of identifiers. This situation, however, only arises in practice for very simple systems.

For systems that clearly exhibit symmetry, we can obtain significant savings in processing time and memory consumption. However, verification is still non-scalable for systems with an unbound number of processes. It also becomes important, therefore, to consider the question of determining "once and for all" if the entire unbounded family of instances satisfies a specification. This problem is, however, generally undecidable.

Since time plays a crucial role in the operation of nowadays computer systems, in this thesis, first we study and develop techniques for real-time model-checking tools to cope with the *state space explosion* and automatic symmetry detection. Then, we explore a new and different form of parameterized verification. Specifically, we ask whether a parameterized family has a compositional proof that the specification is met for all instances.

1.2 Summary of Contributions

The main results of this thesis are summarized as follows:

- We propose to extend the real-time model checker PAT [50] with symmetry reduction. Our work is based on the *state swap* technique [26]. We provide a method to detect structural symmetries arising from parameterized process templates of real-time systems that requires no additional input from a modeler. The method results in a set of generators that forms a group of automorphisms. We even support detecting partial symmetry and rotational symmetry. We have run experiments on Fischer's protocol [1], CSMA/CD protocol [55] and Train-Bridge protocol [54]. As a result, we gain a considerable reduction in the cost of analysis, by a factor exponential in the number of processes.

- State space explosion limits model checking to small protocol instances, however, it becomes important to know whether the entire unbounded family of instances satisfies a specification. Thus, we propose a technique to determine whether a property holds for every instance of a parameterized family of real-time process networks. Our technique incorporates elements of symmetry reduction, compositional reasoning and abstraction. While symmetry reduction partitions network nodes into equivalence classes, compositional reasoning analyzes each representative node of an equivalence class separately, along with an abstraction of its neighboring processes. In certain families of process networks that satisfy the conditions of local symmetry, verification is decidable and relatively effective. We show that verification is decidable in time polynomial in the state space of the smallest verified, "cut-off", instance for networks of Fischer's protocol, CSMA/CD protocol and Train-Bridge protocol.

1.3 Thesis Outline and Overview

In this section, we briefly present the outline of the thesis and overview of each chapter:

- **Chapter 2** contains the preliminary knowledge for this thesis.
- **Chapter 3** presents the theory of symmetry reduction in Timed Automata.
- **Chapter 4** discusses the symmetry reduction package in PAT.

- **Chapter 5** applies the parameterized compositional model checking to Timed Automata.

- **Chapter 6, 7 and 8** illustrate the method through 3 examples: Fischer's protocol, CSMA/CD protocol and Train-Bridge protocol.

- **Chapter 9** gives conclusions and future works.

1.4 Related Work

Many model-checkers have been developed in the last few years for different application areas, such as SPIN [32] for communication protocols, Mur [18] for concurrent and reactive systems, HYTECH [28] for hybrid systems, and especially UPPAAL [35] for real-time systems. These tools have been successfully applied to real cases in practice e.g. [2], [25], [39], [49], [42]. In this section we briefly summarize the well-known techniques in these tools to speed up the verification process.

1.4.1 Partial-order Reduction

In many systems, the ordering of events does not affect the verification process. Based on this observation, all equivalent orderings can be represented by only one single ordering. This technique is so-called *partial-order reduction* [24][51][46]. Pagani first introduces an approach to apply partial-order reduction on timed automata, which is based on global-timed semantics [45]. However, it only limits to transitions that occur at exactly the same time. There is another approach basing on local-time semantics [8]. In [41], Minea extends results given in [8] for reachability analysis to LTL model-checking.

1.4.2 Symmetry Reduction

Symmetry reduction [33] is a method for exploiting structural symmetries in a model of a system with many replicated processes. Symmetry reduction statically detects structural symmetries and then use these knowledge to construct a smaller quotient structure. Since a property ϕ to be checked is invariant under the symmetry, the model-checking algorithm is then applied to the quotient structure rather than the entire Kripke structure. Symmetry reduction has been implemented in the real-time model checker UPPAAL [27].

1.4.3 Symbolic Model-checking

Symbolic model-checking is based on the idea that sets of states can be represented and manipulated in terms of logical formulae. Specifically for timed automata, sets of clock-values are represented and manipulated in term of zones [30][3] - the solution set of clock constraints. Symbolic model-checking will be covered later in this thesis.

1.4.4 Approximation Methods

Approximation methods are aimed at systems whose sizes are too large. In timed automata, we are interested in *over-approximations* - one type of approximation methods where non-reachable states may be considered reachable. *over-approximation* is applied to verification of timed automata in [6] and in [53]. While in [6], the authors focus on the approximate union of two time-zones, Wong-Toi presents techniques for refining the combined results obtained from forwards and backwards analysis [53].

1.4.5 Efficient Representation of Clock Constraints

To cope with *state space explosion*, it is necessary to not only reduce the number of states to visit but also minimize memory spent on storing clock constraints. There are a number of novel techniques addressing this problem. In [16] live-range analysis is used to reduce the number of clocks in a model. In [36], the author presents an algorithm to compute the minimal set of constraints for a given DBM since DBM often contains redundant information.

1.4.6 Symmetry Detection

In order to make model checking feasible for verification of real-life systems, model checkers should be able to statically extract symmetry information from any system (fully symmetric or not). UPPAAL requires a user to manually specify symmetry via *scalarset* [27]. This method not only is error-prone but also requires a modeler to have in-depth background knowledge. Moreover, it is only applicable to fully-symmetric systems.

In [19], the authors introduce a new specification language, *Promela-Lite*, to the model checker SPIN [32]. Then they show how they can detect symmetry from specifications defined in *Promela-Lite*. However, they still need to explicitly define a special datatype named *process identifier* (pid) in a language to facilitate the process. Moreover, their technique is not applicable to real-time systems.

1.5 Publication from The Thesis

The work presented in this thesis incorporates the work in University of Waterloo technical report [CS-04-2017].

Chapter 2

Background

In this chapter, we cover background material of timed automata, model checking and concepts related to this thesis.

2.1 Timed Automata

Timed automata is a well-known formalism for describing the timing behavior of concurrent systems. Through this section, we provide background on timed automata [9] with a focus on the semantics and algorithms based on which PAT is developed.

2.1.1 A Brief Introduction to Timed Automata

A timed automaton is a finite automaton extended by a set of real valued clocks [9]. All clocks start with the value of 0 and time progresses with the same rate. Clock constraints (guards) labeled on the transitions (edges) restrict the behavior of the automaton in the sense that a transition is only enabled when the clocks values satisfy the guard.

A clock constraint is a formula in the form of $x \sim n$ or $x-y \sim n$ where $\sim \in \{=, \leq, <, >, \geq\}$, $n \in \mathbb{R}_+$ and $x, y \in \mathbb{C}$ – a set of clocks. The automaton can reset a subset of the clocks to 0 when a transition is taken.

Figure 2.1 shows one example of a timed automaton, where x and y are two real-valued clocks. At first, the automaton is at the *start* location. It leaves *start* when the value of

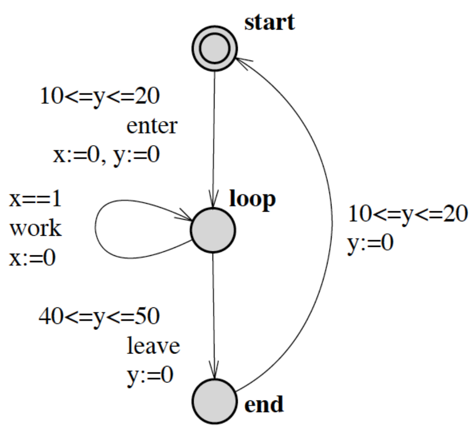


Figure 2.1: A Timed Automaton.

y is between 10 and 20, making a transition from *start* to *loop* and reseting both clocks to 0. At the *loop* location, the automaton makes the self-loop until the value of y is between 40 and 50, the automaton may go from *loop* to *end*. Staying at the *end* location from 10 to 20 time units, the automaton may return back to *start* and restart the process.

In this example, the automaton may get stuck forever in any location since the guards are only enabling conditions, they cannot force a transition to be taken. Alur and Dill [3] solves this problem by introducing *Timed Buchi Automata*. In *Timed Buchi Automata*, some of the locations are marked as *accept* and valid executions must pass through *accept* locations infinitely often.

Back to the example above, assume that the *end* location is marked as *accept*. This implies that all executions of the system enter *end* infinitely often. Consequently, the automaton must leave *start* when the value of y is at most 20. Similarly, it can only stay at the *loop* location at most 50 time units. However, *Timed Buchi Automata* is inconvenient for system modeling and analysis [29].

In [29], the authors introduce *Timed Safety Automata*, which replaces *accepted* locations by *location invariants* - clock constraints in each location. More clearly, the automaton must leave a current location after the invariant condition is violated. Figure 2.2 shows a timed safety automaton that corresponds to the one given in Figure 2.1.

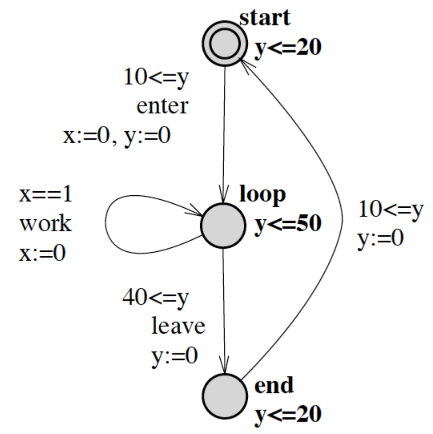


Figure 2.2: A Timed Safety Automaton.

A time automaton $\mathbb{A}$ is a tuple of $\langle L, l_0, E, I \rangle$ where:

- L is a finite set of locations.
- $l_0 \in L$ is the initial location.
- $E \subseteq L$ x $B(\mathbb{C})$ x Σ x $2^{\mathbb{C}}$ x L is the set of edges.
- $I : L \rightarrow B(\mathbb{C})$ is the set of location invariants.

Where we use $B(\mathbb{C})$ to denote a set of clock constraints.

To model concurrent systems, PAT extends the concept of timed automata with parallel composition [40] as well as pair-wise synchronization between processes. Figure 2.3 models a light-switch (left) and its user (right). Two processes synchronize their actions via the *press* label. Figure 2.4 shows the state-space of the combined automaton.

2.1.2 Semantics of Timed Automata

Let v denote the clock assignment that maps all clocks $\in \mathbb{C}$ to their values, $v \in g$ denote that the clock values satisfy the guard g. The semantics of a timed automaton is defined

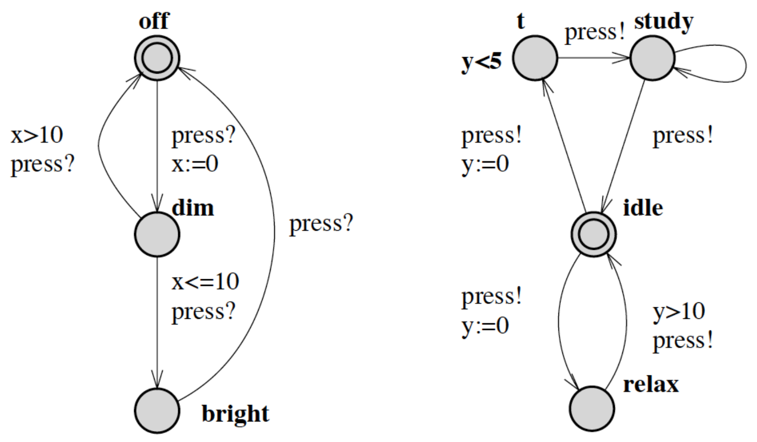


Figure 2.3: Network of Timed Automata. [7]

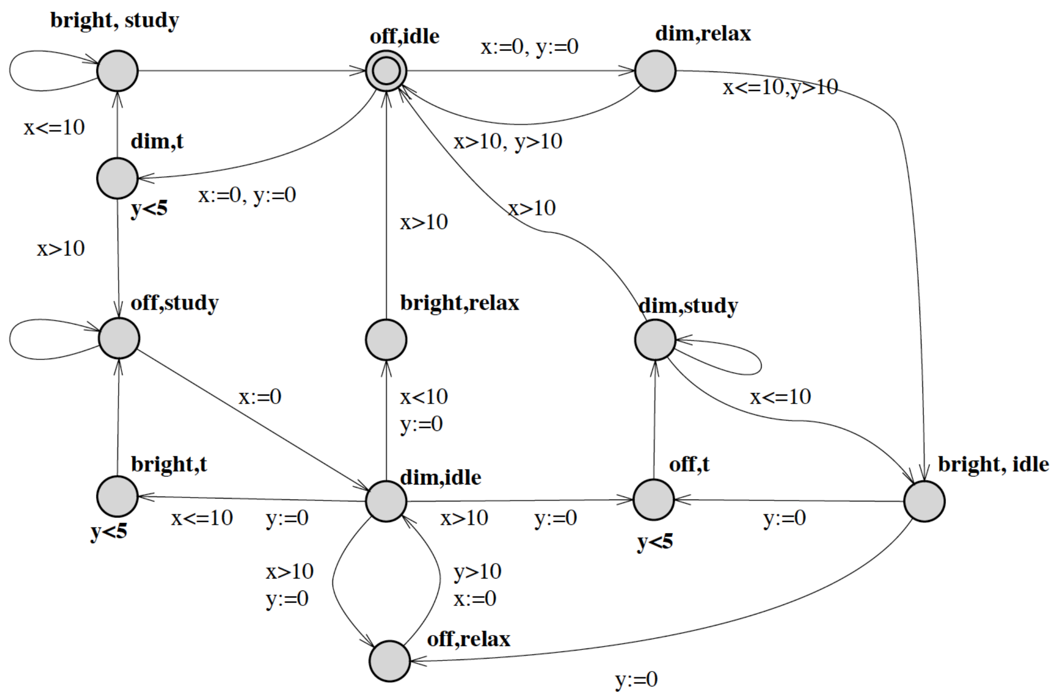


Figure 2.4: Combined Automaton for The Network in Figure 2.3. [7]

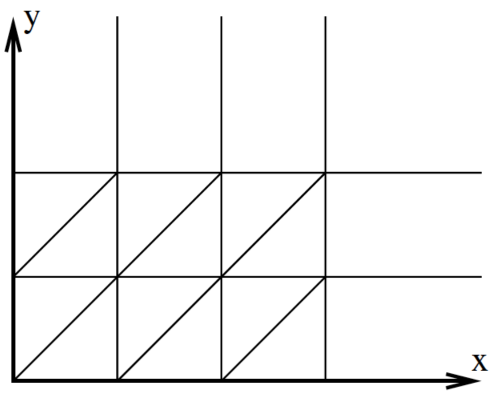


Figure 2.5: Regions for a System with Two Clocks. [7]

as a transition system. A state is a pair $\langle l, v\rangle$ where l is a location and v is a vector of clock values. A transition is either *a delay transition* or *an action transition*, which are defined by the following rules:

$$\textit{Delay Transition}: \langle l, v\rangle \xrightarrow{d} \langle l, v{+}d\rangle \text{ if } v \in I(l),\ v{+}d \in I(l) \text{ and } d \in \mathbb{R}_+ \text{ is the elapsed time.}$$

$$\textit{Action Transition}: \langle l, v\rangle \xrightarrow{a} \langle l', v'\rangle \text{ if } l \xrightarrow{g,a,r} l' \text{ when } \langle l, g, a, r, l'\rangle \in E,\ v \in g \text{ and } v' \in I(l').$$

Because clocks are real-valued, the semantics has the infinite state-space and is inappropriate for automated verification. Many techniques have been developed by attempts to finitely partition the state-space into symbolic states. One technique is to construct a *region graph* [4]. Consider Figure 2.5, where each line segment, each intersection and each area defines one region. The number of possible regions is 60, assuming that $x \leq k = 3$ and $y \leq g = 2$. It means that the region graph, however, may grow exponentially when we increase either the number of clocks or upper bounds (k, g).

Another approach is based on the notion of zone and zone-graph [17]. By definition, a *zone* is the solution set of clock constraints. Figure 2.6 (a) shows a timed automaton and Figure 2.6(b) shows its corresponding zone-graph, which has only eight states. The region-graph for that example has over fifty states.

The symbolic semantics of a timed automaton is defined as a symbolic transition system. A symbolic state is a pair $\langle l, D\rangle$ where l is a location and D is a zone. Therefore, a symbolic state represents a set of states. Symbolic transition relations over symbolic states

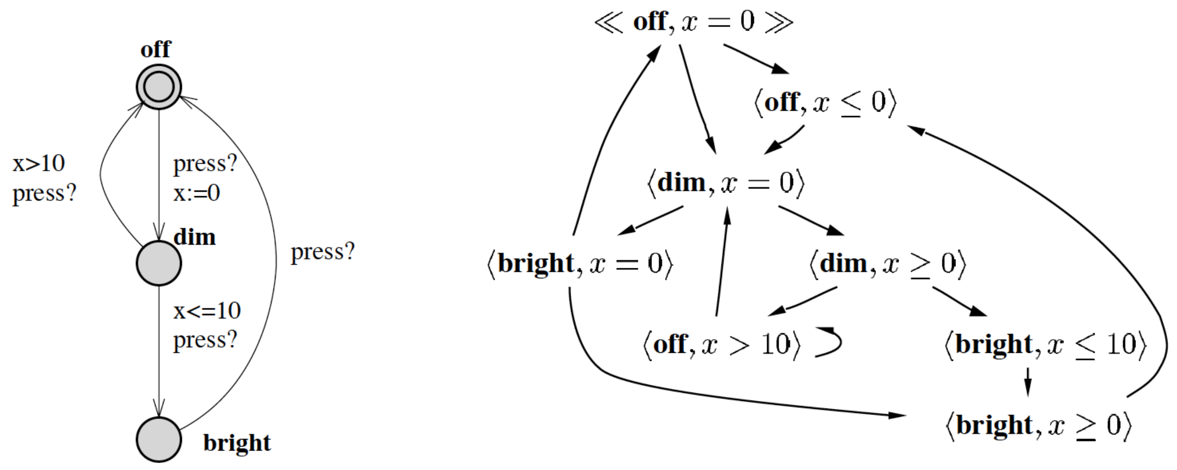


Figure 2.6: A Timed Automaton and The Corresponding Zone Graph. [9]

are defined by the following rules:

$$\textit{Delay Transition}: \langle l, D\rangle \xrightarrow{d} \langle l, D^{\uparrow} \wedge I(l)\rangle \text{ where } D^{\uparrow} = \{v + d \mid v \in D, d \in \mathbb{R}_+\}.$$

$$\textit{Action Transition}: \langle l, D\rangle \xrightarrow{a} \langle l', D' \wedge I(l')\rangle \text{ where } D' = \textit{reset(D)}.$$

where *reset(D)* is one operation on zones that resets selected clock values to zero.

