MODEL CHECKING CONCURRENT AND REAL-TIME SYSTEMS: THE PAT APPROACH

LIU YANG

(B.Sc. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2009

Acknowledgement

First and foremost, I am deeply indebted to my supervisors, Dr. Dong Jin Song and Dr. Rudy Setiono, for their guidance, advice and encouragement throughout the course of my doctoral program. They have given me immense support both in various ways, and have also helped me stay on the track of doing research.

I am deeply grateful to Dr. Sun Jun, who acts like both a friend and co-supervisor in my graduate study. I thank him for introducing me to the exciting area of model checking. His supervision and crucial contribution made him a backbone of this research. His involvement with his originality has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come.

I am grateful to Dr. Joxan Jaffar, Dr. Chin Wei Ngan and Dr. P. S. Thiagarajan for their valuable suggestions and comments on my research works. I have special thanks to Dr. Chen Wei, Dr. Liu Yanhong, Dr. Abhik Roychoudhury, Dr. Pang Jun, etc for their research collaborations.

To my seniors, Dr. Li Yuanfang, Dr. Chen Chunqing, Dr. Sun Jing, Dr. Wang H. Hai and Dr. Qin Shengchao, Feng Yuzhang, and fellow student Zhang Xian - Thank you for your support and friendships through my Ph.D. study.

This study was in part funded by the project “Rigorous Design Methods and Tools for Intelligent Autonomous Multi-Agent Systems” and “Advanced Modeling Checking Systems” supported by Ministry of Education of Singapore and the project “Reliable Software Design and Development for Sensor Network Systems” supported by National University of Singapore Academic Research Fund and the project “Systematic Design Methods and Tools for Developing Location Aware, Mobile and Pervasive Computing Systems” supported by Singapore National Research Foundation-Interactive Digital Media. The School of Computing also provided the finance for me to present papers in several conferences overseas. In addition, I have been encouraged by receiving Microsoft Asia Research Fellowship 2007 and Research Achievement Award 2009. For all this, I am very grateful.

Lastly, I wish to thank sincerely and deeply my parents Liu Maolin and Zhou Xiuling, who have taken care of me with great love in these years. I thank my wife Tan Chen, for all the love.

Contents

Summary

The design and verification of concurrent and real-time systems are notoriously difficult problems. Among the software validation techniques, model checking approach has been proved to be successful as an automatic and effective solution. In this thesis, we study the verification of concurrent and real-time systems using model checking approach.

First, we design an integrated formal language for concurrent and real-time modeling, which combines high-level specification languages with mutable data variables and low-level procedural codes for the purpose of efficient system analysis, in particular, model checking. Timing requirements are captured using behavior patterns like *deadline*, *time out*, etc. A formal semantic model is defined for this language.

Based on this modeling language, we investigate LTL verification problem with focus of fairness assumptions, and refinement checking problem with following results.

1. We propose a unified on-the-fly model checking algorithm to handle a variety of fairness assumptions, which is further tuned to support parallel verification in multi-core architecture with shared memory. We apply the proposed algorithm on a set of self-stabilizing population protocols, which only work under global fairness. One previously unknown bug is discovered in a leader election protocol. Population protocols are designed for networks with large or even unbounded number of nodes, which gives the space explosion problem. To solve this problem, we develop a process counter abstraction technique to handle parameterized systems under fairness. We show that model checking under fairness is feasible, even without the knowledge of process identifiers.

2. Based on the ideas in FDR, we present an on-the-fly model checking algorithm for refinement checking, incorporated with advanced model checking techniques. This algorithm is successfully applied in automatic linearizability verification and conformance checking between Web Services.

Symbolic model checking is capable of handling large state space. We present an alternative solution for LTL verification using bounded model checking approach. Hierarchical systems are encoded as SAT problems. The encoding avoids exploring the full state space for complex systems so as to avoid state space explosion.

To support verification of real-time systems, we propose an approach using a fully automated abstraction technique to build an abstract finite state machine from the real-time model. We show that the abstraction has finite state and is subject to model checking. Furthermore, it weakly bi-simulates the concrete model and we can perform LTL model checking, refinement checking and even timed refinement checking upon the abstraction.

The results of this thesis are embodied in the design and implementation of a self-contained framework: Process Analysis Toolkit (PAT), which supports composing, simulating and reasoning of concurrent and real-time systems. This framework includes all of the proposed techniques: deadlock-freedom, reachability, LTL checking, refinement checking and etc. PAT adopts an extensible design, which allows new languages and verification algorithms to be supported easily. Currently, three modules have been developed in PAT. The experiment results show that PAT is capable of verifying systems with large number of states and complements the state-of-the-art model checkers in several aspects.

Key words: **Formal Verification, Concurrent and Real-time Systems, Model Checking, PAT, LTL Model Checking, Fairness, Partial Order Reduction, Process Counter Abstraction, Refinement Checking, Bounded Model Checking, Timed Zone Abstraction, Timed Refinement Checking, Population Protocol, Linearizability, Web Service Conformance**

List of Figures

Chapter 1

Introduction

The design and verification of concurrent and real-time systems are notoriously difficult problems. In particular, the interaction of concurrent processes in large systems often leads to subtle bugs that are extremely difficult to discover using the conventional techniques of simulation and testing. Automated verification based on model checking promises a more effective way of discovering design errors, which has been used successfully in practice to verify complex software systems.

1.1 Motivation and Goals

With the fast development of IT industry, our reliance on the functioning of software systems is growing rapidly. These systems are becoming more and more complicated and are massively encroaching on daily life, e.g., the Internet, embedded systems, mobile devices and so on. This is especially true for concurrent and real-time systems, which have concurrent executions, shared resources and timing factors. Failure is unacceptable for mission critical systems like electronic commerce, telephone switching networks, air traffic control systems, medical instruments, and numerous other examples. We frequently read of incidents where catastrophic failure is caused by an error in software systems, as some examples listed below.

- Pentium bug: Intel Pentium II chip, released in 1994 produced error in floating point division.
  Cost: $475 million
- Ariane 5 failure: In December 1996, the Ariane 5 rocket exploded 40 seconds after takeoff, by an overflow generated by converting a 64-bit floating-point number into a 16-bit integer.
  Cost: $400 million
- Therac-25 accident: A software failure caused wrong dosages of x-rays.
  Cost: Human Loss.

In a recent example, the 2010 Toyota recall that can cost excess of 2 billion USD has been quite surprising for the people, who have been acquainted with the successful history of this company. Much of the cost, damage to the company's reputation, and the uncertainty regarding the nature of the problems, could have been avoided if the correctness of the software components has been established beyond doubt.

Clearly, the need for reliable software systems is crucial. As the involvement of such systems in our lives increases, so too does the burden for ensuring their correctness. Therefore, it will become more important to develop methods that increase our confidence in the correctness of such systems.

The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking. Model checking is a method of automatically verifying concurrent systems in which a finite state model of a system is compared with a correctness requirement. The process of model checking can be separated into system modeling, requirement specification and verification. It has a number of advantages over other traditional approaches. This method has been used successfully in practice to verify complex circuit design and communication protocols.

In this thesis, our research focuses on model checking of concurrent and real-time systems. In particular, we have tried to address four issues related to model checking: (i) proposing a formal language to model concurrent and real-time systems, (ii) exploring efficient model checking algorithms and reduction techniques, (iii) implementing a toolkit to support effective software verification, and (iv) applying the proposed model checking techniques in different domains. The concrete issues that require more research efforts are elaborated below in the order of the process of model checking.

System Modeling

Formal modeling languages and notations have much to offer in the achievement of technical quality in system development. Precise notations and languages help to make specifications unambiguous while improving intuitiveness, increasing consistency and making it possible to detect errors automatically with the support of effective tools. Over the last few decades, many formal modeling languages have been proposed [108, 222, 75, 183, 150, 120, 155, 10]. Formal modeling languages and notations are generally logic-based formalisms, which are divided into two groups, state-oriented formalisms, including VDM [120], Z [222], Object-Z [75], etc., and event-oriented formalisms, including Communicating Sequential Processes (CSP) [108], CCS [155], Timed CSP [183], π-calculus [155], etc. The formalisms based on the notion of state machines[1] include finite state machines, Statecharts [104], Petri-net [165], Timed Automata [10], etc.

We are particular interested in the event-based modeling languages like CSP, CCS for their rich set of concurrent operators and compositional structure (to achieve a modular design by nature). CSP has passed the test of time. It has been widely accepted and influenced the design of many recent programming and specification languages. Nonetheless, modeling systems with non-trivial data structures and functional aspects completely using languages like CSP remains difficult. In order to solve the problem, many specification languages integrating process algebras like CSP or CCS with state-based specification languages like the Z language or Object-Z have been proposed. The state-based language component is typically used to specify the data states of the system and the associated data operations in a declarative style. Examples include *Circus* [221] (i.e., an integration of CSP and the Z language), CSP-OZ [86] (i.e., an integration of CSP and Object-Z) and TCOZ [152] (i.e., an integration of Timed CSP and Object-Z). However, because declarative specification languages like Z are very expressive and not executable, automated analyzing (in particular,

[1] State machine is a model of behavior composed of a number of states.

model checking) of systems modeled using the integrated languages is extremely difficult.

During the last decade or so, a popular approach for specifying real-time systems is based on the notation Timed Automata [10, 149]. Timed Automata are powerful in designing real-time models with explicit clock variables. Real-time constraints are captured by explicitly setting/reseting clock variables. Models based on Timed Automata often adapt a simple structure, e.g. a network of Timed Automata with no hierarchy [135]. The benefit is that efficient model checking is made feasible. Nonetheless, designing and verifying hierarchical real-time systems is becoming an increasingly difficult task due to the widespread applications and increasing complexity of such systems. As a result, users often need to manually cast high-level compositional time patterns into a set of clock variables with carefully calculated clock constraints. The process is tedious and error-prone.

One goal of this thesis is to design an integrated modeling language for concurrent and real-time systems, which is sufficiently expressive, but is still subject to model checking.

Requirement Specification and Verification

Critical system requirements like safety, liveness and fairness play important roles in system specification, verification and development. Safety properties ensure that something undesirable never happens. Liveness properties state that something desirable must eventually happen. Fairness properties state that if something is enabled sufficiently often, then it must eventually happen. Often, fairness assumptions are necessary to prove liveness properties.

Over the last decades, specification and verification of safety properties (e.g., deadlockfreeness and reachability) have been studied extensively. The concept of liveness itself is problematic [132]. Fairness constraints have been proved to be an effective way of expressing liveness, and is also important in system specification and verification. For instance, without fairness constraints, verifying of liveness properties may often produce counterexamples which are due to un-fair executions, e.g., a process or choice is infinitely ignored. State-based fairness constraints have been well studied in automata theory based on accepting states, e.g., in the setting of Büchi/Rabin/Streett/Muller automata [209]. It has been observed that the notion of fairness is not easily combined with the bottom-up type of compositionality (of process algebra for instance [170]), which is important for attacking the complexity of system development. Existing model checkers are ineffective with respect to fairness [202]. It is desirable to develop effective fairness verification algorithm. Especially it is proven that the correctness of recently developed population protocols requires fairness [14, 88]. In this thesis, we focus on liveness properties expressed in Linear Temporal Logics (LTL)[2], which allows potentially on-the-fly verification algorithms to be developed [61].

In order to verify hierarchical systems, more general specifications like refinement relation are needed. In these cases, the requirement is modeled using an abstract model rather than a logic formula, which gives more expressive power. FDR (Failures-Divergence Refinement) [175] is the *de facto* refinement analyzer, which has been successfully applied in various domains. Based on the model checking algorithm presented in [175] and later improved with other reduction techniques presented in [179], FDR is capable of handling large systems. Nonetheless, since FDR was initially introduced, model checking techniques have evolved much further in the last two decades. A number of effective reduction methods have been proposed which greatly enlarge the size the systems that can be handled. Some noticeable ones include partial order reduction, symmetry reduction, predicate abstraction, etc. It is worth revisiting this algorithm by incorporating new techniques.

For the real-time systems, previous works [135, 35, 35, 217, 207, 33] focus on the flat modeling structure, like Timed Automata. Verification for hierarchical languages is less studied. To the best of our knowledge, there are few verification support for Timed CSP, e.g. the theorem proving approaches documented in [40, 101], the translation to UPPAAL models [70, 71] and the approach based on constraint solving [72]. Regarding the timed refinement checking, tool support is also very limited. One of the reasons is that Timed Automata, which extended Büchi Automata with clocks [10], is designed to capture infinite languages. The refinement checking (or equivalently the language inclusion) problem is undecidable in the setting of Timed Automata [10], because the language of Timed Automata is not closed under complement. Effective verification algorithms and reduction techniques are always desirable when real-time is involved.

[2] For model checking, there is no practical complexity advantage to restrict oneself to a particular temporal logic [80].

Model checking approach works only with finite state space[3] and suffers from the state space explosion problems. We need to develop advanced techniques to cope with this limitation. First, symbolic model checking have been proved as a successful representation for state graph, which can handle large system state up to 10^{20} [44], which may be a promising approach for verifying hierarchical systems. Second, effective reduction techniques have been developed during the last two decades, e.g., partial order reduction [214], symmetry reduction, etc. Therefore, problem specific reduction can be developed when we design the verification algorithms. Third, to handle infinite state spaces, sound abstraction techniques are required, e.g. to handle infinite number of similar processes or apply model checking in real number clocks. Fourth, additional computation resources can be used to speed up the verification, e.g., parallel verification using multi-core CPU.

Tool Support

Effective tool development has always been the focus of model checking community. Since 1980, there is an ample set of model checkers developed. General propose model checkers for concurrent systems include NuSMV [53], SPIN [111], mCRL2 [102] and so on. Verification tools for real-time systems include UPPAAL [135], KRONOS [35], RED [217], Timed COSPAN [207], Rabbit [33] and so on. The success of these tools is shown in establishing the correctness of various systems and finding critical bugs in published algorithm and real-world applications. With the focus on the performance, most tools ignore the following aspects in their design, which may limit their usage.

Usability To be a useful tool, model checkers should be self-contained and offer user friendly interfaces to support system editing, animated simulation, and parameterized verification.

Extensibility Most tools are designed for certain systems or favor for particular requirements. Extensibility of new model checking algorithm or input modeling language is rarely mentioned. A structural design shall make the support of new algorithm or language relative easy by reusing the existing model checking algorithms and libraries.

[3] Though this is generally true, a considerable amount of effort has been expended on applying model checking to infinite state models [140, 84, 4].

To add to our understanding of the field, we aim to develop efficient model checker with the consideration of the above points.

1.2 Summary of Contributions

The main results of this thesis are embodied in the design and implementation of Process Analysis Toolkit (PAT), a self-contained framework for the automatic analysis of concurrent and real-time systems. The contributions of this thesis can be summarized as follows:

- We design an integrated formal language[4] for concurrent and real-time modeling, which combines high-level specification languages with mutable data variables and low-level procedural codes for the purpose of efficient system analysis, in particular, model checking. Timing requirements for real-time systems are captured using behavior patterns like *deadline*, *time out*, etc. Instead of explicitly manipulating clock variables (as in Timed Automata), the time related process constructs are designed to build on implicit clocks. Furthermore, we formally define the semantic model for the language, which facilitates PAT to perform sound and complete system verification.

- We develop a unified on-the-fly model checking algorithm which handles a variety of fairness including process-level weak/strong fairness, event-level weak/strong fairness, strong global fairness, etc. The algorithm extends previous work on model checking based on finding strongly connected components (SCC). To give flexibility, we propose several fairness annotations on individual events, which allows effective reduction techniques used together with the proposed fairness verification. Furthermore, we present a parallel version of the proposed algorithm in multi-core shared-memory architecture. The parallel algorithm is performed on-the-fly with little overhead. Experimental results (see Section 4.7) show that our

[4] The proposed language borrows syntax and semantics of CSP. Though it is not compositional as original CSP, all the verification algorithms presented in this thesis require no compositionality.

algorithm is more effective compared to SPIN to prove or disprove fairness enhanced systems. The parallel algorithm is shown to be scalable to the number of CPU-cores, especially when a system search space contains many SCCs. We apply the proposed fairness model checking algorithms on a set of self-stabilizing population protocols for ring networks, which only work under fairness. We report on our model checking results. Especially, we discover one previously unknown bug in a leader election protocol [118].

- We develop a novel technique for model checking parameterized systems under fairness, against Linear Temporal Logic (LTL) formulae. We show that model checking under fairness is feasible, even without the knowledge of process identifiers. This is done by systematically keeping track of the local states from which actions are enabled / executed within any infinite loop of the abstract state space. We develop necessary theorems to prove the soundness of our technique, and also present efficient on-the-fly model checking algorithms.

- Based on the ideas in FDR [175], we present a on-the-fly model checking algorithm for refinement relations verification. Our algorithm is designed to incorporate advanced model checking techniques, e.g. partial order reduction, to analyze event-based hierarchical system models.

- We apply the refinement checking algorithm to automatically check linearizability [107] based on refinement relations from abstract specifications to concrete implementations. Our method avoids the often difficult task of determining linearization points in implementations, but can also take advantage of linearization points if they are given. We have checked a variety of implementations of concurrent objects, including the first algorithms for the mailbox problem [19] and scalable NonZero indicators [78].

- We apply the refinement checking algorithm to automatically check consistency between Web Service choreography and Web Service orchestration by showing conformance relationship between the choreography and the orchestration. The algorithm is further extended with data support and specialized optimizations for Web services.

- We presents a bounded model checking approach to verify LTL properties using compositional encoding of hierarchical systems as satisfiability (SAT) problems. State-of-the-art SAT solvers are then applied for bounded model checking. The encoding avoids exploring the full state space for complex systems so as to avoid state space explosion. The experiment results show that our approach has a competitive performance for verifying systems with large number of states.

- We propose an approach for modeling and verifying hierarchical real-time systems, which uses a fully automated abstraction technique to build an abstract finite state machine from the real-time model. The idea is to dynamically create clocks to capture constraints introduced by the timed process constructs. A clock may be shared for many constructs in order to reduce the number of clocks. Further, the clocks are deleted as early as possible. During system exploration, a constraint on the active clocks is maintained and solved using techniques based on Difference Bound Matrix (DBM [68]). We show that the abstraction has finite state and is subject to model checking. Further, it weakly bi-simulates the concrete model and, therefore, we may perform sound and complete LTL model checking or refinement checking upon the abstraction. To facilitate timed refinement checking, we formally define a timed trace semantics and a timed trace refinement relationship. We extend the zone abstraction technique to preserve timed event traces; hence timed refinement checking is possible. We provide the first solution for model checking Timed CSP and timed refinement checking.

- We develop Process Analysis Toolkit (PAT), a self-contained tool to support composing, simulating and reasoning of different concurrent systems. PAT implements all proposed model checking techniques catering for checking deadlock-freeness, reachability, LTL checking, refinement checking and etc. To achieve good performance, advanced techniques are implemented like partial order reduction, process counter abstraction, bounded model checking, parallel model checking, etc. PAT is designed to be a generic framework, which can be easily extended to support a system with new languages syntax and verification algorithms. The experiment results show that PAT is capable of verifying systems with large number of states and complements the state-of-the-art model checkers in many aspects.

1.3 Thesis Outline and Overview

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

Chapter 2 is devoted to an introduction of model checking techniques in the order of model checking process. First, systems are modeled using Kripke structures. Second, specification can be written using temporal logic, particular LTL. Last, the verification is done using dedicated algorithms with reduction techniques like partial order reduction. The basics about real-time model checking are explained in the end of this chapter.

Chapter 3 introduces an integrated modeling language with formally defined syntax and operational semantics. The semantics model is interpreted using Labeled Transition System (LTS). A multi-lift system and Fischer's algorithm are used to illustrate the language.

In Chapter 4, we study the LTL verification problem under different fairness assumptions. A fairness model checking approach based on Tarjan's SCC detection algorithm is proposed. To give flexibility, we propose several fairness annotations on individual events, which allow effective reduction techniques used together with the proposed fairness verification. Furthermore, we present a parallel version of the proposed algorithm for multi-core shared-memory architecture.

In Chapter 5, we apply the algorithms developed in Chapter 4 to self-stabilizing population protocols. One previously unknown bug is discovered in a leader election protocol [118]. Population protocols are designed on a large or even unbounded number of similar processes, which raises the state explosion problem. To solve this problem, we propose a process counter abstraction technique.

Chapter 6 introduces trace refinement relations and proposes a refinement checking algorithm incorporated with advanced reduction techniques, like partial order reduction. In Chapter 7, we apply the proposed refinement algorithm on linearizability checking and web service conformance checking.

Chapter 8 presents a compositional encoding of hierarchical processes as satisfiability (SAT) problem and then applies state-of-the-art SAT solvers for bounded model checking. This encoding avoids exploring the full state space for complex systems so as to deal with state space explosion.

Chapter 9 explains our solution to verify hierarchical real-time systems. We develop an automated abstraction technique to build an abstract finite state machine from the real-time model. We show that the abstraction has finite state and is amenable to model checking. Further, we present algorithms for LTL model checking, refinement checking and timed refinement checking upon the abstraction.

Chapter 10 presents PAT, a general framework to support composing, simulating and reasoning of different concurrent systems. The system architecture, workflow, functionalities and details of existing modules are explained in this chapter.

Chapter 11 summaries the contributions of the thesis and discusses future research directions.

1.4 Publications from the Thesis

Most of the work presented in this thesis has been published or accepted in international conference proceedings or journals.

The work in Chapter 3 was presented at *The* 3^{rd} *IEEE International Symposium on Theoretical Aspects of Software Engineering TASE'09 (July 2009)* [194]. The work in Section 4.5 was presented at *The* 10^{th} *International Conference on Formal Engineering Methods ICFEM'08 (November 2008)* [202]. The work in Section 4.6 is accepted at *The* 11^{th} *International Conference on Formal Engineering Methods ICFEM'09 (December 2009)* [147]. The work in Section 5.1 to Section 5.3 was used as a basis for the paper presented at *The* 3^{rd} *IEEE International Symposium on Theoretical Aspects of Software Engineering TASE'09 (July 2009)* [144]. The work in Section 5.4 to Section 5.7 is accepted at *The* 16^{th} *International Symposium on Formal Methods FM'09 (November 2009)* [204]. The work in Chapter 6 was presented in an invited paper at *The* 3^{rd} *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA'08 (October 2008)* [193]. The work in Section 7.1 to Section 7.3 is accepted at *The* 16^{th} *International Symposium on Formal Methods FM'09 (November 2009)* [143]. One case study in Section 7.3 was published in *The* 21^{st} *International Conference on Software Engineering and Knowledge En-*

gineering SEKE'09 (July 2009) [225]. The work in Chapter 8 was presented in a paper at *The* 2^{nd} *IEEE International Symposium on Theoretical Aspects of Software Engineering TASE'08 (June 2008)* [199], and a journal article in 2^{nd} volume of *The Frontiers of Computer Science in China* journal [200]. The work in Chapter 9 is accepted at *The* 11^{th} *International Conference on Formal Engineering Methods ICFEM'09 (December 2009)* [203]. The work in Chapter 10 was the basis for the paper published in *The* 30^{th} *International Conference on Software Engineering ICSE'08 (May 2008)* [146], and the paper published in *The* 21^{st} *International Conference on Computer Aided Verification CAV'09 (June 2009)* [197].

Part of Chapter 4 has been submitted for publication [195]. A paper including the web service conformance checking presented in Section 7.4 to Section 7.7 has been submitted for publication [198]. Part of Section 9.2.3 has been submitted for publication [201]. I also made contributions to other publications [145, 73] which are remotely related this thesis.

For all the publications mentioned above, I have contributed substantially in both theory development and tool implementation.

Chapter 2

Background

The principal validation methods for complex systems include testing, simulation, deductive verification, and model checking. *Simulation* and *testing* approaches send the test signal at input-points and check the signal at the output-points. These two methods may become very expensive for complex, asynchronous systems. More importantly, they cover only a limited subset of possible behaviors. *Deductive verification* uses axioms and proof rules to prove the correctness of the systems, which can handle infinite state systems. However, it is a manual approach, very time-consuming and can only be used by experts. *Model checking* is an automatic approach for verifying finite state systems. It differs from other methods in two crucial aspects: (1) it does not aim of being fully general, and (2) it is fully algorithmic and of low computational complexity[1].

After two decades' development, model checking has covered a wide area including a number of different approaches (e.g. explicit model checking, symbolic model checking, probabilistic model checking, etc.) and techniques (e.g. partial order reduction, binary decision diagrams, abstraction, symmetry reduction, etc.). In this chapter, we cover basic knowledge of model checking and concepts related to this thesis. The remainder of the chapter is organized as follows. Section 2.1 gives a brief introduction to model checking. Section 2.2 explains how the systems should be modeled in

[1] The complexity of most model checking algorithms is proportional to the state space or the product of the state space and property.

model checking approaches. Section 2.3 enumerates several specifications and related algorithms to verify them. Section 2.4 covers the background knowledge of real-time model checking.

2.1 Basics of Model Checking

Model checking [58] is a verification technique that explores all possible system states in a brute-force manner. Therefore, model checking is not feasible for infinite state space systems, which is caused by unbounded data size or infinite number of processes. The performance of model checking approach is related to the size of the system state space. It is a real challenge to examine the largest possible state space that can be handled by limited processors and memories. State-of-the-art model checkers can handles state space of about 10^8 to 10^9 states [111] with explicit state-space enumeration. The main challenge in model checking is dealing with the *state space explosion* problem. This problem occurs in systems with many components that can interact with each other or data structures with many different values. In such cases, the number of global states can be enormous. The linear increment of the interaction or data values will give exponential increment of the state space. For example, a concurrent program with k processes can have a state graph of size $exp(k)$. During the last ten years, considerable research works have been focusing on this problem.

The process of model checking consists of several tasks. First of all, system design is converted into a formalism accepted by model checking tools. The requirement of the systems is abstracted as logic specifications. One common example is temporal logic, which can assert how the behavior of the system evolves over time. The verification of the specification against the system model is generally conducted automatically. If the result is negative, the user is often provided with a witness trace (or counterexample). The analyzing of the error trace may require modifications to the model and repeat the model checking process. Each process of the model checking, namely modeling, specification and verification, will be explained in the following sections.

2.2 System Modeling

First of all, we convert the system, which should be examined, into a formalism accepted by a model checking tool. Formal modeling is a difficult and critical step. Sometimes, owing to limitations on time and memory, the modeling of a design may require the use of abstraction. It may not be so simple to provide the model, because on the one hand relevant or important points must be represented in the model, on the other hand unnecessary details should be eliminated. For example, when reasoning about a communication protocol we may want to focus on the exchange of messages and ignore the actual contents of the messages.

A *state* (or *configuration*) is a snapshot or instantaneous description[2] of the system that captures program counter and the values of the variables at a particular instant of time. The state change as the result of some action of the system is described by transitions, which is a pair of states with an *action* (or *event*) linking them. A computation of a system is a finite or infinite sequence of states where each state is obtained from the previous state by some transition. We use a state transition graph called a *Kripke structure* [113] to formally model a system.

Definition 1 (Kripke structure) *Let AP be a non-empty set of atomic propositions. A* Kripke structure *M over a set of atomic propositions AP is a four-tuple $M = (S, S_0, R, L)$, where S is a finite set of states; $S_0 \subseteq S$ is the set of initial states; $R \subseteq S \times S$ is a transition relation; $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in this state.*

A Kripke structure is a graph having the reachable states of the system as nodes and state transitions of the system as edges. It also contains a labeling of the states of the system with properties that hold in each state. In this thesis, we adopt an event-based formalism. Therefore we extend the transitions in Kripke structure with events (denoted as Σ) to link the pair of states.

[2] The components of state are determined by the actual modeling language. See Section 3.1.2 for the state definition of the proposed modeling language in this thesis.

2.3 Specification and Verification

Specifications are the properties that the design must satisfy. There are different ways to express properties. In state-based formalisms, properties are generally stated using temporal logics to specify how the system evolves over time. These properties are divided into two categories: safety properties and liveness properties. In this section, we present the common properties and algorithms (and techniques) used to verify them.

2.3.1 Safety Property

A safety property is a property stating that “something bad never happens". Generally, safety requirements include the absence of deadlocks and similar critical states that can cause the system to crash.

Deadlock Sequential programs that are not subject to divergence (e.g., endless loops) have a terminal state, which has no outgoing transitions. For concurrent systems, however, computations typically do not terminate. In such case, terminal states are undesirable and mostly represent a design error. Apart from simple design errors where it has been forgotten to indicate certain activities, in most cases such terminal states indicate a *deadlock*. A deadlock occurs if the complete system is in a terminal state, although at least one component is in a (local) nonterminal state. A typical deadlock scenario occurs when components mutually wait for each other to progress.

A more general form of safety property can be stated as a logic formula of the atomic propositions, e.g., $\neg(cs_0 \land cs_1)$ is a safety property for mutual exclusion problem meaning that process 0 and process 1 cannot be in the critical section at the same time.

To verify safety properties, we simply need to conduct a depth first search (or breadth first search) in the state space. During the search, if the reached state is undesirable (e.g., having no outgoing transitions in case of deadlock checking), then we detect an evidence trace as a counterexample. The algorithm (based on depth first search) has been implemented in PAT, which gives a linear time complexity in the size of the state space.

Note that, in the event-oriented world, deadlock-freedom is a liveness property rather than safety property. In practice, safety and liveness do not apply well in a pure event-based formalism. “Something" that happens (or not) is really a configuration of the system and/or its environment, not necessarily an observable event. In this thesis, we treat deadlock as a safety property.

2.3.2 Liveness Properties and Linear Temporal Logics

Different from safety properties, liveness properties mean that “something good" will eventually happen. Safety properties are violated in finite time, liveness properties are violated in infinite time, i.e., by infinite system runs. For example, in a mutual exclusion algorithm, typical examples of liveness properties are including:

- (eventually) each process will eventually enter its critical section;
- (repeated eventually) each process will enter its critical section infinitely often;
- (starvation freedom) each waiting process will eventually enter its critical section.

To model liveness properties, the common practice is to use *Temporal Logics*, e.g. *Computation Tree Logic (CTL)* [55], *Linear Temporal Logic (LTL)* [188] and *CTL** [56]. It is known that, for purposes of model checking, there is no practical complexity advantage to restrict oneself to a particular temporal logic [80]. In this thesis, we focus on LTL as the algorithms proposed later are consistent with LTL model checking.

Definition 2 *Let Pr be a set of propositions and Σ be a set of events. A LTL formula is,*

$$\phi ::= p \mid a \mid \neg\, \phi \mid \phi \wedge \psi \mid X\phi \mid \Box\phi \mid \Diamond\phi \mid \phi U \psi$$

where p ranges over Pr and a ranges over Σ. Let $\pi = \langle s_0, e_0, s_1, e_1, \cdots, e_i, s_i, \cdots\rangle$ be an infinite *execution. Let π^i be the suffix of π starting from s_i.*

$$\begin{array}{ll}
\pi^i \models p & \Leftrightarrow s_i \models p \\
\pi^i \models a & \Leftrightarrow e_{i-1} = a \\
\pi^i \models \neg\, \phi & \Leftrightarrow \neg(\pi^i \models \phi) \\
\pi^i \models \phi \wedge \psi & \Leftrightarrow \pi^i \models \phi \wedge \pi^i \models \psi \\
\pi^i \models X\phi & \Leftrightarrow \pi^{i+1} \models \phi \\
\pi^i \models \Box\phi & \Leftrightarrow \forall\, j \geq i \bullet \pi^j \models \phi \\
\pi^i \models \Diamond\phi & \Leftrightarrow \exists\, j \geq i \bullet \pi^j \models \phi \\
\pi^i \models \phi U \psi & \Leftrightarrow \exists\, j \geq i \bullet \pi^j \models \psi \wedge \forall\, k \mid i \leq k \leq j-1 \bullet \pi^j \models \phi
\end{array}$$

Informally, $X\phi$ means ϕ has to hold at the next state. $\Box\phi$ means ϕ has to hold at all states in the execution. $\Diamond\phi$ means ϕ eventually has to hold in some state of the execution. $\phi U \psi$ means that ϕ has to hold at least until ψ, which holds at the current or a future position.

Using LTL, safety properties can be specified as formula $\Box\neg p$, where p is the undesired property. Regarding the three liveness properties in the mutual exclusion example, they can be formulated as follows, where cs_i means the process i is in the critical section, and $waiting_i$ means the process i is in the waiting state.

- (eventually) $\Diamond cs_i$;
- (repeated eventually) $\Box\Diamond cs_i$;
- (starvation freedom) $\Box(waiting_i \Rightarrow \Diamond cs_i)$.

Note that because we are dealing with an event-based formalism (see Chapter 3 for the proposed language) in this thesis, it would be meaningful if the properties may concern both states and events. For instance, $waiting_i$ and cs_i in the examples above are events. The simplicity of writing formulae concerning events as in the above examples is not purely a matter of aesthetics. It may yield gains in time and space (refer to examples in [50]).

Explicit state model checking uses a graph to represent a Kripke structure with nodes for states and edges for transitions. The input model is converted into a corresponding automaton $\mathcal{M}$, and the negation of the LTL specification ϕ is translated into a Büchi automaton $\mathcal{B}^{\neg\phi}$. Then, the emptiness of the product of $\mathcal{M}$ and $\mathcal{B}^{\neg\phi}$ is checked. If the product is not empty, a counterexample is reported. Algorithms (of model checking finite state systems) based on the explicit state enumeration could be improved if only a fraction of the reachable states are explored. In many cases, it is possible to avoid constructing the entire state space of the model. This is because the states of the automaton are generated only when needed, while checking the emptiness of its intersection with the property automaton $\mathcal{B}$. This tactic is called *on-the-fly* model checking [61, 110]. Instead of constructing the automata for both $\mathcal{M}$ and $\mathcal{B}^{\neg\phi}$ first, we will only construct the property automaton $\mathcal{B}^{\neg\phi}$. We then use it to guide the construction of the system automaton $\mathcal{M}$ while computing the product. In this way, we may frequently construct only a small portion of the state space before we find a counterexample to the property being checked.

One advantage of on-the-fly model checking is that when computing the intersection of the system automaton $\mathcal{M}$ with the property automaton $\mathcal{B}$, some states of $\mathcal{M}$ may never be generated at all. Another advantage of the on-the-fly procedure is that a counterexample may be found before completing the construction of the intersection of the two automata. Once a counterexample has been found and reported, there is no need to complete the construction. Interested readers can refer to Section 4.3 for more details of this approach.

2.3.3 Partial Order Reduction

Systems that consist of a set of components that cooperatively solve a certain task are quite common in practice, e.g. hardware systems, communications protocols, distributed systems, and so on. Typically, such systems are specified as the parallel composition of n processes. The state space of this specification equals in worst case the product of the state spaces of the components. A major cause of this state space explosion is the representation of parallel by means of interleaving. Interleaving is based on the principle that a system run is a totally ordered sequence of actions. In order to represent all possible runs of the systems, all possible interleaving of actions of components need to be represented. For checking a large class of properties, however, it is sufficient to check only some representative of all these interleavings.

For example, if two processes both increment an integer variable in successive steps, the end result is the same regardless of the order in which these assignments occur. The underlying idea of this approach is to reduce the interleaving representation into a partial-order representation. System runs are now no longer totally ordered sequences, but partially ordered sequences. The *partial order reduction* technique [213, 164] aims at reducing the size of the state space that needs to be explored by model checking algorithms. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders.

The method consists of constructing a reduced state graph. The full state graph, which may be too big to fit in memory, is never constructed. The behaviors of the reduced graph are a subset of the behaviors of the full state graph. The justification of the reduction method shows that the behaviors that are not present do not add any information. More precisely, it is possible to define an equivalence relation among behaviors such that the checked property cannot distinguish between equivalent behaviors. If a behavior is not present in the reduced state graph, then an equivalent behavior must be included.

2.4 Model Checking Real-time Systems

Computers are frequently used in critical applications where predictable response times are essential for correctness. Such systems are called *real-time systems*. Examples of such applications include controllers for aircraft, industrial machinery and robots. Due to the nature of such applications, errors in real-time systems can be extremely dangerous, even fatal. Guaranteeing the correctness of a complex real-time system is an important and nontrivial task.

There are two time semantics in the definition of real-time systems. Discrete-time semantics [11] requires that all time readings are integers and all clocks increment their readings at the same time. The other choice is dense-time semantics [9], which means that time readings can be rational numbers or real numbers and all clocks increment their readings at a uniform rate. We discuss the two semantics in the following paragraphs.

2.4.1 Discrete-time Systems

When time is discrete, possible clock values are nonnegative integers, and events can only occur at integer time values. This type of model is appropriate for *synchronous systems*, where all of the components are synchronized by a single global clock. The duration between successive clock ticks is chosen as the basic unit for measuring time. This model has been successfully used for reasoning about the correctness of synchronous hardware designs for many years.

In discrete-time models, we require that there is a single global clock. Therefore, there is a simple and obvious way [133] to support it: introduce a clock variable now, whose value represents the current time, and model the passage of time with a $Tick$ action that increments now. For a continuous-time specification, $Tick$ might increment now by any real number; for a discrete-time specification, it increments now by 1. Timing bounds on actions are specified with one of three kinds of timer variables: a countdown timer is decremented by the $Tick$ action, a count-up timer is incremented by $Tick$, and an expiration timer is left unchanged by $Tick$. A countdown or count-up timer expires when its value reaches some value; an expiration timer expires when its value minus now reaches some value. An upper-bound timing constraint on when an action A must occur is expressed by an enabling condition on the $Tick$ action that prevents an increase in time from violating the constraint; a lower-bound constraint on when A may occur is expressed by an enabling condition on A that prevents it from being executed earlier than it should be.

Alternative approaches are using quantitative temporal analysis for discrete-time systems [81, 46]. These approaches extend CTL with *bounded until* operator [81] to support the specification of timing constraints between two actions. One example query can be that “it is always true that p may be followed by q within 3 time units". Verification algorithms are then developed for this extended modal logic in the similar way of CTL model checking. To increase the scalability, symbolic model checking technique is integrated with this solution [46].

2.4.2 Dense-time Systems

Compared to discrete-time, dense-time is the natural model for *asynchronous systems*, because the separation of events can be arbitrarily small. This ability is desirable for representing causally independent events in an asynchronous system. Moreover, no assumptions need to be made about the speed of the environment when this model of time is assumed. In discrete-time modeling, it is necessary to separate time by choosing some fixed time quantum so that the delay between any two events will be a multiple of this time quantum. This may limit the accuracy with which systems can be modeled. Also the choice of a sufficiently small time quantum to model an asynchronous system accurately may blow up the state space so that verification is no longer possible.

During the last two decades, a number of specification languages are proposed for modeling dense-time systems, e.g. *Timed Automata* [10, 149], *Timed Process Algebra* [223, 174, 182], *Timed Interval Calculus* [85], *Timed Statecharts* [117] and so on. The verification approaches include model checking, simulation, theorem proving and so on. Theorem proving [51, 52] is proved to be a successful means to verify systems with real-valued clock variables. However, model checking technique is difficult to apply in this case since it is designed for finite state systems. Therefore, abstraction is needed to apply model checking. This thesis focuses on the abstraction technique of dense-time models with rational-valued clock values.

Among the specification languages, *Timed Automata* has become the standard modeling technique in designing real-time models. Timed Automata are finite state machines equipped with clocks variables (ranging over rational numbers). The elapse of the time can be modeled as clock variables updating, and execution of the model can be constrained by guard expressions involving clocks. This definition provides a general way to annotate state transition graphs with timing constraints using finitely many rational-valued clock variables. In order to obtain a finite representation for the infinite state space of rational-valued clocks, abstraction techniques like *clock regions* [8] and *clock zones* [68, 224] are used to verify real-time models. Based on these techniques, a number of efficient verification tools for Timed Automata have been developed, e.g. UPPAAL [135], KRONOS [35], RED [217], Timed COSPAN [207] and Rabbit [33].