However, the zone-graph may be also infinite. Consider the model in Figure 2.7, where the values of clocks drift away unboundedly, giving an infinite graph. The solution is to transform zones into their normalized representatives to guarantee termination. In the original theory of timed automata, difference constraints $x - y \sim n$ are not allowed to appear in the guards g. Such automata whose guards contains only atomic constraints in the form of $x \sim n$ are known as diagonal-free automata. For diagonal-free automata, one popular zone-normalization procedure is so-called *k-normalization* [47]. Once the value of a clock x is larger than the maximum constant k, it is no longer significant for the automaton to know its precise value, only that $x \geq k$. Figure 2.8 depicts the *k-normalized* zone-graph of the automaton given in Figure 2.7.

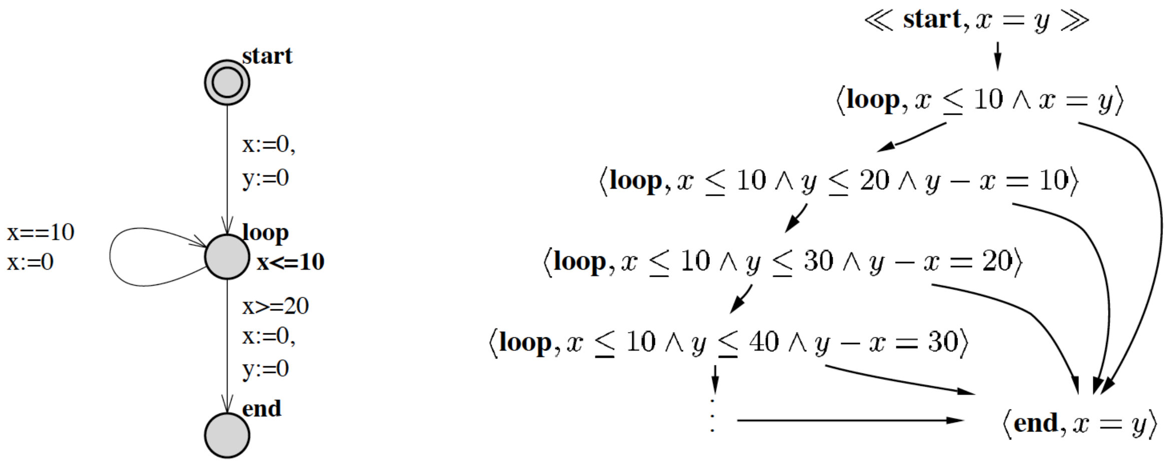

start
x:=0,
y:=0
x==10
x:=0
loop
x<=10
x>=20
x:=0,
y:=0
end
$\ll \textbf{start}, x = y \gg$
$\langle \textbf{loop}, x \leq 10 \land x = y \rangle$
$\langle \textbf{loop}, x \leq 10 \land y \leq 20 \land y - x = 10 \rangle$
$\langle \textbf{loop}, x \leq 10 \land y \leq 30 \land y - x = 20 \rangle$
$\langle \textbf{loop}, x \leq 10 \land y \leq 40 \land y - x = 30 \rangle$
$\langle \textbf{end}, x = y \rangle$

Figure 2.7: A Timed Automaton with an Infinite Zone Graph. [9]

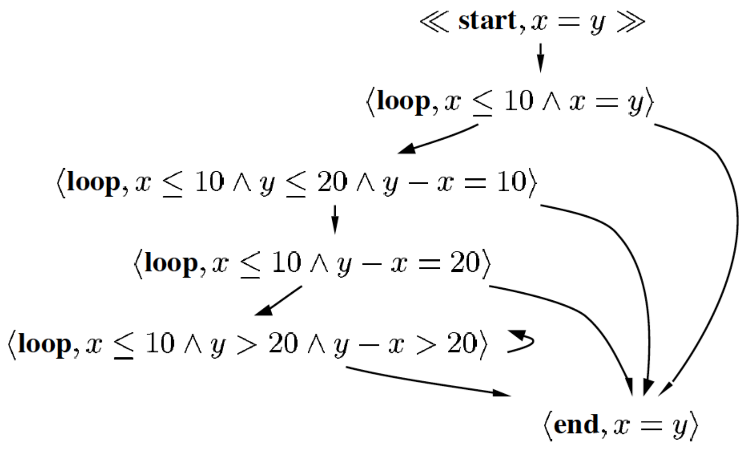

$\ll \textbf{start}, x = y \gg$
$\langle \textbf{loop}, x \leq 10 \land x = y \rangle$
$\langle \textbf{loop}, x \leq 10 \land y \leq 20 \land y - x = 10 \rangle$
$\langle \textbf{loop}, x \leq 10 \land y - x = 20 \rangle$
$\langle \textbf{loop}, x \leq 10 \land y > 20 \land y - x > 20 \rangle$
$\langle \textbf{end}, x = y \rangle$

Figure 2.8: Normalized Zone Graph for The Automaton in Figure 2.7. [9]

2.2 Zone and Difference Bound Matrices

A *zone* is the solution set of clock constraints. Since a zone is a part of a symbolic state, how to represent a zone effectively is a major issue. In many verification tools including PAT, such sets are represented as Difference Bound Matrices (DBMs).

Recall that a clock constraint is a formula in the form of $x \sim n$ or $x - y \sim n$ where $\sim \in (=, \leq, <, >, \geq)$, $n \in \mathbb{N}$ and $x, y \in \mathbb{C}$ – a set of clocks. To have a unified form for clock constraints, we introduce a reference clock $\mathbf{0}$ with the constant value 0 such that $\mathbb{C}_0 = \mathbb{C} \cup \mathbf{0}$. Therefore, any clock constraint can be written in the form of $x - y \sim n$ where $\sim \in (=, \leq, <, >, \geq)$, $n \in \mathbb{N}$ and $x, y \in \mathbb{C}_0$.

DBM representation for a zone D is the matrix, which each row stores lower bounds on the difference between one clock and all other clocks. We denote all clocks in $\mathbb{C}_0$ as $x_0, x_1, ..., x_N$. We use $D_{i,j}$ to denote element (i, j) in the DBM. The matrix elements are then computed as follows:

- $D_{ij} = (k, \leq)$ presents the constraint $x_i - x_j \leq k$.
- $D_{ij} = \infty$ if $x_i - x_j$ is unbounded.
- $D_{ij} = 0$ if $i = j$.
- $D_{ij} = 0$ for the difference between clock $\mathbf{0}$ and other clocks.

Consider the zone $D = x - 0 < 20 \wedge y - 0 \leq 20 \wedge y - x \leq 10 \wedge x - y \leq -10 \wedge 0 - z < 5$, where $0, x, y, z$ are clocks and we denote as x_0, x_1, x_2, x_3. DBM representation for D should be:

$$D = \begin{bmatrix} 0 & 0 & 0 & 5 \\ 20 & 0 & -10 & \infty \\ 20 & 10 & 0 & \infty \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

2.2.1 Canonical DBMs

Although an infinite number of zones share the same solution set, among them there is a unique zone where no atomic constraint can be strengthened without losing solutions [36]. We call it as the canonical representation of an entire family of zones, denoted as *Rep*(*D*).

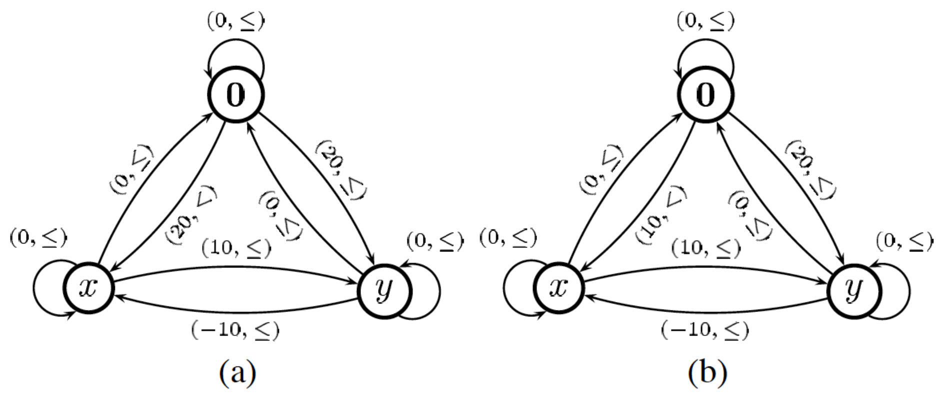


Figure 2.9: Graph Interpretation of The Zone and Its Closed Form. [7]

Rep(D) is derived from a zone D by computing the tightest constraint on each clock difference in D. We can imagine that zone is a weighted graph where clocks are nodes and clock constraints are edges.

As an example, consider the zone $D = x - 0 < 20 \wedge y - 0 \leq 20 \wedge y - x \leq 10 \wedge x - y \leq -10$. By combining the atomic constraints $y - 0 \leq 20 \wedge y - x \leq 10$, we derive $x - 0 \leq 10$, i.e. the bound $x - 0$ on can be strengthened. Figure 2.9(a) presents the graph interpretation of D while Figure 2.9(b) presents the representative *Rep*(D) after $x - 0$ is strengthened.

The problem of computing the canonical representation of a given zone is equivalent to finding the shortest path between every pair of nodes in the graph interpretation of the zone [9].

2.2.2 Operations on DBMs

We briefly list out all operations on DBMs, which are divided into three different classes:

- **Property-Checking** Operations in this class include checking if a DBM is consistent (**consistent(D)**, checking inclusion between zones (**relation(D, D')**), and checking whether a zone satisfies a given atomic constraint (**satisfied(D,** $x_i - x_i \leq m$**)**).
- **Transformation** Operations in this class include computing the strongest post condition of a zone D with respect to delay (**up(D)**), computes the weakest precondi-

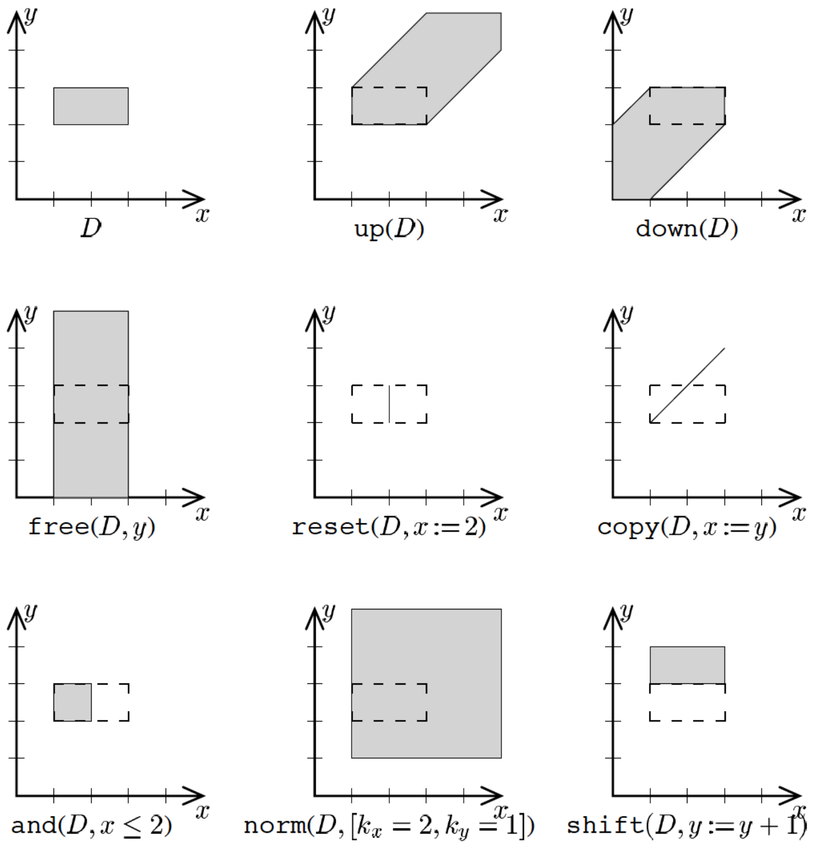


Figure 2.10: All DBM operations applied to the same zone. [9]

tion of D with respect to delay (**down(D)**), adding a constraint to a zone (**and(D,** $x_i - x_j \leq b$**)**), removes all constraints on a given clock (**free(D, x)**), setting a clock to specific values (**reset(D,** $x := m$**)**), copying the value of one clock to another (**copy(D,** $x := y$**)**), adding or subtracting a clock with an integer value (**shift(D,** $x := x + m$**)**).

- **Normalization** Operations in this class include **k-normalization**.

2.3 Basics of Model Checking in Timed Automata

Model Checking [13][11] is one of the most successful approaches for verifying specifications of hardware and software systems. Model-checking uses reachability analysis to verify effectively systems on safety properties.

In timed automata, reachability analysis consists of two steps: 1) compute the normalized zone-graph *on-the-fly* and 2) check if a current state contradicts or satisfies given properties. In comparison to pre-computing, computing the zone-graph *on-the-fly* has an obvious advantage. Specifically, only the part of the state-space needed to prove the property is generated [9]. In most cases, the generated state-space is smaller than the entire state-space. Note that, the method, however, still generates the entire state-space to prove invariant properties.

Algorithm 1 describes the forward reachability analysis algorithm in PAT. Let $\langle l_0, D_0 \rangle$ be

Algorithm 1 Forward Reachability Analysis Algorithm

```
1:  Visited = $\emptyset$
2:  $\langle l_f, D_f \rangle$ = Violated States
3:  Next = $\langle l_0, D_0 \rangle$
4:  **while** Next $\neq \emptyset$ **do**
5:      remove $\langle l, D \rangle$ from Next
6:      **if** $l = l_f$ and $D \cap D_f \neq \emptyset$ **then return** YES
7:      **if** $D \nsubseteq D'$ for all $\langle l, D' \rangle \in$ Visited **then**
8:          add $\langle l, D \rangle$ to Visited
9:          **for** $\langle l', D' \rangle$ such that $\langle l, D \rangle \rightarrow \langle l', D' \rangle$ **do**
10:             add $\langle l', D' \rangle$ to Next
11: **return** NO
```

a set of initial states and a set of bad states is given as $\langle l_f, D_f \rangle$ respectively. **Algorithm 1** computes the normalized zone-graph of the automaton on-the-fly and checks if $\langle l_0, D_0 \rangle$ may evolve to any state whose location is l_f and whose clock assignment satisfies D_f. At each step, the algorithm stores next symbolic states in $Next$ and checks if the reached zones intersect with D_f. Normalized zone-graph is finite [47] and the algorithm is guaranteed to terminate.

Despite model checkers have an obvious advantage over theorem proving, their applications are limited since there are some main reasons as follows:

- *Scalability.* Model checkers must cope with the *state space explosion.* The state space can grow exponentially when the number of components in a system increases.
- *Accessibility.* In fact, model checkers are mostly academic tools whose interfaces are not user-friendly, and they also lack of detailed documentation. Moreover, a modeler is required to have background in formal method to build models.

2.4 System Modeling

The section focuses on the modeling language used in PAT. PAT uses CSP# as its primary modeling language to define processes and computational logic in a process. CSP# is a timed extension of Communication Sequence Process (CSP) [31]. Its grammar is given in Figure 2.11, where P and Q are processes, $e \in \Sigma$ is an observable event, b is a boolean expression, X is a set of event names and d is an integer constant.

To be more detailed:

- *Stop* denotes deadlock, where the process does nothing but idling.
- *Skip* states termination.
- $e \rightarrow P$ performs event e first and then behaves as P. Notice that e is either an abstract event or a data operation, e.g.
- $[b]P$ denotes a guard process. If b is true, then it behaves as P. Otherwise, it does nothing but waits until b becomes true.
- *if b then P else Q* performs a conditional choice. Specifically, if b is true, then it behaves as P, otherwise it behaves as Q.
- $P \,\Box\, Q$ denotes an unconditional choice. Which process it should behave as depends on upcoming events.
- $P \parallel Q$ denotes parallel composition. Two processes communicate via global variables or passing messages on channels.

$P = Stop \mid Skip$	– primitives
$\mid e \rightarrow P$	– event prefixing
$\mid [b]P$	– state guard
$\mid if\ b\ then\ P\ else\ Q$	– if-then-else
$\mid P \Box Q$	– general choice
$\mid P \parallel Q$	– parallel composition
$\mid P;\ Q$	– sequential composition
$\mid P \setminus X$	– hiding
$\mid P \widehat{=} Q$	– process referencing
$\mid Wait[d]$	– delay
$\mid P\ timeout[d]\ Q$	– timeout
$\mid P\ interrupt[d]\ Q$	– timed interrupt
$\mid P\ within[d]$	– react within some time
$\mid P\ waituntil[d]$	– wait until
$\mid P\ deadline[d]$	– deadline

Figure 2.11: CSP# Modeling Language.

- P; Q denotes a sequential execution. The process behaves as P until P terminates and then immediately behaves as Q.
- $P \setminus X$ makes all occurrences of events in X not to be observed or controlled by the process environment.
- $P \equiv Q$ defines P to be exactly the same as Q.
- $Wait[d]$ delays the system execution d time units and then it terminates.
- P *timeout[d]* Q behaves as Q after d time units have elapsed unless the first observable event of P occurs before that.
- P *interrupt[d]* Q only behaves as P within d time units and then behaves as Q.
- P *waituntil[d]* denotes that P executes for at least d time units.
- P *deadline[d]* states that P must terminate within d time units.

Although Timed Automata proves its popularity for modeling real-time systems with explicit clock variables, it has certain drawbacks. Specifically, Timed Automata is not suitable for designing compositional models. However, in industry, high-level requirements for real-time systems are often stated in terms of *deadline, time out, and timed interrupt* [34]. As a result, if users define their models in Timed Automata, they need to manually cast those terms into a set of clock variables with carefully calculated clock constraints, which is tedious and error-prone [48].

2.5 CSMA / CD Protocol

The Carrier Sense, Multiple Access with Collision Detection (CSMA/CD) protocol describes one solution to the problem in Ethernet network, when several agents compete for a single bus. The research group in PAT has successfully done modeling and verification on CSMA/CD protocol [48]. In this section, we show how CSMA/CD protocol is modeled using the CSP# modeling language in PAT.

CSMA/CD protocol consists of two components, namely *Sender* and *Bus*. Two components communicate by pair-wise synchronization channels. Roughly speaking, a *Sender* must first listen to the *Bus*. If the *Bus* is idle, the *Sender* begins to transmit. Otherwise, it must wait and retry later. However, collision may occur when more than one *Sender* are sending message via the *Bus*. Then the *Bus* informs all *Senders* of this collision, and abort their transmission immediately. Therefore, all transmitting messages are discarded.

Modeling Sender Behavior

The behavior of component *Sender* is defined in Figure 2.12. Initially, the *Sender* is in $WaitFor$ location. When there is a message to send, if the *Bus* is idle, the *Sender* goes to $Trans$ location. Otherwise, if the *Bus* is busy or a collision is detected, it moves to *Retry*. If a collision occurs while no message is arrived, the *Sender* remains in $WaitFor$ location.

In $Trans$ location, the *Sender* has two transitions. If a collision is detected within 52 time units, the *Sender* goes to *Retry*. Otherwise, it terminates sending the message after exactly 808 time units, then it goes to $WaitFor$.

In *Retry* location, if the *Bus* is idle, the *Sender* moves back to $Trans$ within 52 time units. Otherwise, it remains in *Retry* location.

$$
\begin{aligned}
\mathit{WaitFor}(i) = {} & (cd?i \rightarrow \mathit{WaitFor}(i)) \\
& \Box\ (newMess!i \rightarrow ((begin!i \rightarrow \mathit{Trans}(i)) \\
& \qquad \Box\ (busy?i \rightarrow \mathit{Retry}(i)) \\
& \qquad \Box\ (cd?i \rightarrow \mathit{Retry}(i))));
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Trans}(i) = {} & (cd?i \rightarrow \mathit{Retry}(i) \mathit{within}[0, 52]) \\
& \Box\ (atomic\{end!i \rightarrow \mathit{Skip}\} \mathit{within}[808, 808]; \\
& \quad \mathit{WaitFor}(i));
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Retry}(i) = {} & (newMess!i \rightarrow ((begin!i \rightarrow \mathit{Trans}(i) \mathit{within}[0, 52]) \\
& \quad \Box\ (busy?i \rightarrow \mathit{Retry}(i) \mathit{within}[0, 52]) \\
& \quad \Box\ (cd?i \rightarrow \mathit{Retry}(i) \mathit{within}[0, 52])));
\end{aligned}
$$

Figure 2.12: Model for a Sender i [48]

Modeling Bus Behavior

The behavior of component *Bus* is showed in Figure 2.13. Initially, the *Bus* is in *Idle* location. The transition from *Idle* to *Active* is enabled when one *Sender* begins to transmit.

In *Active* location, there are three possible transitions. If the *Sender* completes sending, the *Bus* goes back to the initial location. If another *Sender* starts sending messages within 26 time units, the *Bus* moves to *Collision*. Otherwise, after at least 26 time units have elapsed, the *Bus* replies busy signal to any new attempt, then it moves to *Active*1 location.

In *Active*1 location, the *Bus* takes at most 26 time units to inform all *Senders* of this collision, using *BroadcastCD* [48]. After that, the *Bus* moves to *Idle*.

In *Collision* location, the *Bus* replies busy signal to any *Sender* that attempts to send message until the active *Sender* completes transmitting, then the *Bus* moves to *Idle*.

Modeling CSMA/CD Protocol

The whole protocol consists of one *Bus* and N *Senders* interleaving with each other. In PAT, we model this as follows:

$$CSMA = (|||\ \mathrm{i} : \{0, ..., N-1\} \ @\ \mathit{Sender}(i))\ |||\ \mathit{Bus}$$

$$
\begin{aligned}
&\mathit{Idle} = \mathit{newMess}?i \rightarrow \mathit{begin}?i \rightarrow \mathit{Active};\\
&\mathit{Active} = (\mathit{end}?i \rightarrow \mathit{Idle})\\
&\qquad\quad \Box\ (\mathit{newMess}?i \rightarrow\\
&\qquad\qquad ((\mathit{begin}?i \rightarrow \mathit{Collision})\ \mathit{timeout}[26]\\
&\qquad\qquad \Box\ (\mathit{busy}!i \rightarrow \mathit{Active}1)));\\
&\mathit{Active}1 = (\mathit{end}?i \rightarrow \mathit{Idle})\\
&\qquad\quad \Box\ (\mathit{newMess}?i \rightarrow \mathit{busy}!i \rightarrow \mathit{Active}1);\\
&\mathit{Collision} = \mathit{atomic}\{\mathit{BroadcastCD}(0)\}\mathit{within}[0, 26];\ \mathit{Idle};
\end{aligned}
$$

Figure 2.13: Model for the Bus [48].

Chapter 3

Symmetry Reduction in Timed Automata

State-space explosion is the main obstacle to the scalability of model checking. Indeed, it is known that symmetry reduction techniques can be used to combat this problem for networks of replicated components. In this chapter, first we will summarize a theory of symmetry and then explain how to apply symmetry reduction into real-time systems modeled by timed automata.

3.1 A Theory of Symmetry

Consider state graphs, which are tuples containing a set of states S, a set $S_0 \subseteq S$ of initial states and a transition relation $\Delta \subseteq S$ x S.

A state $s \in S$ is *reachable* if a sequence $s_0, s_1, ..., s_{n-1}$ exists, such that:

- $s_0 \in S_0$.
- $s = s_{n-1}$.
- $s_i \in S$ for all $0 \leq i < n$.
- $(s_i, s_{i+1}) \in \Delta$ for all $0 \leq i < n-1$.

Let s be a reachable state and ϕ be a safety property to verify. We denote $s \models \phi$ if the property ϕ is true in state s. **Algorithm 2** depicts a standard forward exploration algorithm to check if a given system satisfies ϕ.

Algorithm 2 Standard Forward Reachability Analysis Algorithm

```
1: $passed = \emptyset$
2: $waiting = S_0$
3: while $waiting \neq \emptyset$ do
4:     remove $s$ from $waiting$
5:     if $s \models \phi$ then return YES
6:     else if $s \notin passed$ then
7:         add $s$ to $passed$
8:         $waiting = waiting \cup succ(s)$
9: return NO
```

Initially, the set *waiting* only contains the initial states S_0. In each while-loop, the algorithm pops out a state s from *waiting* and processes in the following way:

- The algorithm returns YES if state s satisfies the property ϕ.
- If property ϕ is false in state s and if state s has been visited before, then the algorithm discards s and repeats the loop.
- Otherwise, s is added to the set *passed* and all of its successors are added to *waiting*.

If the state space is finite, this algorithm halts. Otherwise, it may not halt. In fact, verification with this algorithm is either undecidable for infinite-state systems or expensive for finite-state systems with a large number of components.

Symmetry reduction exploits structural properties of transition systems to speed up **Algorithm 2**. Authors in [3][21] define symmetry within a state graph as a graph automorphism.

Definition 3.1 (Automorphism) An automorphism [33], which is used to characterize symmetry in a state graph, is a bijection $h : S \to S$ such that:

- $s \in S_0$ if and only if $h(s) \in S_0$.
- If $(s, s') \in \Delta$ if and only if $(h(s), h(s')) \in \Delta$ for all $s, s' \in S$.

Let M be a state graph, $Aut(M)$ be a group of graph automorphisms, and H be a set of graph automorphisms that forms a sub-group under composition of mappings. Then any subgroup G of $Aut(M)$ induces an equivalence relation $\equiv_G$ on the states of M such that:

$$s \equiv_G t \text{ iff } s = \alpha(t)$$

for some $\alpha \in G$. We say that s and t are equivalent and they belong to the same equivalence class. The equivalence class under $\equiv_G$ of a state $s \in S$, denoted $[s]$, can be used to construct a *quotient* Kripke structure $Quot(M) = (S', S'_0, \Delta')$.

Definition 3.2 (Quotient Graph) Let $M = (S, S_0, \Delta)$ be a state graph and let $Aut(M)$ be a group of automorphisms. The quotient graph of M, denoted as $Quot(M)$ and induced by $Aut(M)$, is the graph (S', S'_0, Δ'), where:

- $S' = \{[\text{s}] \mid s \in S\}$.
- $S'_0 = \{[\text{s}] \mid s \in S_0\}$.
- $\Delta' = \{([\text{s}], [\text{k}]) \mid (s, k) \in \Delta\}$.

Theorem 1. Let M be a state graph, G be a subgroup of $Aut(M)$ and ϕ be a safety property. If ϕ is invariant under the group G then:

$$M, s \models \phi \Leftrightarrow Quot(M), [s] \models \phi$$

Therefore, in most cases, $Quot(M)$ is a smaller structure than M and hence, the use of the quotient graph can speed up the verification process. If we can statically detect structural symmetries from the system description and compute a group of automorphisms $Aut(M)$ in advance, then a quotient structure can be constructed. Finally, the model checking tool checks a property ϕ over the quotient graph. We call this process as *Symmetry Reduction.*

3.2 Symmetry Reduction

Symmetry reduction is a technique to tackling state space explosion. For concurrent systems with many replicated processes such as Fischer's protocol or CSMA/CD protocol, the exploitation of structural symmetries in a model can gain a considerable reduction in processing time and memory usage, by a factorial magnitude.

Symmetry reduction in model checking involves replacing a set of equivalent states in a state graph M by a single representative, *rep(s)*, from each equivalence class. It results in a *quotient graph* $Quot(M)$. Providing a property ϕ is invariant under the symmetry used:

$$M \models \phi \text{ if and only if } Quot(M) \models \phi$$

The idea is to build the quotient structure *on-the-fly* using knowledge of a group of automorphisms $Aut(M)$. Instead of backtracking only when a current state s has been reached previously, we now backtrack if $rep(s)$ has been reached previously.

Consequently, we may improve **Algorithm 2** to store and explore only a single representative $rep(s)$ of each equivalence class. **Algorithm 3** presents the modified reachability analysis algorithm. Initially, the set *waiting* still contains the initial states S_0. However, in each while-loop, now the modified algorithm processes in the following way:

- The algorithm returns YES if state s satisfies state property ϕ.
- If property ϕ is false in state s and if state s has been visited before, then the algorithm discards s and repeats the loop.
- Otherwise, s is added to the set *passed* and only representative states of its successors are added to *waiting*.

Since many equivalent states are projected onto the same representative $rep(s)$, the number of visited states may decrease dramatically.

Algorithm 3 Adding Symmetry To The Forward Reachability Analysis.

```
1: $passed = \emptyset$
2: $waiting = rep(S_0)$
3: while $waiting \neq \emptyset$ do
4:     remove $s$ from $waiting$
5:     if $s \models \phi$ then return YES
6:     else if $s \notin passed$ then
7:         add $s$ to $passed$
8:         $waiting = waiting \cup rep(succ(s))$
9: return NO
```

Similarly, we extend the concept of **Algorithm 3** to timed automata. **Algorithm 4** below describes the modified reachability analysis algorithm for timed automata, which improves **Algorithm 1** to store and explore only a single representative $rep(s)$ of each equivalence class.

Algorithm 4 Modified Forward Reachability Analysis Algorithm

```
1: Visited = $\emptyset$
2: $\langle l_f, D_f \rangle$ = Violated States
3: Next = $rep(\langle l_0, D_0 \rangle)$
4: while Next $\neq \emptyset$ do
5:     remove $\langle l, D \rangle$ from Next
6:     if $l = l_f$ and $D \cap D_f \neq \emptyset$ then return YES
7:     if $D \nsubseteq D'$ for all $\langle l, D' \rangle \in$ Visited then
8:         add $\langle l, D \rangle$ to Visited
9:         for $\langle l', D' \rangle$ such that $\langle l, D \rangle \rightarrow \langle l', D' \rangle$ do
10:            add $rep(\langle l', D' \rangle)$ to Next
11: return NO
```

However, there are two challenging problems in the actual implementation of symmetry reduction:

- How to statically detect symmetries from the system description. Then the corresponding symmetry group induces the quotient graph as discussed above.
- Find out optimal solutions to compute $rep(s)$ for any state s.

Moreover, proposed solutions should be computationally inexpensive to protect the gain of using the quotient graph. To cope with the first problem, we propose a technique to detect symmetries automatically from the system description in the next chapter. The second problem is called *orbit problem*. Generally, it is as difficult as testing for graph-isomorphism [10]. Since there is no polynomial algorithms for this problem, in the section we discuss one sub-optimal solution to the orbit-problem in timed automata.

3.3 Representative Function

Given a group of automorphisms, the solution is to convert all explored states to a so-called *normal form*, which represents the equivalence class of the state. The idea is that if two states have the same normal form, then they belong to the same equivalence class.

Let θ be a proposed representative function, then:

$$\theta(s) = \theta(s') \rightarrow [s] = [s'] \text{ for } \forall s, s' \in S.$$

In [26], the authors prove that the orbit problem is even more difficult when the state involves timing. To protect the gain with respect to both time and memory consumption, they revert to the sub-optimal solution, which is adopted in our work.

The idea is to sort states lexicographically within an equivalence class and the representative $rep(s)$ of a state s is defined as the minimal element. Since a symbolic state is a tuple of (L, V, D) - where L and V are sequences of numbers, the problem left is how we sort lexicographically the *zone* attribute.

Let $\mathbb{A}$ be a timed automaton whose local clocks are x and y. $\mathbb{A}$ only performs the operation *reset* on zones. Under these assumptions, there are three possible relations between x and y as follows:

$$x \leq_D y \iff v(x) \leq v(y)$$

$$x \approx_D y \iff v(x) = v(y)$$

$$x <_D y \iff x \leq_D y \wedge \neg(x \approx_D y)$$

where $v(x)$ returns a value of clock x. This is called *diagonal property.*

Lemma 1 (Diagonal Property). [26] Assume that a timed automaton only performs *reset* on zones. For all states and for all clocks x and y, it holds that either $x <_D y$, or $x \approx_D y$ or $y <_D x$.

Let X be a set of clocks. Assume that the equivalence relation $\approx_D$ partitions X into N finite sets $= \{X_1, X_2, ..., X_n\}$, such that:

$$X_i \leq X_j \iff x \leq_D y \ \forall x \in X_i, y \in X_j$$

Clearly the code of X_i, denoted by $C^*(X_i)$, is the lexicographically sorted sequence of the indices of the clocks in X_i. The *zone code* of D, denoted by $C(D)$, is the sequence $(C^*(X_1), C^*(X_2), ..., C^*(X_n))$.

Note that, every zone has exactly one zone code. Instead of sorting *zones*, we can rather lexicographically sort zone codes since zone codes are sequences of numbers.

We define (L, V, D) and (L', V', D') as two symbolic states, we say that (L, V, D) is "smaller" than (L', V', D') if and only if:

$$(L, V, D) < (L', V', D')$$
$$\iff$$
$$(L < L') \vee (L = L' \wedge V < V') \vee (L = L' \wedge V = V' \wedge C(D) < C(D'))$$

This principle is used to sort states lexicographically within an equivalence class effectively. The lexicographically minimum element is selected as the representative.

In the next chapter, we will discuss in detail how to actually "minimize" the state using a group of detected automorphisms.

3.4 Fischer's Protocol

Fischer's protocol ensures mutual exclusion of access to commonly used resources via a shared variable **id**. The protocol relies on location invariants and suitable updates of the variable **id**. We model Fischer's protocol by the real-time system module in PAT. Many examples below in this chapter refer to Fischer's protocol.

Processes of the protocol are instances of a template depicted in Figure 3.1. The template has one local clock **c** and no local variables.

Initially, a process is in location *Start*. The default value of **id** is $Idle$ and clock **c** is set to 0. The transition from *Start* to *Req* is always enabled.

In location *Req*, the process sets **id** to its process identifier and then goes to $Wait$ before 2 time units have elapsed.