Chapter 3

System Modeling

System modeling is very important and highly non-trivial. The choice of specification language is an important factor in the success of the entire development. The language should cover several facets of the requirements and the model should reflect exactly (up to abstraction of irrelevant details) an existing system or a system to be built. The language should have a semantic model suitable to study the behaviors of the system and to establish the validity of desired properties. A formal semantic model is highly desirable, which can act as the basis for a variety of system development activities, e.g., system simulation, visualization, verification or prototype synthesis.

In this chapter, we introduce an event-based modeling language for concurrent systems and real-time systems. This language has a rich set of operators for concurrent communications and time calculations. It is used as the modeling language in the rest of the thesis and our PAT tool.

The remainder of this chapter is organized as follows. Section 3.1 presents CSP#, the modeling language for concurrent systems with formal syntax (in Section 3.1.1) and semantic model (in Section 3.1.2). The discussion in Section 3.1.3 explains the semantic relationship between CSP and CSP#. Section 3.1.4 demonstrates a CSP# model of a multi-lift system. Section 3.2 presents the language extensions of CSP# for modeling real-time systems. Several timed process constructs are introduced in Section 3.2.1, and their semantics is explained in Section 3.2.2. Section 3.2.3 demonstrates the real-time system modeling using Fischer's mutual exclusion algorithm. Section 3.3 discusses related works and summarizes the chapter.

3.1 Concurrent System Modeling

Many specification languages have been proposed for the modeling of concurrent systems. High-level languages like CSP (Communicating Sequential Processes) [108] and CCS (Calculus of Communicating Systems) [155] use mathematical objects as abstractions to represent systems or processes. System behaviors are described as process expressions combined with a rich set of hierarchical operators, e.g., deterministic or nondeterministic choice, parallel composition and recursion. The operators are associated with elegant algebraic laws for system analysis.

The original CSP derives its full name from the built-in syntactic constraint that processes belong to the sequential subset of the language. CSP has passed the test of time. It has been widely accepted and influenced the design of many recent programming and specification languages. Nonetheless, modeling systems with non-trivial data structures and functional aspects completely using languages like CSP remains difficult. A characteristic of CSP is that processes have disjoint local variables, which was influenced by Dijkstra's principle of *loose coupling* [67]. CSP supports inter-process communication through message passing but not shared memory, i.e., shared variables. It has long been known (see [108] and [177], for example) that one can model a variable as a process parallel to the one that uses it. The user processes then read from, or write to, the variable by CSP communication. Though feasible, this is painful for systems with non-trivial data structures (e.g., arrays) and operations (e.g., array sorting). Therefore, 'syntactic sugars' like shared variables are mostly welcome.

In order to solve the problem, many specification languages integrating process algebras like CSP or CCS with state-based specification languages like Z language [222], Object-Z [189], have been proposed. The state-based language component is typically used to specify the data states of the system and the associated data operations in a declarative style. Examples include *Circus* [221] (an integration of CSP and the Z language), CSP+OZ [86, 190] (an integration of CSP and Object-

Z), CCS+Z [205] (an integration of CCS and the Z language) and TCOZ [152] (an integration of Timed CSP [181] and Object-Z). However, because declarative specification languages like Z are very expressive and not executable, automated analyzing (in particular, model checking) of systems modeled using the integrated languages is extremely difficult.

In this chapter, we propose an alternative solution, i.e., instead of specifying data states and operations in declarative languages, they are given as procedural codes. We propose a modeling language named CSP# (short for communicating sequential programs, pronounced as ‘CSP sharp’) which combines high-level modeling operators with low-level programs, for the purpose of concurrent system modeling and verification. We demonstrate that data operations can be naturally modeled as terminating sequential programs, which then can be composed using high-level concurrency operators. *The idea is to treat sequential terminating programs as atomic non-communicating events.* CSP# models are executable with complete operational semantics, and therefore subject to system simulation and, more importantly, fully automated system verification techniques like model checking. CSP# is supported by the CSP module in PAT (see Section 10.3.1) and has been applied to model and verify a number of systems.

3.1.1 Syntax

Integrating a highly abstract language like CSP with programming codes leads to many complications. Our design principle is to maximally maintain CSP’s core elegance in specifying process synchronization, while also allowing state-based behaviors.

A motivating example

We use a multi-lift system as a running example in this section. The reason for choosing the multi-lift system is that it has complicated dynamic behaviors as well as nontrivial data states. Furthermore, the single-lift system has been modeled using many modeling languages including CSP [220]. The system contains multiple components, e.g., the users, the lifts, the floors, the internal button panels, etc. There are non-trivial data components and data operations, e.g., the internal requests and external requests and the operations to add/delete requests. For simplicity, we assume there is no central controller for assigning external requests. Instead, each lift functions on its own to find and serve requests, in the following way. Initially, a lift resides at the ground level ready to travel upwards. Whenever there is a request (from the internal button panel or outside button) for the current residing floor, the lift opens the door and later closes it. Otherwise, if there are requests for a floor on the current traveling direction (e.g., a request for floor 3 when the lift is at floor 1 traveling upwards), then the lift keeps traveling on the current direction. Otherwise, the lift changes its direction. Other constraints on the system include that a user may only enter a lift when the door is open, there could be an internal request if and only if there is a user inside, etc.

Sequential Programs as Events

Shared variables offer an alternative means of communication among processes (which reside at the same computing device or are connected by wires with negligible transmission delay). They record the global state and make the information available to all processes. In the lift example, the internal/external requests can be naturally modeled as shared arrays. Note that the global variables in the lift model are not the real memory used in the distributed lifts, but rather the modeling abstraction of them. To model the real lift systems, more details (e.g. the user request and systems memory) need to be included. In CSP#, they are declared as follows.

1. **#define** $NoOfFloor$ 3;
2. **#define** $NoOfLift$ 2;
3. **var** $extUpReq[NoOfFloor]$;
4. **var** $extDownReq[NoOfFloor]$;
5. **var** $intRequests[NoOfLift][NoOfFloor]$;
6. **var** $doorOpen[NoOfLift]$;

where $define$ and var are reserved keywords. The former defines a global constant, e.g., $NoOfFloor$ which denotes the number of floors and $NoOfLift$ which denotes the number of lifts. The latter defines a variable, e.g., $extUpReq$ array and $extDownReq$ array which store external requests of each floor (the index of the arrays corresponds to the floor level). Two dimensional array $intRequests$ stores internal requests raised from the users inside the lifts, since each lift has $NoOfFloor$ number

```
intRequests[i][level] = 0;
if (dir > 0) {
    extUpReq[level] = 0;
} else {
    extDownReq[level] = 0;
}
```

Figure 3.1: CSP# codes for clearing requests

of internal request buttons. $doorOpen$ array captures the doors' states of all lifts. Each variables are updated during the execution of the lift. CSP# has a weak type system (like JSP) and therefore type information is not necessary for variable declaration. By default, all the above defined are treated as arrays of integers. In particular, elements in $extUpReq$ (or $extDownReq$) are binary: 1 at j-th position means that there is a request for traveling *upwards* (or *downwards*) at j-th floor; 0 means no request. Two dimensional array $intRequests$ stores internal requests from all lifts. In particular, the internal request for the j-th floor from the i-th lift is stored at $intRequests[i][j]$ in the array. Elements in $intRequests$ are binary: 1 means that the floor has been requested and 0 means not requested. Elements in array $doorOpen$ range from -1 to $NoOfFloor - 1$. The i-th element of $doorOpen$ is -1 if and only if the door of i-th lift is closed and it is j such that $j \geq 0$ if and only if the i-th lift has opened door at j-th floor. We assume that initially all doors are closed. We remark if the Z language is used for specification, specific types for elements in the arrays may be defined to constrain their values.

Associated with the variables are data operations which query or modify the variables. In the lift system, whenever a lift opens its door, the requests must be updated accordingly. For instance, the codes shown in Figure 3.1 clear the requests when the i-th lift opens the door at $level$-th floor. Let dir be the current traveling direction (1 for upwards and -1 for downwards). The first line clears internal requests, by simply resetting the respective position in array $intRequests$ to 0. The rest clears external requests. Only the request along the lift's traveling direction is cleared. A more complicated operation is to determine whether there are requests along the current traveling direction, so as to determine whether a lift should keep traveling in the same direction or to change

```
$index = level + dir;$
$result[i] = 0;$
while $(index \geq 0 \wedge index < NoOfFloor \wedge result[i] == 0)$ {
  if $(extUpReq[index] \neq 0 \vee extDownReq[index] \neq 0 \vee intRequests[i][index] \neq 0)$ {
    $result[i] = 1;$
  } else {
    $index = index + dir;$
  }
}
```

Figure 3.2: CSP# codes for searching requests

direction. This operation may be *implemented* by the codes in Figure 3.2, where $level$ is a variable recording the floor that the lift is residing at, $index$ is a loop counter and $result[i]$ records the result (0 for no such request and 1 for yes). A while-loop is used to search for a request along the current traveling direction, e.g., if the lift is traveling upwards, we search for a request for (or from) an upper floor. The search stops when the ground or top floor is reached.

A system may contain multiple data operations, each of which is *terminating* and is assumed to be executed *atomically*. They can be implemented using the CSP# syntax as shown above, or they can be implemented using existing programming languages. For instance, we offer the keyword **call** in PAT to allow invocation of data operations (as atomic events) implemented externally as C# static methods in CSP# models. Data operations may be invoked alternatively or in parallel. From another point of view, data operations are events associated with an optional sequential terminating program. For instance, the program in Figure 3.2 may be labeled as event $checkIfToMove.i.level$, which then can be used to constitute CSP process expressions, e.g., see Figure 3.3 and 3.5. Data races are prevented by not allowing synchronization of events containing procedural code. In this sense, data operations are not “events" as in CSP, but rather (atomic) computations that are inserted in the model.

Composing Programs

The high-level compositional operators in CSP capture common system behavior patterns. They are very useful in system modeling. Furthermore, process equivalence can be proved by appealing to algebraic laws which are defined for the operators. In CSP#, we reuse most of the operators and integrate them with our extensions in a rigorous way so as to maximally preserve the algebraic laws.

A CSP# specification may contain multiple process definitions. A process definition gives a process expression a name, which can be referenced in process expressions. The following is a BNF description[1] of the process expression.

$$
\begin{array}{ll}
P = \mathit{Stop} \mid \mathit{Skip} & \text{– primitives} \\
\quad \mid \ e\{\mathit{prog}\} \rightarrow P & \text{– event prefixing} \\
\quad \mid \ \mathit{ch}!\mathit{exp} \rightarrow P \mid \mathit{ch}?x \rightarrow P & \text{– channel communications} \\
\quad \mid \ P \setminus X & \text{– hiding} \\
\quad \mid \ P\ ;\ Q & \text{– sequential composition} \\
\quad \mid \ P \ \Box \ Q \mid P \sqcap Q & \text{– choice operators} \\
\quad \mid \ \mathit{if}\ b\ \{P\}\ \mathit{else}\ \{Q\} & \text{– conditional choice} \\
\quad \mid \ [b]P & \text{– state guard} \\
\quad \mid \ P \parallel Q & \text{– parallel composition} \\
\quad \mid \ P \mid\mid\mid Q & \text{– interleave composition} \\
\quad \mid \ P \bigtriangleup Q & \text{– interrupt} \\
\quad \mid \ \mathit{ref}(Q) & \text{– process reference}
\end{array}
$$

where P, Q are processes, e is a name representing an event with an optional sequential program[2] prog, X is a set of event names (e.g., $\{e_1, e_2\}$), b is a Boolean expression, ch is a channel, exp is an expression, and x is a variable.

Stop is the process that communicates nothing, also called deadlock. $\mathit{Skip} = \checkmark \rightarrow \mathit{Stop}$, where $\checkmark$ is the special event of termination. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. If e is attached with a program (event prefixing of this type is called *data operation*), the program is executed atomically together with the occurrence of the event. Hiding process $P \setminus X$ makes all occurrences of events in X not to be observed or controlled by the environment of the

[1] Refer to PAT's user manual for ASCII version of the symbols.

[2] The grammar rules for the sequential program can be found in PAT user manual.

process. Sequential composition, $P\ ;\ Q$, behaves as P until its termination and then behaves as Q. External choice $P \ \Box\ Q$ is solved only by the occurrence of a visible event[3]. On the contrast, internal choice $P \sqcap Q$ is solved non-deterministically. Conditional choice[4] $if\ \ b\ \{P\}\ else\ \{Q\}$ behaves as P if b evaluates to true, and behaves as Q otherwise. Process $[b]P$ waits until condition b becomes true and then behaves as P. Note that $[b]P$ does not block and will be dropped in choice operators if other choices are selected. Notice that it is different from $if\ \ b\ \{P\}\ else\ \{Q\}$. One distinguishing feature of CSP is alphabetized multi-processes parallel composition. Let P's alphabet, written as αP, be the events in P excluding the special invisible event τ. Process $P \parallel Q$ synchronizes common events in the alphabets of P and Q excluding *non-communicating events* (see Section 3.1.2 for detailed discussion). In contrast, process $P\ |||\ Q$ runs all processes independently (except for communication through shared variables and synchronous channels[5]). Process $P \bigtriangleup Q$ behaves as P until the first occurrence of a visible event from Q. A process expression may be given a name for referencing. Recursion is supported by process referencing.

CSP# supports global variables, including boolean/integer variables, multi-dimensional arrays and user defined data structure[6], which are globally accessible, process parameters which are accessible in the respective process expression and local variables which are accessible in one data operation. We restrict the use of local variables in general. In particular, local variables introduced as process parameters or variables to store channel inputs cannot be modified by event associated programs. They can, however, be modified indirectly. The following illustrates alternative ways of achieving the same effect.

$$\begin{array}{lll}
P(x) & = add\{x = x+1\} \rightarrow P(x); & -\times \text{ process parameters cannot be modified.}\\
P(x) & = add \rightarrow P(x+1); & -\checkmark\\
\textbf{var}\ x;\ P() & = add\{x = x+1\} \rightarrow P(); & -\checkmark
\end{array}$$

Process parameters cannot be modified, hence they can take their values at the point of introduction.

[3] In CSP, symbol τ is introduced as the internal action in the operational semantics only. In CSP#, we abuse this notation to denote invisible events: internal actions from hiding, resolution of a choice or dereferencing of a process variable. Any event other than τ is called visible event.

[4] In CSP#, we also introduce ifb for conditional choice with busy waiting on the condition.

[5] Note that in original CSP, $P\ |||\ Q$ does not allow communication through shared channels.

[6] In PAT, user defined data structure can be created using external C# library.

This restriction allows us to perform efficient system verification. The reason is that, in this setting, it is sufficient to store only the valuation of the global variables and the process expression (with process parameters replaced with their values) when we explore the system states. Compared to software model checking, we can safely omit the *program stack* (which, combined with recursions, is very complicated to maintain) from the global state.

Besides global variables and data operations, the most noticeable extension to CSP is the use of asynchronous channels, which again can be supported in CSP by explicitly modeling the communication buffer. Nonetheless, explicitly supporting them in CSP# is not only for users' convenience but also for possibly more efficient mechanical system exploration. Given a channel ch with predefined buffer size, process $ch!exp \rightarrow P$ evaluates the expression exp (with the current valuation of the variables) and puts the value into the tail of the respective buffer and behaves as P. Process $ch?x \rightarrow P$ gets the top element in the respective buffer, assigns it to variable x and then behaves as P. Sending/receiving multiple messages at once is supported. If a channel has buffer size 0, it is a synchronous channel, whose input and output communications must occur synchronously.

Definition 3 (System model) *A system model is a 3-tuple $\mathcal{S} = (Var, init_G, P)$, where Var is a set of global variables (including channels), $init_G$ is the initial valuation of the variables and P is a process.*

Figure 3.3 presents a process $Lift$ which concisely models the behavior of one lift. Notice that the process has multiple parameters, namely i which is an identifier of the lift, $level$ which denotes the residing floor and dir which denotes the current traveling direction (1 for traveling upwards and -1 for downwards). The condition at line 7 is used to check whether there is a request for the current floor, with the correct traveling direction if it is external. If yes, then the door is opened, the requests for the floor are cleared (using the code presented in Figure 3.1), and then the door is closed. Otherwise, the lift checks whether to continue traveling on the same direction (using the code presented in Figure 3.2). If the result is 1 (line 13), then the lift moves to the next floor. Otherwise, the lift checks whether it need to change direction (line 21). If there is no request in all directions, then the lift will idle at the current level. In this example, we have events which are

```
$Lift(i, level, dir) =$ if $((dir > 0 \wedge extUpReq[level] == 1)$
    $\vee\ (dir < 0 \wedge extDownReq[level] == 1) \vee\ intRequests[i][level] == dir)\{$
        $opendoor.i\{doorOpen[i] = level;$ *code shown in Figure 3.1*$\} \rightarrow$
        $closedoor.i\{doorOpen[i] = -1\} \rightarrow Lift(i, level, dir)$
    $\}$ else $\{$
        $checkToMoveAlong.i.level\{$*code shown in Figure 3.2*$\} \rightarrow$
        if $(result[i] == 1)\{moving.i.dir \rightarrow$
            if $(level + dir == 0 \vee level + dir == NoOfFloors - 1)\{$
                $Lift(i, level + dir, -1 * dir)$
            $\}$ else $\{Lift(i, level + dir, dir)\}$
        $\}$ else $\{$
            if $((level == NoOfFloors - 1 \wedge dir == -1)$
              $\vee\ (level == 0 \wedge dir == 1))\ \{Lift(i, level, dir)\}$
            else $\{$
                $checkDir.i.level\{$*code shown in Figure 3.2 with -1*dir*$\} \rightarrow$
                if $(result[i] == 1)\{changedir.i.level \rightarrow Lift(i, level, -1 * dir)\}$
                else $\{idle.i.level \rightarrow Lift(i, level, dir)\}$
            $\}$
        $\}$
    $\};$
```

Figure 3.3: CSP# model of the lift

associated with programs and simple events like $moving.i.dir$. The rest of the system model is presented in Section 3.1.4.

3.1.2 Semantics

In the section, we present operational semantics of CSP# models, which translates a model into a Labeled Transition System (LTS). The sets of behaviors can be extracted from the operational semantics, thanks to congruence theorems. The complication due to conflicts between global variables and CSP operational semantics (e.g. calculation of process alphabets) is discussed.

$$\frac{}{(V, \mathit{Skip}) \xrightarrow{\checkmark} (V, \mathit{Stop})}\ [\ \mathit{skip}\ ] \qquad \frac{}{(V, e\{\mathit{prog}\} \rightarrow P) \xrightarrow{e} (\mathit{upd}(V, \mathit{prog}), P)}\ [\ \mathit{prefix}\ ]$$

$$\frac{c \text{ is not full in } V}{(V, c!\mathit{exp} \rightarrow P) \xrightarrow{c!\mathit{eva}(V, \mathit{exp})} (\mathit{app}(V, c!\mathit{exp}), P)}\ [\ \mathit{out}\ ] \qquad \frac{P \mathrel{\widehat{=}} Q, (V, Q) \xrightarrow{*} (V', Q')}{(V, P) \xrightarrow{*} (V', Q')}\ [\ \mathit{def}\ ]$$

$$\frac{c \text{ is not empty in } V}{(V, c?x \rightarrow P) \xrightarrow{c?\mathit{top}(c)} (\mathit{pop}(V, c?x), P)}\ [\ \mathit{in}\ ] \qquad \frac{V \models b, (V, P) \xrightarrow{e} (V', P')}{(V, [b]P) \xrightarrow{e} (V', P')}\ [\ \mathit{guard}\ ]$$

$$\frac{V \models b, (V, P) \xrightarrow{x} (V', P')}{(V, \mathit{if}\ b\ \{P\}\ \mathit{else}\ \{Q\}) \xrightarrow{x} (V', P')}\ [\ \mathit{con}1\ ] \qquad \frac{V \not\models b, (V, Q) \xrightarrow{x} (V', Q')}{(V, \mathit{if}\ b\ \{P\}\ \mathit{else}\ \{Q\}) \xrightarrow{x} (V', Q')}\ [\ \mathit{con}2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P'), x \in \alpha P, x \notin \alpha Q}{(V, P \parallel Q) \xrightarrow{x} (V', P' \parallel Q)}\ [\ \mathit{pa}1\ ] \qquad \frac{(V, Q) \xrightarrow{x} (V', Q'), x \in \alpha Q, x \notin \alpha P}{(V, P \parallel Q) \xrightarrow{x} (V', P \parallel Q')}\ [\ \mathit{pa}2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V, P'), (V, Q) \xrightarrow{x} (V, Q'), x \in \alpha P \cap \alpha Q}{(V, P \parallel Q) \xrightarrow{x} (V, P' \parallel Q')}\ [\ \mathit{pa}3\ ]$$

where $e \in \Sigma$; $e_\tau \in \Sigma \cup \{\tau\}$; $x \in \Sigma \cup \{\checkmark\}$ and $* \in \Sigma \cup \{\tau, \checkmark\}$

Figure 3.4: CSP# firing rules

Operational Semantics

In order to define the operational semantics of a system model, we first define the notion of system configuration to capture the global system state during system executions.

Definition 4 (System configuration) *A system* configuration *(or* state*) is composed of two components (V, P) where V is a function mapping a variable name (or a channel name) to its value (or a sequence of items in the buffer), which we refer to as a valuation function, and P is a process expression.*

The operational semantics is presented as firing rules associated with each process construct. The rules naturally extend the operational semantics for CSP [41] and Timed CSP [181]. Let Σ denote the set of all visible events and τ[7] denote the set of all invisible events[8]. For simplicity, we assume a function $upd(V, prog)$ which, given a sequential program $prog$ and V, returns the modified valuation function V' according to the semantics of the program. We write $V \models b$ (or $V \not\models b$) to denote that condition b evaluates to true (or false) given V. We write $eva(V, exp)$ to denote the value of the expression evaluated with variable valuations in V. To abuse notations, we write $app(V, ch!exp)$ to denote the function V' in which the respective channel buffer is appended with $eva(V, exp)$. We write $pop(V, ch?x)$ to denote the function V' in which the top element (written as $top(V, ch)$) in the respective channel buffer is removed.

Figure 3.4 illustrates part of the firing rules. The rest can be found in the Appendix A. Rule $prefix$ captures how event associated with sequential programs are handled, i.e., the occurrence of the event and program is simultaneous and appears, to the system, to be atomic. Notice that, this is the only way global variables are modified. Rule out and in capture the semantics of channel output/input. We remark that there are two rules ($con1$ and $con2$) associated with $if\ b\ \{P\}\ else\ \{Q\}$, whereas only one rule ($guard$) is associated with $[b]P$. Therefore, if b is false given $[b]P$, then the process will block until b becomes true.

The semantics of parallel composition $P \parallel Q$ are captured using three rules. Either P or Q can make a move if the event x is not in their common alphabets (see rule $par1$ and $par2$), otherwise P and Q have to synchronize on x (see rule $par3$). Notice that the event in a data operation is called *non-communicating event*, which is excluded from the alphabet in CSP#. For instance, assume x is a global variable,

$$P() = a\{x = x + 1\} \rightarrow Stop;$$
$$Q() = a\{x = x + 2\} \rightarrow Stop;$$

Given the above, event a is not synchronized in the parallel composition of $P()$ and $Q()$. The intuition is that data operations are local actions, instead of communications. This prevents synchronizing events associated with different data operations but with the same name (e.g., a in the above example) and syntactically avoids potential data race.

[7] Since invisible events are indistinguishable, we sometimes also use τ to represent an arbitrary invisible event.

[8] Invisible events are internal actions from hiding, resolution of a choice or dereferencing of a process variable.

Labeled Transition Systems

The semantics of a model $\mathcal{S}$ is defined using a Labeled Transition System (LTS) as follows.

Definition 5 (Labeled Transition System (LTS)) *Given a model $\mathcal{S} = (Var, init_G, P)$, let Σ denote the set of all visible events in P and τ*[9] *denote the set of all invisible events, and Σ_τ be $\Sigma \cup \tau$. The labeled transition system corresponding to $\mathcal{S}$ is a 3-tuple $\mathcal{L}^{\mathcal{S}} = (S, init, \rightarrow)$, where S is a set of configurations, $init \in S$ is the initial configuration $(init_G, P)$, and $\rightarrow \subseteq S \times \Sigma_\tau \times S$ is a labeled transition relation.*

Note that the labeled transition relationship $\rightarrow$ for CSP# processes conforms to the operational semantics presented in Figure 3.4 and the Appendix A.

For configurations $s, s' \in S$ and $e \in \Sigma_\tau$, we write $s \xrightarrow{e} s'$ to denote $(s, e, s') \in \rightarrow$, and e is called the engaged event of transition $s \xrightarrow{e} s'$. The set of enabled events at s is $enabled(s) = \{e : \Sigma_\tau \mid \exists\, s' \in S, s \xrightarrow{e} s'\}$. We write $s \overset{e_1, e_2, \cdots, e_n}{\longrightarrow} s'$ iff there exist $s_1, \cdots, s_{n+1} \in S$ such that $s_i \xrightarrow{e_i} s_{i+1}$ for all $1 \leq i \leq n$, $s_1 = s$ and $s_{n+1} = s'$, and $s \xrightarrow{\tau *} s'$ iff $s = s'$ or $s \overset{\tau, \cdots, \tau}{\longrightarrow} s'$. The set of configurations reachable from s by performing zero or more τ transitions is $\tau^*(s) = \{s' : S \mid s \xrightarrow{\tau *} s'\}$. Let Σ^* be the set of finite traces. $tr : \Sigma^*$ is a sequence of visible events. $s \overset{tr}{\Rightarrow} s'$ if and only if there exist $e_1, e_2, \cdots, e_n \in \Sigma_\tau$ such that $s \overset{e_1, e_2, \cdots, e_n}{\longrightarrow} s'$. $tr = \langle e_1, e_2, \cdots, e_n \rangle \upharpoonright \tau$ is the trace with invisible events removed, and $tr \upharpoonright X$ removes the events in X from the sequence tr. The set of traces of $\mathcal{L}$ is $traces(\mathcal{L}) = \{tr : \Sigma^* \mid \exists\, s' \in S, init \overset{tr}{\Rightarrow} s'\}$. A finite $execution$ of $\mathcal{L}$ is a finite sequence of alternating configurations/events $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1} \rangle$ where $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all $0 \leq i \leq n$. An infinite $execution$ is an infinite sequence $\langle s_0, e_0, s_1, e_1, \cdots, e_i, s_i, \cdots \rangle$ where $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all $i \geq 0$.

We remark that the total behaviors of a LTS $\mathcal{L}$ can be represented by its all possible executions. This is similar to the semantic model of TCOZ [171] consisting of events and variable values[10].

[9] Since invisible events are indistinguishable, we sometimes also use τ to represent an arbitrary invisible event.

[10] TCOZ builds on the strengths of Object-Z and Timed CSP. Interested readers can refer to [171] for its semantic model.

The trace semantics (i.e. $traces(\mathcal{L})$) is an event-projection of the executions, which explains the behaviors related to events only.

Given a LTS $(S, init, \rightarrow)$, the size of S can be infinite for two reasons. First, the variables may have infinite domains or the channels may have infinite buffer size. We require (syntactically) that the sizes of the domains and buffers are bounded. Second, processes may allow unbounded replication by recursion, e.g., $P = (a \rightarrow P;\ c \rightarrow \mathit{Skip}) \Box b \rightarrow \mathit{Skip}$, or $P = a \rightarrow P \mid\mid\mid P$. In this thesis (except Section 5.6), we consider only LTSs with a finite number of states for practical reasons. In particular, we bound the sizes of value domains and the number of processes by constants. In our examples, bounding the sizes of value domains also bounds the depths of recursions.

A model is deadlock-free if and only if there does not exist a finite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that s_{n+1} is a deadlock state, i.e., no firing rules are applicable given s_{n+1}. Given a proposition p, a state satisfying the predicate is reachable (or equivalently p is reachable) if and only if there exists a finite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that $s_{n+1} = (V_{n+1}, P_{n+1})$ and $V_{n+1} \models p$.

3.1.3 Discussion

The modeling power of CSP# is demonstrated via its syntax and semantics definitions. For all the operators borrowed from CSP, the operational semantics are kept in CSP#. However, the introduction of variable and states in CSP# sacrifices some algebra properties of the original CSP, particularly $compositionality$. $Compositionality$ means the behaviors meaning of a complex process is determined by the meanings of its constituent processes and the rules used to combine them. CSP language is $compositional$ with respect to properties expressed constraints upon possible traces or upon combinations (called failures) of traces and sets of events that may then be refused (refer to Section 6.1 for detailed definition of failures and refusal). Any universal property proven of the traces or failures of a process remains true when that process is placed in combination with others. Equally, a refinement of a component induces a refinement of the complete system. We refer this property as $event\text{-}based\ compositionality$. For example, given process $P1, P2$ and Q, if

$traces(P1) = traces(P2)$ holds, then $traces(P1 \mid\mid Q) = traces(P2 \mid\mid Q)$. However the following CSP# processes do not preserve this property.

$$\begin{array}{l}
\textbf{var}\ amount\ =\ 0; \\
P1() = pay\{amount = 5\} \rightarrow P1(); \\
P2() = pay\{amount = 15\} \rightarrow P2(); \\
Q() = [amount > 10](accept\{amount = 0\} \rightarrow Q());
\end{array}$$

where $traces(P1) = traces(P2)$, but $traces(P1 \mid\mid Q) \neq traces(P2 \mid\mid Q)$ because the guard condition $amount > 10$ is affected by the assignments in the different pay events.

It is clear that the use of shared variables in CSP# influence process behaviors. It has long been known that one can model a variable as a process parallel to the one that uses it [108, 177]. The user processes then read from, or write to, the variable by CSP communication. However, introducing variable directly in CSP abstracts out these details, which affects some language properties. This result is not surprising, because the semantics of CSP# is explained by executions rather than purely traces. If considering the system behavior as executions (including both event and states), CSP# may be viewed as $execution\text{-}based\ compositional$[11]. In the given example, the executions of $P1$ and $P2$ are not equivalent, therefore it is not a counterexample of $execution\text{-}based\ compositionality$. If focusing on the $event\text{-}based\ compositionality$, CSP# is not a direct extension of CSP. On the other hand, the execution semantic model does not limit the usefulness of CSP#, and all the verification algorithms presented in this thesis require no compositionality and the results do not rely on any semantic relationship between CSP and CSP#.

CSP assumes an open environment, in which we cannot insist that some enabled events can be engaged or an external choice is resolved in a particular direction. To prove any property involving a notion of occurrence, engagement or performance (which is related to the discussion in Chapter 4, 6, 8 and 9), we must assume some closure property or equivalently, that the process is place in an environment where it provides the sole constraint upon the occurrence of the events concerned. That is, that these events are effectively “internal”, or that we have a closed global view.

[11] To prove this, we need to formally present the denotational semantics of CSP#, which can be defined in a similar way as TCOZ [171]. We leave this as one future work.

3.1.4 Case Study: a Multi-lift System

In this section, we complete the case study of the multi-lift model. Our modeling is related to the previous lift system model presented in [151]. In addition, we demonstrate how to write critical system properties as assertions. In this model, we assume there is no central controller to coordinate the lifts, hence multiple lifts may simultaneously compete to serve a single floor.

Figure 3.5 shows the rest of the model. In particular, line 27 defines the rest of the variables (which are used in Figure 3.2). Lines 28 to 38 model users' behavior in the lift system. At line 28, the behavior of three users is defined as the interleaving of each user, where $||| \ x : \{i..j\}@P(x) = P(i) \ ||| \ \cdots \ ||| \ P(j)$. Behavior of a user is specified as process $aUser$ at line 29. Each user may initially be at any floor. This is captured using indexed external choice. The user pushes a button (for traveling upwards or downwards, specified as $ExternalPush(pos)$) and then waits for the lift to come (specified as $Waiting(pos)$).

A **case** statement, which is a syntactic sugar for multiple if-then-else statements, is used in process $ExternalPush(pos)$. We remark that the conditions in the case statement are evaluated in the order until one which evaluates to be true is found. Otherwise, the $default$ branch is taken. In the example, if the user is at the ground floor or the top one, only one direction to travel can be requested. Otherwise, the user may choose either to go upwards or downwards. Lines 31 to 34 capture how the external requests are updated.

The user then waits for the door opened at the user's floor (captured by condition $doorOpen[i] == pos$ at line 36) and then enters the lift. We remark that this model allows users to enter the lift with the wrong traveling direction (which may happen in real world). After making an internal request, the user may exit when the door is opened again at his/her destination floor. The lift system is modeled as the interleaving of users and multiple lifts at line 39. Initially, the lifts are residing at the ground floor, ready to travel upwards. In this example, we demonstrate how variable updates and concurrency operators may be used together seamlessly to capture system behavior.

Once we have a model, we may use PAT to simulate its behaviors. Alternatively, we may write assertions about critical system properties and invoke the PAT model checkers to examine the model

```
27. var index;  var result[NoOfLift];
28. Users() =||| x : {0..2}@aUser();
29. aUser() =□ pos : {0..NoOfFloor − 1}@(ExternalPush(pos);  Waiting(pos));
30. ExternalPush(pos) =  case {
31.        pos == 0 :  pushup.pos{extUpReq[pos] = 1} → Skip
32.        pos == NoOfFloor − 1  :  pushdown.pos{extDownReq[pos] = 1} → Skip
33.        default  :  pushup.pos{extUpReq[pos] = 1} → Skip □
34.                      pushdown.pos{extDownReq[pos] = 1} → Skip
35.        };
36. Waiting(pos) =□ i : {0..NoOfLift − 1}@([doorOpen[i] == pos]enter.i →
37.         □ x : {0..NoOfFloor − 1}@(push.x{intRequests[i][x] = 1} →
38.        [doorOpen[i] == x]exit.i.x → User()));
39. LiftSystem() = Users() ||| (||| x : {0..NoOfLift − 1}@Lift(x, 0, 1));
40. #assert LiftSystem() deadlockfree;
41. #define pr1 extUpReq[0] > 0;
42. #define pr2 extUpReq[0] == 0;
43. #assert LiftSystem() ⊨ □(pr1 ⇒ ◇pr2) ∧ □◇moving.0.1
```

Figure 3.5: CSP# model of the lifts system

in order to find counterexamples. In particular, line 40 asserts that the lift system is deadlock-free. Line 43 states a LTL property (refer to Section 2.3.2) asserting that (1) a request at the ground floor (defined as the proposition at line 41) must eventually be served (defined as the proposition at line 42), and (2) the event $moving.0.1$ must always eventually occur (i.e., 0-th lift must always eventually move upwards).

3.2 Real-time System Modeling

Specification and verification of real-time systems are important research topics which have practical implications. During the last decade or so, a popular approach for specifying real-time systems is based on the notation Timed Automata [10, 149]. Timed Automata are powerful in designing real-time models with explicit clock variables. Real-time constraints are captured by explicitly setting/resetting clock variables. A number of automatic verification support for Timed Automata have proven to be successful [135, 35, 35, 217, 207, 33].

Models based on Timed Automata often adapt a simple structure, e.g. a network of Timed Automata with no hierarchy [135]. The benefit is that efficient model checking is made feasible. Nonetheless, designing and verifying hierarchical real-time systems is becoming an increasingly difficult task due to the widespread applications and increasing complexity of such systems. High-level requirements for real-time systems are often stated in terms of *deadline*, *time out*, and *timed interrupt* [130, 74, 141]. In industrial case studies of real-time system verification, system requirements are often structured into phases, which are then composed sequentially, in parallel, alternatively, etc [105, 134]. Unlike Statechart (with clocks) or timed process algebras, Timed Automata lack of high-level compositional patterns for hierarchical design. As a result, users often need to manually cast those terms into a set of clock variables with carefully calculated clock constraints. The process is tedious and error-prone.

In remaining sections, we investigate an alternative approach for modeling hierarchical real-time systems by extending CSP# with additional behavioral patterns which are useful in modeling and analyzing real-time systems. Examples are $deadline$ (which constrains a process to terminate within some time units), $timed\ interrupt$, etc. Instead of explicitly manipulating clock variables (as in Timed Automata), the time related process constructs are designed to build on implicit clocks.

3.2.1 Syntax

In this section, we introduce the language extensions of CSP# for modeling real-time systems.

Definition 6 (Timed process) *A timed process is defined by the following grammar.*

$$\begin{array}{lll}
P = *CSP\#\ Process\ Constructs* & & \\
\quad \mid\ Wait[d] & & \textit{– delay} \\
\quad \mid\ P\ timeout[d]\ Q & & \textit{– timeout} \\
\quad \mid\ P\ interrupt[d]\ Q & & \textit{– timed interrupt} \\
\quad \mid\ P\ deadline[d] & & \textit{– deadline}
\end{array}$$

where P and Q range over processes, and d is an integer constant.

Based on CSP#, a number of timed process constructs can be used to capture common real-time system behavior patterns. Without loss of generality, we assume d is an integer constant. Process

$Wait[d]$ idles for exactly d time units. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse (since the process starts). Otherwise, Q takes over control after exactly d time units elapse. Process $P\ interrupt[d]\ Q$ behaves exactly as P (which may engage in multiple observable events) until d time units elapse, and then Q takes over control. Process $P\ deadline[d]$ constrains P to terminate before d time units. In this setting, clock variables are made implicit and hence they cannot be compared with each other directly, which potentially allows efficient clock manipulation and hence system verification.

In this thesis, we adopt the dense-time semantic model, in which clock values are isomorphic to a dense series of rational numbers, meaning that there is always a rational number between any two rational numbers. This choice preserves the advantage of dense-time model over discrete-time model (see Section 2.3.2), but still allows us to perform model checking by using some abstraction technique (see Chapter 9). We know that a set of rational numbers can be converted an 'equivalent' set of integer numbers by multiplying their least common multiple. This fact allows us to only consider integer values in the modeling language presented above[12].

3.2.2 Semantics

Similar to the operational semantics of CSP#, we present the firing rules for the timed process constructs. Recall that, a transition of the system is of the form $c \xrightarrow{x} c'$ where $x \in \Sigma \cup \{\tau, \checkmark\}$, and c and c' are the system configurations before and after the transition respectively. In real-time modeling, we introduce the timed transition label. Let t denotes a non-negative integer number, $c \xrightarrow{t} c'$ denotes a transition of t time units elapsing. In the following, we present the firing rules which are associated with the timed process constructs, adopting the approach in [181].

$$\frac{t \leq d}{(V, Wait[d]) \xrightarrow{t} (V, Wait[d-t])}\ [\ de1\ ] \qquad\qquad \frac{}{(V, Wait[0]) \xrightarrow{\tau} (V, Skip)}\ [\ de2\ ]$$

The above captures behaviors of process $Wait[d]$. Rule $de1$ states that the process may idle for any amount of time as long as it is less than or equal to d time units; Rule $de2$ states that the process

[12] Using integer numbers is for the simplification of modeling, but not an approximation.

terminates immediately after d becomes 0.

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P\ \mathit{timeout}[d]\ Q) \xrightarrow{x} (V', P')}\ [\ to1\ ]$$

$$\frac{(V, P) \xrightarrow{\tau} (V', P')}{(V, P\ \mathit{timeout}[d]\ Q) \xrightarrow{\tau} (V', P'\ \mathit{timeout}[d]\ Q)}\ [\ to2\ ]$$

$$\frac{(V, P) \xrightarrow{t} (V, P'), t \leq d}{(V, P\ \mathit{timeout}[d]\ Q) \xrightarrow{t} (V, P'\ \mathit{timeout}[d-t]\ Q)}\ [\ to3\ ]$$

$$\frac{}{(V, P\ \mathit{timeout}[0]\ Q) \xrightarrow{\tau} (V, Q)}\ [\ to4\ ]$$

If an observable event x can be engaged by P, then $P\ \mathit{timeout}[d]\ Q$ becomes P' (rule $to1$). An invisible transition does not solve the choice (rule $to2$). If P may idle for less than or equal to d time units, so is the composition (rule $to3$). When d becomes 0, Q takes over control by a silent transition (rule $to4$).