In location $Wait$, the process waits for at least 2 time units and then reads **id** again. If **id** has kept the old value, the process may enter its critical section CS. Otherwise, the attempt has failed and the process must go back to *Req*.

In CS location, the process is in its critical section. The transition from CS to *Start* is always enabled.

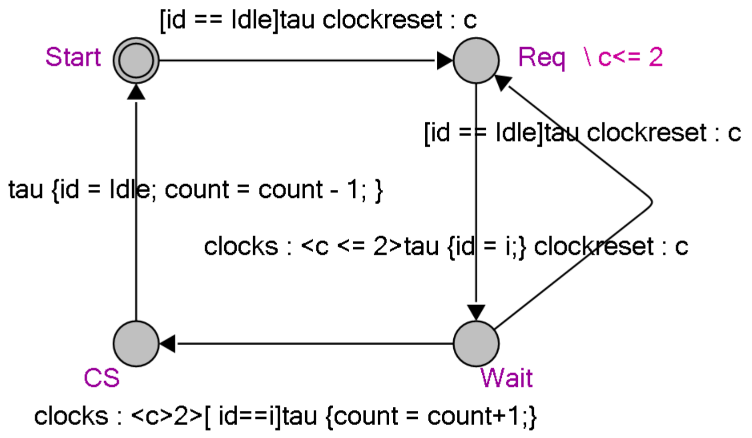

[id == Idle]tau clockreset : c
Start
Req \ c<= 2
[id == Idle]tau clockreset : c
tau {id = Idle; count = count - 1; }
clocks : <c <= 2>tau {id = i;} clockreset : c
CS
Wait
clocks : <c>2>[id==i]tau {count = count+1;}

Figure 3.1: The process template for Fischer's protocol

3.5 Extraction of Automorphisms

In this section, first we define the so-called *state-swap* functions, and second, we define which *state-swap* functions are automorphisms.

3.5.1 State Swap

Based on the concept of *state swap* [26], we define permutations on the state graph, in our case, a PAT model. In a model of a concurrent system with many replicated processes, we restrict attention to automorphisms given by permutations of process identifiers. The state consists of local contributions of various components in the model. Therefore, we use processes which are instantiated from the same template to permute the state.

Let $P(i)$ and $P(j)$ be processes of the system P that originate from the same template T. Therefore, a *state-swap* $swap_{i,j}$: $(L, V, D) \to (L', V', D')$ is defined as follows:

- Process Swap: swaps the contributions to the state given by a pair of $(P(i), P(j))$. Swapping a pair of $(P(i), P(j))$ operates in two steps: 1) interchange the current locations and 2) interchange the values of the local variables and clocks.
- Data Swap: swaps array entries i and j of all dimensions that are indexed by variables of type *pid*. Moreover, it swaps the value i with the value j for all variables of type

pid.

Process Swap is trivial since the processes originate from the same template while in order to perform *Data Swap*, first the tool needs to statically recognize all entities of type *pid* in the model.

Let *Fischer* be an instance of Fischer's protocol with N processes. Processes are instantiated from the process template T given in Figure 3.1. Then *Fischer* is defined in PAT as follows:

$$Fischer = ||| \text{ i} : \{0, ..., N-1\} \text{ @ } T(i)$$

A state of *Fischer* is a tuple (L, V, D), where L is a N-component location vector, V is a set of variable valuations and D is a set of clock valuations.

Example 1: Assume $N = 3$ and also the tool recognizes that **id** has type of *pid*, now we consider the following state s:

- L: $l_0 = Start, l_1 = Wait, l_2 = CS$.
- V: id = 2.
- D: $c_0 = 4, c_1 = 3, c_2 = 2$.

When we apply $swap_{0,2}$ into this state, it results a new state s' by interchanging l_0 with l_2 and c_0 with c_2 (Process Swap), and setting *id* to 0 (Data Swap). s' is given as follows:

- L: $l_0 = CS, l_1 = Wait, l_2 = Start$.
- V: id = 0.
- D: $c_0 = 2, c_1 = 3, c_2 = 4$.

Similarly applying $swap_{1,2}$ to s' gives the following state:

- L: $l_0 = CS, l_1 = Start, l_2 = Wait$.
- V: id = 0.
- D: $c_0 = 2, c_1 = 4, c_2 = 3$.

3.5.2 Action of State Swap on Process Template

When a permutation is applied to a state, it permutes the outgoing actions of the state. Permutation of an action a returns the new action a' with permuted process parameters. Therefore, we define swap functions on the syntax of our models.

A process template in PAT is given in the following syntax:

$$T(x_0, x_1, ..., x_{n-1}) = \{Body\}$$

where T is the template name, $(x_0, ..., x_{n-1})$ is an optional list of template parameters and *Body* determines the computational logic of the process. *Body* consists of local variables v, guards g, updates u and program statements ps over variables and channels [50].

Let T be a process template and π be a *state-swap*. Applying π to T results in a new template $\pi(T)$, where a guard g, an update u and a program statement ps of T is replaced by $\pi(g)$, $\pi(u)$ and $\pi(ps)$ respectively. Finally, $\pi(T)$ is the same as T, except that:

- Any assignment statement $x = val$ is replaced by $x = \pi(val)$, where $type(x) = pid$ and val is a value.
- Any boolean expression $x \sim val$ is replaced by $x \sim \pi(val)$, where $\sim \in (==, \neq)$, $type(x) = pid$ and val is a value.

In other words, *state-swap* is also a syntactic operation on the template T. There is no guarantee that $\pi(T)$ and T are syntactically equivalent.

Example 2: Assume that the system *Fischer* given in *Example 1* does not allow $T(2)$ to enter its critical section. This is achieved by modifying the template T given in Figure 3.1. A guard g on an edge from location *Wait* to location *CS* is now given as:

$$id == i \text{ and } i \neq 2$$

By applying $swap_{0,2}$ into T, it results in a new guard $swap_{0,2}(g)$:

$$id == i \text{ and } i \neq 0$$

It is clear that $swap_{0,2}(T)$ is not identical to T since $T(2)$ is now enabled to enter its critical section in the template $swap_{0,2}(T)$.

3.5.3 Automorphisms

A *state-swap* π is called an automorphism of a system, P, if it satisfies the following three properties:

- It preserves the types of the processes. On other words, it only swaps two processes that are instantiated from the same template.
- It preserves the associations of the processes it maps. It means that given the process associations in P, if a *state-swap* is applied to P, the resulting system $\pi(P)$ should be valid with respect to the given associations.
- It preserves the transition relation of the system such that if there exists a transition between two states s and t in P, then there should be a transition between the states $\pi(s)$ and $\pi(t)$ in $\pi(P)$. On other words, P and $\pi(P)$ must be syntactically equivalent.

Let P be a parameterized system with N processes instantiated from the same template T, where $0 \leq i < N$. We define $Swap(P)$ as a set of possible *state-swaps*, which consists of all permutations of the set of process identifiers $\{0, 1, ..., N-1\}$.

We say that $\pi \in Swap(P)$ is **valid** if it satisfies three properties above. Let $ValidSwap(P)$ be the set of valid *state-swap* functions.

Theorem 2 (Soundness) [26]. Every valid state swap is an automorphism.

3.5.4 Group of Automorphisms

Let G be a group, and let $\alpha_1, \alpha_2, ..., \alpha_n \in G$. The smallest subgroup of G containing the elements $\alpha_1, \alpha_2, ..., \alpha_n$ is denoted $\langle \alpha_1, \alpha_2, ..., \alpha_n \rangle$, and is called the subgroup generated by $\alpha_1, \alpha_2, ..., \alpha_n$. The elements α_i ($1 \leq i \leq n$) are called *generators* for this subgroup. Let X $= \{\alpha_1, \alpha_2, ..., \alpha_n\}$ be a finite subset of G. Then we use $\langle X \rangle$ to denote $\langle \alpha_1, \alpha_2, ..., \alpha_n \rangle$, the subgroup generated by X.

Let H be a subgroup of G, and let $\alpha \in G$. The set $H\alpha = \{\beta\alpha : \beta \in H\}$ is called a right coset of H in G. The set of all right cosets of H in G partitions G into disjoint equivalence classes. In particular, for $\alpha \in H$, we have $H\alpha = H$.

$ValidSwap(P)$ is a set of valid automorphisms, denoted as $\{\pi_1, \pi_2, ..., \pi_n\}$. Under composition of mappings and group theory above, $ValidSwap(P)$ forms a group of automorphisms

$\langle ValidSwap(P) \rangle = \langle \pi_1, \pi_2, ..., \pi_n \rangle$. This means that, $\langle ValidSwap(P) \rangle$ had an identity element π that maps a process p onto itself or $\pi(p) = p$. Also, every other permutation π_i has an inverse element π_i^{-1} such that $\pi_i^{-1}\pi_i(p) = p$.

The next chapter focuses on how we actually implement the symmetry reduction package in PAT.

Chapter 4

Symmetry Reduction Package in PAT

We target systems that are composed of several sub-systems. In each sub-system, processes are instances that are instantiated from the same template. Being motivated by many examples such as Fischer's protocol or CSMA/CD protocol that clearly exhibit structural symmetries, we have extended the real-time model checker PAT with symmetry reduction. It operates in two stages:

- Detect symmetries from the system description. It results in all graph automorphisms that are sound with respect to reachability properties: an automorphism α performs certain permutations on a state s and if a state s has been visited before, then all states $\alpha(s)$ which are obtainable by applying these permutations to s have been also visited.
- Given a group of automorphisms, we generate the symmetry-reduced state space *on-the-fly* and check for a property ϕ using **Algorithm 4**.

4.1 Dining Philosopher

We illustrate the method through the Dining Philosophers example. There are several philosophers who sit around a table with bowls of spaghetti. Forks are placed between every pair of philosophers and a philosopher can only start eating if he acquires both his left fork and the right fork. Figure 4.1 describes the Dining Philosophers example with 6 philosophers and 6 forks, where every edge on the model a shared fork.

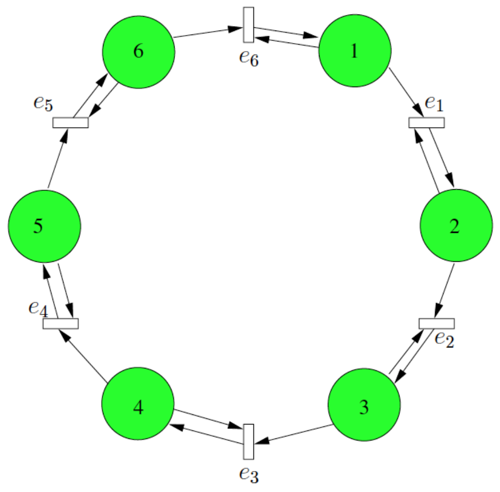


Figure 4.1: Dining Philosophers Example

The philosopher process goes through the internal states T (thinking), *H-1* (hungry-1), *H-2* (hungry-2), E (eating), *R-1* (release-1) and *R-2*(release-2):

- A transition from T to *H-1* is always enabled.
- In state *H-1*, the philosopher tries to acquire the left fork. If the left fork is "busy", it retries in 52 time units. A transition from *H-1* to *H-2* is enabled after the philosopher has acquired the left fork.
- In state *H-2*, the philosopher tries to acquire the right fork. If the fork is "busy", it retries in 52 time units. A transition from *H-2* to E is enabled after the philosopher has acquired the right fork.
- A transition from E to *R-1* is enabled after 52 time units.
- In state *R-1*, the philosopher releases the left fork. A transition from *R-1* to *R-2* is always enabled.
- In state *R-2*, the philosopher releases the right fork. A transition from *R-2* to T is always enabled.

The model has N philosophers instantiated from the template $Phil$ and N forks instantiated from the template $Fork$ below respectively. We define a high-level CSP definition of Dining Philosophers in PAT below:

$$Phil(i) = think.i \rightarrow get.i.(i{+}1)\%N \rightarrow get.i.i \rightarrow eat.i \rightarrow put.i.(i{+}1)\%N \rightarrow put.i.i \rightarrow Phil(i)$$

$$Fork(i) = get.i.i \rightarrow put.i.i \rightarrow Fork(i) \;\Box\; get.i.(i+1)\%N \rightarrow put.i.(i+1)\%N \rightarrow Fork(i)$$

$$P = |||\ \mathrm{i} : \{0, ..., N-1\}\ @\ (Phil(i)\ |||\ Fork(i))$$

4.2 Symmetry Detection In The Specification

In this section, we introduce a technique to automatically detect structural symmetries arising from input process templates. Let P be a parameterized system with N processes instantiated from the same template T, the approach operates in three stages as follows:

- Recognize all valid variables of type *pid* used in T.
- Compute a set of all possible *state-swaps*, denoted $Swap(P)$.
- Then each element $\pi \in Swap(P)$ is checked for validity and finally results in a set of generators $ValidSwap(P)$ that induces a group of automorphisms.

4.2.1 Step 1: Detection of *pid* Variables

As mentioned in the previous chapter, a system composes of a set of process instances and *state-swap* performs certain permutations on a state s, which produces a rearrangement in the following manner: *process-swap* and *data-swap*.

While *process-swap* is trivial since processes originate from the same template, *data-swap* involves swapping variables and array entries of type *pid*. Unlike UPPAAL that a user needs to manually specify entities of type *pid* using the concept of *scalarset*, our method requires no additional information from a modeler.

To do so, we introduce *pid rules*, which allow variables of type *pid* to be used in certain ways as follows:

- (1) The process identifier i has type *pid* by default.
- (2) Given a variable x, $type(x) = pid$ if and only if it is assigned to or compared for equality with another variable var, such that $x \sim var$ where $type(var) = pid$ and $\sim \in [=, \neq, ==]$.
- (3) It is not allowed to perform any arithmetical operations on variables of type *pid*.
- (4) Variable x of type *pid* is only allowed to used in the form of $x \sim v$ where v is either a variable or value and $\sim \in [=, \neq, ==]$.
- (5) $A[N]$ is an array of N elements of type *pid* if and only if $type(A[i]) = pid$.

Given these restrictions, **Algorithm 5** presents the *pid*-type inference algorithm in PAT. **Algorithm 5** runs on the template T. In PAT, T is stored as the syntax tree. Each while-loop iteration involves one pass over the syntax tree. Let $Next$ be a set of *pid* variables to be analyzed and $Visited$ be a set of analyzed *pid* variables. Initially, $Next$ has at most one element - the process identifier i.

Inside a loop, first the model checker extracts a variable x from $Next$. Then the tool visits each guard g, each update u and each program statement ps to verify whether x is used inappropriately (violates any *pid rules*). Otherwise, x is added to $ValidPids$ - the set of valid variables of type *pid*. The algorithm also adds any new variable y, which is related to x by the *second pid rule*, to $Next$. The algorithm only halts when $Next$ is empty.

For any template T whose variables of type *pid* are used inappropriately, T is **invalid**. Currently, we do not support processes instantiated from invalid process templates to run with symmetry reduction.

Clearly, for systems that consist of N sub-systems (means that the system description has N input process templates), **Algorithm 5** will run N times.

Let P be a case of Dining Philosophers with just two philosophers ($phil_1, phil_2$) and the two forks ($fork_1, fork_2$), which contains deadlock. First **Algorithm 5** runs on the template $Phil$. The template parameter i has type of *pid* by default and is added to $Next$. In the first loop, the tool extracts i from $Next$ and goes through the computation logic of the template $Phil$. Since i does not violate or break *pid rules*, i is added to $ValidPids$. Since there is no variable that is related to i in the form defined in (2), **Algorithm 5** halts. Similarly, **Algorithm 5** runs on the template $Fork$. Since there is no invalid *pid* variable used in those templates, we say that P is valid to run with symmetry reduction.

Algorithm 5 *pid*-type Inference Algorithm

```
ValidPids = $\emptyset$
Visited = $\emptyset$
Next = $\{i\}$
isInvalidTemplate = false
**while** Next $\neq \emptyset$ **do**
    remove $x$ from Next
    **for** each guard $g$ / statement $ps$ / update $u$ in T **do**
        **if** $g$ / $ps$ / $u$ contains $x$ **then**
            **if** $g$ / $ps$ / $u$ is in the form defined in (4) **then**
                **if** $g$ / $ps$ / $u$ contains a variable $y$ **then**
                    **if** (y not $\in$ Visited) and (y not $\in$ Next) **then**
                        add $y$ to Next
            **else**
                isInvalidTemplate = true
                break
    **if** isInvalidTemplate == false **then**
        add $x$ to ValidPids
        add $x$ to Visited
**return** FINISHED
```

4.2.2 Step 2: Compute a Set of Possible Permutations

Once all valid variables of type *pid* have been recognized, the next step is to compute a set of all possible *state-swap* from the system description.

In the Dining Philosophers problem considered, assuming $phil_1, phil_2$ are assigned indices of 1, 2 and $fork_1, fork_2$ are assigned indices 3, 4 respectively. We define $Swap(P)$ as a set, which consists of all permutations of the set of process identifiers $\{1, 2, 3, 4\}$. It results in a set of 24 elements to check for validity since only certain permutations are sound with respect to reachability properties.

4.2.3 Step 3: Compute a Set of Generators

A *state-swap* π is called an automorphism of a system, if it satisfies the following three properties:

- It preserves the types of the processes.
- It preserves the associations of the processes it maps.
- It preserves the transition relation of the system.

For a given system P, the set of all possible *state-swap*, $Swap(P)$, is computed from the previous step. For each element in this set, the algorithm checks if it is a valid permutation which means that it satisfies three properties above.

Back to the Dining Philosophers problem considered. The first property means that two processes must be instantiated from the same template. After checking for the first property and eliminating all permutations that map a philosopher to a fork and vice versa, we are left with 4 permutations namely:

$$\{1,2,3,4\}, \{2,1,3,4\}, \{1,2,4,3\}, \{2,1,4,3\}$$

The second property says that if a *state-swap*, π, maps $phil_i$ to $phil_j$ then the permutations of all forks associated with $phil_i$ are now the forks associated with $phil_j$ in the new system. These associations must be consistent with the original system description otherwise π is not an automorphism. In our example, the second property is violated by

$$\{2,1,3,4\}, \{1,2,4,3\}$$

Specifically, in $\{2,1,3,4\}$, $phil_2$ will have $fork_2$ on the right and $fork_1$ on the left which violates the original system description. **Algorithm 6** describes how a *state-swap* is checked for the second property.

Algorithm 6 Implement CheckAssociation

```
1: if processes associated with $p_i = \{p_1, p_2, ....\}$ then
2:     processes associated with $p_j = \{\pi(p_1), \pi(p_2), ....\}$
3:     if $\{\pi(p_1), \pi(p_2), ....\}$ consistent with equations of $p_j$ then
4:         return true;
```

Although processes are obtained as instances of a same parameterized process template, they are not necessarily identical up to renaming. The third property indicates that after the permutation, the new system must be equivalent to the original system. On other words, P and $\pi(P)$ must be syntactically equivalent. This happens in **Algorithm 7**. In PAT, process templates are stored as syntax trees. **Algorithm 7** performs a depth-first-search on both trees and checks for their equivalence.

Algorithm 7 Implement CheckEquivalence

```
1: TreeNode A = $P$
2: TreeNode B = $\pi(P)$
3: public boolean isEquivalent(TreeNode A, TreeNode B):
4: if isEquivalent(A.left, B.left) then
5:     if isEquivalent(A.right, B.right) then
6:         if A is equal to B then
7:             return true;
8: return false;
```

Finally, given a set of all permutations $Swap(P)$, **Algorithm 8** shows how to compute a set of generators $ValidSwap(P)$ that forms a group of automorphisms.

Algorithm 8 Compute $ValidSwap(P)$

```
1: Compute $Swap(P)$
2: $ValidSwap(P) = \emptyset$
3: for each $\pi \in Swap(P)$ do
4:     if $Type(p_i) == Type(p_j)$ then
5:         if $CheckAssociation(p_i, p_j)$ then
6:             if $CheckEquivalence(p_i, p_j)$ then
7:                 add $\pi$ to $ValidSwap(P)$.
8: return $ValidSwap(P)$
```

$ValidSwap(P)$ forms a group of automorphims $Aut(P)$ under functional composition. This means that, $Aut(P)$ had an identity element π that maps a process p onto itself or $\pi(p) = p$. Also, every other permutation π_i has an inverse element π_i^{-1} such that $\pi_i^{-1}\pi_i(p) = p$. In our example, $\pi_1 = \{1, 2, 3, 4\}$ has an inverse element as $\pi_2 = \{2, 1, 4, 3\}$.

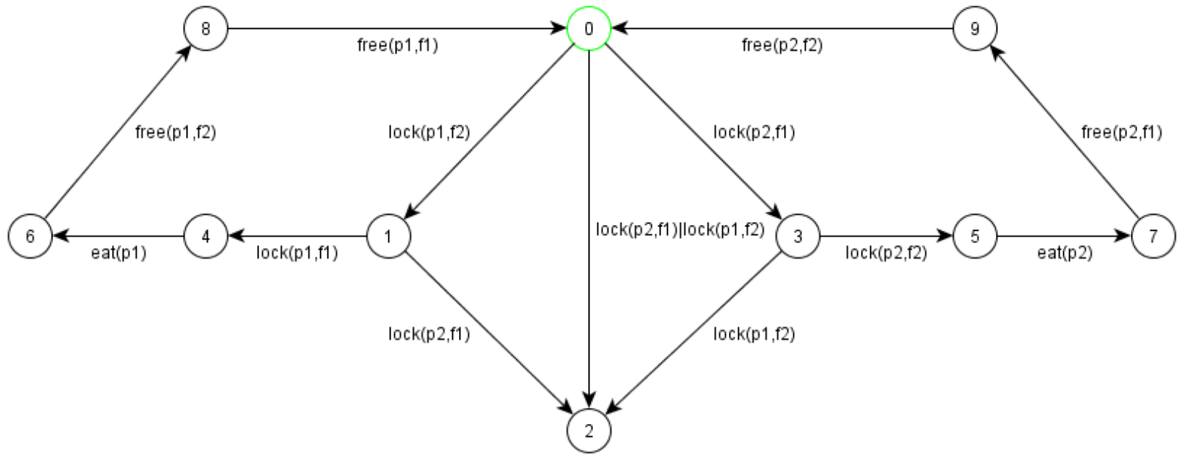


Figure 4.2: The state space of the dining philosopher.

4.3 Generate The Symmetry-reduced State Space

Once symmetry detection has been performed, **Symmetry Reduction** is performed to generate the symmetry-reduced state space on-the-fly using **Algorithm 4**.

Figure 4.2 depicts the state space of our Dining Philosopher example, where *lock* represents a busy fork and $free$ means that a fork is free to use. The state-space is partitioned into 6 equivalence classes namely:

$$\{0\}, \{1,3\}, \{2\}, \{4,5\}, \{6,7\}, \{8,9\}$$

Algorithm 4 presents the modified reachability analysis algorithm, where θ as a representative function that converts a state s to its representative $rep(s)$. Since many equivalent states are projected onto the same representative $rep(s)$, the number of visited states may decrease dramatically.

Assume that a timed automaton only performs $reset$ on zones, we minimize a state s using the **Algorithm 9** below, which results in the representative $rep(s)$. It is clear that $rep(s)$ satisfies the soundness, since states are transformed using automorphisms.

Algorithm 9 Compute The Representative State $Rep(s)$

```
1: $s = (L, V, D)$
2: $ValidSwap(P) \neq \emptyset$
3: $N$ = size(P)
4: for $i = 1$ to $N - 1$ do
5:     for $j = i + 1$ to $N$ do
6:         if $swap_{i,j}(s) < s$ and $swap_{i,j} \in ValidSwap(P)$ then
7:             $s = swap_{i,j}(s)$
```

Chapter 5

Parameterized Compositional Model Checking in Timed Automata

Although symmetry reduction can result in an exponential savings in processing time and memory consumption for highly symmetric networks, state-space explosion generally limits model checking to protocol instances that are much smaller than those that arise in practice. What if an input model has a very large number of components? This is referred to as the parameterized model checking problem (PMCP). The problem is, however, generally undecidable [5].

In [44], we introduce a new and different formulation, which is referred to as the parameterized compositional model checking problem (PCMCP), asks whether a parameterized family has a compositional proof that the specification is met for all instances.

Specifically, verification is reduced to one on a fixed set of representative nodes, making the time complexity for computing the compositional invariant independent of N. Moreover, it is sometimes possible to pick the same set of representatives for all networks in a family. In such case, the compositional invariant computed for a small instance forms a parameterized invariant which holds for all members of the family.

The positive results are based on symmetry arguments that establish the existence of compositional cutoffs: small instances whose compositional verification induces invariants that hold for the entire family. We show that for regular network families, such as the ring, torus, and cube-connected cycles, for the synchronous control-user networks of German and Sistla [23], for asynchronous shared-memory networks from [22], and for distributed memory control-user networks with an index-oblivious control process [44], the verification is decidable in polynomial time.

In this chapter, we extend our previous work in [44] to parametrized timed systems. Our results show that for networks of mutual exclusion protocols, the verification is decidable in polynomial time in the size of the processes. Generally for a control-user system with a non-oblivious controller, both the PCMCP is undecidable since the local state space of a control process is unbounded. However, for Train-Gate protocol and CSMA/CD protocol, we define an abstract control process and show that its compositional invariant is precise to solve the PCMCP.

As this is a new formulation of parameterized verification, we still need to do more work and discuss the implications in more depth. However, our positive results show that many real-time protocols can be verified in polynomial time in the size of the processes since the topology of neighborhoods is usually less complicated than that of the entire graph, which simplifies verification.

5.1 Preliminaries

5.1.1 Process

A real-time process is defined by a tuple (S, I, T, V), where S is a symbolic state space; I is a subset of S, the initial set of states; T is a transition relation, a subset of S x S and V is a set of variables. The transition relation and initial condition induce a set of reachable symbolic states (i.e., states which are obtained from an initial symbolic state through a sequence of transitions).

5.1.2 Internal State

An *internal state* of a real-time process is its symbolic state, which is defined as a pair (l, D) where l is a location and D is a zone. By definition, a zone is the solution set of clock constraints.

5.1.3 Neighborhood

The neighborhood of a real-time process is the set of variables which are shared between that process and other processes. For example, the neighborhood of a node i in a Fischer's protocol of size N is the global variable id.

5.1.4 Local State

Each local state of a node P_i in a real-time system P of size N can be written in the form (l_i, D_i, y), where l_i is a current location of P_i, D_i is a current zone of P_i and y is a vector of states or values of the neighborhoods of P_i.

5.1.5 Inductive Invariant

A real-time process P is defined as a tuple (S, I, T, V). An *invariant* is a *predicate* which holds of all reachable symbolic states.

An inductive invariant is a predicate that includes all initial states and is closed under the transition relation. That is, θ is an inductive invariant of P = (S, I, T, V) if:

- θ includes all initial states.
- θ is closed under transitions.

5.1.6 Interleaved Composition of Processes

An asynchronous, interleaved composition of real-time processes $P_1 = (S_1, I_1, T_1, V_1)$ and $P_2 = (S_2, I_2, T_2, V_2)$, written $P = P_1 \mid\mid\mid P_2$, is defined as the process P = (S, I, T, V) where:

- The set of variables, V, is $V_1 \cup V_2$. The set of shared variables is $V_1 \cap V_2$.
- The set of initial states, I, such that its projection on P_1 is in I_1 and the projection on P_2 is in I_2.
- The transition relation T interleaves transitions of P_1 and P_2, where transitions of one process leave the internal variables of the other process unchanged.

5.1.7 Compositional Invariants

There is a predicate, θ_i for each process P_i; this is a set of local states of P_i. The constraints which the $\{\theta_i\}$ predicates must satisfy to be called a compositional invariant are as follows:

- (init) θ_i includes the initial states of P_i.
- (step) θ_i is inductive for P_i.
- (non-interference) the actions of a neighboring process, P_j, do not falsify θ_i.

5.1.8 Parameterized Compositional Invariants

A compositional invariant for a parameterized family is defined using an unbounded set of compositional constraints. There is a θ-component for each node i in each network N of the family; this is denoted as $\theta_{(i,N)}$. The components must meet the previously defined constraints for compositional invariance:

- (init) $\theta_{(i,N)}$ includes the initial states of $P_{(i,N)}$.
- (step) $\theta_{(i,N)}$ is inductive for $P_{(i,N)}$, and
- (non-interference) the actions of a neighboring process, $P_{(j,N)}$ in network N, do not falsify $\theta_{(i,N)}$.

5.1.9 Compositional Cutoff

Although the vector θ is unbounded, there is still a strongest fix-point solution. As processes from different instances do not in influence one another, this fix-point is the collection of strongest fix-points for each instance. The decidability results in this paper are obtained by collapsing the unbounded collection of constraints to a bounded set through the identification of local (i.e., neighborhood) symmetries.

Network families examined in this paper (Fischer's protocol, CSMA/CD protocol and Train-Gate protocol) have the following property: there is a limit, say K, such that the strongest compositional invariants in networks of size greater than K are identical (up to neighborhood isomorphism) to the strongest compositional invariants in net- works of size at most K. We then refer to K as a *compositional cutoff.*

5.2 Local Symmetry

Two nodes m and n are locally similar, written $m \simeq n$, if there is a bijective function β that the neighborhood of m is isomorphic to the neighborhood of n through β. Tuples of the form (m, β, n) where β is a witnessing bijection for $m \simeq n$, are called *local symmetries.*

For a local symmetry (m, β, n), the isomorphism β maps the neighborhood of m onto the neighborhood of n. We now lift this definition on structure to include the processes running at m and n. Thus β maps a local state (x, y) of m to a local state $(x, \beta(y))$ of n (recall that $x = (l, D)$ is the internal state and y is the neighborhood state), and similarly maps a local transition $((x, y), (x', y'))$ of m to a local transition $((x, \beta(y)), (x', \beta(y')))$ of n. This is lifted to sets of states and transitions in the standard way, that is for every (m, β, n) that respects the local symmetries, it should hole that:

$$[T_n \equiv \beta(T_m) \text{ and } [I_n \equiv \beta(I_m)]$$

Fix a network G and a process assignment. The local symmetries of the network are defined as a relation B with entries of the form (m, β, n), where is itself a relation between the local state spaces of processes m and n. A relation B forms a local symmetry if, for every $(m, \beta, n) \in B$, any step or interference transition of m can be simulated by a step or interference transition of n. More precisely, the following forward-simulation properties hold.

- (initial match) For every initial state x of the process at m, there is an initial state y of the process at n such that $(x, y) \in \beta$.
- (local simulation) If $(x, y) \in \beta$ and there is a transition $(x, x') \in T_m$, then there is y' such that $(y, y') \in T_n$ and $(x', y') \in \beta$.
- (interference simulation) If $(x, y) \in \beta$, and i is a neighbor of m, and there is a joint (i, m) transition $([a, x], [a', x'])$ due to T_i, then either there is a neighbor j of n for which (i, γ, j) is in B and for every b such that $(a, b) \in \gamma$, there is a joint (j, n) transition $([b, y], [b', y'])$ due to T_j, such that $(x', y') \in \beta$, or there is a transition (y, y') in T_n such that $(x', y') \in \beta$.

Corollary 1: Let B be a local symmetry on G. Let θ be the strongest compositional invariant on G. Let P be a property of local states. If (m, β, n) is in symmetry B, and P is invariant under β, then $[\theta_m \implies P]$ if and only if $[\theta_n \implies P]$.

Theorem 3: For a network with global symmetry group G,

$$\mathit{Local}(G) = \{(m, \beta, n) \mid \beta \in G \wedge \beta(m) = n\}$$

is a balance relation.

That is, balanced nodes have isomorphic strongest compositional invariants. A network with a transitive group of automorphisms (i.e., one where any pair of nodes is connected by an automorphism) is called *vertex-transitive*. We have the following corollary.

Corollary 2: A real-time network with a transitive group of automorphisms (i.e., one where any pair of nodes is connected by an automorphism) is called *vertex-transitive*. In a vertex-transitive network, any pair of nodes is balanced and there is a single equivalence class.

Proof: Consider any pair of nodes m, n. As the network has a transitive symmetry group G, there is an automorphism β in G such that $\beta(m) = n$. In that case, the triple $(m, \beta, n$ is in $\mathit{Local}(G)$ by definition. As $\mathit{Local}(G)$ is a balance relation, m and n are balanced and the orbit relation is an equivalence, so that m and n are in the same equivalence class of $\mathit{Local}(G)$. Hence, there is a single equivalence class.

5.3 Parameterized Network Families

For a parameterized network, local symmetries that span members of a network family can be used to reduce the problem of checking a property for all instances to checking it for a small, fixed-size set of representative instances.

We apply the local symmetry definitions to a family of networks by redefining the symmetry relation to relate two nodes in (possibly) different networks. I.e., the relation consists of triples $((G, m), \beta, (H, n))$. We immediately obtain the analogues of Corollary 1 for a parametric network family.

Corollary 2: Let P be a property of local states. If $((G, m), \beta, (H, n))$ is in symmetry B, and P is invariant under β, then $[\theta(G, m) \implies P]$ if and only if $[\theta(H, n) \implies P]$.

Below we show how we generally analyze a real-time system based on the theory in the previous sections:

- Detect a set of automorphisms from the system description. Define a symmetry B (from global symmetry) for the entire network family, with finitely many equivalence classes.
- Find a representative network instance R, whose nodes cover all of the equivalence class. This is achieved by neighborhood abstraction. This technique obtains a uniform invariant which applies to all nodes by abstracting from differences between node neighborhoods (such as the number of neighbors).
- Compute the strongest compositional invariant for R.
- Let property P be invariant under the symmetries in B. By Corollary 2, if P holds for all nodes k in R, it holds of all nodes in the entire family.

We target systems that are composed of several sub-systems. In each sub-system, processes are instances that are instantiated from the same template. They likely exhibit clearly global symmetries and hence induce local symmetries. For those systems that are *vertex-transitive*, in order to extend the global symmetry reduction to a whole family of networks, say that a family of process networks, N, is uniform, we must prove:

- (1) each network in the family is *vertex-transitive*.
- (2) for every pair (M, N) of networks, there is a pair of nodes, $m \in M$ and $n \in N$, that are locally symmetric.
- (3) nodes that are locally symmetric are assigned isomorphic processes, whose state space is independent of network size.

Chapter 6

Verification for Fischer's Protocol

Fischer's protocol ensures mutual exclusion of access to commonly used resources via a shared variable **id**. In fact, the protocol relies on location invariants and suitable updates of **id**. In this chapter, we model Fischer's protocol by the real-time system module in PAT and run the verification with symmetry reduction. The experimental results show that we gain a considerable reduction in the cost of analysis, by a factor exponential in the number of processes. Then we apply our compositional verification technique to determine whether the mutual exclusion property holds for every instance for networks of Fischer's protocol. We prove that verification is decidable in time polynomial in the state space of the smallest verified, "cut-off", instance.