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P\ \mathit{interrupt}[d]\ Q) \xrightarrow{x} (V', P'\ \mathit{interrupt}[d]\ Q)}\ [\ it1\ ]$$

$$\frac{(V, P) \xrightarrow{t} (V, P'), t \leq d}{(V, P\ \mathit{interrupt}[d]\ Q) \xrightarrow{t} (V, P'\ \mathit{interrupt}[d-t]\ Q)}\ [\ it2\ ]$$

$$\frac{}{(V, P\ \mathit{interrupt}[0]\ Q) \xrightarrow{\tau} (V', Q)}\ [\ it3\ ]$$

Rule $it1$ states that if P engages in event x, $P\ \mathit{interrupt}[d]\ Q$ becomes $P'\ \mathit{interrupt}[d]\ Q$. Rule $it2$ states that if P may idle for less than or equal to d time units, so is the composition. When d time units elapse, Q takes over by a τ-transition.

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P\ \mathit{deadline}[d]) \xrightarrow{x} (V', P'\ \mathit{deadline}[d])}\ [\ dl1\ ]$$

$$\frac{(V, P) \xrightarrow{t} (V, P'), t \leq d}{(V, P\ \mathit{deadline}[d]) \xrightarrow{t} (V, P'\ \mathit{deadline}[d-t])}\ [\ dl2\ ]$$

Intuitively, $P\ \mathit{deadline}[d]$ behaves exactly as P except that it must terminate before d time units.

3.2.3 Case Study: Fischer's Algorithm

Fischer's algorithm [87] is a timed mutual exclusion algorithm. It allows n timed processes (identical up to renaming of process identifiers) to access a shared resource in mutual exclusion. The following models Fischer's algorithm of three processes.

```
**var** $x$         $= -1;$
**var** $ct$        $= 0;$
$Proc(i)$     $= [x == -1] Active(i);$
$Active(i)$   $= (update.i\{x = i\} \rightarrow Skip)\ deadline\ [\delta];\ \ Wait[\epsilon];$
                **if** $(x == i)$ $\{$
                    $cs.i\{ct = ct + 1\} \rightarrow exit.i\{ct = ct - 1;\ x = -1\} \rightarrow Proc(i)$
                $\}$ **else** $\{$
                    $Proc(i)$
                $\};$
$Protocol$    $= Proc(0) \parallel Proc(1) \parallel Proc(2);$
```

where δ and ϵ are two integer constants with $\delta < \epsilon$; x and ct are global variables. The protocol is modeled as process $Protocol$, which is the parallel composition of three processes. Each of the three processes attempts to enter the critical section when x is -1, i.e. no other process is currently attempting. Once the process is active, it sets x to its identity i within δ time units (captured by $deadline[\delta]$). Then it idles for ϵ time units (captured by $Wait[\epsilon]$) and then checks whether x is still i. If so, it enters the critical section and leaves later. Otherwise, it restarts from the beginning.

3.3 Summary

In this chapter, we proposed a combination of high-level specification languages with low-level procedural codes and time patterns for analyzing concurrent and real-time systems. A multi-lift system and Fischer's algorithm are used to illustrate the language. We remark that this language has been applied to model and verify a variety of systems, ranging from recently proposed distributed algorithms, concurrent programming algorithms to real-world systems like the pacemaker system. Previously unknown bugs have been discovered. Furthermore, we formally defined the semantic models, which facilitate PAT to perform sound and complete system verification.

This chapter is related to research on integrated formal methods, in particular, works on integrating state-based specification and event-based specification [221, 86, 152, 142, 192, 190, 205, 184, 45]. Different from previous approaches, our modeling language is designed for automated system analysis. Therefore, it is fully operational and supported by PAT. CSP+B [45, 184] (a combination of CSP and the B language) approach is similar to ours, but our language is closer to imperative programming language and accepts external (C#) programs. Two other languages are designed for similar purposes, namely machine readable CSP (which we will refer to as CSP_M) supported by the refinement checker FDR [176] and *Promela* which is supported by the model checker SPIN [110]. Compared to CSP_M, CSP# supports additional language features like shared variables, asynchronous communication channels and event associated programs, which offers users great flexibility in modeling. Furthermore, we give an interpretation of state/event Linear Temporal Logic (see Section 2.3.2) in CSP# semantics framework, which allows temporal logic based model checking of CSP# models. Compared to *Promela*, CSP# supports more process constructs (e.g. parallel operator), i.e., *Promela* is based on a subset of CSP, whereas all CSP models are valid CSP# models. In particular, CSP# inherits[13] the classic trace, stable failures and failures/divergence semantics (formally defined in Chapter 6) from CSP [108] , and therefore, allows us to perform a variety of refinement checking.

The real-time modeling proposed in this chapter is related to hierarchical specification based on process algebras for real-time systems, which has been studied extensively, e.g. the algebra of timed processes ATP [187, 159], CCS + real time [223] and Timed CSP [174, 182]. A remotely related modeling language is Statecharts with clocks [117], which too is compositional. This work follows the approach of Timed CSP and significantly extends the notion to cover a wide range of application domains.

[13] Since variables can be modeled as a process parallel to the one that uses them, then one CSP# model can be converted to a CSP model directly. Hence the trace, stable failures and failures/divergence semantics is inherited from CSP.

Chapter 4

Model Checking Fairness Enhanced Systems

In the area of software system verification, fairness, which is concerned with a fair resolution of non-determinism, is often necessary and important to prove liveness properties (see Section 2.3.2). Fairness is an abstraction of the fair scheduler in a multi-threaded programming environment or the relative speed of the processors in distributed systems. Without fairness, verification of liveness properties often produces unrealistic loops during which one process or event is infinitely ignored by the scheduler or one processor is infinitely faster than others. It is important to rule out those counterexamples and utilize the computational resource to identify the real bugs. However, systematically ruling out counterexamples due to lack of fairness is highly non-trivial. It requires flexible specification of fairness as well as efficient verification under fairness.

In this chapter, we focus on formal system analysis under fairness assumptions. The objective is to deliver a framework which model checks Linear Temporal Logic (LTL) properties (see Section 2.3.2) against concurrent systems functioning under a variety of fairness assumptions.

The remainder of the chapter is organized as follows. The next section gives the background information about model checking with fairness. Section 4.2 gives the formal definitions of a family of fairness notions. Section 4.3 develops necessary theories for model checking. Section 4.4 presents a sequential algorithm for verification under fairness. Section 4.5 introduces an alternative way for specifying and verifying event-based systems with fairness. Section 4.6 proposes a parallel version of the fairness model checking algorithm in the multi-core architecture with share-memory. Section 4.7 gives the experiment results of the proposed algorithms. Section 4.8 discusses related works and summarizes the chapter.

4.1 Background

Fairness and model checking with fairness have attracted much theoretical interests for decades [100, 131, 127, 202]. Their practical implications in system/software design and verification have been discussed extensively. Recent development on distributed systems showed that there are a family of fairness notions, including a newly formulated fairness notion named strong global fairness [88], which are crucial for designing self-stabilizing distributed algorithms [14, 16, 88, 47]. Because the algorithms are designed to function under fairness, model checking of (implementations of) the algorithms thus must be carried out under the respective fairness constraints.

Existing model checkers are ineffective with respect to fairness (which is demonstrated by the experiments below and in Section 4.7). One way to apply existing model checkers for verification under fairness constraints is to re-formulate the property so that fairness constraints become premises of the property. A liveness property ϕ is thus verified by showing the truth value of the following formula.

$fairness\ assumptions \Rightarrow \phi$ – F1

This practice is, though flexible, deficient for two reasons. Firstly, model checking is PSPACE-complete in the size of the formula. In particular, automata-based model checking relies on constructing a Büchi automaton from the LTL formula. The size of the Büchi automaton is exponential to the size of the formulas. Thus, it is infeasible to handle large formulas, whereas a typical system may have multiple fairness constraints. For example, SPIN is a popular LTL model checker [111]. The algorithm it uses for generating Büchi automata handles only a limited number of fairness

Property	**n**	**Time (Sec.)**	**Memory**	**#Büchi States**
$(\bigwedge_{i=1}^{n} \Box\Diamond p_i) \Rightarrow \Box\Diamond q$	1	0.08	466Kb	74
same above	3	4.44	27MB	1052
same above	5	$>$ 3600	$>$ 1Gb	–
$(\bigwedge_{i=1}^{n}(\Box\Diamond p_i \Rightarrow \Box\Diamond q_i)) \Rightarrow \Box\Diamond s$	1	0.13	487Kb	134
same above	2	1.58	10Mb	1238
same above	4	4689.24	$>$ 1Gb	–

Table 4.1: Experiments on LTL to Büchi automata conversion

constraints. Table 4.1 shows experiments on the time and space needed for SPIN to generate the automaton from the standard notions of fairness (see Section 4.2). n is the number of fairness constraints.

The experiments are made on a 3.0GHz Pentium IV CPU and 1 GB memory executing SPIN 4.3, where "$-$" means infeasible. The results show that it takes a non-trivial amount of time to handle 5 fairness constraints. Secondly, partial order reduction which is one of important reduction techniques for model checking distributed systems becomes ineffective. Partial order reduction ignores/postpones invisible events, whereas given F1 all events/propositions presented in *fairness constraints* are visible and therefore cannot be ignored or postponed.

In [161], Pang *et al* applied the SPIN model checker to establish the correctness of a family of population protocols. Only protocols relying on a notion of weak fairness operating on very small networks were verified because of the problems discussed above. Protocols relying on a notion of stronger fairness (e.g., strong fairness or strong global fairness) are beyond the capability of SPIN even for the smallest network[1] (e.g., a network with 3 nodes). It is important to develop an effective approach and toolkit which can handle larger networks because real counterexamples may only be present in larger networks, as shown in Section 4.7.

An alternative method is to design specialized verification algorithms which take fairness into

[1] A network consists of multiple mobile nodes which interact with each other to carry out a computation. Each node can be seen as a process. Refer to Section 5.1 for more details.

account while performing model checking. The focus of existing model checkers has been on process-level weak fairness, which, informally speaking, states that every process shall make infinite progress if always possible (refer to detailed explanation in Section 4.2). For instance, SPIN has implemented a model checking algorithm which handles this kind of fairness. The idea is to copy the global reachability graph $K + 2$ times (for K processes) so as to give each process a fair chance to progress. Process-level strong fairness is not supported because of its complexity. It has been shown that process-level fairness may not be sufficient, e.g., for population protocols.

In this chapter, we present a unified on-the-fly model checking algorithm which handles a variety of fairness including process-level weak/strong fairness, event-level weak/strong fairness, strong global fairness, etc. The algorithm extends previous work on model checking based on finding strongly connected components (SCC). The detailed approach is explained in the rest of the chapter.

4.2 Fairness Definitions

In this section, we present the formal definitions of a variety of fairness assumptions. The modeling language is CSP# (see Section 3.1), which is interpreted as Labeled Transition Systems (LTS).

Recall that, given a LTS $\mathcal{L} = (S, init, \rightarrow)$, $s \xrightarrow{e} s'$ denotes that (s, e, s') is a transition in $\rightarrow$, e is the engaged event of transition $s \xrightarrow{e} s'$, and $enabled(s)$ is the set of enabled events at s (refer to Section 3.1.2). To distinguish with enabled process defined below, we introduce the equivalent notation $enabledEvt(s)$ for $enabled(s)$. If the system is constituted by multiple processes running in parallel, we write $enabledPro(s)$ to be the set of enabled processes, which may make a move given the system state s. Given a transition $s \xrightarrow{e} s'$, we write $engagedEvt(s, e, s')$ to denote $\{e\}$, and $engagedPro(s, e, s')$ to be the set of the participating processes, which have made some progress during the transition. $engagedPro(s, e, s')$ is not empty only when the state s is an interleave or parallel composition[2]. For example, $engagedPro(P_1 \parallel P_2 \parallel P_3, e, P_1 \parallel P_2' \parallel P_3) = \{P_2\}$. Notice that if e is synchronized by multiple processes, the set contains all the participating processes.

[2] Process level fairness is not meaningful if the system has no currency.

Because our targets are nonterminating concurrent systems, and fairness affects infinite not finite system behaviors, we focus on infinite system executions in the following. Finite behaviors are extended to infinite ones by appending infinite idling events at the rear. Without fairness constraints, a system may behave freely as long as it starts with an initial state and conforms to the transition relation. A fairness constraint restricts the set of system behaviors to only those fair ones. Given a LTL property ϕ, verification under fairness is to verify whether all fair executions of the system satisfy ϕ. In the following, we review a variety of fairness assumptions and illustrate their differences using examples.

Definition 7 (Event-level weak fairness) *Let $E = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ be an execution. E satisfies event-level weak fairness, if and only if for every event e, if e eventually becomes enabled forever in E, then $e_i = e$ for infinitely many i, i.e., $\Diamond\Box\ e\ is\ enabled \Rightarrow \Box\Diamond\ e\ is\ engaged$.*

Event-level weak fairness (EWF) [131] states that if an event becomes enabled forever after some steps, then it must be engaged infinitely often. An equivalent formulation is that every computation should contain infinitely many positions at which event e is disabled or has just been engaged. The latter is known as justice condition [138]. Intuitively, it means that an enabled event shall not be ignored infinitely. Or equivalently some state must be visited infinitely often (e.g., accepting states in Büchi automata).

Definition 8 (Process-level weak fairness) *Let $E = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ be an execution. E satisfies process-level weak fairness, if and only if for every process p, if p eventually becomes enabled forever in E, then $p \in engagedProc(s_i, e_i, s_{i+1})$ for infinitely many i, i.e., $\Diamond\Box\ p\ is\ enabled \Rightarrow \Box\Diamond\ p\ is\ engaged$.*

Process-level weak fairness (PWF) states that if a process becomes enabled forever after some steps, then it must be engaged infinitely often. From another point of view, PWF guarantees that each process is only finitely faster than the others; otherwise there will be some always enabled process with no progress, which violates the fairness assumption.

Weak fairness (or justice condition) has been well studied and verification under weak fairness has

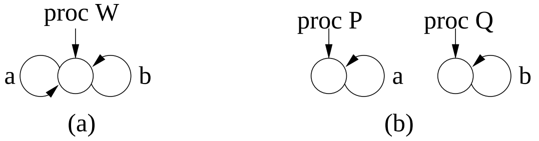


Figure 4.1: Event-level weak fairness vs. process-level weak fairness

been supported to some extent, e.g., PWF is supported by the SPIN model checker [111]. Given the LTS in Figure 4.1(a), the property $\Box\Diamond\ a$ is true under EWF. Event a is always enabled and, hence, by definition it must be infinitely often engaged. The property is, however, false under no fairness or PWF. The reason that it is false under PWF is that the process W may make progress infinitely (by repeatedly engaging in b) without ever engaging in event a. Alternatively, if the system is modeled using two processes as shown in Figure 4.1(b), $\Box\Diamond\ a$ becomes true under PWF (or EWF) because both processes must make infinite progress and therefore both a and b must be engaged infinitely. This example suggests that, different from PWF, EWF is not related to the system structure. In general, process-level fairness may be viewed a special case of event-level fairness. By a simple argument, it can be shown that PWF can be achieved by labeling all events in a process with the same name and applying EWF.

Definition 9 (Event-level strong fairness) *Let $E = \langle s_0, e_0, s_1, e_1, \cdots\rangle$ be an execution. E satisfies event-level strong fairness if and only if, for every event e, if e is infinitely often enabled, then $e = e_i$ for infinitely many i, i.e., $\Box\Diamond\ e\ is\ enabled \Rightarrow \Box\Diamond\ e\ is\ engaged$.*

Event-level strong fairness (ESF) has been identified by different researchers. It is named *strong fairness* in [132] (by contrast to weak fairness defined above). In [88], it is named strong local fairness (in comparison to strong global fairness defined below). It is also known as *compassion* condition [166]. ESF states that if an event is infinitely often enabled, it must be infinitely often engaged. It is particularly useful in the analysis of systems using semaphores, synchronous communication, and other special coordination primitives. Given the LTS in Figure 4.2(a), the property $\Box\Diamond\ b$ is false under EWF but true under ESF. The reason is that b is not always enabled (i.e., it is disabled at the left state) and thus the system is allowed to always take the c branch under EWF. It

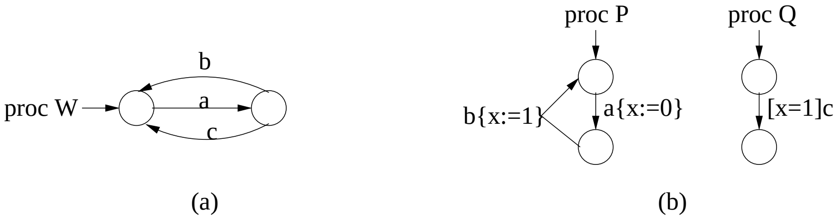


Figure 4.2: Event-level strong fairness and process-level strong fairness

is infinitely often enabled, and thus, the system must engage in b infinitely under ESF.

Definition 10 (Process-level strong fairness) *Let $E = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ be an execution. E satisfies process-level strong fairness if and only if, for every process p, if p is infinitely often enabled, then $p \in engagedProc(s_i, e_i, s_{i+1})$ for infinitely many i, i.e. $\Box\Diamond$ p is enabled $\Rightarrow$ $\Box\Diamond$ p is engaged.*

The process-level correspondence is *process-level strong fairness (PSF)*. Intuitively, PSF means that if a process is repeatedly enabled, it must eventually make some progress. Given the LTS in Figure 4.2(b), the property $\Box\Diamond$ c is false under PWF but true under PSF. The reason is that event c is guarded by condition $x = 1$ and therefore is not repeatedly enabled.

Verification under (event-level/process-level) strong fairness (or compassion condition) has been discussed previously, e.g., in the setting of Streett automata [99, 106], fair discrete systems [124] or programming codes [158]. Nonetheless, there are few established tool support for formal verification under strong fairness [100] to the best of our knowledge. The main reason is the computational complexity. For instance, it is claimed too expensive to support in SPIN [111]. We, however, show that reasonably efficient model checking under strong fairness can be achieved (refer to experiment results in Section 4.7).

Definition 11 (Strong global fairness) *Let $E = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ be an execution. E satisfies strong global fairness if and only if, for every s, e, s' such that $s \xrightarrow{e} s'$, if $s = s_i$ for infinite many i, $s_i = s$ and $e_i = e$ and $s_{i+1} = s'$ for infinitely many i. i.e. $\Box\Diamond$ (s, e, s') is enabled $\Rightarrow$ $\Box\Diamond$ (s, e, s') is engaged.*

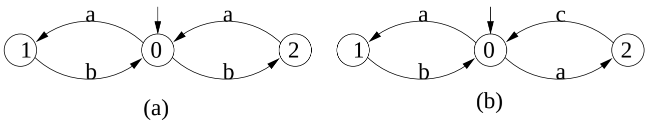


Figure 4.3: Strong global fairness

Strong global fairness (SGF) was suggested by Fischer and Jiang in [88]. It states that if a *step* (from s to s' by engaging in event e) is enabled infinitely often, then it must actually be taken infinitely often[3]. Different from the previous notions of fairness, SGF concerns about both events and states, instead of events only. It can be shown by a simple argument that SGF is stronger than ESF. Because it concerns about both events and states, it is 'event-level' and 'process-level'. Strong global fairness requires that an infinitely enabled event must be taken infinitely often in *all* contexts, whereas ESF only requires the enabled event to be taken in *one* context. Figure 4.3 illustrates the difference with two examples. Under ESF, state 2 in Figure 4.3(a) may never be visited because all events are engaged infinitely often if the left loop is taken infinitely. As a result, property $\Box\Diamond\ state$ 2 is false. Under SGF, all states in Figure 4.3(a) must be visited infinitely often and therefore $\Box\Diamond\ state$ 2 is true. Figure 4.3(b) illustrates their difference when there are non-determinism. Both transitions labeled a must be taken infinitely under SGF, which is not necessary under ESF or EWF. Thus, property $\Box\Diamond\ b$ is true only under SGF. Many population protocols rely on SGF, e.g., protocols presented in [14, 88]. As far as the authors know, there are no previous works on model checking under SGF.

A number of other fairness notions have been discussed by various researchers, e.g., unconditional event fairness [128] which will be discussed in Section 4.5, hyper-fairness which is of only theatrical interests as stated in [132] and event-level weak local/global fairness in [88]. We skip their definitions and remark that our approach can be extended to handle other kinds of fairness.

[3] The definition in [88] is for unlabeled transition systems. We slightly changed it so as to suit the setting of LTS. Nonetheless, both capture the same intuition.

4.3 Model Checking under Fairness as Loop/SCC Searching

Given a LTS $\mathcal{L}$ and a LTL formula ϕ, model checking is about searching for an execution of $\mathcal{L}$ which fails ϕ. In automata-based model checking, the negation of ϕ is translated to an equivalent Büchi automaton $\mathcal{B}^{\neg\phi}$, which is then composed with the LTS representing the system model. We omit the detailed algorithms for translating LTL formula to Büchi automaton. Interested readers can refer to [97]. To be able to explain our algorithm, first we formally define Büchi automaton as follows.

Definition 12 *Büchi automaton A Büchi automaton is a tuple $\mathcal{B} = (\Sigma, B, \rho, b_0, F)$, where Σ is an alphabet, B is a set of Büchi states, $\rho : B \times \Sigma$ is a nondeterministic transition function, $b_0 \in B$ is an initial state, and $F \subseteq B$ is a set of accepting states.*

A run of $\mathcal{B}$ over an infinite word $w = a_1 a_2 \ldots$ is an infinite sequence $\langle b_0, b_1 \ldots \rangle$, where b_0 is initial state and $b_i \in \rho(b_{i-1}, a_i)$ for all $i \geq 1$. A run $\langle b_0, b_1 \ldots \rangle$ is accepting to $\mathcal{B}$ if there is some accepting state in $\mathcal{B}$ that repeats infinitely often.

Model checking under fairness is to search for an infinite execution which is accepting to the Büchi automaton and at the same time satisfies the fairness constraints. In the following, we write $\mathcal{L} \vDash \phi$ to mean that the LTS satisfies the property under no fairness, i.e., every execution of $\mathcal{L}$ satisfies ϕ. We write $\mathcal{L} \vDash_{EWF} \phi$ ($\mathcal{L} \vDash_{PWF} \phi$) to mean that $\mathcal{L}$ satisfies ϕ under event-level (process-level) weak fairness; $\mathcal{L} \vDash_{ESF} \phi$ ($\mathcal{L} \vDash_{PSF} \phi$) to mean that $\mathcal{L}$ satisfies ϕ under event-level (process-level) strong fairness, and $\mathcal{L} \vDash_{SGF} \phi$ to mean that $\mathcal{L}$ satisfies ϕ under strong global fairness.

Without loss of generality, we assume that $\mathcal{L}$ contains only finite states. By a simple argument, it can be shown that the system contains an infinite execution if and only if there exists a loop. An execution of the product of $\mathcal{L}$ and $\mathcal{B}^{\neg\phi}$ is a sequence of alternating states/events

$$R_i^j = \langle (s_0, b_0), e_0, \cdots, (s_i, b_i), e_i, \cdots, (s_j, b_j), e_j, (s_{j+1}, b_{j+1}) \rangle$$

where s_i is a state of $\mathcal{L}$, b_i is a state of $\mathcal{B}^{\neg\phi}$, $s_i = s_{j+1}$ and $b_i = b_{j+1}$. We skip the details on constructing the product and refer the readers to [126]. R_i^j is accepting if and only if the sequence $\langle b_0, b_1, \cdots, b_k, \cdots \rangle$ is accepting to $\mathcal{B}^{\neg\phi}$, i.e., the sequence visits at least one accepting state of $\mathcal{B}^{\neg\phi}$ infinitely often. R_i^j is fair under certain notion of fairness if and only if the sequence $\langle s_0, e_0, s_1, e_1, \cdots, s_k, e_k, \cdots \rangle$ is. Furthermore, we define the following sets.

$$\begin{array}{ll}
\mathit{alwaysEvt}(R_i^j) & = \{e \mid \forall\, k : \{i, \cdots, j\},\ e \in \mathit{enabled}(s_k)\} \\
\mathit{alwaysPro}(R_i^j) & = \{p \mid \forall\, k : \{i, \cdots, j\},\ p \in \mathit{enabledPro}(s_k)\} \\
\mathit{onceEvt}(R_i^j) & = \{e \mid \exists\, k : \{i, \cdots, j\},\ e \in \mathit{enabled}(s_k)\} \\
\mathit{oncePro}(R_i^j) & = \{p \mid \exists\, k : \{i, \cdots, j\},\ p \in \mathit{enabledPro}(s_k)\} \\
\mathit{onceStep}(R_i^j) & = \{(s, e, s') \mid \exists\, k\{i, \cdots, j\},\ s = s_k \wedge (s, e, s') \in \rightarrow)\} \\
\mathit{engagedEvt}(R_i^j) & = \{e \mid \exists\, k : \{i, \cdots, j\},\ e = e_k\} \\
\mathit{engagedPro}(R_i^j) & = \{p \mid \exists\, k : \{i, \cdots, j\},\ p \in \mathit{engagedPro}(s_k, e_k, s_{k+1})\} \\
\mathit{engagedStep}(R_i^j) & = \{(s, e, s') \mid \exists\, k\{i, \cdots, j-1\},\ s = s_k \wedge e = e_k \wedge s' = s_{k+1})\}
\end{array}$$

Intuitively, set $\mathit{alwaysEvt}(R_i^j)/\mathit{alwaysPro}(R_i^j)$ is the set of events/processes which are always enabled during the loop. Set $\mathit{onceEvt}(R_i^j)/\mathit{oncePro}(R_i^j)/\mathit{onceStep}(R_i^j)$ is the set of events/processes/steps which are enabled at least once during the loop. Set $\mathit{engagedEvt}(R_i^j)/\mathit{engagedPro}(R_i^j)/\mathit{engagedStep}(R_i^j)$ is the set of events/processes/step which are engaged during the loop.

Lemma 4.3.1 *Let $\mathcal{L} = (S, init, \rightarrow)$ be a LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of a LTL formula ϕ. Let R_i^j be an arbitrary loop in the product of $\mathcal{L}$ and $\mathcal{B}$.*

- *$\mathcal{L} \vDash_{EWF} \phi$ if and only if there does not exist R_i^j such that $\mathit{alwaysEvt}(R_i^j) \subseteq \mathit{engagedEvt}(R_i^j)$ and R_i^j is accepting.*

- *$\mathcal{L} \vDash_{PWF} \phi$ if and only if there does not exist R_i^j such that $\mathit{alwaysPro}(R_i^j) \subseteq \mathit{engagedPro}(R_i^j)$ and R_i^j is accepting.*

- *$\mathcal{L} \vDash_{ESF} \phi$ if and only if there does not exist R_i^j such that $\mathit{onceEvt}(R_i^j) \subseteq \mathit{engagedEvt}(R_i^j)$ and R_i^j is accepting.*

- *$\mathcal{L} \vDash_{PSF} \phi$ if and only if there does not exist R_i^j such that $\mathit{oncePro}(R_i^j) \subseteq \mathit{engagedPro}(R_i^j)$ and R_i^j is accepting.*

- *$\mathcal{L} \vDash_{SGF} \phi$ if and only if there does not exist R_i^j such that $\mathit{onceStep}(R_i^j) \subseteq \mathit{engagedStep}(R_i^j)$ and R_i^j is accepting.*

The lemma can be proved straightforwardly by contradiction. By the lemma, a system fails the property under certain fairness if and only if there exists a loop which satisfies the fairness but fails the property. Modeling checking under fairness is hence reduced to loop searching.

In the following we introduce the basic definitions in graph theory to ease the discussion later.

Definition 13 (Directed Graph) *A directed graph or digraph is a pair G = (V,E), where set V contains the vertices and set E contains ordered pairs of vertices (i.e. directed edges).*

Definition 14 (Strongly Connected Component (SCC)) *A strongly connected component of a directed graph G is a maximal set of vertices $C \subset V$ such that for every pair of vertices u and v, there is a directed path from u to v and a directed path from v to u.*

Note that each graph can have more than one SCCs. A SCC is terminal if and only if any transition starting from a vertex in the SCC must end with a vertex in the SCC. If a directed graph is the transition system of the product of $\mathcal{L}$ and $\mathcal{B}$, the vertices are of form (s, b), where s is a state in $\mathcal{L}$ and b is a state in $\mathcal{B}$. We say a SCC S of such graph is accepting if and only if there exists one vertex (s, b) in S such that b is an accepting state of $\mathcal{B}$. In an abuse of notations, we refer to S as the strongly connected subgraph in the following context. To further abuse notations, we write $alwaysEvt(S)$ ($alwaysPro(S)$, $onceEvt(S)$, $oncePro(S)$, $onceStep(R_i^j)$, $engagedEvt(S)$, $engagedPro(S)$ or $engagedStep(R_i^j)$) to denote the set obtained by applying the function to a loop which traverses all states of S.

Lemma 4.3.2 *Let $\mathcal{L}$ be a LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of a LTL formula ϕ; S be a strongly connected subgraph in the product of $\mathcal{L}$ and $\mathcal{B}$.*

- *$\mathcal{L} \vDash_{ESF} \phi$ if and only if there does not exist S such that S is accepting and $onceEvt(S) \subseteq engagedEvt(S)$.*
- *$\mathcal{L} \vDash_{PSF} \phi$ if and only if there does not exist S such that S is accepting and $oncePro(S) \subseteq engagedPro(S)$.*

The above lemma can be proved by a simple argument. It shows that model checking under fairness can be reduced to strongly connected subgraph searching.

Lemma 4.3.3 *Let $\mathcal{L}$ be a LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of a LTL formula ϕ. Let S be a SCC in the product of $\mathcal{L}$ and $\mathcal{B}$.*

- *$\mathcal{L} \models_{EWF} \phi$ if and only if there does not exist S such that S is accepting and $alwaysEvt(S) \subseteq engagedEvt(S)$.*
- *$\mathcal{L} \models_{PWF} \phi$ if and only if there does not exist S such that S is accepting and $alwaysPro(S) \subseteq engagedPro(S)$.*

Proof. We prove the event-level weak fairness part of the lemma and remark that the other part can be proved similarly. It can be shown that a system fails ϕ under event-level weak fairness if and only if there exists one strongly connected subgraph C such that C is accepting and $alwaysEvt(C) \subseteq engagedEvt(C)$. Hence, it is sufficient to show that there exists such a C if and only if there exists a SCC S such that S is accepting and $alwaysEvt(S) \subseteq engagedEvt(S)$.
if: If there exists such a SCC S, then we simply let C be S.
only if: If there exists such subgraph C, the SCC S which contains C is accepting and satisfies $alwaysEvt(S) \subseteq engagedEvt(S)$ since $alwaysEvt(S) \subseteq alwaysEvt(C)$ and $engagedEvt(C) \subseteq engagedEvt(S)$. This concludes the proof. □

The lemma shows that model checking under weak fairness can be reduced to SCC searching. The following lemma reduces model checking under strong global fairness to searching for a terminal SCC in $\mathcal{L}$[4].

Lemma 4.3.4 *Let $\mathcal{L}$ be a LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of a LTL formula ϕ. $\mathcal{L} \models_{SGF} \phi$ if and only if there does not exist a SCC S such that S is accepting and $onceStep(R_i^j) \subseteq engagedStep(R_i^j)$.*

[4] A terminal SCC in the product of $\mathcal{L}$ and $\mathcal{B}$ may not be constituted by a terminal SCC in $\mathcal{L}$.

Proof. By lemma 4.3.1, $\mathcal{L}$ fails ϕ under strong global fairness if and only if there exists R_i^j in the product of $\mathcal{L}$ and $\mathcal{B}$ such that R_i^j satisfies strong global fairness and R_i^j is accepting. $\mathcal{L}$ fails ϕ under strong global fairness if and only if there exists a strongly connected subgraph C such that C satisfies strong global fairness and C is accepting. Hence, it is sufficient to show that there exists such a subgraph C if and only if there exists a SCC S such that S which satisfies the constraint.

if: This is proved trivially.

only if: Assume that there exists such a subgraph C. Let $x(C) = \{s \mid \exists\, b\ (s, b) \in C\}$ be the states of $\mathcal{L}$ which constitute C and $t(C) = \{(s, e, s') \mid s \in x(C) \wedge s' \in x(C) \wedge \exists (s, b), (s, b') : C\ (s, b) \xrightarrow{e} (s', b')\}$ be the transition of $\mathcal{L}$ which constitute the strongly connected subgraph. By contradiction, it can be shown that $x(C)$ (together with the transitions in $t(C)$) forms one terminal SCC in $\mathcal{L}$. Let S be the SCC containing C. It can be shown that $x(S)$ (together with the transitions in $t(S)$) forms the same terminal SCC. Therefore, S must satisfy the constraint. □

4.4 An Algorithm for Modeling Checking under Fairness

In the area of LTL model checking, the two best known enumerative sequential algorithms based on fair-cycle detection are the Nested Depth First Search (NDFS) algorithm [61, 109] and SCC-based algorithms [202, 197] based on Tarjan's algorithm for strongly connected components (SCCs) detection [206]. NDFS has been implemented in the model checker SPIN [111]. The basic idea is to perform one DFS first to reach a target state (i.e., an accepting state in the setting of Büchi automata) and then perform second DFS from that state to check whether it is reachable from itself. It has been shown the NDFS works efficiently for model checking under no fairness [111]. Nonetheless, it is not suitable for verification under fairness [111] because whether an execution is fair depends on the whole path instead of one state. In recent years, model checking based on SCC has been re-investigated and it has been shown that it yields comparable performance [99]. In this chapter, we extend the existing SCC-based model checking algorithms [99] to cope with different notions of fairness. The algorithm is inspired by early work on emptiness check of Streett automata [106].

Figure 4.4 presents the algorithm. It is based on Tarjan's algorithm for identifying SCCs (which is

```
procedure $mc(States, Transitions, Fair)$
while there are un-visited states
    let $scc := tarjan(States, Transitions)$;
    mark states in $scc$ as visited;
    if $isFair(scc)$ then                          – *
        generate a counterexample;                 – *
        return $false$;                            – *
    else                                           – *
        $scc = prune(scc, Fair)$;                  – *
        if $\neg mc(scc, Transitions)$ then        – *
            return $false$;                        – *
        endif                                      – *
    endif                                          – *
endwhile
return $true$;
```

Figure 4.4: Algorithm for sequential model checking under fairness

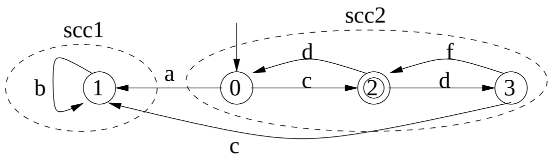


Figure 4.5: Model checking example

linear time in the number of graph edges [206]). It searches for fair strongly connected subgraph on-the-fly. The basic idea is to identify one SCC at a time and then check whether it is fair or not. If it is, the search is over. Otherwise, the SCC is partitioned into multiple smaller strongly connected subgraphs, which are then checked recursively one by one. Figure 4.5 presents a running example, i.e., the product of a LTS and a Büchi automaton. Notice that state 2 is an accepting state.

Assume that $States$ is the set of states and $Transitions$ is the set of transitions[5]. The method takes three inputs, i.e., $States$, $Transitions$ and a fairness type $Fair$ (of value either EWF, PWF, ESF, PSF or SGF). At the top level is a while-loop, which stops only if all states have been visited. At

[5] both of which may be constructed on-the-fly instead of known before-hand.

line 2, Tarjan's algorithm (an improved version) is used to identify a SCC [99]. If the found scc is fair, a fair loop which traverses all states/transitions in the SCC is generated as a counterexample (at line 5) and we conclude that the property is not true at line 6. We skip the details on generating the loop in this thesis and remark that it could be a non-trivial task (refer to [124]). Without fairness assumptions, a SCC is fair if and only if it is accepting to the Büchi automaton (i.e. Büchi fair). Given the LTS presented in Figure 4.5, $tarjan$ method identifies two SCCs, i.e., $scc1$ which is composed of state 1 only and $scc2$ which is composed of state 0, 2 and 3. The order in which SCCs are found is irrelevant to the correctness of the algorithm. If state 2 is explored before state 1, at line 3, scc_states is the set of states in $scc2$.

If scc is not fair, a procedure $prune$ (at line 8) is used to prune *bad states* from scc. Bad states are the reasons why the SCC is not fair. For instance, state 0 (where the event a is enabled) is a bad state in $scc2$ under event-based strong fairness because event a is never engaged in $scc2$ (i.e., $a \not\in engagedEvt(ssc2)$). State 3 is a bad state under strong global fairness because the step from state 3 to state 1 via c is not part of the SCC. The intuition behind the pruning is that there may be a fair strongly connected subgraph in the remaining states after eliminating the bad states. By simply modifying the $prune$ method, the algorithm can be used to handle different fairness. Refer to details in Section 4.4.1.

If some states have been pruned, a recursive call (line 9) is made to check whether there is a fair strongly connected subgraph within the remaining states. The call terminates in two ways. One is that a fair subgraph is found (at line 6) and the other is that all states in scc are pruned (at line 14). If the recursive call returns true, there is no fair subgraph and we continue with another SCC until all states are checked.

4.4.1 Coping with Different Notions of Fairness

In this section, we show how to customize the $prune$ function so as to handle different fairness. Let S be a strongly connected subgraph. The following defines the pruning methods for event-based weak fairness.

$$prune(S, EWF) = \begin{cases} S & \text{if } alwaysEvt(S) \subseteq engagedEvt(S); \\ \varnothing & \text{otherwise}. \end{cases}$$

If there exists an event e which is always enabled (i.e., $e \in alwaysEvt(S)$) but never engaged (i.e., $e \notin engagedEvt(S)$), by definition S does not satisfy event-level weak fairness. If a SCC does not satisfy event-level weak fairness, none of its subgraphs do, because e is always enabled in any of its subgraphs but never engaged. As a result, either all states are pruned or none of them is. Similarly, the following defines the pruning function for process-level weak fairness.

$$prune(S, PWF) = \begin{cases} S & \text{if } alwaysPro(S) \subseteq engagedPro(S); \\ \varnothing & \text{otherwise}. \end{cases}$$

In the case event-level (process-level) strong fairness, a state is pruned if and only if there is an event (process) enabled at this state but never engaged in the subgraph. By pruning the state, the event (process) may become never enabled and therefore not required to be engaged. The following defines the pruning function for event-level and process-level strong fairness.

$$prune(S, ESF) = \{s : S \mid enabledEvt(s) \subseteq engagedEvt(S)\}$$
$$prune(S, PSF) = \{s : S \mid enabledPro(s) \subseteq engagedPro(S)\}$$

By lemma 4.3.4, a SCC may constitute a counterexample to a property under strong global fairness if and only if the SCC satisfies strong global fairness and is accepting. Therefore, we prune all states if it fails strong global fairness.

$$prune(S, SGF) = \begin{cases} S & \text{if } onceStep(S) \subseteq engagedStep(S); \\ \varnothing & \text{otherwise}. \end{cases}$$

We remark that the time complexity of the $prune$ functions are all linear in the number of edges of the SCC.

Given the LTS in Figure 4.5, under event-level strong fairness, state 0 is pruned from $scc2$ because $enabledEvt(state\ 0) = \{a, c\} \not\subseteq engagedEvt(scc2)$. After that the only remaining strongly connected subgraph contains state 2 and 3, now state 3 where c is enabled is considered as a bad state because c is not engaged in the subgraph. State 2 is then pruned for being a trivial strongly connected subgraph which fails event-level strong fairness.

4.4.2 Complexity and Soundness

The time complexities for verification under no fairness, EWF or PWF or SGF are similar, i.e., all linear in the number of system transitions. All states in one SCC are discarded at once in all cases and, therefore, no recursive call is necessary. Furthermore, the $prune$ function is linear in the number of transitions of a SCC. SPIN's model checking algorithm under PWF increases the run-time expense of a verification run by a factor that is linear in the number of running processes. In comparison, our algorithm is less expensive for weak fairness. This is evidenced by the experiment results presented in Section 4.7. Verification under ESF or PSF is in general expensive. In the worst case (i.e., the whole system is strongly connected and only one state is pruned every time), the $tarjan$ method may be invoked at most $\#S$ times, where $\#S$ is the number of system states. Thus, the time complexity is bounded by $\#S \times \#T$ where $\#T$ is the number of transitions. In practice, however, if the property is false, a counterexample is usually identified quickly, because our algorithm constructs and checks SCCs on-the-fly. Even if the property is true, our experience suggests that the worst case scenario is rare in practice. Instead of performing detailed complexity analysis (refer to the discussion presented in [106]), we illustrate the performance of our algorithm using real systems in Section 4.7.

Next, we argue the total correctness of the algorithm. The algorithm is terminating because by assumption, the number of states is finite, and the number of visited states and pruned states are monotonically increasing.

Theorem 4.4.1 *Let $\mathcal{L}$ be a LTS. Let ϕ be a property. Let F be a fairness type (i.e., EWF, PWF, ESF, PSF or SGF). $\mathcal{L} \vDash_F \phi$ if and only if the algorithm returns true.*

Proof. **Case** EWF: By lemma 4.3.1, $\mathcal{L} \vDash_{EWF} \phi$ if and only if there does not exist a SCC S such that $alwaysEvt(S) \subseteq engagedEvt(S)$ and S is accepting. Given any SCC S, the algorithm returns false if and only if it does so at line 8 because the recursive call at line 9 always returns true (by the definition of $prune(S, T, EWF)$). Therefore, it returns false if and only if S is accepting (so that the condition at line 5 is true) and $alwaysEvt(S) \subseteq engagedEvt(S)$ (so that the condition at line 7 is true). If there does not exist such a SCC, the algorithm returns true. If the algorithm returns true, there must not be such a SCC. Therefore, $\mathcal{F} \vDash_{EWF} \phi$ if and only if the algorithm returns true.

Case PWF: Similarly as above.

Case ESF: By lemma 4.3.3, $\mathcal{L} \vDash_{ESF} \phi$ if and only if there does not exist a strongly connected subgraph C such that $onceEvt(C) \subseteq engagedEvt(C)$ and C is accepting. If C itself is a SCC, it must be found (by the correctness of Tarjan's algorithm and function $prune(S, T, ESF)$) and the algorithm returns false if C is accepting. If it is contained in one (and only one) SCC, by the correctness of $prune(S, T, ESF)$, its states are never pruned. As a result, it is identified when all other states in the SCC are pruned or a fair strongly connected subgraph containing all its states is identified. In either case, the algorithm returns false if and only if such a fair strongly connected subgraph is found. Equivalently, it returns true if and only if there are no such subgraphs. Therefore, $\mathcal{F} \vDash_{ESF} \phi$ if and only if the algorithm returns true.

Case PSF: Similarly as above.

Case SGF: By lemma 4.3.4, $\mathcal{L} \vDash_{SGF} \phi$ if and only if there does not exist a SCC S such that S satisfies strong global fairness is accepting. The algorithms returns false if and only if it is at line 8 because the recursive call at line 9 always returns true (by the definition of $prune(S, T, SGF)$). By definition of $prune(S, T, SGF)$, the control reaches line 8 if and only if the SCC is terminal and is accepting. Thus, $\mathcal{F} \vDash_{SGF} \phi$ if and only if the algorithm returns true. □

4.5 Event Annotated Fairness

In this section, we present an alternative (and more flexible) approach, which allows users to associate fairness to only part of the systems or associate different parts with different notions of fairness. The motivation is twofold.

Firstly, previous approaches treat every event or state equally, i.e., fairness is applied to every event/state. In practice, it may be that only certain events are meant to be fair. *For instance, when verifying open systems, fairness/liveness assumptions are often associated with environmental events as a way to capture assumptions on the environment.* In such case, if event-level or process-level fairness is applied, it is clearly overwhelming. Our remedy is to allow users to associate fairness assumptions with individual events by labeling events with fairness marks[6]. PAT supports a number of fairness annotations on events. We examine three of them in the following.

- Unconditional event fairness is written as $f(a)$. An execution of the system is fair if and only if a occurs infinitely often.
- Weak event fairness is written as $wf(a)$. An execution of the system is fair if and only if a occurs infinitely often if it is always enabled from some point on.
- Strong event fairness is written as $sf(a)$. An execution of the system is fair if and only if a occurs infinitely often if it is enabled infinitely often.

Unconditional event fairness [128] does not depend on whether the event is enabled or not, and therefore, is stronger than weak/strong event fairness. It may be used to annotate events which are known to be periodically engaged. For instance, the following process models a natural clock.

$$Clock() = tick \rightarrow Clock();$$

By annotating $tick$ with unconditional event fairness, we require that the clock must progress infinitely and the system (in which the clock and other components execute in parallel) disallows unrealistic *timelock*, i.e., execution of infinite events which takes finite time. Unconditional event fairness (like other event annotations) can be used to mechanically reduce the size of the property. For instance, given the property $\Box\Diamond\ a \Rightarrow \Box\Diamond\ b$. We may mechanically annotate a in the model with unconditional event fairness and verify $\Box\Diamond\ b$ instead. The semantics of weak (strong) event fairness is similar to event-level weak (strong) fairness defined in Section 4.2 except it is associated with individual events (by contrast to all events)[7]. Event annotated fairness may be viewed as the dual image of accepting states in automata theory, e.g., same as only selected states are marked accepting, only selected events are annotated.

[6] An alternative way is to identify a set of events separately from the model itself, which is in theory equivalent, but cumbersome in practice. It is especially true if an event may be constituted by process parameters or even global variables.

[7] Strong global fairness concerns with both events and states and hence no corresponding event annotation is defined.

The other motivation of event annotated fairness is that it makes partial order reduction (see Section 2.3.3) possible (to some extent) for model checking with strong fairness. The algorithm presented in Figure 4.4, i.e., a SCC-based explicit model checking algorithm, undoubtedly suffers from state space explosion, especially when the whole system is strongly connected. Partial order reduction is one of the important techniques to tackle the problem, which sometimes works surprisingly well for concurrent systems. For instance, assume that event a and b are independent and the property to verify is deadlock-freeness, it is sufficient to explore only one of the two outgoing transitions at state 0 in the LTS of Figure 4.3(a).

For classic model checking, a set of conditions which must be satisfied by the chosen subset of enabled events have been proposed to guarantee sound verification against 'X'-free LTL formulas. Efficient heuristic algorithms which calculate over-approximations of the subset were explored [58]. One such heuristic algorithm has been implemented in PAT. However, the conditions and algorithms may not work for verification under fairness. Following results proved in [39], it can be shown that partial order reduction is applicable to verification under EWF or PWF. However, though every execution which satisfies strong fairness in the full state graph has an equivalent execution (up to re-ordering of independent events) in the reduced state graph, it may not satisfy strong fairness and thus verification result over the reduced state graph may not be valid. Similarly, with strong global fairness the reduced state graph may not contain a fair loop even if the full state graph does. For instance, given the LTS in Figure 4.3(a) and assume that a and b are independent. The reduced graph may only contain state 0 and 1, which contains no loop which satisfies strong global fairness. In [163], it was suggested that by considering events dependent to each other if they can enable or disable each other, partial order reduction can be applied to some extent for verification under fairness. Nonetheless, in previous approaches, because all events must be considered, virtually all events become inter-dependent and therefore no reduction is possible. In PAT, partial order reduction is disabled for model checking under strong fairness or strong global fairness. Nonetheless, for systems with event annotated fairness, it remains possible to apply partial order reduction to events which are irrelevant to the fairness annotations.

The algorithm presented in Figure 4.4 can be applied to check systems with event annotated fairness with slight modification. The basic idea remains, i.e., finding a strongly connected subgraph which satisfies the fairness constraints. Only events with fairness annotations are considered this time (by contrast to all events). We remark that annotating all events with weak (strong) fairness is equivalent to associate event-level weak (strong) fairness with the whole system. Method $tarjan$ is modified to cope with partial order reduction, following the heuristic function in [58]. In addition, we define an event to be *fairness visible* if it enables or disables an event annotated with fairness and require that *if the chosen set of events is a strict subset of the enabled events, the subset must not contain fairness visible events.* The intuition is that only independent events which are irrelevant to fairness are subject to partial order reduction. Notice that this checking has time complexity linear in the number of enabled events. The soundness follows from the discussion in [39, 163].

Function $prune$ is also modified to examine only the annotated events. It is defined as follows.

$$prune(S, Anno) = \begin{cases} \varnothing & \text{if there exists } f(e) \text{ and } e \notin engagedEvt(S); \\ \varnothing & \text{if } alwaysEvt(S) \cap wf(\Sigma) \not\subseteq engagedEvt(S); \\ \{s : S \mid enabledEvt(s) \cap sf(\Sigma) \subseteq engagedEvt(S)\}; & \\ & \text{otherwise}. \end{cases}$$

A SCC S is fair with respect to the event annotated fairness if and only if: all events which are annotated with unconditional event fairness are contained in the set $engagedEvt(S)$; if an event is annotated with weak event fairness and is enabled at *every* state in the SCC, then the event is contained in $engagedEvt(S)$; and if an event is annotated with strong fairness and is enabled at *some* state in the SCC, then the event is contained in $engagedEvt(S)$. If a SCC does not satisfy unconditional or weak event fairness, it is abandoned all together. If a state enables an event annotated with strong fairness which is never engaged in the SCC, then it is pruned. For instance, given the LTS in Figure 4.5, if event a is annotated with strong fairness, state 0 is a bad state in $scc2$. It is not if it is annotated with no or weak fairness. By a similar argument (to that of Theorem 4.4.1), the soundness of the algorithm can be proved.

4.6 A Multi-Core Model Checking Algorithm

Rapid development in hardware industry has brought the prevalence of multi-core systems with shared-memory, which enabled the speedup of various tasks by using parallel algorithms. The LTL model checking problem is one of the difficult problems to be parallelized or scaled up to multi-core systems. In this section, we present an on-the-fly parallel model checking algorithm based on the Tarjan's SCC detection algorithm presented in Section 4.4. The proposed parallel algorithm allows the verification to make full use of a multi-core CPU in the shared-memory architecture. The approach can be applied to general LTL model checking or with different fairness assumptions. Further, it is orthogonal to state space reduction techniques like partial order reduction.

4.6.1 Shared-Memory Platform

We work with a model based on threads that share all memory, although they have separate stacks in their shared address space and a special thread-local storage to store thread-private data. Our working environment is .NET framework (version 2.0) in Microsoft Windows platform, with its implementation of threads as lightweight processes. Switching contexts among different threads is cheaper than switching contexts among full-featured processes with separate address spaces, so using threads in the system incurs only a minor penalty.

Critical Sections, Locking and Lock Contention In a shared-memory setting, access to memory, that may be used for writing by more than a single thread, has to be controlled through the use of mutual exclusion, otherwise, race conditions will occur. This is generally achieved through use of mutex[8]. A thread wishing to enter a critical section has to lock[9] the associated mutex, which may block the calling thread if the mutex is locked already by some other thread. An effect called resource or lock contention is associated with this behavior. This occurs, when two or more threads happen to need to enter the same critical section (and therefore lock the same mutex), at the same

[8] A mutex is a common name for a program object that negotiates mutual exclusion among threads, also called a lock.

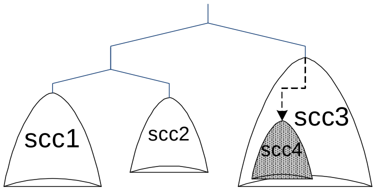
[9] In .NET framework, keyword $lock$ is used to achieve this effect.

time. If critical sections are long or they are entered very often, contention starts to cause observable performance degradation, as more and more time is spent waiting for mutexes.

Memory Management and Thread Communication In our setting, we assume that all resources are allocated from the managed heap. Objects are automatically freed when they are no longer needed by the application. The communication between threads can be achieved simply by object reference passing.

4.6.2 Parallel Fairness Model Checking Algorithm

The SCC-based verification algorithm presented in the previous section is highly recursive and employs a sequential DFS search, which exhibits some challenges in parallelism.



The sequential algorithm in Figure 4.4 can be illustrated in the figure above. When a SCC is detected, it will be analyzed and pruned until empty or there is a counterexample detected ($scc4$ in above graph). Taking a close look at the algorithm, we observe that there are four actions applied in each detected SCC: (1) fairness testing (line 4), (2) bad states pruning (line 8), (3) counterexample generation (line 5), (4) recursive sub-SCC detection (line 9). The first three actions are local to the detected SCC. Although the recursive sub-SCC detection is complicated, we can create a local copy of the Tarjan algorithm to search for “SCC” in the pruned states. In this way, each SCC can be processed independent. Therefore, we can put the workload of SCC analysis into separate threads to achieve concurrency. Inspired by these observations, we present a SCC-based parallel model checking algorithm with four parts: $Tarjan\ thread$, $SCC\ worker\ thread$, $SCC\ worker\ thread\ pool$ and $parallel\ model\ checker$. The detailed algorithms are illustrated as follows.

```
$stopped = false$;
procedure $run(threadPool, States, Transitions)$
1. $visited = \emptyset$;
2. while there are states in $States$ but not in $visited$
3.     if $stopped$ then {return; }
4.     let $scc = tarjan(States, Transitions)$;
5.     $visited = visited \cup scc$;
6.     if $forking\ conditions$ then
7.         $threadPool.forkWorkerThread(scc, Transitions)$;
8.     else
9.         process $scc$ locally
10.    endif
11. endwhile
12. return;
```

Figure 4.6: Tarjan thread implementation

Tarjan thread Figure 4.6 presents the implementation of $Tarjan\ thread$, which identifies all SCCs using Tarjan's algorithm. Tarjan thread has one public variable $stopped$ and the thread starting procedure run. $stopped$ is a control variable to stop this thread (line 3) as soon as a worker thread reports a counterexample. When $Tarjan\ thread$ starts, the run procedure will perform a DFS to detect all SCCs in the search space using Tarjan's algorithm. This process is similar to mc procedure in Figure 4.4. When a SCC scc is detected at line 4, if the forking conditions at line 6 are satisfied, then a new SCC worker thread will be forked and added in to the worker thread pool (line 7). Otherwise scc will processed locally in the $Tarjan\ thread$ (line 9). This local process is the same as the $SCC\ worker\ thread$ (which will be explained later), which stops this thread if there is a counterexample is found. Forking conditions can be that the size of scc is bigger than some threshold or the thread pool is full. We add this checking to achieve better efficiency and workload balance. If the size of scc is small (e.g., only few nodes), the overhead of creating a thread is much bigger than processing it locally. If the thread pool is full, processing the found scc locally is probably more efficient than creating a long waiting queue in the thread pool.

```
$threadQueue = empty\ queue;\ jobFinished = false;$
procedure $forkWorkerThread(States, Transitions)$
lock$(threadQueue);$
    if$(\neg jobFinished)$
        let $wt = new\ workerThread(States, Transitions);$
        register $wt.termination$ to $threadTermination$ procedure
        $threadQueue.enqueue(wt);$
    endif
unlock$(threadQueue);$

procedure $threadTermination(thread)$
lock$(threadQueue);$
    if thread produces counterexample $\wedge \neg jobFinished$ then
        terminate all other threads
        terminate tarjan thread
        $jobFinished = true;$
    endif
    $threadPool.remove(thread)$
unlock$(threadQueue);$

procedure $allThreadsJoin()$
while(has running threads)
    busy wait
endwhile
```

Figure 4.7: Thread pool implementation

SCC worker thread *SCC worker thread* works on a detected SCC to report whether the SCC contains a counterexample or not within the given SCC states and transitions. It basically resembles the code from line 4 to 12 (highlighted using _*) in Figure 4.4. If the detected SCC is not fair, it will prune the states according to the given fairness type. Otherwise it will terminate and return false. If the pruned *scc* has fewer states, a local copy of the Tarjan's algorithm will continue the searching. Upon the termination of *SCC worker thread*, a notification will send to the thread pool to notify the result.

SCC worker thread pool The implementation of the $SCC\ worker\ thread\ pool$ is presented in Figure 4.7. The thread pool has a working queue $threadQueue$[10] to store all active worker threads. Private variable $jobFinished$ indicates whether a counterexample has been found or not. Procedure $forkWorkerThread$ creates a new worker thread (line 3) and puts it into the working queue (line 5), if the counterexample is not found (line 2). A lock is used on $threadQueue$ (at line 1 and 7) to prevent $Tarjan\ thread$ working too fast to add two or more threads at same time. This is possible because during the process of forking the first thread, $Tarjan\ thread$ may find another SCC and want to fork a new thread. At line 4, we register the termination event of the $worker\ thread$ to procedure $threadTermination$, which means upon the termination of the worker thread, the thread pool will be notified and procedure $threadTermination$ will be triggered. When procedure $threadTermination$ is triggered, if the termination thread has located a counterexample and no one does it before (line 9), thread pool will terminate[11] all other active threads (line 10) and $Tarjan\ thread$ (by setting $stopped$ flag to true) (line 11). Flag $jobFinished$ is set to true at line 12, hence new threads shall not be forked anymore. $!jobFinished$ checking in line 9 is necessary to prevent terminating same threads twice. In the end, the termination thread is removed from thread pool in line 12. During this process $threadQueue$ is locked to prevent data race. Procedure $allThreadsJoin$ does busy-waiting until all threads terminate.

Parallel model checker Lastly, $parallel\ model\ checker$ is shown in Figure 4.8. It conducts the verification by creating the $Tarjan\ thread$ and thread pool. Once $Tarjan\ thread$ starts, it will wait for $Tarjan\ thread$ to join (i.e., successfully terminate) (line 3). The termination can be that all states are explored, or a counterexample is found locally, or $stopped$ flag is set to false. Afterwards, it will wait for thread pool to terminate (line 4). The procedure will return false if any counterexample is found in $tarjan\ thread$ or any worker thread.

[10] In our implementation, $threadQueue$ is realized by System.Threading.ThreadPool object in .NET Framework. The thread scheduling is managed by the thread pool automatically.

[11] Thread termination can be achieved by thread killing or asking the thread to voluntarily give up. The second way is safer and adopted in our approach. One example is the stopped flag in $Tarjan\ thread$.

```
**procedure** $pmc(States, Transitions)$
initialize worker thread pool $threadPool$
**let** $tarjan = tarjanThread.run(threadPool, States, Transitions);$
$tarjan.join();$
$threadPool.allThreadsJoin();$
**if** counterexample is found **then**
    **return** $false;$
**return** $true;$
```

Figure 4.8: Parallel model checker implementation

4.6.3 Complexity and Soundness

In this section, we discuss the complexity of the parallel model checking algorithm and prove its soundness.

For the parallel version of the algorithm, the time and space complexity is exactly same as the sequential version as discussed in Section 4.4.2. This is not surprising because the parallel algorithm simply splits SCC analysis into worker threads. The parallel algorithm is designed for a shared memory framework, the SCCs and their transitions are shared between $Tarjan\ thread$ and worker threads. There is no communication overhead. If to migrate this approach into distributed systems, we may consider to pass SCC only and let the worker threads to build the transitions locally to avoid the communication overhead. This is because the number of transitions of a SCC is often much larger than the number of vertices.

If the verification result is true, the number of states and transitions visited in the parallel and sequential version are same. If there is a counterexample, the parallel version may visit more states depending on when the counterexample is identified. If a counterexample is present in the first few SCCs encountered during the search, then the sequential version may find one quickly, while the parallel version may have forked multiple threads to search in more SCCs. Hence parallel version visits more states and transitions. On the other hand, if a counterexample is present only in the last few SCCs, the parallel version can be faster than the sequential version if the counterexample is identified quickly in one worker thread, which then terminates all other SCC checking. This is evidenced by the experiment results presented in Section 4.7.2. Notice that when there are more than one counterexamples in the system, it is possible that the parallel verification may produce different counterexample at different runs.

Regarding the soundness, the following theorem states the correctness of the parallel algorithm pmc. We argue the total correctness of the parallel algorithm by showing it is terminating and equivalent to the sequential mc algorithm.

Theorem 4.6.1 *Let $\mathcal{L}$ be a LTS. Let ϕ be a property. Let F be a fairness type (i.e., EWF, PWF, ESF, PSF or SGF). $\mathcal{L} \models_F \phi$ if and only if the algorithm pmc returns true.*

Proof. Firstly, we show that the pmc algorithm is terminating. By the assumption, we know that the number of states is finite, so is the number of the SCCs. In $Tarjan\ thread$, the number of visited states and the pruned states are monotonically increasing, hence the Tarjan thread is terminating. Worker threads are terminating since they are working on the detected SCC and the number of pruned states are monotonically increasing. Since the number of SCC is finite, worker thread pool is terminating. Therefore pmc is terminating.

Secondly, we show that pmc returns the same result as mc. The key of this proof is to prove that each SCC analysis is independent of each other. If this true, then checking the SCCs in parallel is same as checking them sequentially. We have listed the four actions performed in the SCCs in Section 4.6.2, which can be applied independently.

Lastly, the correctness of data sharing and race condition prevention by using locks have been discussed in Section 4.6.1. We skip it here. □

Following the above theorem, we conclude that the sequential algorithm and the parallel algorithm are equivalent in terms of correctness. Therefore as long as the reduction is compatible with sequential algorithm, then it is compatible with the parallel algorithm. For example, Section 4.5 shows that partial order reduction is possible[12] by employing fairness annotations on individual events,

[12] In general, fairness verification conflicts with partial order reduction.

which means this technique can also be used with our parallel algorithm. We remark that pmc is orthogonal to state reduction techniques like partial order reduction, symmetry reduction or data abstraction. Intuitively, the parallel algorithm would perform better since it may utilize more CPU power. Nonetheless, thread forking/terminating or communication between threads can be costly. We present detailed analysis using real-world examples as well as hand crafted examples in the next section.

4.7 Experiments

In this section, we demonstrate the effectiveness of the sequential version and parallel version of fairness model checking algorithms using experiments on both benchmark systems as well as recently developed population protocols, which require fairness for correctness.

4.7.1 Experiments for Sequential Fairness Verification

Table 4.2 summarizes the verification statistics on recently developed population protocols. The unit of time measurement is second. Notice that “$-$” means either out of memory or more than 4 hours. The protocols include leader election for complete networks (LE_C) [88], for rooted trees (LE_T) [47], for odd sized rings (LE_OR) [118], for network rings (LE_R) [88] and token circulation for network rings (TC_R) [14]. The property is that eventually always there is one and only one leader in the network, i.e., $\Diamond\Box oneLeader$. Correctness of all these algorithms relies on different notions of fairness. For simplicity, fairness is applied to the whole system. As a result, partial order reduction is only applied for verification under no or weak fairness, but not strong fairness or strong global fairness.

We remark that event-level fairness or strong global fairness is required for these examples. As discussed in Section 4.2, PWF is different from EWF. In order to compare PAT with SPIN for verification with EWF, we twist the models so that each event in population protocols is modeled as a process. By a simple argument, it can be shown that for such models, EWF is equivalent to PWF.

Model	Size	EWF			ESF		SGF	
		Result	PAT	SPIN	Result	PAT	Result	PAT
LE_C	6	Yes	26.7	229	Yes	26.7	Yes	23.5
LE_C	7	Yes	152.2	1190	Yes	152.4	Yes	137.9
LE_C	8	Yes	726.6	5720	Yes	739.0	Yes	673.1
LE_T	9	Yes	10.2	62.3	Yes	10.2	Yes	9.6
LE_T	11	Yes	68.1	440	Yes	68.7	Yes	65.1
LE_T	13	Yes	548.6	3200	Yes	573.6	Yes	529.6
LE_OR	3	No	0.2	0.3	No	0.2	Yes	11.8
LE_OR	5	No	1.3	8.7	No	1.8	–	–
LE_OR	7	No	15.9	95	No	21.3	–	–
LE_R	4	No	0.3	< 0.1	No	0.7	Yes	19.5
LE_R	6	No	1.8	0.2	No	4.6	–	–
LE_R	8	No	11.7	1.7	No	28.3	–	–
TC_R	5	No	< 0.1	< 0.1	No	< 0.1	Yes	0.6
TC_R	7	No	0.2	0.1	No	0.2	Yes	13.7
TC_R	9	No	0.4	0.2	No	0.4	Yes	640.2

Table 4.2: Population protocols experiments: with Core 2 CPU 6600 at 2.40GHz and 2GB RAM

Nonetheless, model checking under PWF in SPIN increases the verification time by a factor that is linear in the number of processes. By modeling each event as a process, we increase the number of processes and therefore un-avoidably increase the SPIN verification time by a factor that is constant (in the number of events per process for network rings) or linear (in the number of network nodes for complete network). SPIN has no support for ESF, PSF or SGF. Thus, the only way to model check under strong fairness or strong global fairness in SPIN is to encode the fairness constraints as part of the property. However, even for the smallest network (with 3 nodes), SPIN needs significant amount of time to construct (very large) Büchi automata from the property. Therefore, we conclude that it is infeasible to use SPIN for such a purpose and omit the experiment results from the table. We remark that in theory, strong fairness can be transformed to weak fairness by paying the price of one Boolean variable [124]. Nonetheless, the property again needs to be augmented with additional clauses after the translation, which is again infeasible.

All of the algorithms fail to satisfy the property without fairness. The algorithm for complete networks (LE_C) or trees (LE_T) requires at least EWF, whereas the rest of the algorithms require SGF. It is thus important to be able to verify systems under strong fairness or strong global fairness. Notice that the token circulation algorithm for network rings (TC_R) functions correctly for a network of size 3 under EWF. Nonetheless, EWF is not sufficient for a network with more nodes, as shown in the table. The reason is that a particular sequence of message exchange which satisfies EWF only needs the participation of at least 4 network nodes. This suggests that our improvement in terms of the performance and ability to handle different forms of fairness has its practical value.

A few conclusions can be drawn from the results in the table. Firstly, in the presence of counterexamples, PAT usually finds one quickly (e.g., on LE_R and TC_R under EWF or strong fairness). It takes longer to find a counterexample for LE_OR mainly because there are too many possible initial configurations of the system (exactly 2^{5*N} where N is network size) and a counterexample is only present for particular initial configurations. Secondly, verification under ESF is more expensive than verification with no fair, EWF or SGF. This conforms to theoretical time complexity analysis. The worst case scenario is absent from these examples (e.g., there are easily millions of transitions/states in many of the experiments). Lastly, PAT outperforms the current practice of verification under fairness. PAT offers comparably better performance on verification under weak fairness (e.g., on LE_C and LE_T) and makes it feasible to verify under strong fairness or strong global fairness. This allows us to discover bugs in systems functioning with strong fairness. Experiments on LE_C and LE_T (for which the property is only false under no fairness) show minor computational overhead for handling a stronger fairness.

Table 4.3 shows verification statistics of benchmark systems to show other aspects of our algorithm. Because of the deadlock state, the dining philosophers model ($DP(N)$ for N philosophers and forks) does not guarantee that a philosopher always eventually eats ($\Box\Diamond eat0$) whether with no fairness or strong global fairness. This experiment shows PAT takes little extra time for handling the fairness assumption. We remark that PAT may spend more time than SPIN on identifying a counterexample in some cases. This is both due to the particular order of exploration and the

Model	Property	Result	Fairness	PAT(sec)	SPIN(sec)
DP(10)	$\Box\Diamond eat0$	No	no	0.8	< 0.1
DP(13)	$\Box\Diamond eat0$	No	no	9.8	< 0.1
DP(15)	$\Box\Diamond eat0$	No	no	56.1	< 0.1
DP(10)	$\Box\Diamond eat0$	No	strong global fairness	0.8	–
DP(13)	$\Box\Diamond eat0$	No	strong global fairness	9.8	–
DP(15)	$\Box\Diamond eat0$	No	strong global fairness	56.0	–
MS(10)	$\Box\Diamond work0$	Yes	event-level strong fairness	9.3	–
MS(12)	$\Box\Diamond work0$	Yes	event-level strong fairness	105.5	–
MS(100)	$\Box\Diamond work0$	Yes	event annotated strong fairness	3.1	–
MS(200)	$\Box\Diamond work0$	Yes	event annotated strong fairness	15.5	–
PETERSON(3)	bounded bypass	Yes	process-level weak fairness	0.1	1.25
PETERSON(4)	bounded bypass	Yes	process-level weak fairness	1.7	> 671
PETERSON(5)	bounded bypass	Yes	process-level weak fairness	58.9	–

Table 4.3: Experiment results on benchmark systems

difference between model checking based on nested DFS and model checking based on identifying SCCs. PAT's algorithm relies on SCCs. If a large portion of the system is strongly connected, it takes more time to construct the SCC before testing whether it is fair or not. In this example, the whole system contains one large SCC and a few trivial ones including the deadlock state. If PAT happens to start with the large one, the verification may take considerably more time.

Milner's cyclic scheduler algorithm ($MS(N)$ for N processes) is a showcase for the effectiveness of partial order reduction. We apply fairness in two different ways, i.e., one applying ESF to the whole system and the other applying only to inter-process communications[13]. In the latter case, partial order reduction allows us to prove the property over a much larger number of processes (e.g., 200 vs 12). Peterson's mutual exclusive algorithm ($PETERSON(N)$) requires at least PWF to guarantee bounded by-pass [6], i.e., if a process requests to enter the critical section, it eventually will. The

[13] A general guideline for annotating event fairness is that, communication events are more unlikely to be annotated as local events are under the control of the local scheduler.

property is verified under PWF in both PAT and SPIN. PAT outperforms SPIN in this setting as well.

4.7.2 Experiments for Multi-core Fairness Verification

In this section, we present the experiments results on parallel fairness model checking algorithm. Table 4.4 summarizes the verification statistics on dinning philosophers problem (DP), and recently developed population protocols (LE_C [88], LE_R [88] and TC_R [14]). We modify the DP model so that it is deadlock-free (i.e., by letting one of the philosophers to pick up the forks in a different order). The property is that a philosopher never starves indefinitely, i.e., $\Box\Diamond eat.0$, where $eat.0$ is the event of 0-th philosopher eating. The property for the leader election protocols is that eventually always there is one and only one leader in the network, i.e., $\Diamond\Box oneLeader$. Correctness of all these algorithms relies on different notions of fairness.

In our experiments below, $Size$ denotes the number processes in the models. Besides the execution time of the sequential algorithm (mc) and parallel algorithm (pmc), we present additional measurements which reflect the how much workload pmc can put in parallel if the verification result is true[14]. One is the average size of nontrivial SCCs (denoted as $Avg\ SCC\ Size$) and the number of SCC (denoted as $\#SCC$). A SCC is trivial if and only if it has only one state. Intuitively, the parallel algorithm gains more saving with larger and more SCCs. The other is the ratio of the number of states of all (non-trivial) SCCs and the whole state space (denoted as $SCC\ Ratio$). Intuitively, a higher $SCC\ Ratio$ shall lead to more saving. The forking condition is that the SCC must have at least 100 states. '-' means out of memory. The unit of time measurement is second.

Regarding the threads scheduling, there are two approaches. The first approach is to manually assign a newly created thread to a free CPU-core. If all CPU-cores are used, the new thread is pushed into the working queue and wait.The second approach is to make each thread as operating system thread[15], and let the OS CPU scheduler to do the scheduling. We compared the two approaches,

[14] When the property is false, $SCC\ Ratio$ can be different for different runs.

[15] In our implementation, we use System.Threading.ThreadPool object in .NET framework 2.0 to create system threads in Microsoft Windows system.

Model	Size	Avg SCC	SCC	EWF			ESF			SGF		
		/#SCC	Ratio	Res.	mc	pmc	Res.	mc	pmc	Res.	mc	pmc
DP	5	67/13	0.36	No	0.08	0.08	Yes	0.22	0.20	Yes	0.19	0.19
DP	6	178/21	0.38	No	0.13	0.13	Yes	0.97	0.84	Yes	0.86	0.78
DP	7	486/31	0.4	No	0.38	0.37	Yes	4.62	3.39	Yes	4.42	3.38
DP	8	1368/43	0.41	No	1.41	1.33	Yes	29.3	19.5	Yes	32.9	22.1
LE_C	5	34/43	0.58	Yes	4.04	3.66	Yes	4.03	3.65	Yes	3.66	3.49
LE_C	6	48/103	0.64	Yes	23.1	21.3	Yes	23.1	21.5	Yes	21.9	20.1
LE_C	7	66/227	0.68	Yes	128	124	Yes	129	124	Yes	134	127
LE_C	8	86/479	0.71	Yes	604	600	Yes	615	607	Yes	721	684
LE_R	3	9/268	0.36	No	0.11	0.11	No	0.12	0.12	Yes	1.40	1.27
LE_R	4	9/2652	0.4	No	0.11	0.28	No	0.59	0.60	Yes	21.7	15.7
LE_R	5	9/25274	0.42	No	0.71	0.72	No	2.22	2.19	Yes	587	456
TC_R	6	84/2	0.01	No	0.11	0.11	No	0.11	0.11	Yes	2.20	2.38
TC_R	7	210/2	0.01	No	0.14	0.14	No	0.15	0.16	Yes	11.3	12.3
TC_R	8	330/3	0.01	No	0.19	0.20	No	0.25	0.23	Yes	69.6	72.9
TC_R	9	756/3	0.01	No	0.27	0.31	No	0.36	0.37	Yes	494	573

Table 4.4: Experiment results on a PC running Windows XP with 2.83 GHz quad-core Intel Q9550 CPU and 2 GB memory

it shows that when the size of the SCCs is big, the two approaches have same results. When the number of SCCs is big, the second approach is more efficient. We applied second approach in our experiments.

In Table 4.4, we can see that when the verification result is false, either pmc or mc can be faster, which is expected. When the verification result is true, pmc is faster in most of the cases, except in the case of model checking the TC_R example under strong global fairness. In this particular example, $SCC\ ratio$ is very low (0.01), which means that there are many trivial SCCs. Furthermore, there are only few non-trivial SCCs. As a result, there is little work that can be separated out for the worker threads to speed up the model checking, and the communication overhead makes pmc slower. On the other hand, the pmc slowdown in this case is only several percents of mc, which

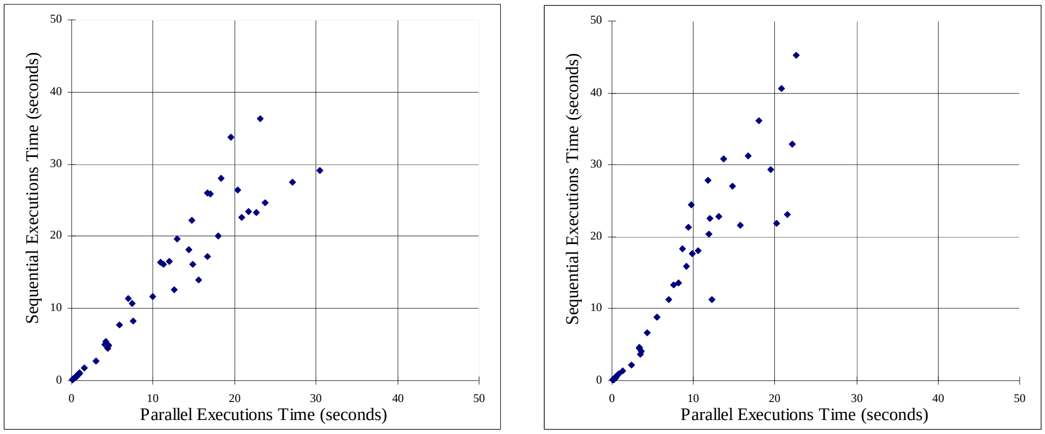
Model	Size	Avg SCC/	SCC	EWF			ESF			SGF		
		#SCC	Ratio	Res.	mc	pmc	Res.	mc	pmc	Res.	mc	pmc
$PAR1$	5	10001/5	0.2	No	1.75	2.11	Yes	22.5	12.0	Yes	11.3	6.97
$PAR1$	6	10001/6	0.2	No	1.74	2.07	Yes	27.1	14.8	Yes	13.6	8.13
$PAR1$	7	10001/7	0.2	No	1.71	2.29	Yes	31.2	16.7	Yes	15.9	9.14
$PAR1$	8	10001/8	0.2	No	1.71	2.16	Yes	36.1	18.0	Yes	18.1	10.6
$PAR1$	9	10001/9	0.2	No	1.71	2.15	Yes	40.6	20.9	Yes	20.4	11.9
$PAR1$	10	10001/10	0.2	No	1.73	2.15	Yes	45.3	22.6	Yes	22.81	13.07
$PAR2$	4	20000/5	0.5	No	5.46	7.12	NA	-	-	Yes	8.87	5.52
$PAR2$	5	20000/6	0.5	No	6.05	9.53	NA	-	-	Yes	18.3	8.64
$PAR2$	6	20000/7	0.5	No	6.39	10.5	NA	-	-	Yes	21.4	9.32
$PAR2$	7	20000/8	0.5	No	6.90	11.4	NA	-	-	Yes	24.5	9.69
$PAR2$	8	20000/9	0.5	No	7.77	11.7	NA	-	-	Yes	27.9	11.82
$PAR2$	9	20000/10	0.5	No	8.06	12.8	NA	-	-	Yes	30.9	13.68
$PAR3$	7	2000/8	1	No	0.29	0.20	Yes	412	118	Yes	0.41	0.28
$PAR3$	8	2000/9	1	No	0.21	0.24	Yes	463	136	Yes	0.45	0.29
$PAR3$	9	2000/10	1	No	0.25	0.23	Yes	516	156	Yes	0.49	0.31

Table 4.5: Experiment results on a PC running Windows XP with 2.83 GHz quad-core Intel Q9550 CPU and 2 GB memory

shows that the communication overhead in pmc is low.

Table 4.5 summarizes the verification statistics on some hand craft examples to show the potential effectiveness of the parallel algorithm. We create three models ($PAR1$, $PAR2$ and $PAR3$) such that their state spaces contain several SCCs, each of which has a big number of states. As a result, worker threads can be dispatched with substantial workload. Correctness of all these algorithms requires ESF and SGF.

In Table 4.5, we can see that pmc is working well in $PAR1$ example, where the average SCC size is big and the SCC ratio is not very low. The performance is even better (60% speedup) when the SCC ratio increases to 0.5 in $PAR2$ example. The $PAR3$ example almost produces the ideal case (72% speedup) such that the four cores are fully loaded. Since there are more SCCs than cores,



(a) Results on Intel Core2 6600 CPU

(b) Results on Intel Q9550 CPU

Figure 4.9: Experimental results for scalability testing

further speedup could be achieved if there were more cores. ESF case in $PAR2$ gives a worst case mentioned in Section 4.3 for strong fairness checking, hence it ends up with out of memory exception.

The experiment results in Table 4.4 and 4.5 confirm that the speedup of parallel verification relies on the size and the number of non-trivial SCCs. Each SCC has four analysis actions as described in Section 4.6.2. If the size of SCCs is big and/or the number of SCCs is more than the number of cores, each worker thread will make full use of the available CPU-cores. Overall, pmc performs better than mc for big average SCC size and high SCC ratio.

To study the scalability of our approach with different number of CPU cores, we conduct the same experiments (model checking examples in Table 1 and $PARA1$ example under strong global fairness)[16] on a dual-core CPU (Figure 4.9 (a)) and a quad-core CPU[17] (Figure 4.9 (b)). The coordinate of each point (x, y) in the graphs represents mc execution time and pmc execution time of a model correspondingly. From the figures we can see that, points in Figure 4.9 (a) are scattered between line

[16] $PARA2$ and $PARA3$ have high average SCC size and SCC ratio which is rare in real systems, so we exclude them in the salability testing.

[17] Since we calculate the speedup of pmc compared to mc, the absolute speed of the two CPUs is not important.

$y = x$ and $y = 2x$, while points in Figure 4.9 (b) are scattered between line $y = 2x$ and $y = 3x$. The average speedup of the parallel algorithm is 22.9% for quad-core CPU and 11.2% for dual-core CPU. This suggests that our approach is scalable for more CPU cores in general.

Besides PAT, there are a number of model checkers which are designed for similar application domains. It is, however, not easy to compare PAT with them. For instance, the refinement checker FDR does not support shared variables/arrays, and therefore, FDR's model is significantly different from PAT's. Further, FDR has no support for multi-core. The model checker SPIN supports verification of LTL properties. The multi-core parallel algorithm in SPIN is designed for model checking based on nested depth-first search. Nested depth-first-search works well for verification under no fairness. It can be twisted to perform model checking under fairness in the price of significant computational overhead, which has been shown in [197]. As a result, it makes little sense here to compare performance of our parallel algorithm with SPIN's.