6.1 Modeling Process Behavior

Processes of the protocol are instances of a template depicted in Figure 6.1. The template has one local clock **c** and no local variables.

Initially, a process is in location *Start*. The default value of **id** is *Idle* and clock **c** is set to 0. The transition from *Start* to *Req* is always enabled.

In location *Req*, the process sets **id** to its process identifier and then goes to *Wait* before 2 time units have elapsed.

In location *Wait*, the process waits for at least 2 time units and then reads **id** again. If **id** has kept the old value, the process may enter its critical section *CS*. Otherwise, the

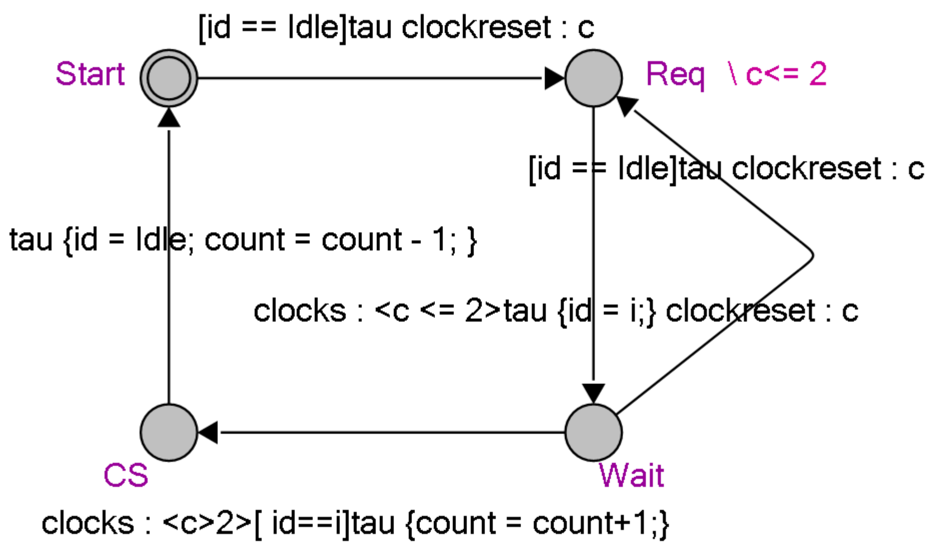

[id == Idle]tau clockreset : c
Start
Req \ c<= 2
[id == Idle]tau clockreset : c
tau {id = Idle; count = count - 1; }
clocks : <c <= 2>tau {id = i;} clockreset : c
CS
Wait
clocks : <c>2>[id==i]tau {count = count+1;}

Figure 6.1: The process template for Fischer's protocol

attempt has failed and the process must go back to *Req*.

In *CS* location, the process is in its critical section. The transition from *CS* to *Start* is always enabled.

6.2 Modeling Fischer's Protocol

The protocol consists of N processes interleaving with each other. In PAT, we model this as follows:

$$Fischer = ||| \text{ i} : \{0, ..., N-1\} \text{ @ } Process(i)$$

6.3 Verification Properties

MutualExclusionFail is a boolean condition *true* of global states, where more than one process are in the local state *CS* at the same time.

In PAT, *MutualExclusionFail* is defined as follows:

$$define\ MutualExclusionFail\ count > 1;$$

In order to formally verify our model of Fischer's protocol is correct, we check whether the model reaches to *MutualExclusionFail*:

$$assert\ Fischer\ reaches\ MutualExclusionFail$$

6.4 Symmetry Detection

In order to explain how symmetry detection is performed in PAT, a simple model $Fischer$ of Fischer's protocol has been considered. It consists of 3 processes, T_1, T_2 and T_3, which are instantiated from the same template T given in Figure 6.1.

Symmetry detection consists of 3 following steps:

- Stage 1: Detect all variables of type *pid* used in T.
- Stage 2: Compute $Swap(Fischer)$.
- Stage 3: Compute $ValidSwap(Fischer)$.

Stage 1. Algorithm 5 shows how the *pid-type* inference technique is performed on T.

The template parameter i has type of *pid* by default. In the first while-loop, the tool extracts i from $Next$. Then the program goes through the computation logic of the process template T (including guards g, updates u, and program statements ps over variables and channels) to make sure that i does not violate or break *pid* rules. Finally, i is added to $ValidPids$. The algorithm also adds the global variable id, which is related to i by the guard g as follows:

$$id == i$$

to $Next$. In the second loop, id is extracted from $Next$ and its validity is verified. The algorithm halts since $Next$ is empty now. We conclude that processes instantiated from T are valid to run with symmetry reduction and id is also the variable of type *pid*.

Stage 2. Assuming that T_1, T_2, T_3 are assigned indices of 1, 2, and 3 respectively, all possible permutations of the set of indices $\{1, 2, 3\}$, denoted $Swap(Fischer)$ are obtained. This results in 6 permutations as follows:

$$\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}$$

Stage 3. A permutation π, of process indices is called an automorphism of a system, if it satisfies the following three properties:

- It preserves the types of the processes.
- It preserves the associations of the processes it maps.
- It preserves the transition relation of the system.

All possible permutations $Swap(Fischer)$ ensure the first property since they all originate from the same template T.

For networks of Fischer's protocol, there is no association between processes. Therefore, all possible permutations also satisfies the second property.

We apply each element $\pi \in Swap(Fischer)$ into $Fischer$ and hence, form a new system $\pi(Fischer)$. In other words, $\pi(Fischer)$ is a syntactic operation on the system $Fischer$. We show that all $\pi \in Swap(Fischer)$ satisfy the third property.

So we conclude that $Fischer$ is fully symmetric since all processes are identical up to renaming. Therefore, we have:

$$ValidSwap(Fischer) = Swap(Fischer)$$

From Theorem 3, $Fischer$ consists of N balanced processes.

6.5 Symmetry Reduction and Experimental Results

We have run experiments on PAT for different numbers of processes and Table 6.1 summarizes the verification results. The environment is an i5-dual-core machine with 4 GB memory.

To demonstrate the effectiveness of symmetry reduction, we ran each experiment twice,

Processes	8		10		15		20		30	
Mode	Sym	No	Sym	No	Sym	No	Sym	No	Sym	No
Time (s)	0.04	3.3	0.13	122	0.85	N/A	4.35	N/A	53	N/A
Visited State	190	85495	360	1827331	1379	N/A	3880	N/A	17717	N/A
Memory (Mb)	9.3	57.7	11	1347.5	11.8	N/A	22	N/A	109	N/A

Table 6.1: Experimental Results for Fischer's Protocol

with and without symmetry reduction. Experiments were run with a 300 second timeout. We focus on three criteria: processing time (s), the number of visited states and memory usage (MB). The data shows that the regular PAT's limit for Fischer's protocol is less than 15 processes while the verification for 30 processes can be done within 53 seconds using 109MB of memory with symmetry reduction.

Using symmetry reduction, the verification tool gains a considerable reduction in processing time and memory usage, by a factorial magnitude, however verification is still not feasible for an instance of Fischer's protocol with a very large number of processes (> 100). Here is the moment when our compositional verification method will prove its advantage.

6.6 Compositional Verification

Consider a family of instances of Fischer's protocol $\{R_i\}$, where each instance consists of i identical processes. We combine elements of compositional proofs, abstraction and local symmetry to verify whether mutual exclusion holds for every instance of Fischer's protocol.

Since Fischer's protocol is fully symmetric, every instance R_i is *vertex-transitive*. In a instance R_x, we define *Rep* as a single representative process and θ_x as a compositional invariant for *Rep*. A state in θ_x is a tuple $(l, D, \mathbf{id})$, where (l, D) is an internal state of *Rep* and **id** is a *neighborhood* ranging from $\{0, ..., x-1\}$. Since x could have any possible positive value, θ_x may unbounded. We define an abstraction of *Rep*, denoted $\widehat{Rep}$ and show that its compositional invariant is sufficiently precise to solve the PCMCP.

In the abstract representative process $\widehat{Rep}$, the transitions are the same as in *Rep*, except that **id** has a value in the set of $\{k, \hat{k}, -1\}$, where k is the process identifier of *Rep* and $\hat{k}$ is the abstract process identifier of any process in R_x that its process identifier is not equal to k. The abstraction is a Galois connection (α, γ) where $\alpha(s, \mathbf{id}) = (s, a)$ where a is the set of three possible values $\{k, \hat{k}, -1\}$ and $\gamma(s, a) = \{(s, \mathbf{id}) \mid \alpha(s, \mathbf{id}) = (a, s)\}$.

We define Δ_x is the strongest compositional invariant on the abstract process $\widehat{Rep}$, the first lemma says that the compositional invariant of the abstract process over-approximates the concrete one.

Lemma 1 For each state in θ_x, there is an α-related state in Δ_x.

Proof: We define D_i as a finite set of zones at the location i. In location *Start*, Δ_x^{Start} is just the state $(Start, D_{Start}, -1)$ with the *location* = *Start*, and $\mathbf{id} = -1$. So $\Delta_x^{Start} = \theta_x^{Start}$. Hence, the hypothesis holds for Δ_x^{Start}.

In *Req*, $\theta_x^{Req} = \{(Req, D_{Req}, i) \mid i \in \{-1, .., x-1\}/k\}$, where x is the size of the instance R_x. Δ_x^{Req} has one of two possible values including $(Req, D_{Req}, -1), (Req, D_{Req}, \hat{k})$. It is clear that each state of *Req* in the set of $\{(Req, D_{Req}, i) \mid i \in \{0, ..., x-1\}/k\}$ is related to $Req, D_{Req}, \hat{k})$ by α. So it satisfies the condition required.

In location $Wait$, there are two possible cases. If $\mathbf{id} = k$, $(Wait, D_{Wait}, k) \in \theta_x^{Wait}$ and also $(Wait, D_{Wait}, k) \in \Delta_x^{Wait}$. Otherwise, $\{(Rep, D_{Wait}, i) \mid i \in \{-1, ..., x-1\}/k\} \in \theta_x^{CS}$, which are related to $(Rep, D_{Wait}, \hat{k}) \in \Delta_x^{Wait}$ by α. Hence, the hypothesis holds for Δ_x^{Wait}.

In location CS, $\theta_x^{CS} = \Delta_x^{CS} = (CS, D_{CS}, k)$. **End Proof**

Theorem 4 The PCMCP is decidable in polynomial time for Fischer's protocol.

Proof Consider an instance R_x as the smallest verified instance ($x = 2$) and a random instance R_y. From Lemma 1, all states in Δ_x also satisfy the mutual exclusion property. Δ_x and Δ_y are isomorphic since they have the same local states. We can say Δ_x and Δ_y are locally symmetric and hence, from Corollary 1, all states in Δ_y also ensure mutual exclusion. Since for each state in θ_y, there is an α-related state in Δ_y. It is clear that θ_y satisfies mutual exclusion as expected. So the PCMCP is decidable in polynomial time. **End Proof**.

Chapter 7

Verification for CSMA / CD Protocol

The Carrier Sense, Multiple Access with Collision Detection (CSMA/CD) protocol describes one solution to the problem in Ethernet network, when several agents compete for a single bus. The simplified algorithm of CSMA/CD is shown in Figure 7.1. The research group in PAT has successfully done modeling and verification on CSMA/CD protocol [48]. In this chapter, we extend the previous work [48] to verify CSMA/CD protocol with symmetry reduction. Our results show that we gain a considerable reduction in the cost of analysis, by a factor exponential in the number of processes. We also prove that for networks of CSMA/CD protocol, verification is decidable in time polynomial in the state space of the smallest verified, "cut-off", instance.

7.1 Model For CSMA/CD Protocol

CSMA/CD protocol consists of two components, namely *Sender* and *Bus*. Two components communicate by pair-wise synchronization channels. Roughly speaking, a *Sender* must first listen to the *Bus*. If the *Bus* is idle, the *Sender* begins to transmit. Otherwise, it must wait and retry later. However, collision may occur when more than one *Sender* are sending message via the *Bus*. Then the *Bus* informs all *Senders* of this collision, and abort their transmission immediately. Therefore, all transmitting messages are discarded. Also, we make assumptions that no messages are lost during transmitting. We list all variables and processes of this model with a simplified description, as illustrated in Figure 7.2.

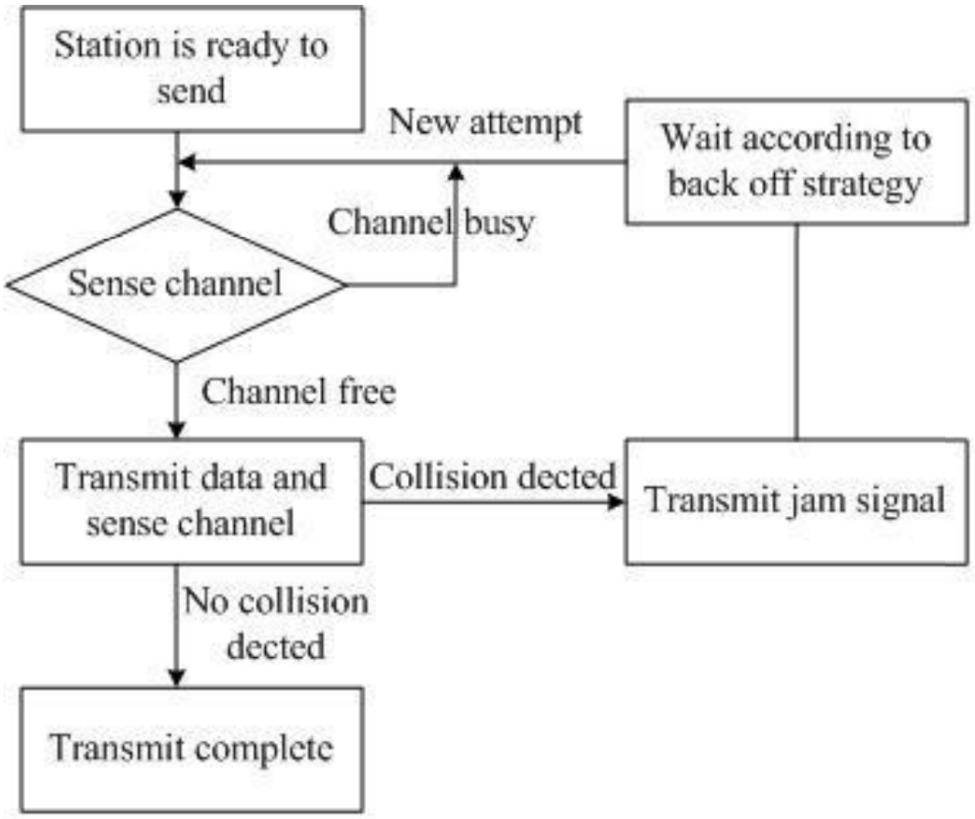
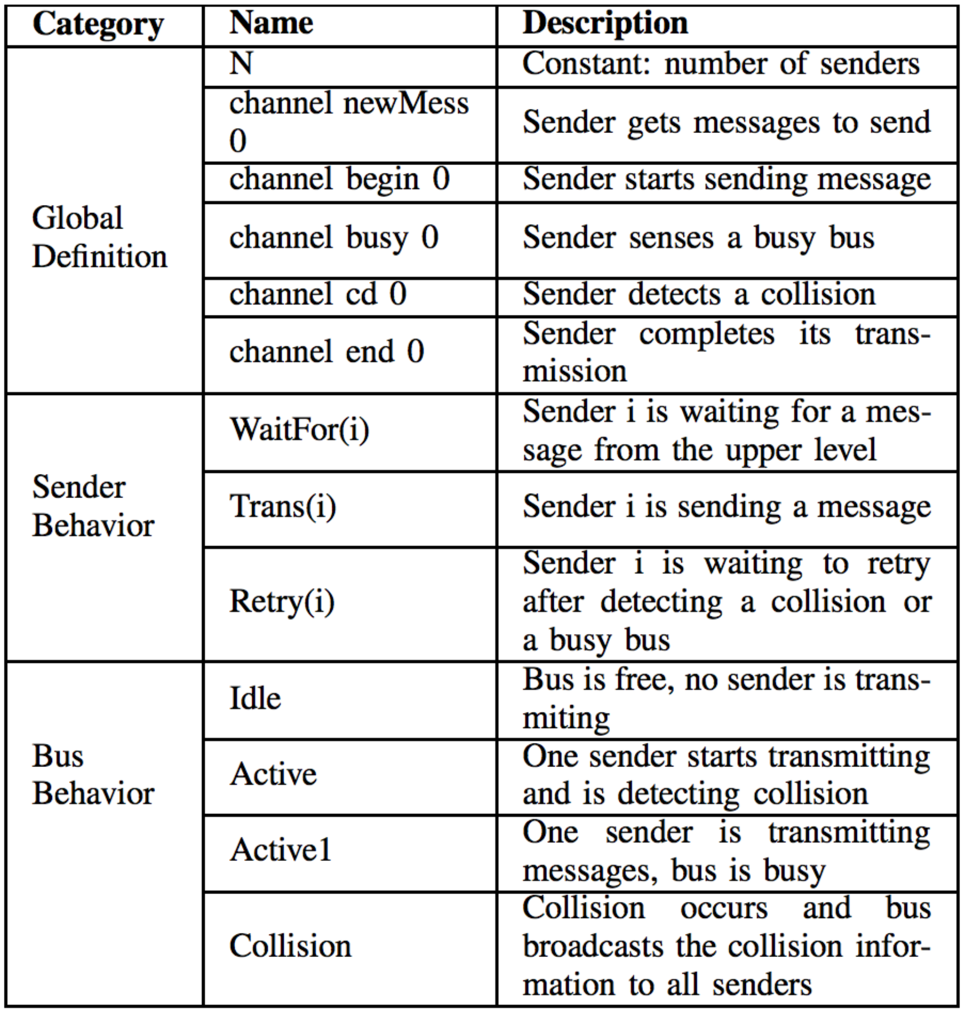