4.8 Summary

In summary, we developed a model checking approach to verify fairness enhanced systems based on Tarjan's SCC detection algorithm. Our approach is holistic, which does not only take care of LTL verification but also checks the fairness constraints satisfaction in one goal. Furthermore, we presented a parallel version of the proposed algorithm for in multi-core shared-memory architecture. We showed that our sequential algorithm is effective to prove or disprove both benchmark systems and newly proposed distributed algorithms. The experimental results on real world systems suggested our parallel algorithm is efficient and scalable to multi-cores.

The research on categorizing fairness/liveness, motivated by system analyzing of distributed or concurrent systems, has a long history [131, 138, 18, 96]. A rich set of fairness notions have been identified during the last decades, e.g., weak or strong fairness in [131], justice or compassion conditions in [138], hyperfairness in [21, 132], strong global or local fairness recently in [88], etc. This chapter summarizes a number of fairness notions which are closely related to distributed system verification and provides a framework to model check under different fairness constraints. Other works on categorizing fairness/liveness have been evidenced in [173, 128, 166, 216].

This chapter is related to research on system verification under fairness [100, 131, 88]. Our model checking algorithm is related to previous works on emptiness checking for Büchi automata [126, 111, 99] and Streett automata [124, 99, 137, 106]. In this chapter, we apply the idea to the automata-based model checking framework and generalize it to handle different fairness assumptions. In a way, our algorithm integrates the two algorithms presented in [99, 106] and extends them in a number of aspects to suit our purpose. Furthermore, model checking algorithms under strong fairness have been proposed in [99] and [137]. In both works, a similar pruning process is involved.

Regarding LTL parallel verification, there are various approaches in the literature. Holzmann proposed a parallel extension of the SPIN model checker in [112]. In this SPIN extension, the algorithm for checking safety properties scales well to N-core systems. However, the algorithm for liveness checking, which is based on the original SPIN's nested DFS algorithm, can only be applied in dual-core systems. Whereas, our approach is scalable to N-core system and can also handle different forms of fairness, while SPIN handles only process level weak fairness. Lafuente [129] presented a cycle localization algorithm based on nested DFS, which is very similar to our ideas. In their approach, the main thread performs the first DFS to identify an accepting state, and the worker threads perform the second DFS to detect the fair cycle starting from the accepting state. Compared to this solution, our approach has the advantage that each SCC will be checked by one and only one worker thread. A different approach to shared-memory model checking is presented in [114], based on CTL* translation to Hesitant Alternating Automata. The proposed algorithm uses so-called non-emptiness game for deciding validity of the original formula and is therefore largely unrelated to the algorithms based on fair-cycle detection. Barnat, Chaloupka and Pol gave a comprehensive survey [24] in the distributed SCC decomposition algorithms [37, 38, 89, 49, 36, 23, 25]. However, these algorithms are designed for distributed systems and have quadratical or cubic order of complexity.

Chapter 5

Applications of Fairness Model Checking

Recently, the population protocol model [20] has emerged as an elegant computation paradigm for describing mobile ad hoc networks. One essential property of population protocols is that with respect to all possible initial configurations all nodes must eventually converge to the correct output values (or configurations), which is a typical liveness property. To guarantee that such kind of properties can be achieved, fairness assumption is required. A number of population protocols have been proposed and studied [14, 16, 88, 118, 15]. Most of them only work if *global fairness* (see Section 5.1) is imposed. Furthermore, it was shown that without global fairness uniform self-stabilizing leader election in rings is impossible [88].

In this chapter, we apply the fairness model checking algorithms developed in Chapter 4 on a set of self-stabilizing population protocols for ring networks to show the effectiveness of the proposed algorithms. The choice of ring topology reduces the interactions of nodes and also makes our models scale up easily. We select protocols for two-hop coloring and orienting nodes and protocols for leader election and token passing. All these protocols only work under global fairness. We report on our model checking results. Especially, we present one previously unknown bug in a leader election protocol [118], which can only be identified using PAT (as far as we know).

We notice that population protocols are designed on a large or even unbounded number of behaviorally similar processes, which raises the state explosion problem of the model checking approach. To solve this problem, we propose a fair model checking algorithm with process counter abstraction. Our on-the-fly checking algorithm enforces fairness by keeping track of the local states from where actions are enabled / executed within an execution trace without maintaining the process identifiers. We show the usability of this technique via the automated verification of several real-life protocols.

The remainder of the chapter is organized as follows. In Section 5.1, we introduce the population protocol model. Section 5.2 presents the population protocols studied in this chapter. The model checking results are summarized in Section 5.3. The process counter abstraction starts from Section 5.4 with the system model definitions and process counter representation. Section 5.5 presents how to perform model checking under fairness without the knowledge of process identifiers. Section 5.6 discusses how to handle infinitely many processes. Section 5.7 discusses experimental results of the counter abstraction. Section 5.8 concludes this chapter.

5.1 The Population Protocol Model

The population protocol model [20] is proposed recently as a computation paradigm for describing mobile ad hoc networks, consisting of multiple mobile nodes which interact with each other to carry out a computation. Application domains of the protocols include wireless sensor networks and biological computers. In this section, we briefly introduce the population protocol model. More details are available in [14, 88].

In the population protocol model, the underlying network can be described as a directed graph $G = (V, E)$ without multi-edges and self-loops. Each vertex represents a simple finite-state sensing device, and each edge (u, v) means that u as an *initiator* could possibly interact with v as a *responder*.

Definition 15 (Population Protocol Model) *A* population protocol model *is specified as a six-tuple* $\mathcal{P} = (Q, \mathcal{C}, X, Y, O, \delta)$*, which contains*

- *a finite set Q of states,*
- *a set $\mathcal{C}$ of configurations,*
- *a finite set X of input symbols,*
- *a finite set Y of output symbols,*
- *an output function $O : Q \to Y$, and*
- *a transition function $\delta : (Q \times X) \times (Q \times X) \to 2^{Q \times Q}$.*

If $(p', q') \in \delta((p, x), (q, y))$, then we write $((p, x), (q, y)) \to (p', q')$ and call it a transition. When δ always maps to a set that only contains a single pair of states, then we call the protocol *deterministic*.

A *configuration* C is a mapping $C : V \to Q$ assigning to each node its internal state, and an *input assignment* $\alpha : V \to X$ specifies the input for each node. Let C and C' be configurations, α be an input assignment, and u, v be different nodes. If there is a pair $(C'(u), C'(v)) \in \delta((C(u), \alpha(u)), (C(v), \alpha(v)))$, we say that C goes to C' via edge $e = (u, v)$ by transition $((C(u), \alpha(u)), (C(v), \alpha(v))) \to (C'(u), C'(v))$, abbreviated to $(C, \alpha) \xrightarrow{e} C'$. A pair of a transition r and an edge e constitutes an *action* $\sigma = (r, e)$. If C goes to C' via some edge, then C can go to C' in one *step*, written as $(C, \alpha) \to C'$.

An *execution* is an infinite sequence of configurations and assignments $(C_0, \alpha_0), (C_1, \alpha_1), \ldots,$ $(C_i, \alpha_i), \ldots$, such that $C_0 \in \mathcal{C}$ and for each i, $(C_i, \alpha_i) \to C_{i+1}$. Different fairness assumptions on population protocol models can be defined on the system executions in the same way as in Section 4.2. The fairness constraint is imposed on the scheduler to ensure that the protocol makes progress. In population protocols, the required fairness condition will make the system behave nicely eventually, although it can behave arbitrarily for an arbitrarily long period [20]. That is why most of population protocols [14, 16, 88, 118, 15] only work if *global fairness* is assumed. For instance, Fischer and Jiang [88] have proved that without global fairness uniform self-stabilizing leader election in rings is impossible. Several protocols are presented in the next section to further explain the ideas.

5.2 Population Ring Protocol Examples

In this section, we take a set of self-stabilizing population protocols for ring networks. A distributed system or a population protocol is said to be *self-stabilizing* [66] if it satisfies the following two properties:

- *convergence*: starting from an arbitrary configuration, the system is guaranteed to reach a correct configuration;
- *closure*: once the system reaches a correct configuration, it cannot become incorrect any more.

This means that in our modeling of these protocols, we have to take all possible initial configurations into account, and the checked properties have the form of $\Diamond\Box property$. The choice of ring topology makes it less demanding when we model the interactions of nodes and it also makes our models easily scale up to larger instances. We have selected protocols for two-hop coloring and orienting nodes and protocols for leader election and token passing. Note that all these protocols only work under global fairness.

In the population protocol model, one protocol consists of N nodes, numbered from 0 to $N-1$.[1] A protocol is usually described by a set of interaction rules between an initiator u and a responder v. Such rules have conditions on the state and the input of the initiator and the responder, and specify the state of the initiator and the responder if a transition can be taken.

5.2.1 Two hop coloring

A protocol to make nodes to recognize their neighbors in a ring is presented in [15]. In fact, it is a general algorithm that enables each node in a degree-bounded graph to distinguish between its neighbors. The graph is colored such that any two nodes adjacent to the same node have different colors. More precisely, for each node v, if u and w are distinct neighbors of v, then u and w must

[1] In the following discussion, we set N as three for simplicity.

have different colors. (u, w) is called a *two-hop* pair. In this chapter, we restrict ourselves to rings, and three colors suffice the purpose (see [15]).

Each node u in a ring has two state components, $color[u]$ encodes the color of node u and $F[u]$ is a bit array, indexed by colors. Initially, $color[u]$ and $F[u]$ can have arbitrary values. The following description defines the interaction between an initiator u and a responder v.

```
**if** $F[u][color[v]] \neq F[v][color[u]]$ **then**
    $color[u] \leftarrow color'[u];\ \ F[u][color[v]] = F[v][color[u]];$
**else**
    $F[u][color[v]] = \neg F[u][color[v]];\ \ F[v][color[u]] = \neg F[v][color[u]];$
**endif**
```

One edge (or interaction) (u, v) is synchronized if $F[u][color[v]] = F[v][color[u]]$, then these two nodes do not change their color but flip their bits ($F[u][color[v]]$ and $F[v][color[u]]$). On the other hand, node u is nondeterministically recolored, and it copies $F[v][color[u]]$ of node v as its bit $F[u][color[v]]$. The statement $color[u] \leftarrow color'[u]$ means one of the three possible colors is nondeterministically assigned as the new color of u. The CSP# model of this protocol is shown in detailed in example 5.2.1. In [15], a deterministic version of two-hop coloring is given as well (see below). Instead of nondeterministically assigning all possible colors to the initiator u, its color is updated as $color[u] \leftarrow (color[u] + r[u]) \bmod C$. The additional state component $r[u]$ is a local bit for node u that flits whenever u acts as the initiator of an interaction. We also model and analyze this protocol in CSP#.

```
**if** $F[u][color[v]] \neq F[v][color[u]]$ **then**
    $color[u] \leftarrow (color[u] + r[u]) \bmod C;\ \ F[u][color[v]] = F[v][color[u]];$
**else**
    $F[u][color[v]] = \neg F[u][color[v]];\ \ F[v][color[u]] = \neg F[v][color[u]];$
**endif**
$r[u] \leftarrow \neg r[u];$
```

Example 5.2.1 (Two-hop Coloring Protocol) A self-stabilizing population protocol for two-hop coloring is proposed [15]. This algorithm can guarantee that the neighbors of a node in a ring have different colors. Figure 5.1 presents (part of) its model in CSP#. Line 1 defines two global constants (N and C of value 3) and global variables. N models the network size, i.e., number of nodes and C

```
1. #define $N$ 3; #define $C$ 3; var $color[N]$; var $F[N][C]$;
2. $Inter(u, v) =$
3.     if $(F[u][color[v]] \neq F[v][color[u]])\{$
4.         $act1.u.v\{F[u][color[v]] = F[v][color[u]];\ color[u] = 0;\ \} \rightarrow Inter(u, v)$
5.         $\Box\ act2.u.v\{F[u][color[v]] = F[v][color[u]];\ color[u] = 1;\ \} \rightarrow Inter(u, v)$
6.         $\Box\ act3.u.v\{F[u][color[v]] = F[v][color[u]];\ color[u] = 2;\ \} \rightarrow Inter(u, v)$
7.     $\}$ else $\{$
8.         $act4.u.v\{F[u][color[v]] = 1 - F[u][color[v]];$
9.                 $F[v][color[u]] = 1 - F[v][color[u]];\ \} \rightarrow Inter(u, v)$
10.     $\};$
11. $Init() = ...$
12. $Protocol() = Init();\ |||\ x : \{0..N-1\}@(Inter(x, (x+1)\%N)\ |||\ Inter((x+1)\%N, x));$
13. #define $thcolor\ (color[0] \neq color[2] \wedge color[1] \neq color[2] \wedge color[0] \neq color[1]);$
14. #assert $Protocol() \vDash \Diamond\Box thcolor;$
```

Figure 5.1: CSP# Model for two hop coloring protocol

models the number of colors. Array $color$ models the color of each node. F is a bit array for each node, indexed by colors. Next, line 2 to 10 defines how an initiator u interacts with a responder v, which captures the essence of the protocol. Every time there is an interaction in the network, the initiator and responder must update themselves according to a set of pre-defined rules. A rule is applicable only if the guarding condition (e.g., $F[u][color[v]] \neq F[v][color[u]]$) is satisfied. An action (e.g., $act1.u.v$) may be attached with variables updating (e.g., $color[u] = 0$). Line 12 models the two-hop coloring protocol as process $Protocol$, which starts with process $Init$ (which initializes the system in every possible configuration and is omitted here). After initialization, the system is the interleaving (modeled by the operator $|||$) of nodes' interactions in the network. Which nodes can interact reflects the topology of the network. The LTL property is $\Diamond\Box thcolor$ (defined as an assertion at line 14), where $\Diamond$ and $\Box$ are modal operators which read as *eventually* and *always* respectively (refer to Section 2.3.2 for details). $thcolor$ (defined at line 13) is a proposition which states that the neighbors of a node in a ring have different colors (for rings of size three). **end**

5.2.2 Orienting undirected rings

Given a ring colored by protocols in Section 5.2.1, it is possible to have a protocol that gives a sense of orientation to each node on an undirected ring [15]. After the orienting, (1) each node has exactly one predecessor and one successor, the predecessor and successor of a node are different; (2) for any two nodes u and v, u is the predecessor of v if and only if v is the successor of u, for any edge (u, v), either u is the predecessor of v or v is the predecessor of u.

Each node u in a ring has three state components: $color[u]$ encodes the color of node u, $precolor[u]$ the color of its predecessor, and $succolor[u]$ the color of its successor. Initially, all nodes are two-hop colored (array $color$ satisfies the two-hop coloring property), $precolor[u]$ and $succolor[u]$ can have arbitrary values. The following description defines the interaction between an initiator u and a responder v. The CSP# model of this protocol is shown in Figure B.1 in Appendix B.

```
**if** $color[v] == precolor[u] \wedge color[v] \neq succolor[u]$ **then**
    $succolor[v] \leftarrow color[u]$;
**elseif** $color[v] == succolor[u] \wedge color[v] \neq precolor[u]$ **then**
    $precolor[v] \leftarrow color[u]$;
**else**
    $precolor[u] \leftarrow color[v]$;   $succolor[v] \leftarrow color[u]$;
**endif**
```

5.2.3 Leader election

In this section, we study a leader election protocol in oriented odd rings. The following description is partially taken from [118, 15]. Supposing each node has a *label* bit, a maximal sequence of alternating labels is called a segment. According to the orientation of the ring, the head and tail of a segment can be defined in a natural way. One edge of the form $(0, 0)$ or $(1, 1)$ connecting the tail of one segment to the head of another segment is called a *barrier* edge. For a node u in a ring, it has four state components: $leader[u]$ states whether the node is a leader, $label[u]$ is its label, $probe[u]$ is 1 if u holds a probe token, and $phase[u]$ alternates between 0 and 1 to make each barrier alternate between firing a probe and moving forward. The protocol consists of several parts. In the basic part, the barriers move clockwise around the ring. Each barrier advances by flipping the label bit of the second node on the barrier (the head of the next segment). When two barriers collide, they cancel out each other. Because the ring size is odd, there is always at least one barrier. In the rest of the protocol, the leader bullet and probe marks are manipulated. Probes are sent out by the barrier in a clockwise direction and absorbed by any leader they run into. If a probe meets the barrier on its way back, it is converted to leader. Leaders fire *bullets* counter-clockwise around the ring. Bullets are absorbed by the barrier, but they kill any leaders they encounter along the way. The description of an interaction between an initiator u and a responder v in the protocol (taken from [118], p.66) is as follows. The CSP# model of this protocol is shown in Figure B.2 in Appendix B.

```
if $label[u] == label[v]$ then
  if $probe[u] == 1$ then $leader[u] \leftarrow 1;\ probe[u] \leftarrow 0$ endif
    $bullet[v] \leftarrow 0$
  if $phase[u] == 0$ then $phase[u] \leftarrow 1;\ probe[v] \leftarrow 1;$
  elseif $probe[v] == 0$ then
    $label[v] = \neg label[v];\ phase[v] \leftarrow 0$
  endif
elseif $leader[v] == 1$ then
  if $bullet[v] == 1$ then $leader[v] \leftarrow 0$ else $bullet[u] \leftarrow 1$ endif
else
  if $bullet[v] == 1$ then $bullet[v] \leftarrow 0;\ bullet[u] \leftarrow 1$ endif
  if $probe[u] == 1$ then $probe[u] \leftarrow 0;\ probe[v] \leftarrow 1$ endif
endif
```

Counterexample. We have analyzed this protocol in PAT, and found one counterexample. We consider a ring of size three, nodes are numbered as 0, 1 and 2. The counterexample found by PAT can be described as follows: it is an infinite execution containing a loop, u is the node for the initiator and v for the responder of one interaction according to the protocol description. The execution can start with a configuration $bullet = [1, 1, 1], label = [1, 1, 1], leader = [1, 1, 0], phase = [1, 1, 1], probe = [1, 1, 0]$.

1. Since $label[2] == label[0]$, $probe[2] == 0$, $phase[2] == 1$ and $probe[0] == 1$, we have $bullet[0] \leftarrow 0$. ($u = 2$ and $v = 0$)

2. Then since $label[0] == label[1]$, $probe[0] == 1$, $phase[0] == 1$ and $probe[1] == 1$, we have $leader[0] \leftarrow 1$, $probe[0] \leftarrow 0$, and $bullet[1] \leftarrow 0$. ($u = 0$ and $v = 1$)

3. Then since $label[2] == label[0]$, $probe[2] == 0$, $phase[2] == 1$ and $probe[0] == 0$, we have $bullet[0] \leftarrow 0$, $label[0] \leftarrow 1 - label[0]$, and $phase[0] \leftarrow 0$. ($u = 2$ and $v = 0$)

4. Then since $label[1] == label[2]$, $probe[1] == 1$, $phase[1] == 1$ and $probe[2] == 0$, we have $leader[1] \leftarrow 1$, $probe[1] \leftarrow 0$, $bullet[2] \leftarrow 0$, $label[2] \leftarrow 1 - label[2]$ and $phase[2] \leftarrow 0$. ($u = 1$ and $v = 2$)

5. Then since $label[2] == label[0]$, $probe[2] == 0$ and $phase[2] == 0$, we have $bullet[0] \leftarrow 0$, $phase[2] \leftarrow 1$ and $probe[0] \leftarrow 1$. ($u = 2$ and $v = 0$)

Now, we have reached a configuration with $bullet = [0, 0, 0], label = [0, 1, 0], leader = [1, 1, 0], phase = [0, 1, 1], probe = [1, 0, 0]$.[2] From here, we have a loop. Within this loop, all actions enabled at reachable configurations of the loop are executed. But these configurations contain more than two leaders. Hence, this infinite execution is global fair but not self-stabilizing for leader election. The loop is given below.

1. Since $label[2] == label[0]$, $probe[2] == 0$, $phase[2] == 1$ and $probe[0] == 1$, we have $bullet[0] \leftarrow 0$. ($u = 2$ and $v = 0$)

2. Then since $label[0] \neq label[1]$, $leader[1] == 1$ and $bullet[1] == 0$, we have $bullet[0] \leftarrow 1$. ($u = 0$ and $v = 1$)

3. Then since $label[0] \neq label[1]$, $leader[1] == 1$ and $bullet[1] == 0$, we have $bullet[0] \leftarrow 1$. ($u = 0$ and $v = 1$)

4. Then since $label[2] == label[0]$, $probe[2] == 0$, $phase[2] == 1$ and $probe[0] == 1$, we have $bullet[0] \leftarrow 0$. ($u = 2$ and $v = 0$)

The last step in the loop leads us back to the starting configuration of the loop. We have communicated this counterexample to the author of [118], it is confirmed as a valid counterexample

[2] As the protocol is self-stabilizing, the counterexample can start directly from here. We keep the first part just to faithfully represent the infinite trace found by PAT.

which has escaped simulations of the protocol [119]. The reason to the counterexample is the following [119]. In the explanation of the protocol, it says that “probes are sent out by the barrier in a clockwise direction and absorbed by any leader they run into". The second half of the sentence is missing from the pseudo code description. The protocol also requires consistent ordering of the position of tokens within each node (in the order of leader, bullet, and probe clockwise). A barrier edge should only generate a probe at the responder if the responder is not a leader. Otherwise, the probe would be able to pass the leader token. In the description, this property is not preserved either. Modifications of the description have been made in [15]. We also modeled the revised version of the protocol, and found no counterexample. By this case study, we emphasize that without the newly developed model checking algorithm in Chapter 4 for efficient verification under (global) fairness, it is impossible to find such an error in a pseudo code description of a population protocol, especially when a protocol tends to be intuitively more complicated.

5.2.4 Token circulation

The token circulation protocol in directed rings depicted below is proposed in [14, 15]. The desired behavior of this protocol can be described as follows: (1) there is only one node who holds the token; (2) a node does not obtain again until every other node has obtained a token once; (3) each node can have the token infinitely often.

Rule 1. $((*\ b,\ \ N),(*\ b,\ \ L))\ \rightarrow\ ((-\ b),(+\ \bar{b}))$
Rule 2. $((*\ b,\ \ *),(*\ \bar{b},\ \ N))\ \rightarrow\ ((-\ b),(+\ b))$

It is assumed that every node passes the token to next one right after getting it. Furthermore, the protocol also requires the existence of a leader. Informally, there is a static node with the leader mark L, and all other nodes have the non-leader mark N in every configuration. The state of each node is represented by a pair in $\{-,+\} \times \{0,1\}$. $+$ means that the node is holding a token and $-$ means the opposite. The second part of a state of a node is called the label. The $*$ here denotes an always-matched symbol. On the left hand side, the symbol b matches either 0 or 1 and $\bar{b}$ is its complement. It should be noticed that different occurrences of b in a same rule refer to the same value. The input for each node informs them who is leader, which is unique in the network. When two nodes interact, if the responder is the leader, it sets its label to the complement of the initiator's label; otherwise the responder copies the label from the initiator. If an interaction triggers a label change, a token is passed from the initiator to the responder. If a token is not present at the initiator, a new token is generated.

The CSP# model of this protocol is shown in Figure B.3 in the Appendix B. We only give the assertion for the first property. The other two can be defined in a similar way. The states of the whole system are represented by three arrays of bits ($leader[N]$, $token[N]$ and $label[N]$). Without loss of generality, we assume that node 0 is always the leader. Therefore, we could simply set each node a fixed input ($leader[i]$) for leader election without considering complicated details of a dynamic leader election process, which we have analyzed in Section 5.2.3.

5.3 Experiments of Population Protocols

In this section, we present the experimental results on the set of ring protocols presented in previous sections. Table 5.1 collects the experimental results. For the two-hop coloring protocol, there are two version: ^{1} for nondeterministic and ^{2} for deterministic. For the orienting undirected ring protocol, both properties in Figure B.1 are checked. Leader election protocol is only checked for odd rings as required. The experiment testbed is a PC running Windows XP SP3 with 2.83GHz Intel Q9550 CPU and 4 GB memory. In the table, "$-$" means out of memory.

From the table, firstly it shows that the number of states, transitions and running time increase rapidly (exponentially) with the number of nodes in rings, especially for two-hop coloring and leader election protocols. The reason is that these protocols use more state components than the others, e.g., the arrays. This conforms to the theoretical results. Secondly, we show that PAT is effective, it can handle millions states in hundreds of seconds (which is compatible to SPIN). Notice that SPIN is infeasible for verifying the protocols because it does not support the fairness notions[3].

[3] SPIN supports only process-level weak fairness.

Model	Property	Ring Size	Result	#States	#Transitions	Time (Sec)
two-hop coloring[1]	$\Diamond\Box thcolor$	3	Yes	122856	1972174	43.3
two-hop coloring[1]	$\Diamond\Box thcolor$	4	NA	–	–	–
two-hop coloring[2]	$\Diamond\Box thcolor$	3	Yes	983016	9473998	627
two-hop coloring[2]	$\Diamond\Box thcolor$	4	NA	–	–	–
orienting rings	$\Diamond\Box property1$	3	Yes	3200	28540	0.61
orienting rings	$\Diamond\Box property2$	3	Yes	3221	28163	0.64
orienting rings	$\Diamond\Box property1$	4	Yes	69766	883592	18.1
orienting rings	$\Diamond\Box property2$	4	Yes	66863	794662	17.5
orienting rings	$\Diamond\Box property1$	5	Yes	1100756	18216804	601
orienting rings	$\Diamond\Box property2$	5	Yes	1021851	15486265	536
orienting rings	$\Diamond\Box property1$	6	NA	–	–	–
leader election	$\Diamond\Box oneleader$	3	Yes	55100	216699	10.6
leader election	$\Diamond\Box oneleader$	5	NA	–	–	–
token circulation	$\Diamond\Box onetoken$	6	Yes	21559	105577	2.86
token circulation	$\Diamond\Box onetoken$	7	Yes	91954	514703	14.9
token circulation	$\Diamond\Box onetoken$	8	Yes	388076	2446736	88.6

Table 5.1: Experiment results of population protocols

Although we are bound to check relatively small instances of the protocols, the newly developed verification techniques in Chapter 4, does complement existing model checkers with the improvement in terms of performance and ability to handle different forms of fairness. It enables us to establish the correctness of these protocols under global fairness or, in the case of the leader election protocol, to identify bugs. Readers can compare the result presented in Section 4.7 on a similar verification practice using SPIN. The argument for using model checking techniques in general, is that, if there is a bug in the protocol design, probably it is present in a small network. On the other hand, to apply model checking on large number of nodes, we need to apply advanced reduction or abstraction techniques in the verification to conquer the state explosion problem. This motivates the algorithm of fairness model checking with process counter abstraction, which is presented in following sections.

5.4 Process Counter Abstraction

All the protocols in the previous sections are designed on a large (or even unbounded) number of behaviorally similar processes of the same type. Such systems, refereed as parameterized systems, frequently arise in distributed algorithms and protocols (*e.g.*, cache coherence protocols, control software in automotive / avionics), where the number of behaviorally similar processes is unbounded during system design, but is fixed later during system deployment. Thus, the deployed system contains fixed, finite number of behaviorally similar processes. However during system modeling/verification it is convenient to not fix the number of processes in the system for the sake for achieving more general verification results. A parameterized system represents an infinite family of instances, each instance being finite-state. Property verification of a parameterized system involves verifying that *every finite state instance of the system* satisfies the property in question.

A common practice for analyzing parameterized systems can be to fix the number of processes to a constant (as we did in the verification of population protocols). To avoid state space explosion, the constant is often small (smaller than 10 in our population protocols experiments), compared to the size of real applications. Model checking is then performed in the hope of finding a bug which is exhibited by a fixed (and small) number of processes. This practice can be incorrect because the real size of the systems is often unknown during system design (but fixed later during system deployment). It is also difficult to fix the number of processes to a “large enough" constant such that the restricted system with fixed number of processes is observationally equivalent to the parameterized system with unboundedly many processes. Computing such a *large enough constant* is undecidable after all, since the parameterized verification problem is undecidable [17]. It is difficult to apply model checking to the problem directly.

Since parameterized systems contain a large number of behaviorally similar processes, a natural state space abstraction is to group the processes based on which state of the local transition systems they reside in [167, 63, 168]. Thus, instead of saying “process 1 and 3 are in state s, and process 2 is in state t", we simply say “2 processes are in state s and 1 is in state t". Such an abstraction reduces the state space by exploiting a powerful state space symmetry, as often evidenced in real-life concurrent systems such as a caches, mutual exclusion protocols and network protocols. Verification by traversing the abstract state space here produces a sound and complete verification procedure. However, if the total number of processes is unbounded, the aforementioned counter abstraction still does not produce a finite state abstract system. The count of processes in a local state can still be ω (unbounded number), if the total number of processes is ω. We can adopt a *cutoff number*, so that any count greater than the *cutoff* number is abstracted to ω. This yields a finite state abstract system, which allows sound but incomplete verification procedure, e.g., any LTL property verified in the abstract system holds for all concrete finite-state instances of the system, but not vice-versa.

In this chapter, we develop a novel technique for model checking parameterized systems under fairness, against LTL formulae. We show that model checking under fairness is feasible, even without the knowledge of process identifiers. This is done by systematically keeping track of the local states from which actions are enabled / executed within any infinite loop of the abstract state space. We develop necessary theorems to prove the soundness of our technique, and also present efficient on-the-fly model checking algorithms.

5.4.1 System Models

We begin with formally defining our system model, which is a specialized one (compared to Definition 3 in Section 3.1.1) that the system process is a parallel composition of identity processes.

Definition 16 (System Model) *A system model is a structure $\mathcal{S} = (Var_G, init_G, Proc)$ where Var_G is a finite set of global variables, $init_G$ is their initial valuation and $Proc$ is a parallel composition of multiple processes $Proc = P_1 \parallel P_2 \parallel \cdots$ such that each process $P_i = (S_i, init_i, \rightarrow_i)$ is a labeled transition system.*

We assume that all global variables have finite domains and each P_i has finitely many local states. A local state represents a program text together with its local context (e.g. valuation of the local variables). Two local states are equivalent if and only if they represent the same program text and the same local context. Let $State$ be the set of all local states. We assume that $State$ has finitely

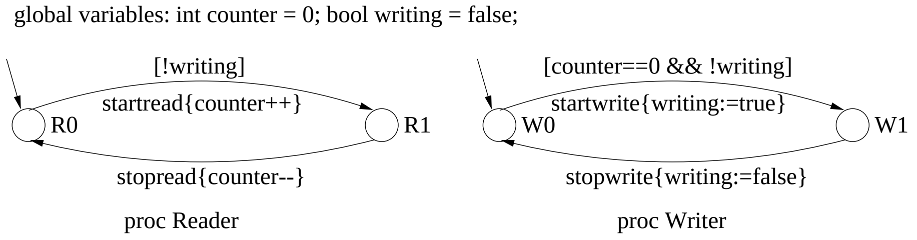

global variables: int counter = 0; bool writing = false;
[!writing]
startread{counter++}
R0
R1
stopread{counter--}
proc Reader
[counter==0 && !writing]
startwrite{writing:=true}
W0
W1
stopwrite{writing:=false}
proc Writer

Figure 5.2: Readers/writers model

many elements. This disallows unbounded non-tail recursion which results in infinite different local states. $Proc$ may be composed of infinitely many processes. Each process has a unique identifier. In an abuse of notation, we use P_i to represent the identifier of process P_i when the context is clear. Notice that two local states from different processes are equivalent only if the process identifiers are irrelevant to the program texts they represent. Processes may communicate through global variables, (multi-party) barrier synchronization or synchronous/asynchronous message passing. It can be shown that parallel composition || is symmetric and associative.

Example 5.4.1 Figure 5.2 shows a model of the readers/writers problem, which is a simple protocol for the coordination of readers and writers accessing a shared resource. The protocol, referred as RW, is designed for arbitrary number of readers and writers. Several readers can read concurrently, whereas writers require exclusive access. Global variable $counter$ records the number of readers which are currently accessing the resource; $writing$ is true if and only if a writer is updating the resource. A transition is of the form $[guard]name\{assignments\}$, where $guard$ is a guard condition which must be true for the transition to be taken and $assignments$ is a simple sequential program which updates global variables. The following are properties which are to be verified.

$$\Box !(counter > 0 \wedge writing) \qquad – Prop_1$$
$$\Box \Diamond counter > 0 \qquad – Prop_2$$

Property $Prop_1$ is a safety property which states that writing and reading cannot occur simultaneously. Property $Prop_2$ is a liveness property which states that always eventually the resource can be accessed by some reader. **end**

In order to define the operational semantics of this specialized system model, we re-define the notion of system configuration defined in Definition 4 in Section 3.1.2, which is referred to as *concrete configurations*. This terminology distinguishes the notion from the state space abstraction and the abstract configurations which will be introduced later.

Definition 17 (Concrete Configuration) *Let $\mathcal{S}$ be a system model. A concrete configuration of $\mathcal{S}$ is a pair $(v, \langle s_1, s_2, \cdots \rangle)$ where v is the valuation of the global variables (channel buffers may be viewed as global variables), and $s_i \in S_i$ is the local state in which process P_i is residing.*

A system transition is of the form $(v, \langle s_1, s_2, \cdots \rangle) \rightarrow_{Ag} (v', \langle s_1', s_2', \cdots \rangle)$ where the system configuration after the transition is $(v', \langle s_1', s_2', \cdots \rangle)$ and Ag is a set of participating processes. For simplicity, set Ag (short for $agent$) is often omitted if irreverent. A system transition could be one of the following forms:

(i) a local transition of P_i which updates its local state (from s_i to s_i') and possibly updating global variables (from v to v'). An example is the transition from $R0$ to $R1$ of a reader. In such a case, P_i is the participating process, i.e., $Ag = \{P_i\}$.

(ii) a multi-party synchronous transition among processes $P_i, \cdots, P_j$. Examples are message sending/receiving through channels with buffer size 0 (e.g., as in Promela [111]) and alphabetized barrier synchronization in the classic CSP. In such a case, local states of the participating processes are updated simultaneously. The participating processes are $P_i, \cdots, P_j$.

(iii) process creation of P_m by P_i. In such a case, an additional local state is appended to the sequence $\langle s_1, s_2, \cdots \rangle$, and the state of P_i is changed at the same time. Assume for now that the sequence $\langle s_1, s_2, \cdots \rangle$ is always finite before process creation. It becomes clear in Section 5.6 that this assumption is not necessary. In such a case, the participating processes are P_i and P_m.

(iv) process deletion of P_i. In such case, the local state of P_i is removed from the sequence $(\langle s_1, s_2, \cdots \rangle)$. The participating process is P_i.

Definition 18 (Concrete Transition System) *Let $\mathcal{S} = (Var_G, init_G, Proc)$ be a system model,* *where $Proc = P_1 \parallel P_2 \parallel \cdots$ such that each process $P_i = (S_i, init_i, \rightarrow_i)$ is a local transition system. The concrete transition system corresponding to $\mathcal{S}$ is a 3-tuple $T_{\mathcal{S}} = (C, init, \hookrightarrow)$ where C is the set of all reachable system configurations, $init$ is the initial concrete configuration $(init_G, \langle init_1, init_2, \cdots \rangle)$ and $\hookrightarrow$ is the global transition relation obtained by composing the local transition relations $\rightarrow_i$ in parallel.*

An execution of $\mathcal{S}$ is an infinite sequence of configurations $E = \langle c_0, c_1, \cdots, c_i, \cdots \rangle$ where $c_0 = init$ and $c_i \hookrightarrow c_{i+1}$ for all $i \geq 0$. Given an execution in the setting of parameterized system, we can define *process-level weak fairness* (see Definition 8), *process-level strong fairness* (see Definition 10) and *strong global fairness* (see Definition 11) in a similar way as in Section 4.2.

Given the RW model presented in Figure 5.2, it can be shown that $RW \vDash Prop_1$. It is, however, not easy to prove it using standard model checking techniques. The challenge is that many or unbounded number of readers and writers cause state space explosion. Also, RW fails $Prop_2$ without fairness constraint. For instance, a counterexample is $\langle startwrite, stopwrite \rangle^{\infty}$, i.e., a writer keeps updating the resource without any reader ever accessing it. This is unreasonable if the system scheduler is well-designed or the processors that the readers/writers execute on have comparable speed. To avoid such counterexamples, we need to perform model checking under fairness.

5.4.2 Process Counter Representation

Parameterized systems contain behaviorally similar or even identical processes. Given a configuration $(v, \langle \cdots, s_i, \cdots, s_j, \cdots \rangle)$, multiple local states[4] may be equivalent. A natural "abstraction" is to record only how many copies of a local state are there.

Let $\mathcal{S}$ be a system model. An alternative representation of a concrete configuration is a pair (v, f) where v is the valuation of the global variables and f is a total function from a local state to the set of processes residing at the state. For instance, given that $R0$ is a local state in Figure 5.2, $f(R0) = \{P_i, P_j, P_k\}$ if and only if reader processes P_i, P_j and P_k are residing at state $R0$. This

[4] The processes residing at the local states may or may not have the same process type.

representation is sound and complete because processes at equivalent local states are behavioral equivalent and $\|$ composition is symmetric and associative (so that processes ordering is irrelevant).

Furthermore, given a local state s and processes residing at s, we may consider the processes indistinguishable (as the process identifiers must be irrelevant given the local states are equivalent) and abstract the process identifiers. That is, instead of associating a set of process identifiers with a local state, we only count the number of processes. Instead of setting $f(R0) = \{P_i, P_j, P_k\}$, we now set $f(R0) = 3$. *In this and the next section, we assume that the total number of processes is bounded.*

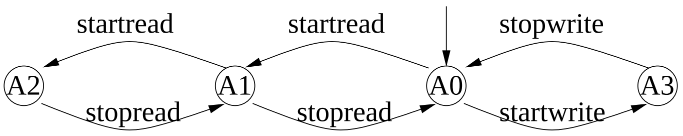
Definition 19 (Abstract Configuration) *Let $\mathcal{S}$ be a system model. An abstract configuration of $\mathcal{S}$ is a pair (v, f) where v is a valuation of the global variables and $f : State \rightarrow \mathbb{N}$ is a total function*[5] *such that $f(s) = n$ if and only if n processes are residing at s.*

Given a concrete configuration $cc = (v, \langle s_0, s_1, \cdots \rangle)$, let $\mathcal{F}(\langle s_0, s_1, \cdots \rangle)$ returns the function f (refer to Definition 19) — that is, $f(s) = n$ if and only if there are n states in $\langle s_0, s_1, \cdots \rangle$ which are equivalent to s. Further, we write $\mathcal{F}(cc)$ to denote $(v, \mathcal{F}(\langle s_0, s_1, \cdots \rangle))$. Given a concrete transition $c \rightarrow_{Ag} c'$, the corresponding abstraction transition is written as $a \hookrightarrow_{Ls} a'$ where $a = \mathcal{F}(c)$ and $a' = \mathcal{F}(c')$ and Ls (short for local-states) is the local states at which processes in Ag are. That is, Ls is the set of local states from which there is a process leaving during the transition. We remark that Ls is obtained similarly as Ag is.

Given a state s and an abstract configuration a, $enabled(s, a)$ *to be true if and only if* $\exists\, a',\ a \hookrightarrow_{Ls} a' \wedge s \in Ls$, i.e., a process is enabled to leave s in a. For instance, given the transition system in Figure 5.3, $Ls = \{R0\}$ for the transition from $A0$ to $A1$ and $enabled(R0, A1)$ is true.

Definition 20 (Abstract Transition System) *Let $\mathcal{S} = (Var_G, init_G, Proc)$ be a system model, where $Proc = P_1 \parallel P_2 \parallel \cdots$ such that each process $P_i = (S_i, init_i, \rightarrow_i)$ is a local transition system. An abstract transition system of $\mathcal{S}$ is a 3-tuple $A_{\mathcal{S}} = (C, init, \hookrightarrow)$ where C is the set of all reachable abstract system configurations, $init \in C$ is $(init_G, \mathcal{F}(init_G, \langle init_1, init_2, \cdots \rangle))$ and $\hookrightarrow$ is the abstract global transition relation.*

[5] In PAT, the mapping from a local state to 0 is always omitted for memory saving.



A0: ((writing,false),(counter,0),(R0,2),(R1,0),(W0,2),(W1,0))
A1: ((writing,false),(counter,1),(R0,1),(R1,1),(W0,2),(W1,0))
A2: ((writing,false),(counter,2),(R0,0),(R1,2),(W0,2),(W1,0))
A3: ((writing,true),(counter,0),(R0,2),(R1,0),(W0,1),(W1,1))

Figure 5.3: Readers/writers model

We remark that the abstract transition relation can be constructed *without* constructing the concrete transition relation, which is essential to avoid state space explosion. Given the model in Figure 5.2, if there are 2 readers and 2 writers, then the abstract transition system is shown in Figure 5.3.

A concrete execution of $T_{\mathcal{S}}$ can be uniquely mapped to an execution of $A_{\mathcal{S}}$ by applying $\mathcal{F}$ to every configuration in the sequence. For instance, let $X = \langle c_0, c_1, \cdots, c_i, \cdots \rangle$ be an execution of $T_{\mathcal{S}}$ (i.e., a concrete execution), the mapped execution of $A_{\mathcal{S}}$ is $L = \langle \mathcal{F}(c_0), \mathcal{F}(c_1), \cdots, \mathcal{F}(c_i), \cdots \rangle$ (i.e., the abstract execution). In an abuse of notation, we write $\mathcal{F}(X)$ to denote L. Notice that the valuation of the global variables are preserved. Essentially, no information is lost during the abstraction. It can be shown that $A_{\mathcal{S}} \vDash \phi$ if and only if $T_{\mathcal{S}} \vDash \phi$.

5.5 Fair Model Checking Algorithm with Counter Abstraction

Process counter abstraction may significantly reduce the number of states. It is useful for verifying safety properties. However, it conflicts with the notion of fairness. A counterexample to a liveness property under fairness must be a fair execution of the system. By Definition 8 and 10, the knowledge of which processes are enabled or engaged is necessary in order to check whether an execution is fair or not. In this section, we develop the necessary theorems and algorithms to show that model checking under fairness constraints is feasible even without the knowledge of process identifiers.

By assumption the total number of processes is finite, the abstract transition system $A_{\mathcal{S}}$ has finitely many states. An infinite execution of $A_{\mathcal{S}}$ must form a loop (with a finite prefix). Assume that the loop starts with index i and ends with k, written as $L_i^k = \langle c_0, \cdots, c_i, c_{i+1}, \cdots, c_{i+k}, c_{i+k+1}\rangle$ where $c_{i+k+1} = c_i$. We define the following functions to collect loop properties and use them to define fairness later.

$$\begin{array}{ll} \mathit{always}(L_i^k) & = \{s : \mathit{State} \mid \forall\, j : \{i, \cdots, i+k\},\ \mathit{enabled}(s, c_j)\} \\ \mathit{once}(L_i^k) & = \{s : \mathit{State} \mid \exists\, j : \{i, \cdots, i+k\},\ \mathit{enabled}(s, c_j)\} \\ \mathit{leave}(L_i^k) & = \{s : \mathit{State} \mid \exists\, j : \{i, \cdots, i+k\},\ c_j \hookrightarrow_{Ls} c_{j+1} \wedge s \in Ls\} \end{array}$$

Intuitively, $\mathit{always}(L_i^k)$ is the set of local states from where there are processes, which are ready to make some progress, throughout the execution of the loop; $\mathit{once}(L_i^k)$ is the set of local states where there is a process which is ready to make some progress, at least once during the execution of the loop; $\mathit{leave}(L_i^k)$ is the set of local states from which processes leave during the loop. For instance, given the abstract transition system in Figure 5.3, $X = \langle A0, A1, A2\rangle^\infty$ is a loop starting with index 0 and ending with index 2. $\mathit{always}(X) = \varnothing$; $\mathit{once}(X) = \{R0, R1, W0\}$; $\mathit{leave}(X) = \{R0, R1\}$.

The following lemma checks whether an execution is fair by only looking at the abstract execution.

Lemma 5.5.1 *Let $\mathcal{S}$ be a system model; X be an execution of $T_\mathcal{S}$; $L_i^k = \mathcal{F}(X)$ be the respective abstract execution of $A_\mathcal{S}$. (1). $\mathit{always}(L_i^k) \subseteq \mathit{leave}(L_i^k)$ if X is weakly fair; (2). $\mathit{once}(L_i^k) \subseteq \mathit{leave}(L_i^k)$ if X is strongly fair.*

Proof: (1). Assume X is weakly fair. By definition, if state s is in $\mathit{always}(L_i^k)$, there must be a process residing at s which is enabled to leave during every step of the loop. If it is the same process P, P is always enabled during the loop and therefore, by Definition 8, P must participate in a transition infinitely often because X is weakly fair. Therefore, P must leave s during the loop. By definition, s must be in $\mathit{leave}(L_i^k)$. If there are different processes enabled at s during the loop, there must be a process leaving s, so that $s \in \mathit{leave}(L_i^k)$. Thus, $\mathit{always}(L_i^k) \subseteq \mathit{leave}(L_i^k)$.

(2). Assume X is strongly fair. By definition, if state s is in $\mathit{once}(L_i^k)$, there must be a process residing at s which is enabled to leave during one step of the loop. Let P be the process. Because P is infinitely often enabled, by Definition 8, P must participate in a transition infinitely often because X is strongly fair. Therefore, P must leave s during the loop. By definition, s must be in $\mathit{leave}(L_i^k)$.
□

The following lemma generates a concrete fair execution if an abstract *fair* execution is identified.

Lemma 5.5.2 *Let $\mathcal{S}$ be a model; L_i^k be an execution of $A_{\mathcal{S}}$. (1). There exists a weakly fair execution X of $T_{\mathcal{S}}$ such that $\mathcal{F}(X) = L_i^k$ if $always(L_i^k) \subseteq leave(L_i^k)$; (2). If $once(L_i^k) \subseteq leave(L_i^k)$, there exists a strongly fair execution X of $T_{\mathcal{S}}$ such that $\mathcal{F}(X) = L_i^k$.*

Proof: (1). By a simple argument, there must exist an execution X of $T_{\mathcal{S}}$ such that $\mathcal{F}(X) = L_i^k$. Next, we show that we can unfold the loop (of the abstract fair execution) as many times as necessary to let all processes make some progress, so as to generate a weakly fair concrete execution. Assume P is the set of processes residing at a state s during the loop. Because $always(L_i^k) \subseteq leave(L_i^k)$, if $s \in always(L_i^k)$, there must be a transition during which a process leaves s. We repeat the loop multiple times and choose a different process from P to leave each time. The generated execution must be weakly fair.

(2). Similarly as above. □

The following theorem shows that we can perform model checking under fairness by examining the abstract transition system only.

Theorem 5.5.3 *Let $\mathcal{S}$ be a system model. Let ϕ be a LTL property. (1). $\mathcal{S} \vDash_{wf} \phi$ if and only if for all executions L_i^k of $A_{\mathcal{S}}$ we have $always(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$; (2). $\mathcal{S} \vDash_{sf} \phi$ if and only if for all execution L_i^k of $A_{\mathcal{S}}$ we have $once(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$.*

Proof: (1). **if** part: Assume that for all L_i^k of $A_{\mathcal{S}}$ we have $L_i^k \vDash \phi$ if $always(L_i^k) \subseteq leave(L_i^k)$, and $\mathcal{S} \nvDash_{wf} \phi$. By definition, there exists a weakly fair execution X of $T_{\mathcal{S}}$ such that $X \nvDash \phi$. Let L_i^k be $\mathcal{F}(X)$. By lemma 5.5.1, $always(L_i^k) \subseteq leave(L_i^k)$ and hence $L_i^k \vDash \phi$. Because our abstraction preserves valuation of global variables, $L_i^k \nvDash \phi$ as $X \nvDash \phi$. We reach a contradiction.
only if part: Assume that $\mathcal{S} \vDash_{wf} \phi$ and there exists L_i^k of $A_{\mathcal{S}}$ such that $always(L_i^k) \subseteq leave(L_i^k)$, and $L_i^k \nvDash_{wf} \phi$. By lemma 5.5.2, there must exist X of $T_{\mathcal{S}}$ such that X is weakly fair. Because process counter abstraction preserves valuations of global variables, $X \nvDash \phi$. Hence, we reach contradiction.

(2). Similarly as above. □

Thus, in order to prove that $\mathcal{S}$ satisfies ϕ under fairness, we need to show that there is no execution L_i^k of $A_{\mathcal{S}}$ such that $L_i^k \not\models \phi$ and the execution satisfies an additional constraint for fairness, i.e., $always(L_i^k) \subseteq leave(L_i^k)$ for weak fairness or $once(L_i^k) \subseteq leave(L_i^k)$ for strong fairness. Or, if $\mathcal{S} \not\models_{wf} \phi$, then there must be an execution L_i^k of $A_{\mathcal{S}}$ such that L_i^k satisfies the fairness condition and $L_i^k \not\models \phi$. In such a case, we can generate a concrete execution.

Following the above discussion, fair model checking parameterized systems is reduced to searching for particular loops in $A_{\mathcal{S}}$. We amend those SCC based algorithms presented in Chapter 4 to cope with weak or strong fairness and process counter abstraction. Given $A_{\mathcal{S}}$ and a property ϕ, model checking involves searching for an execution of $A_{\mathcal{S}}$ which fails ϕ. In automata-based model checking, the negation of ϕ is translated to an equivalent Büchi automaton $\mathcal{B}_{\neg\,\phi}$, which is then composed with $A_{\mathcal{S}}$. Notice that a state in the produce of $A_{\mathcal{S}}$ and $\mathcal{B}_{\neg\,\phi}$ is a pair (a, b) where a is an abstract configuration of $A_{\mathcal{S}}$ and b is a state of $\mathcal{B}_{\neg\,\phi}$.

Given a transition system, a strongly connected subgraph is a subgraph such that there is a path connecting any two states in the subgraph. An MSCC is a maximal strongly connected subgraph. Given the product of $A_{\mathcal{S}}$ and $\mathcal{B}_{\neg\,\phi}$, let scg be a set of states which, together with the transitions among them, forms a strongly connected subgraph. We say scg is accepting if and only if there exists one state (a, b) in scg such that b is an accepting state of $\mathcal{B}_{\neg\,\phi}$. In an abuse of notation, we refer to scg as the strongly connected subgraph in the following. The following lifts the previously defined functions on loops to strongly connected subgraphs.

$$\begin{array}{ll} always(scg) & = \{y : State \mid \forall\, x : scg,\ enabled(y, x)\} \\ once(scg) & = \{y : State \mid \exists\, x : scg,\ enabled(y, x)\} \\ leave(scg) & = \{z : State \mid \exists\, x, y : scg,\ z \in leave(x, y)\} \end{array}$$

$always(scg)$ is the set of local states such that for any local state in $always(scg)$, there is a process ready to leave the local state for every state in scg; $once(scg)$ is the set of local states such that for some local state in $once(scg)$, there is a process ready to leave the local state for some state in scg; and $leave(scg)$ is the set of local states such that there is a transition in scg during which there is a process leaving the local state. Given the abstract transition system in Figure 5.3, $scg = \{A0, A1, A2, A3\}$ constitutes a strongly connected subgraph. $always(scg)$ = nil; $once(scg) = \{R0, R1, W0, W1\}$; $leave(scg) = \{R0, R1, W0, W1\}$.

```
procedure $checkingUnderWeakFairness(A_{\mathcal{S}}, \mathcal{B}_{\neg\,\phi})$
while there are un-visited states in $A_{\mathcal{S}} \otimes \mathcal{B}_{\neg\,\phi}$
    use the improved Tarjan's algorithm to identify one SCC, say $scg$;
    if $scg$ is accepting to $\mathcal{B}_{\neg\,\phi}$ and $always(scg) \subseteq leave(scg)$
        generate a counterexample and return false;
    endif
endwhile
return $true$;
```

Figure 5.4: Model checking algorithm under weak fairness

Lemma 5.5.4 *Let $\mathcal{S}$ be a system model. There exists an execution L_i^k of $A_{\mathcal{S}}$ such that $always(L_i^k) \subseteq leave(L_i^k)$ if and only if there exists an MSCC scc of $A_{\mathcal{S}}$ such that $always(scc) \subseteq leave(scc)$.*

Proof: The **if** part is trivially true. The **only if** part is proved as follows. Assume there exists execution L_i^k of $A_{\mathcal{S}}$ such that $always(L_i^k) \subseteq leave(L_i^k)$, there must exist a strongly connected subgraph scg which satisfies $always(scg) \subseteq leave(scg)$. Let scc be the MSCC which contains scg. We have $always(scc) \subseteq always(scg)$, therefore, the MSCC scc satisfies $always(scc) \subseteq always(scg) \subseteq leave(scg) \subseteq leave(scc)$. □

The above lemma allows us to use MSCC detection algorithms for model checking under weak fairness. Figure 5.4 presents an on-the-fly model checking algorithm based on Tarjan's algorithm for identifying MSCCs. The idea is to search for an MSCC scg such that $always(scg) \subseteq leave(scg)$ and scg is accepting. The algorithm terminates in two ways, either one such MSCC is found or all MSCCs have been examined (and it returns true). In the former case, an abstract counterexample is generated. In the latter case, we successfully prove the property. Given the system presented in Figure 5.3, $\{A0, A1, A2, A3\}$ constitutes the only MSCC, which satisfies $always(scg) \subseteq leave(scg)$. The complexity of the algorithm is linear in the number of transitions of $A_{\mathcal{S}}$.

Lemma 5.5.5 *Let $\mathcal{S}$ be a system model. There exists an execution L_i^k of $A_{\mathcal{S}}$ such that $once(L_i^k) \subseteq leave(L_i^k)$ if and only if there exists a strongly connected subgraph scg of $A_{\mathcal{S}}$ such that $once(scg) \subseteq leave(scg)$.*

```
procedure $checkingUnderStrongFairness(A_{\mathcal{S}}, \mathcal{B}_{\neg\phi}, states)$
1. while there are un-visited states in $states$
2.     use Tarjan's algorithm to identify a subset of $states$ which forms a SCC, say $scg$;
3.     if $scg$ is accepting to $\mathcal{B}_{\neg\phi}$
4.         if $once(scg) \subseteq leave(scg)$
5.             generate a counterexample and return false;
6.         else if $checkingUnderStrongFairness(A_{\mathcal{S}}, \mathcal{B}_{\neg\phi}, scg \setminus bad(scg))$ is false
7.             return false;
8.         endif
9.     endif
10. endwhile
11. return $true$;
```

Figure 5.5: Model checking algorithm under strong fairness

We skip the proof of the lemma as it is straightforward. The lemma allows us to extend the algorithm proposed in [202] for model checking under strong fairness. Figure 5.5 presents the modified algorithm. The idea is to search for a strongly connected subgraph scg such that $once(scg) \subseteq leave(scg)$ and scg is accepting. Notice that a strongly connected subgraph must be contained in one and only one MSCC. The algorithm searches for MSCCs using Tarjan's algorithm. Once an MSCC scg is found (at line 2), if scg is accepting and satisfies $once(scg) \subseteq leave(scg)$, then we generate an abstract counterexample. If scg is accepting but fails $once(scg) \subseteq leave(scg)$, instead of throwing away the MSCC, we prune a set of *bad states* from the SCC and then examinate the remaining states (at line 6) for strongly connected subgraphs. Intuitively, *bad states* are the reasons why the SCC fails the condition $once(scg) \subseteq leave(scg)$. Formally,

$$bad(scg) = \{x : scg \mid \exists\, y,\ y \notin leave(scg) \wedge y \in enabled(y, x)\}$$

That is, a state s is bad if and only if there exists a local state y such that a process may leave y at state s and yet there is no process leaving y given all transitions in scg. By pruning all bad states, there may be a strongly connected subgraph in the remaining states satisfying the fairness constraint.

The algorithm is partly inspired by the one presented in [106] for checking emptiness of Streett automata. Soundness of the algorithm follows the discussion in [202, 106]. It can be shown that any state of a strongly connected subgraph which satisfies the constraints is never pruned. As a result, if there exists such a strongly connected subgraph scg, a strongly connected subgraph which contains scg or scg itself must be found eventually. Termination of the algorithm is guaranteed because the number of visited states and pruned states are monotonically increasing. The complexity of the algorithm is linear in $\#states \times \#trans$ where $\#states$ and $\#trans$ are the number of states and transitions of $A_{\mathcal{S}}$ respectively. A tighter bound on the complexity can be found in [106].

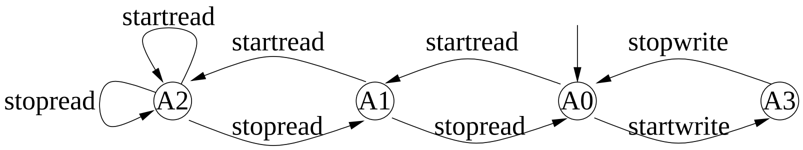
5.6 Counter Abstraction for Infinitely Many Processes

In the previous sections, we assume that the number of processes (and hence the size of the abstract transition system) is finite and bounded. If the number of processes is unbounded, there might be unbounded number of processes residing at a local state, e.g., the number of reader processes residing at $R0$ in Figure 5.2 might be infinite. In such a case, we choose a $cutoff$ number and then apply further abstraction. In the following, we *modify* the definition of abstract configurations and abstract transition systems to handle unbounded number of processes.

Definition 21 *Let $\mathcal{S}$ be a system model with unboundedly many processes. Let K be a positive natural number (i.e., the cutoff number). An abstract configuration of $\mathcal{S}$ is a pair (v, g) where v is the valuation of the global variables and $g : State \rightarrow \mathbb{N} \cup \{\omega\}$ is a total function such that $g(s) = n$ if and only if $n (\leq K)$ processes are residing at s and $g(s) = \omega$ if and only if more than K processes are at s.*

Given a configuration $(v, \langle s_0, s_1, \cdots \rangle)$, we define a function $\mathcal{G}$ similar to function $\mathcal{F}$, i.e., $\mathcal{G}(\langle s_0, s_1, \cdots \rangle))$ returns function g (refer to Definition 21) such that given any state s, $g(s) = n$ if and only if there are n states in $\langle s_0, s_1, \cdots \rangle$ which are equivalent to s and $g(s) = \omega$ if and only if there are more than K states in $\langle s_0, s_1, \cdots \rangle$ which are equivalent to s. Furthermore, $\mathcal{G}(c) = (v, \mathcal{G}(\langle s_0, s_1, \cdots \rangle))$.

The abstract transition relation of $\mathcal{S}$ (as per the above abstraction) can be constructed without constructing the concrete transition relation. We illustrate how to generate an abstract transition in the following. Given an abstract configuration (v, g), if $g(s) > 0$, a *local transition* from state s to state



A0: ((writing,false),(counter,0),(R0,inf),(R1,0),(W0,inf),(W1,0))
A1: ((writing,false),(counter,1),(R0,inf),(R1,1),(W0,inf),(W1,0))
A2: ((writing,false),(counter,inf),(R0,inf),(R1,inf),(W0,inf),(W1,0))
A3: ((writing,true),(counter,0),(R0,inf),(R1,0),(W0,inf),(W1,1))

Figure 5.6: Abstract readers/writers model

s', creating a process with initial state $init$ may result in different abstract configurations (v, g') depending on g. In particular, g' equals g except that $g'(s) = g(s) - 1$ and $g'(s') = g(s') + 1$ and $g'(init) = g(init) + 1$ assuming $\omega + 1 = \omega$, $K + 1 = \omega$ and $\omega - 1$ is either ω or K. We remark that by assumption $State$ is a finite set and therefore the domain of g is always finite. This allows us to drop the assumption that the number of processes must be finite before process creation. Similarly, we abstract synchronous transitions and process termination.

The *abstract transition system* for a system model $\mathcal{S}$ with unboundedly many processes, written as $R_{\mathcal{S}}$ (to distinguish from $A_{\mathcal{S}}$), is now obtained by applying the aforementioned abstract transition relation from the initial abstract configuration.

Example 5.6.1 Assume that the cutoff number is 1 and there are infinitely many readers and writers in the readers/writers model. Because $counter$ is potentially unbounded and, we mark $counter$ as a special process counter variable which dynamically counts the number of processes which are reading (at state $R1$). If the number of *reading* processes is larger than the cutoff number, $counter$ is set to ω too. The abstract transition system A_{RW} is shown in Figure 5.6. The abstract transition system may contain spurious traces. For instance, the trace $\langle start, (stopread)^{\infty} \rangle$ is spurious. It is straightforward to prove that $A_{RW} \vDash Prop_1$ based on the abstract transition system. **end**

The abstract transition system now has only finitely many states even if there are unbounded number of processes and, therefore, is subject to model checking. As illustrated in the preceding example, the abstraction is sound but incomplete in the presence of unboundedly many processes. Given an execution X of $T_{\mathcal{S}}$, let $\mathcal{G}(X)$ be the corresponding execution of the abstract transition system. An execution L of $R_{\mathcal{S}}$ is spurious if and only if there does not exist an execution X of $T_{\mathcal{S}}$ such that $\mathcal{G}(X) = L$. Because the abstraction only introduces execution traces (but does not remove any), we can formally establish a simulation relation (but not a bisimulation) between the abstract and concrete transition systems, that is, $R_{\mathcal{S}}$ simulates $T_{\mathcal{S}}$. Thus, while verifying a LTL property ϕ we can conclude $T_{\mathcal{S}} \vDash \phi$ if we can show that $R_{\mathcal{S}} \vDash \phi$. Of course, $R_{\mathcal{S}} \vDash \phi$ will be accomplished by model checking under fairness.

The following re-establishes Lemma 5.5.1 and (part of) Theorem 5.5.3 in the setting of $R_{\mathcal{S}}$. We skip the proof as they are similar to that of Lemma 5.5.1 and Theorem 5.5.3 respectively.

Lemma 5.6.2 *Let $\mathcal{S}$ be a system model, X be an execution of $T_{\mathcal{S}}$ and $L_i^k = \mathcal{G}(X)$ be the corresponding execution of $R_{\mathcal{S}}$. We have (1). $always(L_i^k) \subseteq leave(L_i^k)$ if X is weakly fair; $(2).once(L_i^k)$ $\subseteq leave(L_i^k)$ if X is strongly fair.*

Theorem 5.6.3 *Let $\mathcal{S}$ be a system model and ϕ be a LTL property. (1). $\mathcal{S} \vDash_{wf} \phi$ if for all execution traces L_i^k of $R_{\mathcal{S}}$ we have $always(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$; (2). $\mathcal{S} \vDash_{sf} \phi$ if for all execution traces L_i^k of $R_{\mathcal{S}}$ we have $once(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$;*

The reverse of Theorem 5.6.3 is not true because of spurious traces. We remark that the model checking algorithms presented in Section 5.5 are applicable to $R_{\mathcal{S}}$ (as the abstraction function is irrelevant to the algorithm). By Theorem 5.6.3, if model checking of $R_{\mathcal{S}}$ (using the algorithms presented in Section 5.5 under weak/fairness constraint) returns true, we conclude that the system satisfies the property (under the respective fairness).

5.7 Experiments of Process Counter Abstraction

In this section, we conduct experiments on real-life systems to demonstrate the effectiveness of the process counter abstraction technique. The experimental results are summarized in the Table 5.2, where NA means not applicable (hence not tried, due to limit of the tool); NF means not feasible (out of 2GB memory or running for more than 4 hours). The data is obtained with Intel 9550 CPU at

Model	#Proc	Property	No Fairness			Weak Fairness			Strong Fairness		
			Result	PAT	SPIN	Result	PAT	SPIN	Result	PAT	SPIN
LE	10	$\Diamond\Box\ one\ leader$	false	0.04	0.02	true	0.06	320	true	0.06	NA
LE	100	$\Diamond\Box\ one\ leader$	false	0.04	0.02	true	0.27	NF	true	0.28	NA
LE	1000	$\Diamond\Box\ one\ leader$	false	0.04	NA	true	2.26	NA	true	2.75	NA
LE	10000	$\Diamond\Box\ one\ leader$	false	0.04	NA	true	23.9	NA	true	68.8	NA
LE	∞	$\Diamond\Box\ one\ leader$	false	0.06	NA	true	265	NA	true	464	NA
KV	2	$Prop_{Kvalue}$	false	0.05	0	true	0.6	1.14	true	0.6	NA
KV	3	$Prop_{Kvalue}$	false	0.05	0	true	4.56	61.2	true	4.59	NA
KV	4	$Prop_{Kvalue}$	false	0.05	0.02	true	29.2	NF	true	30.2	NA
KV	5	$Prop_{Kvalue}$	false	0.06	0.02	true	175	NF	true	187	NA
KV	∞	$Prop_{Kvalue}$	false	0.12	NA	?	NF	NA	?	NF	NA
$Stack$	5	$Prop_{stack}$	false	0.06	0.02	false	0.78	NF	false	0.74	NA
$Stack$	7	$Prop_{stack}$	false	0.06	0.02	false	11.3	NF	false	12.1	NA
$Stack$	9	$Prop_{stack}$	false	0.06	0.02	false	159	NF	false	192	NA
$Stack$	10	$Prop_{stack}$	false	0.05	0.02	false	596	NF	false	780	NA
ML	10	$\Box\Diamond\ access$	true	0.11	21.5	true	0.11	107	true	0.11	NA
ML	100	$\Box\Diamond\ access$	true	1.04	NF	true	1.04	NF	true	1.04	NA
ML	1000	$\Box\Diamond\ access$	true	11.0	NA	true	11.1	NA	true	11.1	NA
ML	∞	$\Box\Diamond\ access$	true	13.8	NA	true	13.8	NA	true	13.8	NA

Table 5.2: Experimental results of process counter abstraction

2.83GHz and 2GB RAM. We compared PAT with SPIN [111] on model checking under no fairness or weak fairness. Notice that SPIN does not support strong fairness and is limited to 255 processes.

The first model (LE) is a self-stabilizing leader election protocol for complete networks (i.e., any pair of nodes are connected) [88]. The property is that eventually always there is one and only one leader in the network, i.e., $\Diamond\Box\ one\ leader$. PAT successfully proved the property under weak or strong fairness for many or unbounded number of network nodes (with cutoff number 2). SPIN took much more time to prove the property under weak fairness. The reason is that the fair model checking algorithm in SPIN copies the global state machine $n+2$ times (for n processes) so as to give each process a fair chance to progress, which increases the verification time by a factor that is linear in the number of network nodes.

The second model (KV) is a K-valued register [22]. A shared K-valued multi-reader single-writer register R can be simulated by an array of K binary registers. The complete model can be found in Example 7.2.1 in Section 7.2.1. A *progress* property we tested is that $Prop_{Kvalue} = \Box(read\_inv \rightarrow \Diamond read\_res)$, i.e., a reading operation ($read\_inv$) eventually returns some valid value ($read\_res$). With no fairness, both PAT and SPIN identified a counterexample quickly. Because the model contains many local states, the size of A_S increases rapidly. PAT proved the property under weak/strong fairness for 5 processes, whereas SPIN was limited to 3 processes with weak fairness.

The third model ($Stack$) is a lock-free stack [210]. In concurrent systems, in order to improve the performance, stacks can be implemented by a linked list shared by arbitrary number of processes. Each push or pop operation keeps trying to update the stack until no other process interrupts. The property of interest is that a process must eventually be able to update the stack, which can be expressed as the LTL $Prop_{stack} = \Box(push\_inv \rightarrow \Diamond push\_res)$ where event $push\_inv$ ($push\_res$) marks the starting (ending) of $push$ operation. The property is false even under strong fairness.

The fourth model (ML) is the Java meta-lock algorithm [5]. In Java language, any object can be synchronized by different threads via synchronized methods or statements. The Java meta-locking algorithm is designed to ensure the mutually exclusive access to an object. A synchronized method first acquires a lock on the object, executes the method and then releases the lock. The property is that always eventually some thread is accessing the object, i.e., $\Box\Diamond\ access$, which is true without fairness. This example shows that the computational overhead due to fairness is negligible in PAT.

In another experiment, we use a model in which processes all behave differently (so that counter abstraction results in no reduction) and each process has many local states. We then compare the verification results with or without process counter abstraction. The result shows the computational and memory overhead for applying the abstraction is negligible. In summary, the enhanced PAT model checker complements existing model checkers in terms of not only performance but also the ability to perform model checking under weak or strong fairness with process counter abstraction.

5.8 Summary

In the literature, a number of population protocols have been proposed to solve problems in wireless sensor networks. The correctness of these protocols relies on global fairness, which makes their automatic verification using existing model checkers expensive or even infeasible. In this chapter, we have applied proposed fairness model checking algorithms, designed to handle verification under fairness more efficiently, to a set of self-stabilizing population ring protocols. We have shown that the model checking algorithm allows us to successfully verify instances of these protocols. Moreover, it has helped us to identify one previously unknown bug in a leader election protocol.

During the analysis, we have faced the infamous state explosion problem (see Table 5.1). To solve this problem, we studied model checking under fairness with process counter abstraction. We presented a fully automatic method for property checking under fairness with process counter abstraction. We showed that fairness can be achieved without the knowledge of process identifiers. We have shown the effectiveness of our approach by conducting experiments on real-world systems.

Pang et al [161] applied the SPIN model checker to establish the correctness of a family of population protocols. Only small networks (i.e., with few nodes) were verified under weak fairness in SPIN. Theorem proving is used in verifying population protocols with arbitrary size [64]. However, translating protocols into theorem prover accepted formalism and manual proof make this approach difficult to apply. PAT can handle global fairness required for the correctness of most population protocols, which makes PAT an ideal candidate for automatically verifying population protocols.

The works closest to counter abstraction approach are the methods presented in [63, 168, 167]. In particular, verification of liveness properties under fairness is addressed in [167]. In [167], the fairness constraints for the abstract system are generated manually (or via heuristics) from the fairness constraints for the concrete system. Different from the above works, our method handles one (possibly large) instance of parameterized systems at a time and uses counter abstraction to improve verification effectiveness. In addition, fairness conditions are integrated into the on-the-fly model checking algorithm which proceeds on the abstract state representation - making our method fully automated. Our method is also related to work on symmetry reduction [82, 57].

Chapter 6

Refinement Checking

The previous chapters have been focused on temporal logic model checking. An alternative approach is refinement checking, which has been successfully demonstrated by FDR [175]. In this chapter, we enrich PAT with this capability.

Hoare's classic Communicating Sequential Processes (CSP [108]) has been a rather successful event-based modeling language for decades. Theoretical development on CSP has advanced formal methods in many ways. Its distinguishable features like alphabetized parallel composition have proven to be useful in modeling a wide range of systems.

FDR (Failures-Divergence Refinement) is the *de facto* analyzer for CSP, which has been successfully applied in various domains. Based on the model checking algorithm presented in [175] and later improved with other reduction techniques presented in [179], FDR is capable of handling large systems. Nonetheless, since FDR was initially introduced, model checking techniques have evolved much further in the last two decades. Quite a number of effective reduction methods have been proposed which greatly enlarge the size the systems that can be handled. Some noticeable ones include partial order reduction, symmetry reduction, predicate abstraction, etc. In this chapter, we present a on-the-fly refinement checking algorithm designed to incorporate advanced model checking techniques to analyze event-based hierarchical systems.

The remainder of the chapter is organized as follows. We briefly introduce FDR and its refinement checking in Section 6.1. Section 6.2 proposes a refinement checking algorithm integrated with partial order reduction. Section 6.3 presents the experimental results of the proposed algorithm, compared with FDR model checker. Section 6.4 concludes this chapter.

6.1 FDR and Refinement Checking

Failures-Divergence Refinement (FDR [175]) is a well-established model checker for CSP. Different from temporal logic based model checking, using FDR, safety, liveness and combination properties can be verified by showing a refinement relation from the CSP model of the system to a CSP process capturing the properties. In addition, FDR verifies whether a process is deadlock-free or not. In the following, we review the notions of different refinement/equivalence relationship in terms of labeled transition systems (see Section 3.1.2).

Given a model $\mathcal{S} = (Var, init_G, P)$, let $\mathcal{L} = (S, init, \rightarrow)$ be the LTS of $\mathcal{S}$. Let s, s' be members of S, Σ denote the set of all visible events in P, τ denote the set of all invisible events, and Σ_τ be $\Sigma \cup \tau$. We define the following notions.

- $enabled(s) = \{e : \Sigma_\tau \mid \exists\, s' \bullet s \xrightarrow{e} s'\}$. A state s is stable if and only if $\tau \notin enabled(s)$.
- $mrefusal(s) = \Sigma \setminus enabled(s)$ is the maximum refusal set, i.e., the maximum set of events which can be refused.
- A state is a divergence state $div(s)$ if and only if s can be extended with infinite τ transitions.
- The set of divergence traces of $\mathcal{L}$, written as $divergence(\mathcal{L})$, is $\{tr : \Sigma^* \mid \exists\, tr', tr'$ is a prefix of $tr \wedge \exists\, s : S, init \xrightarrow{tr'} s \wedge div(s)\}$. Note that if some prefix of a given trace is a divergence trace, the given trace is too.
- The set of failures of $\mathcal{L}$, written as $failures(\mathcal{L})$, is $\{(tr, X) : \Sigma^* \times 2^\Sigma \mid \exists\, s : S, init \xrightarrow{tr} s \wedge X \subseteq \Sigma \setminus enabled(s)\} \cup \{(tr, \Sigma) : \Sigma^* \times 2^\Sigma \mid tr \in divergence(\mathcal{L})\}$. Note that the system state reached by a divergence state may refuse all events.

The following defines refinement and equivalence.

Definition 22 (Refinement and Equivalence) *Let $\mathcal{L}_{im} = (S_{im}, init_{im}, \rightarrow_{im})$ be a LTS representing an implementation. Let $\mathcal{L}_{sp} = (S_{sp}, init_{sp}, \rightarrow_{sp})$ be a LTS representing a specification. $\mathcal{L}_{im}$ refines $\mathcal{L}_{sp}$ in the trace semantics, written as $\mathcal{L}_{im} \sqsupseteq_T \mathcal{L}_{sp}$, if and only if $traces(\mathcal{L}_{im}) \subseteq traces(\mathcal{L}_{sp})$. $\mathcal{L}_{im}$ refines $\mathcal{L}_{sp}$ in the stable failures semantics, written as $\mathcal{L}_{im} \sqsupseteq_F \mathcal{L}_{sp}$, if and only if $failures(\mathcal{L}_{im}) \subseteq failures(\mathcal{L}_{sp})$. $\mathcal{L}_{im}$ refines $\mathcal{L}_{sp}$ in the failures/divergence semantics, written as $\mathcal{L}_{im} \sqsupseteq_D \mathcal{L}_{sp}$, if and only if $failures(\mathcal{L}_{im}) \subseteq failures(\mathcal{L}_{sp})$ and $divergence(\mathcal{L}_{im}) \subseteq divergence(\mathcal{L}_{sp})$. $\mathcal{L}_{im}$ equals $\mathcal{L}_{sp}$ in the trace (stable failures/failures divergence) semantics if and only if they refine each other in the respective semantics.*

Different refinement relationship can be used to establish different properties. Safety can be verified by showing a trace refinement relationship. Combination of safety and liveness is verified by showing a stable failures refinement relationship if the system is divergence-free or otherwise by showing a failures/divergences refinement relationship. The reader should refer to [178] for a discussion on the expressiveness of CSP refinement. In the following, we write $Im \sqsupseteq Sp$ to mean $\mathcal{L}_{Im} \sqsupseteq \mathcal{L}_{Sp}$ whenever it will not cause confusion. Internally, equivalence relationships may be used to simplify process expressions, e.g., $P \sqcap P$ is replaced by P for simplicity.

Example 6.1.1 The following models the classic dining philosophers [108],

$$\begin{aligned}
Phil(i) &= get.i.(i+1)\%N \rightarrow get.i.i \rightarrow eat.i \rightarrow \\
&\quad \rightarrow put.i.(i+1)\%N \rightarrow put.i.i \rightarrow think.i \rightarrow Phil(i) \\
Fork(i) &= get.i.i \rightarrow put.i.i \rightarrow Fork(i) \;\Box \\
&\quad get.(i-1)\%N.i \rightarrow put.(i-1)\%N.i \rightarrow Fork(i) \\
Pair(i) &= (Phil(i) \parallel Fork(i)) \setminus \{get.i.i, put.i.i, think.i\} \\
College &= (\|_{i=0}^{N-1} Pair(i)) \setminus \bigcup_{i=0}^{N-1} \{get.i.(i+1)\%N, put.i.(i+1)\%N\}
\end{aligned}$$

where N is a global constant (i.e., the number of philosophers), $get.i.j$ ($put.i.j$) is the action of the i-th philosopher picking up (putting down) the j-th fork and fc is a global variable recording the amount of food that has been consumed. The system is composed of N philosopher-fork-pairs running in parallel. Figure 6.1 is the transition system of $College$ with $N = 2$. All events except the bolded ones are invisible.

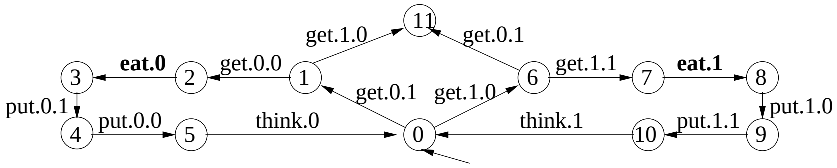


Figure 6.1: LTS for 2 dining philosophers

Assume that the following process is used to capture the property for the dining philosophers: $Prop \mathrel{\widehat{=}} \|_{i=0}^{N-1} Eat(i)$ where $Eat(i) = eat.i \rightarrow Eat(i)$. It can be shown that $College$ trace-refines $Prop$ (given a particular N). Informally speaking, that means it is possible for each philosopher to eat, i.e., $\{eat.0, \cdots, eat.(N-1)\}^*$ are traces of $College$. In order to show that it is *always* possible for him/her to eat, we need to establish $College \sqsupseteq_F Prop$, which is not true, i.e., assume $N = 2$, $(\langle get.0.1, get.1.0\rangle, \{eat.0, eat.1\})$ is in $failures(College)$ but not $failures(Prop)$. **end**

In order to check refinement, every reachable state of the implementation reachable from the initial state via some trace must be compared with every state of the specification reachable via the same trace. There may be many such states in the specification due to nondeterminism. In FDR, the specification is firstly normalized so that there is exactly one state corresponding to each possible trace. A state in the normalized LTS is a set of states which can be reached by the same trace from the initial state. For instance, Figure 6.2 shows the normalized LTS of the one presented in Example 6.1.1.

Definition 23 (Normalized LTS) *Let $(S, init, \rightarrow)$ be a LTS. The normalized LTS is $(NS, Ninit, NT)$ where NS is the set of subsets of S, $Ninit = \tau^*(init)$, and $NT = \{(P, e, Q) \mid P \in NS \wedge Q = \{s : S \mid \exists\, v_1 : P,\ \exists\, v_2 : S,\ (v_1, e, v_2) \in \rightarrow \wedge\ s \in \tau^*(v_2)\}\}$.*

Given a normalized state $s \in NS$,

- *$enabled(s)$ is $\bigcup_{x \in s} enabled(x)$,*
- *$mrefusal(s)$ is $\{mrefusal(x) \mid x \in s\}$, which is a set of maximum refusal sets,*
- *$div(s)$ is true if and only if there exists $x \in s$ such that $div(x)$ is true.*

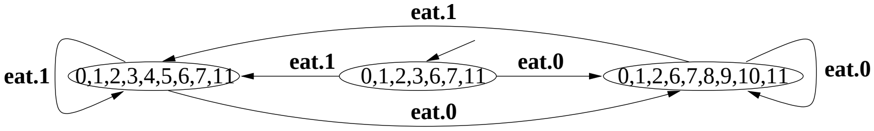


Figure 6.2: Normalized LTS for 2 dining philosophers

Given a LTS constructed from a process, the normalized LTS corresponds the normalized process. A state in the normalized LTS groups a set of states in the original LTS which are all connected by τ-transitions. Given a trace, exactly one state in the normalized LTS is reached. FDR then traverses through every reachable states of the implementation and compares them with the corresponding normalized states in the specification (refer to the algorithm presented in [175]).