Figure 7.1: An Overview of CSMA/CD Protocol



Category	Name	Description
Global Definition	N	Constant: number of senders
	channel newMess 0	Sender gets messages to send
	channel begin 0	Sender starts sending message
	channel busy 0	Sender senses a busy bus
	channel cd 0	Sender detects a collision
	channel end 0	Sender completes its transmission
Sender Behavior	WaitFor(i)	Sender i is waiting for a message from the upper level
	Trans(i)	Sender i is sending a message
	Retry(i)	Sender i is waiting to retry after detecting a collision or a busy bus
Bus Behavior	Idle	Bus is free, no sender is transmiting
	Active	One sender starts transmitting and is detecting collision
	Active1	One sender is transmitting messages, bus is busy
	Collision	Collision occurs and bus broadcasts the collision information to all senders

Figure 7.2: Components of CSMA/CD Protocol

$$
\begin{aligned}
\mathit{WaitFor}(i) = {} & (cd?i \rightarrow \mathit{WaitFor}(i)) \\
& \Box\ (\mathit{newMess}!i \rightarrow ((\mathit{begin}!i \rightarrow \mathit{Trans}(i)) \\
& \qquad \Box\ (\mathit{busy}?i \rightarrow \mathit{Retry}(i)) \\
& \qquad \Box\ (cd?i \rightarrow \mathit{Retry}(i))));
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Trans}(i) = {} & (cd?i \rightarrow \mathit{Retry}(i)\mathit{within}[0, 52]) \\
& \Box\ (\mathit{atomic}\{\mathit{end}!i \rightarrow \mathit{Skip}\}\mathit{within}[808, 808]; \\
& \quad \mathit{WaitFor}(i));
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Retry}(i) = {} & (\mathit{newMess}!i \rightarrow ((\mathit{begin}!i \rightarrow \mathit{Trans}(i)\mathit{within}[0, 52]) \\
& \quad \Box\ (\mathit{busy}?i \rightarrow \mathit{Retry}(i)\mathit{within}[0, 52]) \\
& \quad \Box\ (cd?i \rightarrow \mathit{Retry}(i)\mathit{within}[0, 52])));
\end{aligned}
$$

Figure 7.3: Model for a Sender i [48]

7.1.1 Modeling Sender Behavior

The behavior of component *Sender* is defined in Figure 7.3. Initially, the *Sender* is in location $WaitFor$. When there is a message to send, if the *Bus* is idle, the *Sender* goes to location $Trans$. Otherwise, if the *Bus* is busy or a collision is detected, it moves to location *Retry*. If a collision occurs while no message is arrived, the *Sender* remains in location $WaitFor$.

In location $Trans$, the *Sender* has two transitions. If a collision is detected within 52 time units, the *Sender* goes to location *Retry*. Otherwise, it terminates sending the message after exactly 808 time units, then it goes to location $WaitFor$.

In location *Retry*, if the *Bus* is idle, the *Sender* moves back to location $Trans$ within 52 time units. Otherwise, it remains in location *Retry*.

7.1.2 Modeling Bus Behavior

The behavior of component *Bus* is showed in Figure 7.4. Initially, the *Bus* is in location $Idle$. The transition from $Idle$ to *Active* is enabled when one *Sender* begins to transmit.

In location *Active*, there are three possible transitions. If the *Sender* completes sending, the *Bus* goes back to the initial location. If another *Sender* starts sending messages

```
$Idle = newMess?i \rightarrow begin?i \rightarrow Active;$
$Active = (end?i \rightarrow Idle)$
          $\Box$ $(newMess?i \rightarrow$
              $((begin?i \rightarrow Collision)$ $timeout[26]$
             $\Box$ $(busy!i \rightarrow Active1)));$
$Active1 = (end?i \rightarrow Idle)$
           $\Box$ $(newMess?i \rightarrow busy!i \rightarrow Active1);$
$Collision = atomic\{BroadcastCD(0)\}within[0, 26];$  $Idle;$
```

Figure 7.4: Model for the Bus [48].

```
$BroadcastCD(x) = \textbf{if}\ (x < N)\{$
    $(cd!x \rightarrow BroadcastCD(x + 1))$
    $\Box$
    $(newMess?[i{==}x]i \rightarrow cd!x \rightarrow BroadcastCD(x + 1))$
   $\}$
   else {
     Skip
    };
```

Figure 7.5: Model for the BroadcastCD [48].

within 26 time units, the *Bus* moves to location *Collision*. Otherwise, after at least 26 time units have elapsed, the *Bus* replies busy signal to any new attempt, then it moves to location *Active*1.

In location *Active*1, the *Bus* takes at most 26 time units to inform all *Senders* of this collision, using *BroadcastCD* given in Figure 7.5. After that, the *Bus* moves to location *Idle*.

In location *Collision*, the *Bus* replies busy signal to any *Sender* that attempts to send message until the active *Sender* completes transmitting, then the *Bus* moves to location *Idle*.

7.1.3 Modeling CSMA/CD Protocol

The whole protocol consists of one *Bus* and N *Senders* interleaving with each other. In PAT, we model this as follows:

$$CSMA = (||| \text{ i} : \{0, ..., N-1\} \ @ \ Sender(i)) \ ||| \ Bus$$

7.2 Verification

In order to formally verify our model for CSMA/CD protocol is correct, we define *deadlock-free* safety property. Informally, safety property states "bad things" never happen during the execution. *deadlockfree* is a safety property so that it is always possible to move from one state to another. *deadlockfree* property in PAT is defined as follows:

assert CSMACD deadlockfree;

7.3 Symmetry Detection

A simple case $CSMA$ of CSMA/CD protocol has been considered. It consists of 1 *Bus* and 3 *Senders* that are instantiated from the templates given in Figure 7.1 and Figure 7.2 respectively.

Symmetry detection consists of 3 following steps:

- Stage 1: Detect all variables of type *pid* used in process templates *Bus* and *Sender*.
- Stage 2: Compute *Swap*($CSMA$).
- Stage 3: Compute $ValidSwap(CSMA)$.

Stage 1. First **Algorithm 5** runs on the template *Sender*. The template parameter i has type of *pid* by default and is added to $Next$. **Algorithm 5** halts after $Next$ is empty. Since no invalid *pid* variable is used in *Sender*, we say that all *Sender* processes are valid to run with symmetry reduction.

Similarly, **Algorithm 5** runs on the template *Bus*. The algorithm is able to detect a local variable x as the valid *pid* variable, which is related to *is* by the guard g as follows:

$$i == x$$

And the *Bus* process is also valid to run with symmetry reduction.

Stage 2. In the CSMA/CD protocol problem considered, there are 3 $Senders$, namely $Sender_1$, $Sender_2$ and $Sender_3$ and 1 *Bus*. Assuming that $Sender_1$, $Sender_2$, $Sender_3$ are assigned indices of 1, 2, and 3 respectively, and *Bus* is assigned an index of 4. All possible permutations of the set of indices $\{1, 2, 3, 4\}$, denoted $Swap(CSMA)$ are obtained. This results in 24 permutations which need to be checked for validity.

Stage 3. A *state-swap* π is called an automorphism of the model $CSMA$ if it satisfies the following three properties. The first property says that the permutation preserves the types of the processes being mapped onto each other. After eliminating all permutations that map *Sender* to *Bus* and vice versa, we are left with 6 permutations namely:

$$\{1, 2, 3, 4\}, \{1, 3, 2, 4\}, \{2, 1, 3, 4\}, \{2, 3, 1, 4\}, \{3, 1, 2, 4\}, \{3, 2, 1, 4\}$$

The second property checks if associations between the processes are not violated in the system obtained after applying the permutations. Since in this particular example, $Sender_1$, $Sender_2$ and $Sender_3$ are all associated with *Bus*, and these associations are consistent after applying the permutations to the *Sender* processes. Therefore, all remaining permutations ensures the second property.

The third property indicates that after the permutation, the new system must be equivalent to the original system. The third property is checked by using **Algorithm 7**. We show no remaining permutation violates the third property. Finally we have:

$$ValidSwap(P) = \{1, 2, 3, 4\}, \{1, 3, 2, 4\}, \{2, 1, 3, 4\}, \{2, 3, 1, 4\}, \{3, 1, 2, 4\}, \{3, 2, 1, 4\}$$

From Theorem 3, $CSMA$ consists of N balanced $Senders$.

7.4 Experimental Results

We have run experiments on PAT for different numbers of processes. Table 7.1 summarizes the verification results for CSMA/CD protocol.

To demonstrate the effectiveness of symmetry reduction, we ran each experiment twice, with and without symmetry reduction. Experiments were run with a 300 second timeout. We focus on three criteria: processing time (s), the number of visited states and memory usage (MB). The data shows that the regular PAT's limit for CSMA/CD protocol is around 10 processes, while verification for 30 processes can be done in 3 seconds using less than 23MB with symmetry reduction.

We keep increasing the number of *Senders* until the verification takes over 300 seconds. Although the verification tool gains a considerable reduction in processing time and memory usage, by a factorial magnitude, however verification is still not feasible for an instance of CSMA/CD protocol with a very large number of processes (> 100). Similarly to the case of Fischer's protocol, we are interested in applying our compositional verification method to CSMA/CD protocol when model checking is no longer feasible.

Processes	8		10		15		25		30	
Mode	Sym	No	Sym	No	Sym	No	Sym	No	Sym	No
Time (s)	0.037	5.6	0.07	99.8	0.26	N/A	1.45	N/A	2.76	N/A
Visited State	99	30953	149	291869	299	N/A	775	N/A	1067	N/A
Memory (Mb)	10.1	29.8	10.6	256	10.5	N/A	16.5	N/A	23.3	N/A

Table 7.1: Experimental Results for CSMA/CD Protocol

7.5 Compositional Verification

Consider a family of instances of CSMA/CD protocol $\{R_i\}$, where each instance consists of one *Bus* and i identical *Senders*. We verify whether $deadlockfree$ holds for every instance of CSMA/CD protocol. In the simplest compositional formulation, we define two invariants: θ_{B_i}, which represents local states of the *Bus* in R_i and θ_{S_i}, which represents local states of the single representative *Sender* in R_i (since all *Senders* are identical up to renaming).

Theorem 5 The PCMCP is decidable in polynomial time for CSMA/CD protocol.

Proof: A state in θ_S is a tuple $(l_S, D_S, cd, begin, end, busy)$, where (l_S, D_S) is an internal state of the *Sender* and a vector of values for its neighborhoods is $(cd, begin, end, busy)$.

A state in θ_B is a tuple $(l_B, D_B, cd, begin, end, busy)$, where (l_B, D_B) is an internal state of the *Bus*, and a vector of values for its neighborhoods is also $(cd, begin, end, busy)$.

Clearly, the neighborhoods of the *Bus* and the *Sender* are a group of synchronization channels and they are valueless. Therefore, the *Bus* and the *Sender* have finite local states. Consider an instance R_x and an instance R_y. Assume R_x is the smallest verified instance ($x = 2$). Then we have that θ_{B_x} is isomorphic to θ_{B_y} and θ_{S_x} is isomorphic to θ_{S_y}. The smallest instance consists of 1 *Bus* and 2 *Senders*. Its strongest compositional invariant can be calculated automatically. So the PCMCP is decidable in polynomial time.
End Proof.

Chapter 8

Verification for Train-Bridge Protocol

A problem arises that how to design a railway control system which only assigns the usage of the bridge to only one of many trains competing for. Train-Bridge protocol describes one solution to this problem. Roughly speaking, whenever a train approaches the bridge, it must first listen to the bridge and wait for absence signal before going. When the absence signal comes which means the bridge is idle, the train begins to go through the bridge. If the bridge is busy, the train waits until the bridge assigns it as the next train to go. The protocol uses channels to communicate and synchronize actions between trains and the bridge, together with time constraints on each location to guarantee that the bridge is a critical shared resource that is accessed only by one train at a time. In this chapter, we run the verification on Train-Bridge protocol with symmetry reduction. Our results show that we gain a considerable reduction in the cost of analysis, by a factor exponential in the number of processes. We also prove that for networks of Train-Bridge protocol, verification is decidable in time polynomial in the state space of the smallest verified, "cut-off", instance.

8.1 Model For Train-Bridge Protocol

We make the following assumptions that no messages are lost during transmission by channels. Based on the above assumptions, we then model the Train-Gate protocol in the real-time system module in our PAT tool. The model for this protocol consists of two components, namely *Train* (request for access) and *Bridge* (control access). *Train* and *Bridge* communicate by synchronous events, so we define this communication by pair-wise synchronization channels. We list all variables and processes of this model with a simplified description, as illustrated below.

Category	Name	Description
Global Definition	N	Constant: number of trains.
	channel appr 0	Train requests to go through.
	channel stop 0	Train senses a busy bridge.
	channel go 0	Train is assigned to go through.
	channel leave 0	Train completes its request.
Train Behavior	Safe(i)	The start location.
	Appr(i)	Train i is approaching the bridge.
	Stop(i)	Train i is waiting for the approval after detecting the busy bridge.
	Start(i)	Train i is allowed to go through.
	Cross(i)	Train i is acrossing the bridge.
Bridge Behavior	Free	Bridge is free.
	Active	No train is waiting and one train is approaching.
	Active1	At least one train is waiting.
	Control	Control access through the bridge to avoid collision.

8.1.1 Modeling Train Behavior

The behavior of component $Train$ is showed in Figure 8.1. Initially, the $Train$ is in location $Safe$. When the $Train$ is approaching the *Bridge*, the transition from $Safe$ to *Appr* is enabled.

In location *Appr*, the $Train$ has two transitions, which is modeled as two external choices in PAT. If a *stop* signal is not received after 10 time units have elapsed, the $Train$ goes to $Cross$. Otherwise, it then goes to *Stop*.

In location *Stop*, the $Train$ just waits until the *Bridge* assigns it as the next train to go through by sending a *go*! signal. Then it moves to *Start*.

In location *Start*, the transition to $Cross$ is enabled. The $Train$ is allowed to take maximum 15 time units to go across the *Bridge*.

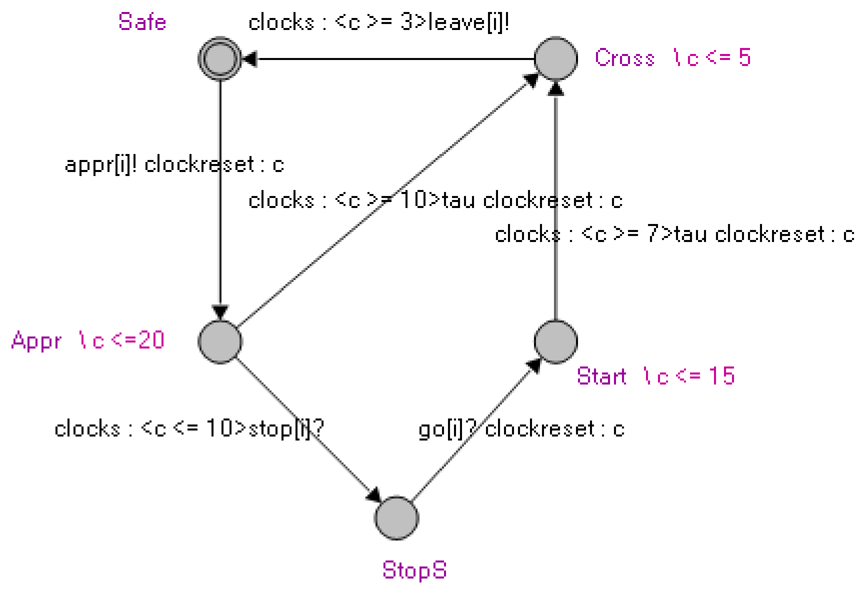

Safe
clocks : <c >= 3>leave[i]!
Cross \c <= 5
appr[i]! clockreset : c
clocks : <c >= 10>tau clockreset : c
clocks : <c >= 7>tau clockreset : c
Appr \c <=20
Start \c <= 15
clocks : <c <= 10>stop[i]?
go[i]? clockreset : c
StopS

Figure 8.1: Model for the Train

In location *Cross*, the *Train* goes through the bridge within 5 unit times and notices the *Bridge* when it has passed.

8.1.2 Modeling Bridge Behavior

The behavior of component *Bridge* is showed in Figure 8.2. The *Bridge* governs the access to the critical shared resource (Bridge). It guarantees that only one train is given access at a time. Whenever a particular train is approaching, the *Bridge* enqueues its process identifier, checks for the availability and replies an appropriate response signal. The *Bridge* serves the next element in a queue when the critical shared resource is available again.