6.2 An Algorithm for Refinement Checking

This section is devoted to algorithms for refinement checking. We start with reviewing a slightly modified on-the-fly checking algorithm based on the one implemented in FDR and then improve it with partial order reduction.

6.2.1 On-the-fly Refinement Checking Algorithm

Let $Spec = (S_{sp}, init_{sp}, \rightarrow_{sp})$ be a specification and $Impl = (S_{im}, init_{im}, \rightarrow_{im})$ be an implementation. Refinement checking is reduced to reachability analysis of the product of $Impl$ and normalized $Spec$. It has been shown that such an approach works well for certain models [179]. Nonetheless, because normalization in general is computationally expensive, it may not be always desirable. Thus, we adopted an alternative approach. Figure 6.3 presents the on-the-fly refinement checking algorithm. The algorithm similarly performs a reachability analysis in the product of the implementation and the normalized specification. The difference is that normalization is brought on-the-fly as well.

```
procedure $refines(Impl, Spec)$
$checked := \varnothing;$
$pending.push((init_{im}, \tau^*(init_{sp})));$
while $pending \neq \varnothing$
    $(Im, NSp) := pending.pop();$
    $checked := checked \cup \{(Im, NSp)\};$
    if $\neg(enabled(Im) \setminus \{\tau\} \subseteq enabled(NSp))$                       – C1
        $\vee\ (\tau \notin Im \wedge \neg\ existSuperSet(mrefusal(Im), mrefusal(NSp)))$   – C2
        $\vee\ (\neg\ div(NSp) \wedge div(Im))$                                         – C3
          return $false;$
    else
          foreach $(Im', NSp') \in \underline{next(Im, NSp)}$
               if $(Im', NSp') \notin checked$
                    $pending.push((Im', NSp'));$
               endif
          endfor
     endif
endwhile
return $true;$
```

Figure 6.3: Algorithm: $refines(Impl, Spec)$

Details of the following procedures are skipped for brevity. Procedure $tau(S)$ explores all outgoing transition of S and returns the set of states reachable from S via a τ-transition. We remark that this procedure will be refined later. Procedure $\tau^*(S)$ is implemented using a depth-first-search procedure. The set of states reachable from S via only τ transitions is returned. For instance, given the LTS in Example 6.1.1, $\tau^*(0)$ returns $\{0, 1, 2, 6, 7, 11\}$. The procedure $tau(S)$ is applied repeatedly until all τ-reachable states are identified. Procedure $existSuperSet(x, Y)$ where x is a set and Y is a set of sets returns true if and only if there exists y in Y such that $x \subseteq y$.

Depending on the type of refinement relationship, the algorithm performs a depth-first search for a pair (Im, NSp) where Im is a state of the implementation and NSp is a state of the normalized specification such that, the enabled events of Im is not a subset of those of NSp (C1), or Im is stable and there does not exist a state in NSp which refuse all events which are refused by Im (C2), or Im diverges but not NSp (C3). The algorithm returns true if no such pair is found. Note

```
procedure $next(Im, NSp)$
$toReturn := \varnothing$
foreach $e \in enabled(Im)$
    if $e == \tau$
        foreach $Im' \in tau(Im)$
            $toReturn := toReturn \cup \{(Im', NSp)\};$
        endfor
    else
        $NSp' := \{s \mid \exists\, x : NSp, x \xrightarrow{e} x' \wedge s \in \tau^*(x')\};$
        foreach $Im'$ such that $Im \xrightarrow{e} Im'$
            $toReturn := toReturn \cup \{(Im', NSp')\};$
        endfor
    endif
endfor
return $toReturn;$
```

Figure 6.4: Algorithm: $next(Im, NSp)$

that if `C1` is satisfied, a counterexample is found for any refinement checking; if `C2` is satisfied, a counterexample is found for stable failures refinement checking or failure/divergence refinement checking; if `C3` is satisfied, a counterexample is found for fairlure/divergence refinement checking only. The procedure for producing a counterexample is skipped for simplicity. Producing the shortest counterexample requires a breath-first-search after identifying the faulty state. Line 10 to 14 of algorithm *refines* explores new states of the product and pushes them into the stack *pending*. The procedure *next* is presented in Figure 6.4. Given a pair (Im, NSp), it returns a set of pairs of the form (Im', NSp') for each enabled event in Im. If the event is visible, NSp' is a successor of NSp via the event and Im' is the successor of Im via the same event. Otherwise, Im' is a successor of Im via a τ-transition and NSp' is Sp. Because normalization is brought on-the-fly, it is sometimes possible to find a counterexample before the specification is completely normalized. The soundness of the algorithm follows the soundness discussion in [175].

6.2.2 Partial Order Reduction

As any model checking algorithm, refinement checking suffers from state space explosion. A number of attempts have been applied to reduce the search space [179]. This section describes the one implemented in PAT based on partial order reduction. Our reduction realizes and extends the early works on partial order reduction for process algebras and refinement checking presented in [214] and [219]. The inspiration of the reduction is that events may be independent, e.g., $think.i$ is mutually independent of each other. Given $P = P_1 \parallel \cdots \parallel P_n$ and two enabled events e_1 and e_2, e_1 is dependent of e_2, written as $dep(e_1, e_2)$, and vice versa only if one of the following is true,

- e_1 and e_2 are from the same process P_i.
- $e_1 = e_2$ so that they may be synchronized, e.g., $get.i.i$ of process $Phil(i)$ and $get.i.i$ of process $Fork(i)$.
- e_1 updates a variable which e_2 depends on or vice versa, e.g., because $eat.i$ updates a global variable, all $eat.i$ are inter-dependent.

Two events are independent if they are not dependent. Because the ordering of independent events may be irrelevant to a given property, we may deliberately ignore some of the ordering so as to reduce the search space. Partial order reduction may be applied to a number of places in algorithm $refines$, namely, the procedure $tau(S)$ (and therefore $tauclosure$) and $next$. Since indexed parallel composition (and indexed interleaving) is the main source of state space explosion, we assume that Im is of the form $(V, (P_1 \parallel P_2 \parallel \cdots \parallel P_n) \setminus X)$ in the following and show how it is possible to only explore a subset of the enabled transitions and yet preserve the soundness.

We start with applying partial order reduction to the procedure tau. Note that tau is applied to the specification or implementation independently. Thus, as long as we guarantee that the reduced state space (of either $Impl$ or $Spec$) is failures/divergence equivalent to the full state space, we prove that there is a refinement relationship in the reduced state space if and only if there is one in the full state space. Figure 6.5 show our algorithm for selecting a subset of the τ-transitions. In the algorithm

```
procedure $tau'(Im)$
$nextmoves := stubborn\_tau(Im)$;
if $(nextmoves \neq \emptyset)$ then return $nextmoves$;   else return $tau(Im)$;

procedure $stubborn\_tau(Im)$
foreach $P_i$
    $por := enabled_{P_i}(Im) \subseteq \{\tau\} \cup X \land enabled_{P_i}(Im) = current(P_i)$
    foreach $e \in enabled_{P_i}(Im)$
        $por := por \land \neg\, loop(e) \land \forall\, e' : \Sigma_j, j \neq i \Rightarrow \neg\, dep(e, e')$
    endfor
    if $por$ then
        return $\{(((\cdots \parallel P_i' \parallel \cdots) \setminus X), V, C') \mid (P_i, V, C) \xrightarrow{e} (P_i', V, C')\}$;
    endif
endfor
return $\emptyset$;
```

Figure 6.5: Algorithm: $tau'(Im)$ and $stubborn\_tau(Im)$

tau', we try to identify one set of τ-transitions which are independent of the rest. If such a subset is found (i.e., the algorithm $stubborn\_tau$ returns a non-empty set of successors), only the subset is explored further. Otherwise (i.e., $stubborn\_tau$ returns an empty set), all possible τ-transitions are explored. In $stubborn\_tau$, the idea is to identify one process P_i such that all τ-transitions from P_i are independent of those from other processes. Note that this approach is most effective with τ-transition generated from one process only. It is possible to handle τ-transition generated from multiple processes with a slightly more complicated procedure (which we skip for brevity). The details of the following simple procedures have been skipped. Given $Im = (V, P)$, $enabled_{P_i}(Im)$ is the set of enabled event from component P_i, i.e., $enabled(Im) \cap enabled((V, P_i))$. For instance, given $College$ with $N = 2$, $enabled(Pair(0))$ is $\{get.0.1\}$. $current(P_i)$ is the set of events that could be enabled in process P_i given the most cooperative environment. For instance, $current(Phil(i)) = \{get.i.(i+1)\%N\}$ despite whether the fork is available or not. $loop(e)$ is true if and only if performing this event results in a state on the search stack, i.e., forming a cycle.

A process P_i is considered a candidate only if all enabled events from P_i result in τ-transitions (i.e., $enabled_{P_i}(Im) \subseteq \{\tau\} \cup X$) and no other transition could be possibly enabled given a different environment (i.e., $enabled_{P_i}(Im) = current(P_i)$). The former is required because we are only interested in τ-transitions. The latter (partly) ensures that no disabled event from P_i is enabled before executing an event from P_i. Furthermore, all enabled events from P_i must not form a cycle (so that an enabled event is not skipped for ever) or dependent on an enabled event from some other component. For detailed discussion on the intuition behind these conditions, refer to [58].

Example 6.2.1 Assume that $N = 2$ and the following is the current process expression,

$$\begin{array}{l} ((think.0 \rightarrow Phil(0) \parallel put.1.0 \rightarrow Fork(0)) \setminus \{get.0.0, put.0.0, think.0\}) \parallel \\ (((get1.1 \rightarrow eat.1 \rightarrow put.1.0 \rightarrow put.1.1 \rightarrow think.1 \rightarrow Phil(1)) \parallel Fork(1)) \\ \quad \setminus\{get.1.1, put.1.1, think.1\}) \setminus \{get.0.1, get.1.0, put.0.1, put.1.0\} \end{array}$$

where the first philosopher has just put down both forks while the second one has just picked up his first fork. Two τ-transitions are enabled, i.e., one due to $think.0$ and the other due to $get.1.1$. The algorithm tau' would return only the successor state after performing $get.1.1$ (assuming it is not on the stack). This is the only event enabled for the second component of the outer parallel composition is the τ transition due to $get.1.1$ (and thus the condition at line 1 of $stubborn\_tau$ is satisfied). Because $get.1.1$ is local to the component, por is true after the loop from line 2 to line 4.
end

The above algorithms apply partial order reduction to τ-transitions only. $tauclosure$ is refined as well since it is based on tau'. Unlike FDR, PAT is capable of applying partial order reduction to visible events. Because both $Impl$ and $Spec$ must make corresponding transitions for a visible event, reduction for visible events is complicated. A conservative approach has been implemented in PAT. Figure 6.6 present the algorithm, i.e., the refined $next$. If Im is not stable, we apply the algorithm $stubborn\_tau$ to identify a subset of τ-transitions (line 1). If no such subset exists, the pair (Im, Sp) is fully expanded (line 11). An algorithm $stubborn\_visible$ similar to $stubborn\_tau$ is used to check if a given visible event e is a candidate for partial order reduction. Function $processes(e)$ returns all process components (of the parallel composition) whose alphabet contains e. Firstly, we choose a possible candidate from Im using the algorithm $stubborn\_visible$. Event e is chosen if and only if, for each process in $processes(e)$, e is the only event from the process which can be enabled and all

```
procedure $next'(Im, Sp)$
0. if $\tau \in enabled(Im)$
1.     $nextmoves := stubborn\_tau(Im);$
2.     if $(nextmoves \neq \varnothing)$ then return $nextmoves;$
3. else
4.     foreach $e \in enabled(Im)$
5.         $por := stubborn\_visible(Im, e);$
6.         foreach $S \in Sp$
7.             $por := por \wedge stubborn\_visible(S, e);$
8.         endeach
9.         if $por$ then return $\{(Im', tauclosure(Sp')) \mid Im \xrightarrow{e} Im' \wedge Sp \xrightarrow{e} Sp'\}$
10.     endeach
11. return $next(Im, Sp);$

procedure $stubborn\_visible(Im, e)$
0. $por := \neg\, loop(e) \wedge \forall\, e' : \Sigma_j \bullet e' \neq e \Rightarrow \neg\, dep(e, e');$
1. foreach $P_i \in processes(e)$
2.     $por := por \wedge enabled_{P_i}(Im) = current(P_i) = \{e\};$
3. return $por;$
```

Figure 6.6: Algorithm: $next'(Im, Sp)$ and $stubborn\_visible(Im, e)$

other enabled events are independent of e and performing e does not result in a state on the stack. Next, we check if e satisfies the same set of conditions for each state in the normalized state of the specification. If it does, e is used to expand the search tree at line 9 (and all other enabled events are ignored). In order to find such e efficiently, the candidate events are selected in a pre-defined order, i.e., events which have the least number of associated processes are chosen first. The soundness of the partial order reduction algorithms is proved as follows.

Proof: The proof consists of two steps. Firstly, because the algorithm tau' applies to one model only (whereas $next'$ must coordinate both the implementation and the specification), it is sufficient to show that the reduction regarding τ-transitions (i.e., the algorithm $stubborn\_tau$) preserves failures/divergence equivalence. Secondly, we show that the reduction regarding visible events (i.e., the algorithm $next'$) is sound.

Given a LTS $(S, init, \rightarrow)$, let Σ be the set of events, Σ_I be the set of invisible events and Σ_V be the set of visible events. A stubborn set generator is a function Ä: $S \rightarrow 2^{\Sigma}$ such that for every s'_0, s'_n, and $s_0, s_1, \ldots \in S$, $e \in$ Ä(s_0), and $e_1, e_2, \ldots \in \Sigma-$ Ä(s_0). The set Ä(s) is called a stubborn set. In [214], the following sufficient conditions has been proved to preserve CSP failures divergence equivalence.

Ä$_0$ If $enabled(s_0) \neq \varnothing$ then Ä$(s_0) \cap enabled(s_0) \neq \varnothing$

Ä$_1$ If there are a trace $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_n, \rangle$ and (s_n, e, s'_n), then there are $s'_0, \ldots, s'_{n-1} \in S$ such that $\langle s'_0, e_0, s_1, e_1, \cdots, e_n, s'_n, \rangle$ and (s_0, e, s'_0).

Ä$_2$ If there are a trace $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_n, \rangle$ and (s_0, e, s'_0), then there are $s'_1, \ldots, s'_n \in S$ such that $\langle s'_0, e_0, s_1, e_1, \cdots, e_n, s'_n, \rangle$ and (s_n, e, s'_n).

Ä$_3$ If there are a trace $\langle s_0, e_0, s_1, e_1, \cdots\rangle$ and (s_0, e, s'_0), then there are $s'_1, s'_2, \ldots, \in S$ such that $\langle s'_0, e_0, s'_1, e_1, \cdots, \rangle$.

Ä$_4$ For every $s \in$ S̈, either $\Sigma_V \cap$ Ä(s) $\cap enabled(s) = \varnothing$ or $\Sigma_V \subseteq$ Ä(s) (or both).

Ä$_5$ $\forall\, s \in$ S̈: $\forall\, e \in enabled(s) : \exists\, s' \in$S̈: $s \overset{e_1, e_2, \cdots, e_n}{\longrightarrow} s' \wedge e \in$Ä$(s)$.

Ä$_6$ For every $s \in$ S̈, if $\Sigma_I \cap enabled(s) \neq \varnothing$, then Ä$(s) \cap \Sigma_I \cap enabled(s) \neq \varnothing$.

It is thus sufficient to prove that the reduction regarding τ-transitions satisfies the sufficient conditions. In the following, let E be the reduced set of successors (i.e., the stubborn set Ä(s)) and F be the full set. Notice that the result returned by algorithm $stubborn\_tau$ is returned by algorithm tau' or $next'$ if and only if it is not empty (line 1 of tau' and line 2 of $next'$). Thus, as long as F is not empty, E is not empty. By line 3 of algorithm $stubborn\_tau$, transitions other than those selected in E are all independent of those in E. By line 1 of $stubborn\_tau$, because the set of possibly enabled events must be the same of the set enabled event from the component, a transition from the component must remain disabled unless a transition from the components has been taken. By theorem 3.2 of [214], Ä$_0$, Ä$_1$, Ä$_2$, Ä$_3$ hold. Because only τ-transitions are reduced in tau', condition Ä$_4$ is trivial. By the condition $\neg\, loop(e)$ at line 3 of $stubborn\_tau$, no action will be ignored forever, and thus Ä$_5$ holds. Ä$_6$ is trivial for the same reason as for Ä$_4$. By theorem 4.2 and 5.3 in [214], the reduction regarding τ-transitions preserves trace/failures/divergence equivalence and thus is sound.

In order to prove that algorithm $next'$ is sound, we need to prove (in addition to the above) that the reduction regarding visible events are sound as well. We reuse the results which have been proved in [219] and show that the sufficient conditions proposed in [219] have been full-filled. Firstly, C1 and C3 in [219] are trivial true. Because of line 0 and 2 of $stubborn\_visible$, an action dependent (say e) on an action selected can only be executed after some action selected has been executed. There are two cases in which this might be violated. In both of these cases, some transition (say a) independent of e are executed, eventually enabled a transition that is dependent on e. In the first case, if a belongs to some other components. A necessary condition for this to happen is that a is dependent on e. This is prevented by line 1. In the other case, a belongs to the same component of e, which is not possible because we require that $current(P_i) = \{e\}$. The same argument applies to line 6 to 8 which guarantees that no action dependent on e is executed before e is executed (and there C1in [219] is proved). C2 in [219] is guaranteed by the condition $\neg\, loop(e)$. Therefore, we conclude the reduction is sound. □

Example 6.2.2 Let $P(i) = a.i \rightarrow b.i \rightarrow P(i)$. Assume the specification and implementation is defined as: $Spec = \|_{i=0}^{2} P(i)$ and $Impl = \|_{i=0}^{1} P(i)$. Assume we need to show that $Impl$ trace-refines $Spec$. Initially, two events are enabled in $Impl$, i.e., $a.0$ and $a.1$. Assume that $a.0$ is selected first, because $loop(a.0)$ is false and $a.0$ is independent of all other enabled events (i.e., $a.1$), the condition at line 0 of algorithm $stubborn\_visible$ is satisfied. Because $a.0$ is the only event that would possibly be enabled from $P(0)$, the condition at line 2 is satisfied too. Thus, $a.0$ is a possible candidate for partial order reduction for $Impl$. Similarly, it is also a candidate for $Spec$ (which is the only state in the normalized initial state). Therefore, we only need to explore $a.0$ initially. **end**

6.3 Experiments

We compare PAT with FDR using benchmark models for refinement checking. For the sake of a fair comparison, all models use only standard CSP features which are supported by both. Table 6.1 shows the experiment results for three models, obtained on a 2.0 GHz Intel Core Duo CPU and 1 GB memory. In the table, "$-$" means out of memory. Since FDR has no direct support for variables, the experiments use three examples with process definitions only.

The first example is the classic dining philosopher problem, where N is the number of philosophers and forks. Because of the modeling, partial order reduction is not effective for this example. As a result, PAT handles about 10^7 states (about 11 philosophers and forks) in a reasonable amount of time. FDR performs extremely well for this example because of the strategy discussed in [179]. Namely, it builds up a system gradually, at each stage compressing the subsystems to find an equivalent process with (for this particular example) many less states. Notice that with manual hiding (to localize some events), PAT performs much better. The second example is the classic readers/writers problem, in which the readers and writers coordinate to ensure correct read/write ordering. N is the number of readers/writers. Reduction in PAT is very effective for this example. As a result, PAT handles a few hundreds of readers/writers efficiently, whereas FDR suffers from state space explosion quickly (for $N = 18$). The third example is the Milner's cyclic scheduling algorithm, in which multiple processes are scheduled in a cyclic fashion. Partial order reduction is extremely effective for this model. As a result, PAT handles hundreds of processes, whereas FDR handles less than 14 processes. The experiment results show our best effort so far on automated model checking of an extended version of CSP. It by no means suggests the limit of our tool. We believe that by incorporating more reduction techniques (e.g., symmetry reduction) as well as fine-tuning the implementation, the performance of PAT can be improved significantly.

We remark that we are using the intuitive modeling in these examples. FDR compression techniques are not fully explored.

Model	N	Property	Result	PAT (sec.)	FDR (sec.)
Dining Philosophers	5	P [T= S	true	0.28125	0.067
Dining Philosophers	6	P [T= S	true	0.8593	0.069
Dining Philosophers	8	P [T= S	true	13.78	0.076
Dining Philosophers	10	P [T= S	true	430.28	0.107
Dining Philosophers	12	P [T= S	true	-	0.319
Reader/Writers	12	P [T= S	true	< 1	0.812
Reader/Writers	14	P [T= S	true	< 1	6.906
Reader/Writers	16	P [T= S	true	< 1	81.247
Reader/Writers	18	P [T= S	true	< 1	-
Reader/Writers	50	P [T= S	true	1.097	-
Reader/Writers	100	P [T= S	true	9	-
Reader/Writers	200	P [T= S	true	77.515	-
Milner’s Cyclic Scheduler	11	P [T= S	true	< 1	19.011
Milner’s Cyclic Scheduler	12	P [T= S	true	< 1	89.421
Milner’s Cyclic Scheduler	13	P [T= S	true	< 1	419.021
Milner’s Cyclic Scheduler	14	P [T= S	true	< 1	-
Milner’s Cyclic Scheduler	50	P [T= S	true	2.406	-
Milner’s Cyclic Scheduler	100	P [T= S	true	9.765	-
Milner’s Cyclic Scheduler	200	P [T= S	true	60.453	-

Table 6.1: Experiment results for refinement checking

6.4 Summary

In this chapter, we studied refinement checking problem. We started with the definitions of trace refinement and equivalence. Based on the FDR approach, we proposed an on-the-fly refinement checking algorithm, incorporated with partial order reduction. Experiments suggest that our approach is effective compared to FDR. Though we have shown cases where PAT outperforms FDR, we believe that a full comparison is yet to be carried out with more experiments.

Refinement checking of CSP programs is an area that has been widely explored. There has been research on techniques that require human assistance. This work ranges from completely manual proof techniques, such as Josephs' work on relational approaches to CSP refinement checking [122], to semi-automated techniques where humans must provide hints to guide a theorem prover in checking refinements [76, 208, 116, 190]. Second, there has been work on fully automated techniques to CSP refinement checking [175], the state of the art being embodied in the FDR tool [175]. Leuschel and Butler [139] proposed a refinement checking approach for B language based on the searching and tabling techniques of logic programming. However their approach is not on-the-fly. Kundu, Lerner and Gupta [125] developed an automatic solution for refinement checking of infinite data size. Their approach uses theorem proving to handle infinite state space, which can also be combined with our solution. Regarding partial order reductions, a number of algorithms have been previously proposed for partial order reduction which is trace/failures/divergence preserving, e.g. [214, 219]. The algorithms presented in the chapter may be considered as lightweight realization and extension of those presented in [214, 219].

Chapter 7

Applications of Refinement Checking

Refinement checking is a useful verification technique, which has many applications. As one example, Allen and Garlen have shown how CSP programs can be used as types to describe the interfaces of software components [7]. The refinement relation becomes a sub-typing relation, and refinement checking can then be used to determine if two components, whose interfaces are specified using CSP programs, are compatible. In this chapter, we demonstrate the usefulness of refinement checking using two examples, namely, linearizability verification for concurrent objects and Web Service conformance checking.

Linearizability is an important correctness criterion for implementations of concurrent objects. Automatic checking of linearizability is challenging because it requires checking that 1) all executions of concurrent operations be serializable, and 2) the serialized executions be correct with respect to the sequential semantics. In the first half of this chapter, we describe a new method to automatically check linearizability based on refinement relations from abstract specifications to concrete implementations. Our method avoids the often difficult task of determining linearization points in implementations, but can also take advantage of linearization points if they are given. The refinement verification algorithm developed in the previous chapter is used to automatically check a variety of implementations of concurrent objects, including the first algorithms for the mailbox problem [19] and Scalable NonZero Indicators [78]. Our system is able to prove or find all known and injected bugs in these implementations.

In recent years, many Web Service composition languages have been proposed. Web Service *choreography* describes collaboration protocols of cooperating Web Service participants from a global view. Web Service *orchestration* describes collaboration of the Web Services in predefined patterns based on local decision about their interactions with one another at the message/execution level. In the second part of this chapter, we present a model-based method to close the gap between the two views. Building on the strength of advanced model checking techniques, Web Service choreography and orchestration are verified against against each other (to show that they are consistent). Specialized optimization techniques are developed to handle large Web Service models.

The rest of the chapter is structured in two parts. The first part talks about linearizability checking. Section 7.1 gives the definition of linearizability. Section 7.2 shows how to express linearizability using refinement relations in general. Section 7.3 presents experimental results of linearizability checking. The second part talks about Web Service conformance checking. Section 7.4 introduces the background about Web Service and conformance checking. Section 7.5 introduces the modeling languages to capture Web Service compositions. Section 7.6 discusses how to verify Web Services. Section 7.7 presents experimental results to show its scalability. Section 7.8 concludes this chapter.

7.1 Linearizability

Linearizability [107] is an important correctness criterion for implementations of objects shared by concurrent processes, where each process performs a sequence of operations on the shared objects. Informally, a shared object is *linearizable* if each operation on the object can be understood as occurring instantaneously at some point, called the *linearization point*, between its invocation and its response, and its behavior at that point is consistent with the specification for the corresponding sequential execution of the operation.

One common strategy for proving linearizability of an implementation (used in manual proofs or automatic verification) is to determine linearization points in the implementation of all operations and then show that these operations are executed atomically at the linearization points [78, 13, 215]. However, for many concurrent algorithms, it is difficult or even impossible to statically determine all linearization points. For example, in the K-valued register algorithm (Section 10.2.1 of [22]), linearization points differ depending on the execution history. Furthermore, the linearization points determined might be incorrect, which can give wrong results of linearizability. Therefore, it is desirable to have automatic solutions to verifying these algorithms without knowing linearization points. However, existing methods for automatic verification without using linearization points either apply to limited kinds of concurrent algorithms [218] or are inefficient [215].

In this chapter, we describe a new method for automatically checking linearizability based on refinement relations from abstract specifications to concrete implementations. Our method does not rely on knowing linearization points, but can take advantage of them if given. The method exploits model checking of finite state systems specified as concurrent processes with shared variables, and is not limited to any particular kinds of concurrent algorithms. We exploit powerful optimizations to improve the efficiency and scalability of our checking method.

Refinement requires that the set of execution traces of a concrete implementation be a subset of that of an abstract specification. Thus, we express linearizability as trace refinement of operation invocations and responses from the abstract specification to the concrete implementation, where the abstract specification is correct with respect to sequential semantics. The idea of refinement has been explored before: Alur et al. [12] showed that linearizability can be cast as containment of two regular languages on a semi-commutative alphabet, and Derrick et al. [65] expressed linearizability as non-atomic refinement of Object-Z and CSP models. Some similar approaches [59, 69, 148] prove linearizability using trace simulation. In this following, we give a general and rigorous definition of linearizability, regardless of the modeling language used, using refinement.

7.1.1 Formal Definition

Linearizability [107] is a safety property of concurrent systems, over sequences of events corresponding to the invocations and responses of the operations on shared objects.

In a shared memory model $\mathcal{M}$, $O = \{o_1, \ldots, o_k\}$ denotes a set of k shared objects, $P = \{p_1, \ldots, p_n\}$ denotes a set of n processes accessing the objects. Shared objects support a set of *operations*, which are pairs of invocations and matching responses. Every shared object has a set of states that it could be in. A *sequential specification* of a (deterministic) shared object[1] is a function that maps every pair of invocation and object state to a pair of response and a new object state.

The behavior of $\mathcal{M}$ is defined as H, the set of all possible sequences of invocations and responses together with the initial states of the objects. A history $\sigma \in H$ induces an irreflexive partial order $<_\sigma$ on operations such that $op_1 <_\sigma op_2$ if the response of operation op_1 occurs in σ before the invocation of operation op_2. Operations in σ that are not related by $<_\sigma$ are concurrent. σ is sequential iff $<_\sigma$ is a strict total order. Let $\sigma|_i$ be the projection of σ on process p_i, which is the subsequence of σ consisting of all invocations and responses that are performed by p_i. Let $\sigma|_{o_i}$ be the projection of σ on object o_i, which is the subsequence of σ consisting of all invocations and responses of operations that are performed on object o_i.

A sequential history σ is *legal* if it respects the sequential specifications of the objects. More specifically, for each object o_i, if s_j is the state of o_i before the invocation of the j-th operation op_j in $\sigma|_{o_i}$, then response of op_j and the resulting new state s_{j+1} of o_i follow the sequential specification of o_i. For example, a sequence of read and write operations of an object is legal if each read returns the value of the preceding write if there is one, and otherwise it returns the initial value. Every history σ of a shared memory model $\mathcal{M}$ must satisfy the following basic properties:

Correct interaction For each process p_i, $\sigma|_i$ consists of alternating invocations and matching responses, starting with an invocation. This property prevents *pipelining* operations.

Closeness Every invocation has a matching response. This property prevents *pending* operations.

In addition to these two, liveness property is also important here to guarantee the progress of the systems. Even if the model satisfies linearizability, it may not progress as desired. For instance,

[1] More rigorously, the sequential specification is for a *type* of shared objects. For simplicity, however, we refer to both actual shared objects and their types interchangeably in this chapter.

even under a fair scheduler push/pop in Treiber's stack algorithm [210] might never terminate if there is always another concurrent push/pop. According to Section 2.3.2, liveness properties can be formulated as Linear Temporal Logic (LTL) formulae and checked using standard LTL model checkers (with or without the assumption of a fair scheduler).

Given a history σ, a *sequential permutation* π of σ is a sequential history in which the set of operations as well as the initial states of the objects are the same as in σ. The formal definition of linearizability is given as follows.

Linearizability There exists a sequential permutation π of σ such that

1. for each object o_i, $\pi\mid_{o_i}$ is a legal sequential history (i.e. π respects the sequential specification of the objects), and
2. if $op_1 <_\sigma op_2$, then $op_1 <_\pi op_2$ (i.e., π respects the run-time ordering of operations).

Linearizability can be equivalently defined as follows: In every history σ, if we assign increasing time values to all invocations and responses, then every operation can be shrunk to a single time point between its invocation time and response time such that the operation appears to be completed instantaneously at this time point [148, 22]. This time point for each operation is called its *linearization point*. Linearizability is a safety property [148], so its violation can be detected in a finite prefix of the execution history.

Linearizability is defined in terms of the interface (invocations and responses) of high-level operations. In a real concurrent program, the high-level operations are implemented by algorithms on concrete shared data structures, e.g., using a linked list to implement a shared stack object. Therefore, the execution of high-level operations may have complicated interleaving of low-level actions. Linearizability of a concrete concurrent algorithm requires that, despite complicated low-level interleaving, the history of high-level invocation and response events still has a sequential permutation that respects both the run-time ordering among operations and the sequential specification of the objects. This idea is formally presented in the next section using refinement relations in a process algebra extended with shared variables.

7.2 Linearizability as Refinement Relations

In this section, we show how to create high-level linearizable specifications and how to use a refinement relation from an implementation model to a specification model to define linearizability.

7.2.1 Model Construction

To create a high-level linearizable specification for a shared object, we rely on the idea that in any linearizable history, any operation can be thought of as occurring at some linearization point. We define the specification LTS $\mathcal{L}_{sp} = (S_{sp}, init_{sp}, \rightarrow_{sp})$ for a shared object o as follows. Every execution of an operation op of o on a process p_i includes three atomic steps: the invocation action $inv(op)_i$, the linearization action $lin(op)_i$, and the response action $res(op, resp)_i$. The linearization action $lin(op)_i$ performs the computation based on the sequential specification of the object. In particular, it maps the invocation and the object state before the operation to a new object state and a response, changes the object to the new state, and buffers the response $resp$ locally. The response action $res(op, resp)_i$ generates the actual response $resp$ using the buffered result from the linearization action. Each of the three actions is executed atomically without being interfered by any other action, but the three actions of one operation may interleave with the actions of other operations. In $\mathcal{L}_{sp}$, all $inv(op)_i$ and $res(op, resp)_i$ are visible events, while $lin(op)_i$ are invisible events.

In a LTS $\mathcal{L}_{sp} = (S_{sp}, init_{sp}, \rightarrow_{sp})$, each process p_i has (a) an idle state $s_{p_i,0}$, (b) a state $s(op)_{p_i,1}$ for every operation op of object o, representing the state after the invocation of op but before the linearization action of op, and (c) $s(op, resp)_{p_i,2}$ for every operation op and every possible response $resp$ of this operation, representing the state after the linearization action of op but before the response of op. Then S_{sp} is the cross product of all object values and all process states. $init_{sp}$ is the combination of the initial value of object o and $s_{p_i,0}$'s for all processes p_i. For $s \in S_{sp}$, let s_{v_o} be the value of object o encoded in s, s_{p_i} be the state of p_i in s, and s_{-p_i} and $s_{-p_i,-v_o}$ be the state s excluding s_{p_i} and excluding s_{p_i} and s_{v_o}, respectively. The labeled transition relation $\rightarrow_{sp}$ is such that for $(s, e, s') \in\rightarrow$, (a) if $e = inv(op)_i$, then $s_{-p_i} = s'_{-p_i}$, $s_{p_i} = s_{p_i,0}$, and $s'_{p_i} = s(op)_{p_i,1}$; (b) if $e = lin(op)_i$, then $s_{-p_i,-v_o} = s'_{-p_i,-v_o}$, $s_{p_i} = s(op)_{p_i,1}$, and $s'_{p_i} = s(op, resp)_{p_i,2}$, such that s'_{v_o} and $resp$ are the new object value and the response, respectively, based on the sequential specification of object o as well as the old object state s_{v_o} and the state $s_{p_i} = s(op)_{p_i,1}$ of process p_i; (c) if $e = res(op, resp)_i$, then $s_{-p_i} = s'_{-p_i}$, $s_{p_i} = s(op, resp)_{p_i,2}$, and $s'_{p_i} = s_{p_i,0}$.

Example 7.2.1 (K-valued register) We use a shared K-valued single-reader single-writer register algorithm (Section 10.2.1 of [22]) to demonstrate the ideas above. The linearizable abstract model is defined as follows, where R is the shared register with initial value K, and M is a local variable to store the value read from R.

$$\begin{array}{ll}
\mathit{ReaderA}() & = \mathit{read\_inv} \rightarrow \mathit{read}\{M = R\} \rightarrow \mathit{read\_res}.M \rightarrow \mathit{ReaderA}();\\
\mathit{WriteA}(v) & = \mathit{write\_inv}.v \rightarrow \mathit{write}\{R = v\} \rightarrow \mathit{write\_res} \rightarrow \mathit{Skip};\\
\mathit{WriterA}() & = (\mathit{WriteA}(0) \,\Box\, \mathit{WriteA}(1) \,\Box \ldots \Box\, \mathit{WriteA}(K-1));\ \mathit{WriterA}();\\
\mathit{RegisterA}() & = (\mathit{ReaderA}() \mid\mid\mid \mathit{WriterA}()) \backslash \{\mathit{read}, \mathit{write}\};
\end{array}$$

The $\mathit{ReaderA}$ process repeatedly reads the value of register R and stores the value in local variable M. Event $\mathit{read\_res}.M$ returns the value in M. $\mathit{WriteA}(v)$ writes the given value v into R. Event $\mathit{write\_inv}.v$ stores the value v to be written into the register. The $\mathit{WriterA}$ process repeatedly writes a value in the range of 0 to $K-1$. External choices are used here to enumerate all possible values. $\mathit{RegisterA}$ interleaves the reader and writer processes and hides the read and write events (linearization actions). The only visible events are the invocation and response of the read and write operations. This model generates all the possible linearizable traces.

Given a LTS $\mathcal{L}_{im} = (S_{im}, init_{im}, \rightarrow_{im})$ that supposedly implements object o, the visible events of $\mathcal{L}_{im}$ are those $inv(op)_i$'s and $res(op, resp)_i$'s. For example, the following models an implementation of a K-valued register using an array B of K binary registers (storing only 0 and 1).

$$\begin{array}{ll}
\mathit{Reader}() & = \mathit{read\_inv} \rightarrow \mathit{UpScan}(0);\\
\mathit{UpScan}(i) & = \textbf{if}(B[i] == 1)\{\mathit{DownScan}(i-1, i)\}\textbf{else}\{\mathit{UpScan}(i+1)\};\\
\mathit{DownScan}(i, v) & = \textbf{if}(i < 0)\{\mathit{read\_res}.v \rightarrow \mathit{Reader}()\}\\
 & \quad \textbf{else}\{\textbf{if}(B[i] == 1)\{\mathit{DownScan}(i-1, i)\}\textbf{else}\{\mathit{DownScan}(i-1, v)\}\};\\
\mathit{Write}(v) & = \mathit{write\_inv}.v \rightarrow \tau\{B[v] = 1;\ \} \rightarrow \mathit{WriterScan}(v-1);\\
\mathit{WriterScan}(i) & = \textbf{if}(i < 0)\{\mathit{write\_res} \rightarrow \mathit{Skip}\}\\
 & \quad \textbf{else}\{\tau\{B[i] = 0;\ \} \rightarrow \mathit{WriterScan}(i-1)\};\\
\mathit{Writer}() & = (\mathit{Write}(0) \,\Box\, \mathit{Write}(1) \,\Box \ldots \Box\, \mathit{Write}(K));\ \mathit{Writer}();\\
\mathit{Register}() & = \mathit{Reader}() \mid\mid\mid \mathit{Writer}();
\end{array}$$

The $Reader$ process first does a upward scan from element 0 to the first non-zero element i, and then does a downward scan from element $i-1$ to element 0 and returns the index of last element whose value is 1. Event $read\_res.v$ returns this index as the return value of the read operation. The $Write(v)$ process first sets the v-th element of B to 1, and then does a downward scan to set all elements before i to 0. Note that in this implementation, the linearization point for $Reader$ is the last point where the parameter v in $DownScan$ process is assigned in the execution. Therefore, the linearization point can not be statically determined. Instead, it can be in either $UpScan$ or $DownScan$. We remark that one liveness property can be verified by model checking $\Box read\_inv \Rightarrow \Diamond read\_res$ where $\Box$ and $\Diamond$ are modal operators which read as *always* and *eventually* respectively. **□end**

Theorem 7.2.2 characterizes linearizability of the implementation through a refinement relation and thus establishes our approach to verifying linearizability. Different versions of this result appeared in distributed computing literature, for example, in Lynch's book [148], Theorems 13.3-13.5.

Theorem 7.2.2 *All traces of L_{im} are linearizable iff $L_{im} \sqsupseteq_T L_{sp}$.*

Proof (sketch). Sufficient condition: For any trace $\sigma \in traces(L_{im})$, because $L_{im} \sqsupseteq_T L_{sp}$, σ is also a trace of L_{sp}. Let ρ be the execution history of L_{sp} that generates the trace σ. We define the sequential permutation π of σ as the reordering of operations in σ in the same order as the linearization actions $lin(op)_i$'s of all operations op and all processes p_i in ρ. If $op_1 <_\sigma op_2$, the linearization action of op_1 must be ordered before the linearization action of op_2 in ρ, and thus $op_1 <_\pi op_2$. It is also easy to verify that π is a legal sequential history of object o, since the linearization action of every operation in ρ is the only action in the operation that affects the object state based on its sequential specification, and the order of operations in π respects the order of linearization actions in ρ.