8.1.3 Modeling Train-Bridge Protocol

The whole protocol consists of one *Bridge* and *N Trains* interleaving with each other. In PAT, we model this as follows:

$$TrainBridge = (||| \text{ i} : \{0, ..., N-1\} \ @ \ Train(i)) \ ||| \ Bridge$$

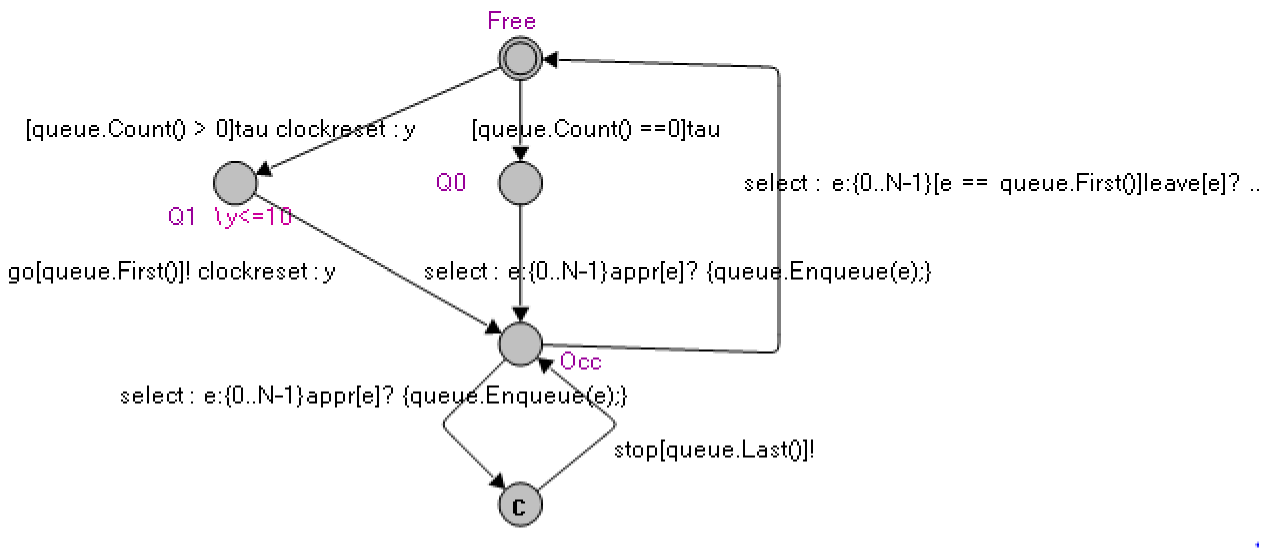

Free
[queue.Count() > 0]tau clockreset : y
[queue.Count() ==0]tau
Q0
select : e:{0..N-1}[e == queue.First()]leave[e]? ..
Q1 \y<=10
go[queue.First()]! clockreset : y
select : e:{0..N-1}appr[e]? {queue.Enqueue(e);}
Occ
select : e:{0..N-1}appr[e]? {queue.Enqueue(e);}
stop[queue.Last()]!
C

Figure 8.2: The template for the Bridge

8.2 Verification Properties

In order to formally verify our model for Train-Bridge protocol is correct, we define a *deadlockfree* safety property. Informally, safety property states "bad things" never happen during the execution. *deadlockfree* is the safety property so that it is always possible to move from one state to another. *deadlockfree* in PAT is defined as follows:

assert TrainGate deadlockfree;

We also define $overflow$ as "bad states" that there is more than N elements in the *queue* (N is the number of $Trains$). We check whether there is any reachable state that satisfies $overflow$. *overflow* in PAT is defined as follows:

define overflow (queue.count() > N);

assert TrainGate reaches overflow;

8.3 Symmetry Detection

A simple case of Train-Bridge protocol has been considered. It consists of 3 $Trains$ and 1 *Bridge* that are instantiated from the templates given in Figure 8.1 and Figure 8.2 respectively.

System detection consists of 3 following steps:

- Stage 1: Detect all variables of type *pid* from process templates *Bridge* and $Trains$.
- Stage 2: Compute *Swap*($TrainBridge$).
- Stage 3: Compute $ValidSwap$($TrainBridge$).

Stage 1. First **Algorithm 5** runs on the template $Train$. The template parameter i has type of *pid* by default and is added to $Next$. **Algorithm 5** halts after $Next$ is empty. Since no invalid *pid* variable is used in $Train$, we say that all $Train$ processes are valid to run with symmetry reduction.

Similarly, **Algorithm 5** runs on the template *Bridge*. It detects e as a variable of type *pid*, *queue* as an array of type *pid* and also shows that they are used appropriately.

Stage 2. In the Train-Bridge protocol problem considered, there are 3 $Trains$, namely $Train_1$, $Train_2$ and $Train_3$ and 1 *Bridge*. Assuming that $Train_1$, $Train_2$, $Train_3$ are assigned indices of 1, 2, and 3 respectively, and *Bridge* is assigned an index of 4. All possible permutations of the set of indices $\{1, 2, 3, 4\}$, denoted *Swap*(*TrainBridge*) are obtained. This results in 24 permutations which need to be checked for validity.

Stage 3. A *state-swap* π is called an automorphism of the model $TrainBridge$ if it satisfies the following three properties. The first property says that the permutation preserves the types of the processes being mapped onto each other. After eliminating all permutations that map $Train$ to *Bridge* and vice versa, we are left with 6 permutations namely:

$$\{1, 2, 3, 4\}, \{1, 3, 2, 4\}, \{2, 1, 3, 4\}, \{2, 3, 1, 4\}, \{3, 1, 2, 4\}, \{3, 2, 1, 4\}$$

The second property checks if associations between the processes are not violated in the system obtained after applying the permutations. Since in this particular example, $Train_1$, $Train_2$ and $Train_3$ are all associated with *Bridge*, and these associations are consistent after applying the permutations to the *Sender* processes. Therefore, all remaining permutations ensures the second property.

The third property indicates that after the permutation, the new system must be equivalent to the original system. The third property is checked by using **Algorithm 7**. We show no remaining permutation violates the third property. Finally we have:

$$ValidSwap() = \{1,2,3,4\}, \{1,3,2,4\}, \{2,1,3,4\}, \{2,3,1,4\}, \{3,1,2,4\}, \{3,2,1,4\}$$

From Theorem 3, $TrainBridge$ consists of N balanced $Trains$.

8.4 Experimental Results

We have run experiments on PAT for different numbers of processes. Table 8.1 summarizes the verification results for Train-Bridge protocol.

To demonstrate the effectiveness of symmetry reduction, we ran each experiment twice, with and without symmetry reduction. Experiments were run with a 300 second timeout. We focus on three criteria: processing time (s), the number of visited states and memory usage (MB). The data shows that the regular PAT's limit for Train-Bridge protocol is around 10 processes.

We keep increasing the number of $Trains$ until the verification takes over 300 seconds. Although the verification tool gains a considerable reduction in processing time and memory usage, by a factorial magnitude, however verification is still not feasible for an instance of Train-Bridge protocol even with 25 $Train$ processes. Similarly to Fischer's protocol and CSMA/CD protocol, we are interested in applying our compositional verification method to Train-Bridge protocol when model checking is no longer feasible.

Processes	8		10		15		20		25	
Mode	Sym	No	Sym	No	Sym	No	Sym	No	Sym	No
Time (s)	0.18	44.3	0.84	295.6	9.7	N/A	140	N/A	N/A	N/A
Visited State	450	796138	1094	6867701	5494	N/A	44006	N/A	N/A	N/A
Memory (Mb)	12.1	641	59.4	8096	125.7	N/A	865.5	N/A	N/A	N/A

Table 8.1: Experimental Results for Train-Bridge Protocol

8.5 Compositional Verification

Consider a family of instances of Train-Bridge protocol $\{R_i\}$, where each instance consists of one *Bridge* and i identical $Trains$. We verify whether $deadlockfree$ holds for every instance of Train-Bridge protocol.

From the previous section, we prove that any pair of $Trains$ in $\{R_i\}$ is balanced since they are connected by appropriate automorphisms. In the simplest compositional formulation, we define two invariants: θ_{B_i}, which represents local states of the *Bridge* in R_i and θ_{T_i}, which represents local states of the single representative $Train$ in R_i.

Theorem 6 The PCMCP is decidable in polynomial time for Train-Bridge protocol.

To prove the local states of the $Bridge \in R_i$ and $Bridge \in R_j$, that are locally symmetric or θ_{B_i} is isomorphic to θ_{B_j}, we need to prove:

- The isomorphism β maps the neighborhoods of the $Bridge \in R_i$ and $Bridge \in R_j$.
- Internal states of the $Bridge \in R_i$ and initial states of the $Bridge \in R_j$ are related by β.

Proof: For the Train-Bridge protocol, the *Bridge* has a finite state-space. the *Bridge* shares and updates the list of following global variables with $Trains$: channel *appr*, channel *go*, channel *stop*, and channel *leave*. It is interesting that the global variable *queue* is only used in the *Bridge*, which does not affect local states of the *Bridge*. Therefore, we do not abstract the original *Bridge* like we have done with Fischer's protocol. Therefore, a state in θ_{B_i} is a tuple $(l_{B_i}, D_{B_i}, appr, go, stop, leave)$, where (l_{B_i}, D_{B_i}) is an internal state of the *Bridge* and a vector of values for its neighborhoods is $(appr, go, stop, leave)$. Since its neighborhoods $(appr, go, stop, leave)$ are all stateless and the *Bridge* has finite internal states, we can conclude that local states of the $Bridge \in R_i$ and $Bridge \in R_j$, that are locally symmetric.

Similarly a state in θ_{T_i} is a tuple $(l_{T_i}, D_{T_i}, appr, go, stop, leave)$, where (l_{T_i}, D_{T_i}) is an internal state of the $Train$ and a vector of values for its neighborhoods is $(appr, go, stop, leave)$. So local states of the $Train \in R_i$ and $Train \in R_j$, that are also locally symmetric.

Consider an instance R_x and an instance R_y. Assume R_x is the smallest verified instance $(x = 2)$. Then we have that θ_{B_x} is isomorphic to θ_{B_y} and θ_{T_x} is isomorphic to θ_{T_y}. The smallest instance consists of 1 *Bridge* and 2 *Trains*. Its strongest compositional invariant can be calculated automatically. The strongest (θ_B, θ_T) pair can be calculated by turning the compositional rules into a simultaneous least fix-point formulation, and iterating until convergence. The computation time is polynomial in the number of states of *Trains* and of *Bridge*. **End Proof**.

Conclusion

In this thesis, we focused on the verification of concurrent and real-time systems using model checking approach. Although model checking has an obvious advantage over formal methods, it must cope with the *state space explosion* - the state space can grow exponentially since the number of components in a system increases.

State-space explosion is the main obstacle to the scalability of model checking. Indeed, it is known that symmetry reduction techniques can be used to combat this problem for networks of replicated components. Therefore, we extended the real-time model checker PAT with symmetry reduction. We then proposed a mechanism that allows structural symmetry detection arising from process templates that requires no additional input from a user (no special data type is needed). By analyzing a template syntax, we even support detecting partial symmetry and rotational symmetry. Our method operates in two stages:

- Detect symmetries from process templates. It results in all graph automorphisms that are sound with respect to reachability properties: an automorphism α performs certain permutations on a state s and if a state s has been visited before, then all states $\alpha(s)$ which are obtainable by applying these permutations to s have been also visited.
- Given a group of valid automorphisms, we generate the symmetry-reduced state space *on-the-fly* and check for a property ϕ.

We run experiments on Fischer's protocol, CSMA/CD protocol and Train-Bridge protocol. Our results shows that we are able to obtain significant savings by only performing over a quotient state-space. However, the verification is still non-scalable for instances with an arbitrarily large number of parallel processes.

Being directly motivated by attempts to verify Fischer's protocol, CSMA/CD protocol and Train-Bridge protocol with a large number of components, we extended our previous work [43] [44] to timed automata. The method combines elements of automatic symmetry detection, symmetry reduction, compositional reasoning and abstraction. Automatic symmetry detection infers symmetries of the state-space underlying a model without explicitly constructing the state-space. Symmetry reduction partitions network nodes into equivalence classes. Compositional reasoning analyzes each process of an equivalence class separately along with an abstraction of its neighboring processes. The benefit is that the computation can be reduced to one on a fixed set of representative nodes and works in time polynomial in the number of processes.

In a nutshell, our compositional verification method works as follows: First, the verification tool detects structural symmetries directly from input templates, requiring no additional input from a user. It results in a set of possible permutations and each of them is checked individually against the model to see if it induces a valid automorphism. Second, valid automorphisms enable the tool to finitely partition input processes into equivalence classes such that any pair of nodes in an equivalence class are balanced and equivalent. For networks that have significant global symmetry, we are able to obtain a uniform invariant which applies to all nodes by making use of the local symmetries. As a results, the verification is only performed on a fixed set of representative nodes R and local symmetries suffice to ensure that for a property ϕ that is preserved by the symmetries, it follows that ϕ holds for all instances if it holds for R. The method is only partly automated for systems consisting of a large number of processes.

Our results show that verification is decidable in time polynomial in the state space of the smallest verified, "cut-off", instance for networks of Fischer's protocol, CSMA/CD protocol and Train-Bridge protocol.

In our previous work [43] [44], the technique requires local symmetries of a parametric system to be manually given as input by a system designer. Now local symmetries are derived automatically from the global symmetry detection method that is applied to instances of the parametric protocol. Moreover, the symmetry detection is both automatic and relatively inexpensive to run.

Since parametric system analysis is in general an undecidable problem, we do not expect that a single analysis technique applies to all cases. Currently, we target systems that are composed of several sub-systems. In each sub-system, processes are instances that are instantiated from the same template. The future work includes exploring other real-life protocols. Moreover, we are interested in the verification of liveness properties and allow the use of process identifiers in arithmetic operations.

References

[1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1543–1571, 1994.

[2] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. *Formal methods for industrial applications: Specifying and programming the steam boiler control*, volume 9. Springer Science & Business Media, 1996.

[3] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *International Colloquium on Automata, Languages, and Programming*, pages 322–335. Springer, 1990.

[4] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[5] Krzysztof R Apt and Dexter C Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[6] Felice Balarin. Approximate reachability analysis of timed automata. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 52–61. IEEE, 1996.

[7] Johan Bengtsson. *Clocks, dbms and states in timed systems.* Acta Universitatis Upsaliensis, 2002.

[8] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. *CONCUR'98 Concurrency Theory*, pages 485–500, 1998.

[9] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on concurrency and petri nets*, pages 87–124. Springer, 2004.

[10] E Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verification*, pages 450–462. Springer, 1993.

[11] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[12] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal methods in system design*, 9(1):77–104, 1996.

[13] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[14] Edmund M Clarke, Orna Grumberg, and Doron Peled. Model checking. 2000, 2000.

[15] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool kronos. *Hybrid Systems III*, pages 208–219, 1996.

[16] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 73–81. IEEE, 1996.

[17] David L Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Conference on Computer Aided Verification*, pages 197–212. Springer, 1989.

[18] David L Dill, Andreas J Drexler, Alan J Hu, and C Han Yang. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD'92. Proceedings, IEEE 1992 International Conference on*, pages 522–525. IEEE, 1992.

[19] Alastair F Donaldson and Alice Miller. Automatic symmetry detection for model checking using computational group theory. In *International Symposium on Formal Methods*, pages 481–496. Springer, 2005.

[20] E Emerson and A Sistla. Symmetry and model checking. In *Computer Aided Verification*, pages 463–478. Springer, 1993.

[21] E Allen Emerson and A Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1):105–131, 1996.

[22] Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. *Journal of the ACM (JACM)*, 63(1):10, 2016.

[23] Steven M German and A Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 39(3):675–735, 1992.

[24] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification*, pages 176–185. Springer, 1991.

[25] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using spin. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[26] Martijn Hendriks. *Enhancing uppaal by exploiting symmetry.* Nijmegen Institute for Computing and Information Sciences, Faculty of Science, University of Nijmegen, 2002.

[27] Martijn Hendriks, Gerd Behrmann, Kim Larsen, Peter Niebert, and Frits Vaandrager. Adding symmetry reduction to uppaal. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 46–59. Springer, 2003.

[28] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):110–122, 1997.

[29] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 394–406. IEEE, 1992.

[30] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.

[31] Charles AR Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.

[32] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[33] C Norris Ip and David L Dill. Better verification through symmetry. *Formal methods in system design*, 9(1-2):41–75, 1996.

[34] Luming Lai and Phil Watson. A case study in timed csp: The railroad crossing problem. In *Hybrid and Real-Time Systems*, pages 69–74. Springer, 1997.

[35] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[36] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24. IEEE, 1997.

[37] Kim Laxsen, Paul Pettersson, and Wang Yi. Diagnostic model-checking for real-time systems. *Hybrid Systems III*, pages 575–586, 1996.

[38] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 281–297, 1998.

[39] S Mauw, R Mateescu, and W Janssen. Verifying business processes using spin. In *Proceedings of the International SPIN Workshop*, pages 21–36, 1998.

[40] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[41] Marius Minea. Partial order reduction for model checking of timed automata. In *CONCUR*, volume 1664, pages 431–446. Springer, 1999.

[42] John C Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur/spl phi. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151. IEEE, 1997.

[43] Kedar S Namjoshi and Richard J Trefler. Local symmetry and compositional verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 348–362. Springer, 2012.

[44] Kedar S Namjoshi and Richard J Trefler. Parameterized compositional model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 589–606. Springer, 2016.

[45] Florence Pagani. Partial orders and verification of real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 327–346. Springer, 1996.

[46] Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993.

[47] Paul Pettersson. *Modelling and verification of real-time systems using timed automata: theory and practice*. Department of Computer systems, Univ., 1999.

[48] Ling Shi and Yan Liu. Modeling and verification of transmission protocols: A case study on csma/cd protocol. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 143–149. IEEE, 2010.

[49] Ulrich Stern and David Dill. Automatic verification of the sci cache coherence protocol. *Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[50] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[51] Antti Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification*, pages 156–165. Springer, 1991.

[52] Farn Wang and Karsten Schmidt. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 50–64. Springer, 2002.

[53] Howard Wong-Toi. Symbolic approximations for verifying real-time systems. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1994.

[54] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII*, pages 243–258. Springer, 1995.

[55] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):123–133, 1997.