Necessary condition: Let σ be a trace of L_{im}. By assumption σ is linearizable. We need to show that σ is also a trace of L_{sp}. Since σ is linearizable, there is a sequential permutation π of σ such that π respects both the sequential specification of object o and the run-time ordering of the operations in

σ. We construct an execution history ρ of L_{sp} from σ and π as follows. Starting from the first event of σ, for any event e in σ, (a) if it is an invocation event, append it to ρ; (b) if it is a response event $res(op, resp)_i$, locate the operation op in π, and for each unprocessed operation op' by a process j before op in π, process op' by appending a linearization action $lin(op')_j$ to ρ, following the order of π; finally append $lin(op)_i$ and $res(op, resp)_i$ to ρ. It is not difficult to show that the execution history ρ constructed this way is indeed a history of L_{sp}. Moreover, obviously the trace of ρ is σ. Therefore, σ is also a trace of L_{sp}. □

The above theorem shows that to verify linearizability of an implementation, it is necessary and sufficient to show that the implementation LTS is a refinement of the specification LTS as we defined above. This provides the theoretical foundation of our verification of linearizability. Notice that the verification by refinement given above does not require identifying low-level actions in the implementation as linearization points, which is a difficult (and sometimes even impossible) task. In fact, the verification can be automatically carried out without any special knowledge about the implementation beyond knowing the implementation code.

In some cases, one may be able to identify certain events in an implementation as linearization points. We call these linearization events. For example, three linearization events have been identified in the stack algorithm [13]. In these cases, we can make these events visible and hide other events (including the invocation and response events) and verify refinement relation only for these events. More specifically, we obtain a specification LTS L'_{sp} by the following two modifications to L_{sp}: (a) for each linearization action $lin(op)_i$, we change it to $lin(op, resp)_i$ so that the response $resp$ computed by this linearization action is included; and (b) all linearization actions are visible while all $inv(op)_i$ and $res(op, resp)_i$ are invisible. Let L'_{im} be an implementation LTS such that its linearization events are visible and all other events are invisible, and its linearization events are also specified as $lin(op, resp)_i$.

Theorem 7.2.3 *Let L'_{sp} and L'_{im} be the specification and implementation LTSs such that linearization events are specified as $lin(op, resp)_i$ and are the only visible events. If $L'_{im} \sqsupseteq_T L'_{sp}$, then the implementation is linearizable. Conversely, if the implementation is linearizable, and* it can be shown that no other actions in the implementation can be linearization actions, *then $L'_{im} \sqsupseteq_T L'_{sp}$.*

The proof of this theorem is straight forwards, hence ignored. With this theorem, the verification of linearizability could be more efficient based on only linearization events. However, one important remark is that, as stated in the theorem, to make refinement a necessary condition of linearizability in this case, one has to show that no other actions in the implementation can be linearization points. In other words, the determined linearization points have to be complete. Otherwise, even if the verification finds a counterexample for the refinement relation, we cannot conclude that the implementation is not linearizable since we may have failed in determining all possible linearization events. Examples of implementations modeled using linearization points can be found in [196].

7.2.2 Verification of Linearizability

With the results from Theorem 7.2.2 and 7.2.3, we can use the refinement checking algorithm presented in Section 6 to verify linearizability of an implementation of concurrent objects. Several improvements that are specialized to linearizability verification are suggested in the following.

Partial order reduction (POR) in the refinement checking works without knowledge of linearization points. Nonetheless, having the knowledge would allow us to take full advantage of POR. Because linearization points are the only places where data consistency must be checked, we may amend the above algorithm to perform data consistency check at the linearization points. As a result, encoding relevant data as part of the event is not necessary and the model contains fewer events, which translates to fewer traces. Furthermore, because only the linearization points need to be synchronized, we may hide all other visible events to τ-transitions that are subject to POR.

Besides partial order reduction, our approach is compatible with other state space reduction techniques or abstract interpretation techniques. Distributed algorithms and protocols are usually designed for a large number of similar processes. They are therefore subject to symmetric reduction [83]. For instance, different writers (i.e., $WriterA(i)$) in Example 7.2.1 are symmetric and therefore, it is sound (subject to property-specific conditions) to only explore one writer and conclude the same for all other writers. If the processes are identical, then it is subject to process counter abstraction 5.4. For example, in the concurrent stack algorithm, the processes invoking push and pop are symmetric and therefore, we only keep track of the number of processes, instead of the exact processes. In this way, we may prove the property for arbitrary number of processes.

7.3 Experiments of Linearizability Checking

Our method has been applied to a number of concurrent algorithms, including *register*—the K-valued register algorithm[2] in Example 7.2.1, *stack*—a concurrent stack algorithm [210], *queue*—a concurrent non-blocking queue algorithm in Figure 3 of [154], *buggy queue*—an incorrect queue algorithm [186], and *mailbox* and *SNZI*—the first algorithms for the mailbox problem [19] and scalable Non-Zero indicators [78], respectively. We remark that the mailbox problem and SNZI are complicated algorithms that are not formally verified before. Both algorithms use sophisticated data structures and control structures, so the linearization points are difficulty to determine. The verification details of the two algorithms can be found in [196] and [225] respectively.

Table 7.1 summarizes part of our experiments, where '-' means out of memory or more than 4 hours, and '(points)' means that linearization points are given. The number of states and running time increase rapidly with data size and the number of processes, e.g., 3 processes for *register*, *stack*, *queue*, and *SNZI* vs. 2 processes. The results conform to theoretical results [12]: model checking linearizability is in EXPSPACE for both time and space. When linearization points are known, the complexity is still EXPSPACE, but the state space reduces significantly since the state spaces of implementation and specification are smaller. We show that the speedup of knowing linearization points is in the order of $O(2^{k \cdot 2^n \cdot (k^{2n} - k^n)})$, where k is the size of the shared object and n is the number of processes [196]. Use of partial order reduction effectively reduces the search space and running time in most cases, including *stack* and *queue,* and especially *mailbox* and *SNZI* because their algorithms have multiple internal transitions. For *register*, the state space is reduced but running time increases because of computational overhead. For *buggy queue* [186], the counterexamples (discovered firstly in [60]) are produced quickly after exploring only part of the state space.

[2] We extend this example with 2 readers and 1 writer. The correctness is verified using PAT.

Algorithm	**#Proc.**	**Linearizable**	**Time(sec)**	**#States**	**Time(sec)**	**#States**
			w/o POR	w/o POR	with POR	with POR
4-valued *register*	2	true	6.14	50893	5.72	43977
5-valued *register*	2	true	44.9	349333	60.4	307155
6-valued *register*	2	true	297	2062437	789	1838177
3-valued *register* with 2 readers and 1 writer	3	true	294	479859	393	361255
stack of size 12	2	true	138	540769	65.9	395345
stack of size 14	2	true	411	763401	99.4	599077
stack of size 2	3	true	-	-	4321	4767519
stack of size 12 (points)	2	true	0.62	9677	0.82	9677
stack of size 14 (points)	2	true	0.82	12963	1.11	12963
stack of size 2 (points)	3	true	1.14	10385	1.56	10385
stack of size 2 (points)	4	true	37.6	219471	49.4	219471
queue of size 6	2	true	134	432511	86.2	343446
queue of size 8	2	true	256	104582	218	938542
buggy queue of size 10	2	false	10.9	32126	6.87	32126
buggy queue of size 20	2	false	52.73	105326	41.1	105326
mailbox of 3 operations	2	true	71.6	272608	27.8	120166
mailbox of 4 operations	2	true	2904	9928706	954	3696700
SNZI of size 2	2	true	1298	712857	322	341845
SNZI of size 3	3	true	-	-	6214	8451568

Table 7.1: Experiment results on a PC with 2.83 GHz Intel Q9550 CPU and 2 GB memory

Vechev and Yahav [215] also provided automated verification. Their approach needs to find a linearizable sequence for each history, whose worst-case time is exponential in the length of the history, as it may have to try all possible permutations of the history. As a result, the number of operations they can check is only 2 or 3. In contrast, our approach handles all possible interleaving of operations given sizes of the shared objects. Because of partial order reduction and other optimizations, our approach is more scalable than theirs. For instance, we can verify stacks of size 14, which means any number of stack operations that contain up to 14 consecutive push operations.

Note that experiments in Section 6.3 suggest that PAT is faster than FDR for systems without variables. Modeling variables using processes and lack of partial order reduction will make FDR even slower. Therefore we skip comparison with FDR on these examples.

7.4 Web Service and Conformance Checking

The Web Services paradigm promises to enable rich, dynamic, and flexible interoperability of highly heterogeneous and distributed Web-based platforms. In recent years, many Web Service composition languages have been proposed. There are two different viewpoints in the area of Web Service composition. Web Service *choreography* describes collaboration protocols of cooperating Web Service participants from a global point of view. An example is WS-CDL (Web Service Choreography Description Language [48]). Web Service *orchestration* refers to Web Service descriptions which take a local point of view, which describes collaborations of the Web Services in predefined patterns based on local decision about their interactions with one another at the message/execution level. A representative is WS-BPEL (Web Service Business Process Execution Language [121]), which models business processes by specifying the work flows of carrying out business transactions.

Informally, a choreography may be viewed as a contract among multiple corporations, i.e., a specification of requirements (which may not be executable). An orchestration is the composition of concrete services provided by each corporation who realizes the contract. The distinction between choreography and orchestration resembles the well studied distinction between sequence diagrams (which describes inter-object system interactions, taking a global view) and state machines (which may be used to describe intra-object state transitions, taking a local view).

In this chapter, we focus on the conformance checking problem between a choreography or an orchestration, i.e., whether they are consistent with each other. Solving either problem is however highly non-trivial. For instance, because Web Services are designed for potentially large number of users[3] (who may invoke the services simultaneously), verifying Web Services based on model checking techniques must cope with state space explosion due to concurrent service invocations.

[3] In reality, the number is bounded by the thread pool size of the underlying operating system. See discussion in [93].

The solution for the consistency checking is to show conformance relationship (i.e., existence of a weak simulation relationship) between the choreography and the orchestration. The algorithm is based on the refinement checking algorithm in Section 6, further extended with data support and specialized optimizations for Web Services.

7.5 Web Service Modeling

In this section, we present modeling languages which are expressive enough to capture all core features of Web Service choreography and orchestration. There are two reasons for introducing intermediate modeling languages for Web Services. First, heavy languages like WS-CDL or WS-BPEL are designed for machine consumption and therefore are lengthy and complicated in structure. Moreover, there are mismatches between WS-CDL and WS-BPEL. For instance, WS-CDL allows channel passing whereas WS-BPEL does not. The intermediate languages focus on the interactive behavioral aspect. The languages are developed based on previous works of formal models for WS-CDL and WS-BPEL [48, 172, 169]. They cover all main features like synchronous/asynchronous message passing, channel passing, process forking, parallel composition, shared variables, etc. Second, based on the intermediate languages and their semantic models (namely, labeled transition systems), our verification and synthesis approaches is not bound to one particular Web Service language. For instance, newly proposed orchestration languages like Orc [156] can be easily supported. This is important because Web Service languages evolves rapidly. Being based on intermediate languages gives us opportunity to quickly cope with new syntaxes or features (e.g., by tuning the preprocessing component).

7.5.1 Choreography: Syntax and Semantics

The following is the core syntax for building models of Web Service choreography[4], e.g., in WS-CDL. Additional language features like variables, arrays and assertions are illustrated using exam-

[4] The ASCII version of the syntax can be found in PAT user manual.

ples in later sections. Let $\mathcal{I}$ (short of *interaction*), $\mathcal{J}$ be terms of choreography. Let A, B range over Web service roles; ch range over communication channels; svr range over a set of pre-setup service invocation channels (refer to discussion later); $\tilde{ch}$ denote a sequence of channels; x range over variables; b be a predicate over only the variables and exp be an expression.

$$\begin{array}{lll}
\mathcal{I} ::= & Stop & \text{– inaction} \\
\mid & Skip & \text{– termination} \\
\mid & svr(A, B, \tilde{ch}) \rightarrow \mathcal{I} & \text{– service invocation} \\
\mid & ch(A, B, exp) \rightarrow \mathcal{I} & \text{– channel communication} \\
\mid & x := exp;\ \mathcal{I} & \text{– assignment} \\
\mid & if\ b\ \mathcal{I}\ else\ \mathcal{J} & \text{– conditional} \\
\mid & \mathcal{I} \;\Box\; \mathcal{J} & \text{– choice} \\
\mid & \mathcal{I} \;|||\; \mathcal{J} & \text{– service interleaving} \\
\mid & \mathcal{I};\ \mathcal{J} & \text{– sequential}
\end{array}$$

We assume that each role is associated with a set of local variables and there are no globally shared variables among roles. This is a reasonable assumption as each role (which is a service) may be realized in a remote computing device. Informally, $svr(A, B, \tilde{ch})$, where svr is pre-defined service invocation channel, states that role A invokes a service provided by role B through channel svr. A service invocation channel is one which is registered with a service repository so that the service is subject for invocation. $\tilde{ch}$ is a sequence of session channels which are created for this service invocation only. Notice that because the same service shall be available all the time, service channel svr is reserved for service invocation only. $ch(A, B, exp)$ where ch is a session channel states that role A sends the message exp to role B through channel ch.

$x := exp$ assigns the value of exp to the variable x. Without loss of generality, we always require that the variables constituting exp and x must be associated with the same role[5]. If b evaluates to true, $if\ b\ \mathcal{I}\ else\ \mathcal{J}$ behaves as $\mathcal{I}$, otherwise $\mathcal{J}$. Given a variable x (a condition b), we write $role(x)$ ($role(b)$) to denote the associated role. $\mathcal{I} \;\Box\; \mathcal{J}$ is an unconditional choice (i.e., choice of two unguarded working units in WS-CDL) between $\mathcal{I}$ and $\mathcal{J}$, depending on whichever executes first. $\mathcal{I} \;|||\; \mathcal{J}$ denotes two interactions running in parallel. Notice that there are no message communications between $\mathcal{I}$ and $\mathcal{J}$. Two choreographes executing in a sequential order is written as $\mathcal{I};\ \mathcal{J}$. We remark that recursion is supported by referencing a choreography name.

[5] This is a well-formedness rule which can be checked easily.

```
1.  $BuySell() = B2S(Buyer, Seller, \{Bch\}) \rightarrow Session();$
2.  $Session() = Bch(Buyer, Seller, QuoteRequest) \rightarrow Bch(Seller, Buyer, QuoteResponse.x) \rightarrow$
3.      if $(x \leq 1000)\{$
4.          $Bch(Buyer, Seller, QuoteAccept) \rightarrow Bch(Seller, Buyer, OrderConfirmation) \rightarrow$
5.          $S2H(Seller, Shipper, \{Bch, Sch\}) \rightarrow$
6.          $(Sch(Shipper, Seller, DeliveryDetails.y) \rightarrow Stop$
7.            $||| \; Bch(Shipper, Buyer, DeliveryDetails.y) \rightarrow Stop)$
8.      $\}$else$\{$
9.          $Bch(Buyer, Seller, QuoteReject) \rightarrow Session()$
10.           $\Box \; Bch(Buyer, Seller, Terminate) \rightarrow Stop$
11.     $\};$
```

Figure 7.1: A sample choreography

The syntax above is expressive enough to capture the core Web service choreography features[6]. For instance, channel passing is supported as we are allowed to transfer a sequence of channels on service invocation. Figure 7.1 presents a sample choreography, which illustrates how to use this language to model Web Services. The choreography coordinates three roles (i.e., $Buyer$, $Seller$ and $Shipper$) to complete a business transaction. At line 1, the $Buyer$ communicates with the $Seller$ through service channel $B2S$ to invoke its service. Channel Bch which is sent along the service invocation is to be used as a session channel for the session only. In the $Session$, the $Buyer$ firstly sends a message $QuoteRequest$ to the $Seller$ through channel Bch. At line 2, the $Seller$ responds with some quotation value x, which is a variable. Notice that in choreography, the value of x may be left unspecified at this point. At line 5, the $Seller$ sends a message through the service channel $S2H$ to invoke a shipping service. Notice that the channel Bch is passed onto the $Shipper$ so that the shipper may contact the $Buyer$ directly. At line 6 and 7, the $Shipper$ sends delivery details to the $Buyer$ and $Seller$ through the respective channels. The rest is self-explanatory.

Given a choreography model, a system configuration is a 2-tuple $(\mathcal{I}, V)$, where $\mathcal{I}$ is a choreography and V is a mapping from the variables to their values, i.e., from data variables to their valuations or from channel variables to channel instances. A transition is expressed in the form of $(\mathcal{I}, V) \xrightarrow{e}$

[6] Higher order processes, which can be used to model Java applet, are excluded. They can be easily integrated into the framework.

$(\mathcal{I}', V')$. The transition rules are presented in Figure 7.2. Rule $inv1$ captures service invocation, where event $svr!\tilde{ch}$ occurs. Afterwards, rule $inv2$ becomes applicable so that the service invoking request is ready to be received. At the same time, a copy of the choreography is forked. This is because a service may be invoked multiple times, possibly simultaneously, by different service users and all service invocations must conform to the choreography. In fact, in the standard practice of Web Services, a service is embodied by a shared channel in the form of URLs or URIs through which many users can throw their requests at any time. For instance, different processes acting as $Buyer$s may invoke the service provided by the $Seller$. All $Buyer$s must follow the communication sequence. Furthermore, in order to match the reality (and make things more interesting), we assume that both service invocation and channel communication are asynchronous in this work. As a result, service invocation (or channel communication) is divided into two events, i.e, the event of issuing a service invocation (or channel output) and the event of receiving a service invocation (or channel input). This is captured by rules $inv1$, $inv2$, $ch1$ and $ch2$. For simplicity, we assume that a function $eval$ returns the value of an expression exp given the valuation of variables V. Rule $assign$ updates variable valuations. Rule ref captures referencing, i.e., if $\mathcal{I}$ is defined to be $\mathcal{J}$, they have the same behavior. The rules for other constructs resembles those for CSP# in Section 3.1, hence ignored.

Given a choreography $\mathcal{I}$, we build a labeled transition system (LTS) $\mathcal{L}^{\mathcal{I}} = (S, init, \rightarrow)$ in the same way as in Section 3.1.2 based on the operational semantic rules. We skip the details here. In order to verify properties about the choreography, we use model checking techniques to explore all traces of the transition system. One complication is that the choreography's behavior may depend on environmental input which is only known during runtime with the execution of an orchestration. For instance, the price quote provided by the $Seller$ is unknown given only the choreography in Figure 7.1. We discuss this issue in Section 7.6.

7.5.2 Orchestration: Syntax and Semantics

A Web Service orchestration $\mathcal{O}$ is composed of multiple roles, each of which is specified as an individual process. A slightly different syntax is used to build orchestration models. The reason is that orchestration takes a local view and therefore all primitive actions are associated with a single

$$\frac{}{(svr(A,B,\tilde{ch}) \to \mathcal{I},\, V) \overset{svr!\tilde{ch}}{\to} (svr?(B,\tilde{ch}) \to \mathcal{I} \mid\mid\mid svr(A,B,\tilde{ch}) \to \mathcal{I},\, V)}\ [\ inv1\ ]$$

$$\frac{}{(svr?(B,\tilde{ch}) \to \mathcal{I},\, V) \overset{svr?\tilde{ch}}{\to} (\mathcal{I},\, V)}\ [\ inv2\ ]$$

$$\frac{}{(ch(A,B,exp) \to \mathcal{I},\, V) \overset{ch!v}{\to} (ch?(B,v) \to \mathcal{I},\, V)}\ [\ ch1\ ]$$

$$\frac{}{(ch?(B,v) \to \mathcal{I},\, V) \overset{ch?v}{\to} (\mathcal{I},\, V)}\ [\ ch2\ ]$$

$$\frac{eval(exp,\, V) = v}{(x := exp;\ \mathcal{I},\, V) \overset{\tau}{\to} (\mathcal{I},\, V' \oplus x \mapsto v)}\ [\ assign\ ]$$

where $\checkmark$ is the special event of termination

Figure 7.2: Choreography structural operational semantics

role. Let P (short of *process*) and Q be the processes, which describe behaviors of a role. Assume that P plays the role A in the orchestration, written as $P@A$.

$$\begin{array}{rll}
P ::= & Stop \mid Skip & \text{– primitives} \\
\mid & inv!\tilde{ch} \to P & \text{– service invoking} \\
\mid & inv?\tilde{x} \to P & \text{– service being invoked} \\
\mid & ch!exp \to P & \text{– channel output} \\
\mid & ch?x \to P & \text{– channel input} \\
\mid & x := exp;\ P & \text{– assignment} \\
\mid & if\ b\ P\ else\ Q & \text{– conditional branching} \\
\mid & P \,\Box\, Q & \text{– orchestration choice} \\
\mid & P \mid\mid\mid Q & \text{– interleaving} \\
\mid & P;\ Q & \text{– sequential}
\end{array}$$

Process $inv!\tilde{ch} \to P$ invokes a service through service channel inv and then behaves as specified by P. Or a service can be invoked by $inv?\tilde{x} \to P$ where $\tilde{x}$ is a sequence of channel variables which store the received channels. A process may send (receive) a message through a channel ch by $ch!exp \to P$ ($ch?x \to P$). To match the reality, we always assume that the communication channels between different processes are asynchronous (and with a fixed buffer size) in this work.

Let c be a channel; C be a channel valuation function; $notempty(c, C)$ be true iff the buffer is not empty; $notfull(c, C)$ be true iff the buffer is not full; $top(c, C)$ be the top element in the buffer. For simplicity, let $C \oplus (c, x)$ be $C \oplus c \mapsto C(c) \frown \langle x \rangle$ (where $\frown$ is sequence concatenation), i.e., adding the element to the respective channel. Similarly, let $C \ominus c$ be the resultant channel valuation function with the top element in c removed.

$$\frac{notfull(inv, C)}{(inv!\tilde{ch} \rightarrow P, V_A, C) \overset{inv!\tilde{ch}}{\rightarrow} (P \mid\mid\mid (inv!\tilde{ch} \rightarrow P), V_A, C \oplus (inv, \tilde{ch}))} [\ invoking\ ]$$

$$\frac{notempty(inv, C)}{(inv?\tilde{x} \rightarrow P, V_A, C) \overset{inv?top(inv, C)}{\rightarrow} (P, V_A \oplus \tilde{x} \mapsto top(inv, C), C \ominus inv)} [\ invoked\ ]$$

$$\frac{notfull(ch, C)}{(ch!exp \rightarrow P, V_A, C) \overset{ch!exp}{\rightarrow} (P, V_A, C \oplus (ch, exp))} [\ output\ ]$$

$$\frac{notempty(ch, C)}{(ch?x \rightarrow P, V_A, C) \overset{ch?top(ch, C)}{\rightarrow} (P, V_A \oplus x \mapsto top(ch, C), C \ominus ch)} [\ input\ ]$$

Figure 7.3: Orchestration structural operational semantics

The rest are similar to those of choreography.

Similarly, we define the operational semantics. Let V_A be the valuation of the variables associated with the role A. Let C be a valuation function of the channels, which maps a channel to the sequence of items in the buffer. C is a set of tuples of the form $c \mapsto \tilde{msg}$. A configuration of the process is a 3-tuple (P, V_A, C). Part of the firing rules for local steps of a process are presented in Figure 7.3 and the rest are skipped for the their similarity with the constructs in CSP# (refer to Section 3.1.2). We remark that as in choreography, service invocation in orchestration forks a new copy of the service (see rule inv in Figure 7.3) and thus allows potentially many concurrent service invocations. In reality, however, the number of overlapping service invocations is bounded by the maximum number of threads the underlying operating system allows [93]. In next section, we discuss how to capture this constraint and at the same time perform efficient verification. Because an orchestration

```
$Role\ Buyer\ \{$
    $var\ counter = 0;$
    $Main()$      $= B2S!\{Bch\} \rightarrow Session();$
    $Session()$   $= Bch!QuoteRequest \rightarrow counter{+}{+};\ Bch?QuoteResonse.x \rightarrow$
                  **if** $(x \leq 1000)\{$
                      $Bch!QuoteAccept \rightarrow Bch?OrderConfirmation$
                          $\rightarrow Bch?DeliveryDetails.y \rightarrow Stop$
                  $\}$
                  **else if** $(counter > 3)\{Bch!QuoteReject \rightarrow Session()\}$ **else** $\{Stop\};$
$\}$
$Role\ Seller\ \{$
    $var\ x$      $= 1200;$
    $Main()$      $= B2S?\{ch\} \rightarrow Session();$
    $Session()$   $= ch?QuoteRequest \rightarrow ch!QuoteResonse.x \rightarrow (ch?QuoteAccept \rightarrow$
                  $ch!OrderConfirmation \rightarrow S2H!\{ch, Sch\} \rightarrow Sch?DeliveryDetails.y \rightarrow$
                  $Stop \ \Box\ ch?QuoteReject \rightarrow Session());$
$\}$
$Role\ Shipper\ \{$
    $var\ detail$ $=$ “01/11/2009”$;$
    $Main()$      $= S2H?\{ch1, ch2\} \rightarrow$
                  $(ch1!DeliveryDetails.detail \rightarrow Stop\ |||\ ch2!DeliveryDetails.detail \rightarrow Stop);$
$\}$
```

Figure 7.4: A simple orchestration

is the cooperation of multiple roles/processes, the behaviors of the processes must be composed in order to obtain the global behavior. Given two processes, e.g., P and Q, playing different roles, e.g., A and B, the composition is written as $P@A \parallel Q@B$. Figure 7.5 shows the respective semantic rules, i.e., a global step is constituted of a local step by either P or Q.

Following the rules, given an orchestration with multiple roles, each of which is specified as a process defined above, we may build a LTS. The executions of the orchestration equal to the executions the LTS. Similarly, we define traces of an orchestration as τ-filtered traces of the LTS. Given an orchestration $\mathcal{O}$, let $traces(\mathcal{L}^{\mathcal{O}})$ be the set of finite executions. Figure 7.4 presents an orchestration which implements the choreography in Figure 7.1. Each role is implemented as a separate component. Each component contains variable declarations (optional) and process definitions. We assume that the process $Main$ defines the computational logic of the role after initialization. We remark that the orchestration generally contains more details than the choreography, e.g., the variable $counter$ in $Buyer$ constraints the number of attempts the buyer would try before giving up.

7.6 Web Service Conformance Verification

In this section, we define conformance between a choreography and an orchestration based on trace refinement and present an approach to verify it by showing refinement relationships. The verification is performed under the constraints of bounded (many) service invocations.

As discussed above, both choreography and orchestration can be translated into LTSs. An orchestration $\mathcal{O}$ is valid if and only if it is weakly equivalent to the choreography $\mathcal{I}$, i.e., $traces(\mathcal{L}^{\mathcal{O}}) = traces(\mathcal{L}^{\mathcal{I}})$. This can be verified by showing that the orchestration weakly simulates the choreography and *vice versa*. By the assumption that the ranges of the variables are finite and the number of concurrent service invocations are bounded, the LTS has finite number of states. As a result, we can use the refinement checking algorithm proposed in Chapter 6 to check trace refinement. Given a choreography and an orchestration, the algorithm works by constructing the synchronous product of the two LTSs and then performing a reachability analysis. For instance, in order to verify that the orchestration conforms to the choreography, the algorithm searches for a pair of states, one of the choreography and the other of the orchestration, such that the state of the orchestration can perform more (visible) events than that of the choreography. If such a state is found, then we find a counterexample. Otherwise, we establish the refinement relationship.

A main challenge for verifying practical Web Services by model checking is state space explosion. There are multiple causes of state space explosion. Two of them are 1) the numerous different interleaving of processes executing concurrently in service orchestration and 2) the large number[7] of concurrent service invocations. In the following, we discuss two optimization techniques which have been adopted to cope with the above causes.

Firstly, the algorithm is improved with partial order reduction, to reduce the number of possible

[7] An operating system would like dozens or even thousands of concurrent threads.

interleaving (particularly for orchestration). Events performed by single service role (e.g., local variable updates in service choreography or orchestration) are often independent with the rest of the system and hence are subject to reduction. For instance, the action of updating variable $counter$ in Figure 7.4 results in an invisible event, which is independent of actions performed by other roles like $Seller$ or $Shipper$. During model checking, if this action is enabled (together with actions performed by other roles), we only expand the system graph using this action and postpone the rest[8]. By this way, we build a smaller LTS and therefore checks deadlock-freeness, safety and liveness more efficiently. For refinement checking, we apply this reduction in two ways. One is to apply partial order reduction separately to invisible events of either the choreography or the orchestration. Notice that this reduction is trace preserving and therefore is sound for refinement checking. The other one is to apply reduction to visible events of the choreography and the orchestration at the same time.

Secondly, by a simple argument, it can be shown that the following algebraic laws are true; where $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{K}$ are choreographes.

$$\begin{aligned} \mathcal{I} \mid\mid\mid \mathcal{J} &= \mathcal{J} \mid\mid\mid \mathcal{I} \\ (\mathcal{I} \mid\mid\mid \mathcal{J}) \mid\mid\mid \mathcal{K} &= \mathcal{I} \mid\mid\mid (\mathcal{J} \mid\mid\mid \mathcal{K}) \end{aligned}$$

Naturally, different invocations of the same Web Service are similar or even identical. By the above laws, the interleaving of multiple choreographes can be sorted (in certain fixed ordering) without changing the system behaviors. Therefore, if the choreography is in the form of $\mathcal{I} \mid\mid\mid \cdots \mid\mid\mid \mathcal{I} \mid\mid\mid \cdots$, it is equivalent whether the first $\mathcal{I}$ makes a transition or the second does. For verification of deadlock-freeness, safety or liveness properties, it is thus sound to pick one of the transitions and ignore the others. In general, this reduction could reduce the number of states up to the factor of $N!$ where N is the number of identical components. This reduction is inspired by research on model checking parameterized systems [115] and [79]. Furthermore, the process counter abstraction presented in Section 5.4 can also be applied here.

There are a number of other algebraic laws which may help to reduce the number of states (e.g., $\mathcal{I} \;\Box\; \mathcal{J} = \mathcal{J} \;\Box\; \mathcal{I}$). Nonetheless, it is a balance between the computational overhead (for the additional checking) and gain in state reduction. In our implementation (refer to Section 10.3.3),

[8] This reduction is subject to other constraints, e.g., the resultant state must not be on the search stack.

$$\frac{(P, V_A, C) \xrightarrow{e} (P', V_A', C')}{(P@A \parallel Q@B, V_A \cup V_B, C) \xrightarrow{e} (P'@A \parallel Q@B, V_A' \cup V_B, C')}$$

$$\frac{(Q, V_B, C) \xrightarrow{e} (Q', V_B', C')}{(P@A \parallel Q@B, V_A \cup V_B, C) \xrightarrow{e} (P@A \parallel Q'@B, V_A' \cup V_B, C')}$$

Figure 7.5: Process composition rules

a set of specially chosen algebraic laws are used to detect equivalence of system configurations, including the symmetry and associativity laws of $|||$, $\parallel$ and $\Box$, etc.

Choreography may contain free variables (for environment inputs), which must be instantiated during execution time. This is achieved by synchronizing the valuations of the choreography and orchestration whenever a free variable is used. For instance, when checking conformance between the orchestration in Figure 7.4 and the choreography in Figure 7.1, after the $Seller$ sends out the message $QuoteResponse.1200$ (where x is 1200), the variable x in the choreography is instantiated to 1200. Similarly, the session channel names of the service invocations in the choreography must be instantiated to the actual channel names of service invoking in the orchestration. For instance, channel Bch in Figure 7.1 is instantiated to bch after the synchronization step between $B2S(Buyer, Seller, \{Bch\})$ in Figure 7.1 and $B2S!\{bch\}$ in Figure 7.4.

7.7 Experiments of Conformance Checking

We have conducted experiments on multiple case studies, including ones from *www.oracle.com* and from [48, 172]. We are currently applying PAT to several large WS-CDL and WS-BPEL models. We thus demonstrate the scalability of our verification approach, using two models. One is the online store example presented in Figure 7.1 and Figure 7.4. Instead of one buyer and one service invocation, we amend the model so that multiple users are allowed to use the services multiple times. The other is the service for travel arrangement. Its WS-BPEL model and WS-CDL specification are

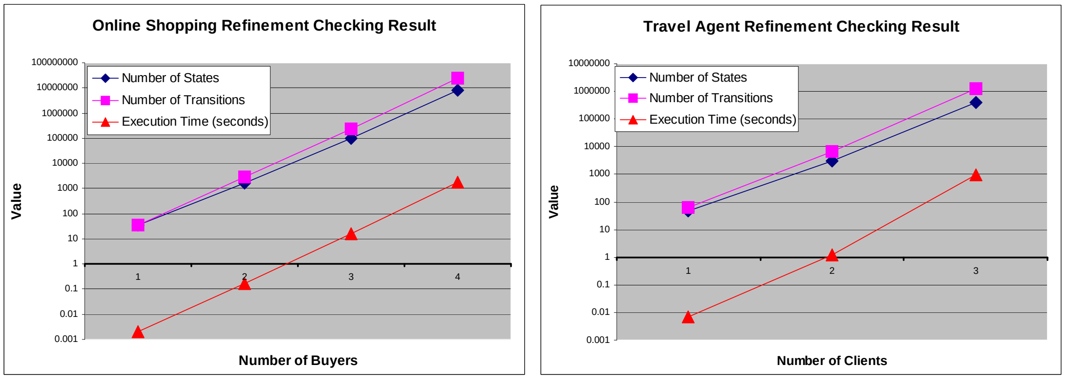


Figure 7.6: Experiments for conformance verification

available at *http://www.comp.nus.edu.sg/~pat/cdl/*. A number of clients invoke the business process, specifying the name of the employee, the destination, the departure date, and the return date. The BPEL process checks the employee travel status (through a Web Service). Then it checks the prices for the flight ticket with multiple airlines (through Web Services). Finally, the BPEL process selects the lowest price and returns the travel plan to each client.

Figure 7.6 shows PAT's efficiency using the two examples, obtained on a PC with Intel Q9500 CPU at 2.83GHz and 4GB RAM. In the online store example, we allow buyers to invoke the service repeatedly. As a result, the orchestration is deadlock-free. In the travel arrangement example, one client invokes the service only once. Because the number of concurrent service invocations is bound by the maximum number of threads allowed, the system reaches a deadlock state after exhausting all threads. This is consistent with the finding in [93]. We verify that the orchestration conforms to the choreography using the refinement checking algorithm, as shown in Figure 7.6. In both cases, the number of states and the verification time increase rapidly. Yet, PAT is able to confirm that the orchestration conforms to the choreography with a few buyers/clients using the service concurrently. In a nutshell, PAT explores 10^7 states in a reasonable amount of time, which suggests that PAT is comparable to FDR in terms of efficiency[9].

[9] LTSA-WS or WS-Engineer takes a different approach and therefore we skip the performance comparison here.

7.8 Summary

In this chapter, we presented two applications of refinement checking.

First, we showed that linearizability can be expressed using a refinement relation, hence can be verified using refinement checking algorithm. Several case studies show that our approach is capable of verifying practical algorithms and identifying bugs in faulty implementations. Several future directions are possible. Algorithms that accept an infinite number of threads or unbound data structures make model checking impossible. Symmetric properties among threads can reduce infinite number of threads to a small number. Shape analysis or integration with thereat prover can also be incorporated into the model checking to handle unbounded data size.

Second, we present model-based methods for fully automatic analysis of Web Service compositions, in particular, linking two different views of Web Services. Our methods build on the strength of advanced model checking techniques. In particular, we verify whether designs of Web Services from two different views are consistent or not, by on-the-fly refinement checking with specialized optimizations.

In terms of modeling of linearizability, our approach is based on the trace refinement of LTSs, which is similar to [12]. The non-atomic refinement defined in [65] separates the data explicitly as state-based formalism Object-Z. This modeling requires to have the knowledge of linearization points, and also prevents automatic verification techniques such as model checking to be used. Linearizability checking is a much studied research area, since it is a central property for the correctness of concurrent algorithms. Herlihy and Wing [107] present a methodology for verifying linearizability by defining a function that maps every state of an concurrent object to the set of all possible abstract values representing it. Vafeiadis et. al. [212] use rely-guarantee reasoning to verify linearizability for a family of implementations for linked lists. Neither of them requires statically determined linearization points, but they are manual. Verification using theorem provers (e.g., PVS) is another approach [69, 59]. In these works, algorithms are proved to be linearizable by using simulation between input/output automata modeling the behavior of an abstract set and the implementation. However, theorem prover based approach is not automatic. Conversion to IO automata and use of

PVS require strong expertise. Wang and Stoller [218] present a static analysis that verifies linearizability for an unbounded number of threads. Their approach detects certain coding patterns, which are known to be atomic regardless of the environment. This solution is not complete (i.e., not applicable to all algorithms). Amit et al. [13] presented a shape difference abstraction that tracks the difference between two heaps. This approach works well if the concrete heap and the abstract heap have almost identical shapes. Recently, Manevich et al. [153] and Berdine et al. [31] extended it to handle larger number and unbounded number of threads, respectively. Vafeiadis [211] further improved this solution to allow linearization points in different threads. The main limitation of these approaches is that users need to provide linearization points, which are unknown for some algorithms. In [215], Vechev and Yahav provided two methods for linearizability checking. The first method is a fully automatic, but inefficient as discussed in Section 7.3. The second method requires algorithm-specific user annotations for linearization points, which is not generic.

Web Service conformance checking is related to research on verifying Web Services, particularly, the line of work by Foster *et al* presented in [94, 95, 93, 91]. They proposed to apply model-based verification for Web Services. Their approach is to build Finite State Processes (FSP) model from Web services and then apply verification techniques based on FSP to verify Web Services. For instance, conformance between choreography and orchestration is verified by showing a bi-simulation relationship between the respective FSP models. In particular, they identified the model of resource constraint in Web Service verification [93] and proposed to perform verification under resource constraints. In addition, they developed a tool named LTSA-WS [95] (and later WS-Engineer [92]). Our work can also be categorized as model-based verification, and is similar to theirs. Our approach complements their works in a number of aspects. Firstly, our model is based on a modeling language which is specially designed for Web service composition with features like channel passing, shared variables/arrays, service invocation with service replication, etc. Secondly, our verification algorithms employ specialized optimizations for Web Services verification, e.g., model reduction based on algebraic properties of the models, partial order reduction for orchestration with multiple local computational steps, etc. These optimizations allow us to handle large state space and potentially large Web Services. Our work is remotely related to formal modeling and verification of Web Services [73, 157, 169, 123]. Our languages are inspired from the languages proposed in [48, 172].

Chapter 8

Bounded Model Checking of Compositional Processes

Model checking techniques presented in previous chapters rely on exhaustive search through explicit representations of the state space and suffers from the state space explosion problem. To overcome this problem, verification techniques like SAT-based bounded model checking [54] have been proved to be successful in verifying a variety of system models.

Applying bounded model checking to compositional process algebras is, however, not a trivial task. One challenge is that the number of system states for process algebra models is not statically known, whereas exploring the full state space is computationally expensive. This chapter presents a compositional encoding of hierarchical processes as SAT problems and then applies state-of-the-art SAT solvers for bounded model checking. The encoding avoids exploring the full state space for complex systems so as to deal with state space explosion. The bounded model checking technique is used to validate system models against event-based temporal properties. The experiment results show that this approach can handle large systems.

The remainder of this chapter is organized as follows. Section 8.1 discusses some background of bounded model checking. Section 8.2 presents how to encode the semantics of compositional processes as Boolean formulae at the same time avoiding state space explosion. Section 8.3 introduces the encoding of the LTL properties and the verification problem. Section 8.4 presents the experimental results. Section 8.5 concludes this chapter.

8.1 Background

The original proposal of model checking relies on exhaustive search through explicit representations of reachable system states [58], which is known as *explicit model checking*. It suffers from the *state space explosion* problem. Later, *symbolic model checking* was proposed to overcome this problem by enumerating states symbolically (typically based on the notion of BDDs [42, 44]).

However, human intervention may be required to fine-tune the variable ordering so as to reduce the size of BDDs. In recent years, *bounded model checking* [54] have been proposed to complement explicit model checking and symbolic model checking with great success. The idea is to encode finite state machines (as well as the properties to be verified) as a Boolean formula that is satisfiable if and only if the underlying state machine can realize a finite sequence of transitions that reaches states of interest, and then apply state-of-the-art SAT solvers [2] to produce counterexamples (if any) efficiently. If such a path segment cannot be found at a given length k, the search is continued for larger k. With the rapid development of SAT solvers, we believe bounded model checking is promising for formal verification.

Previous works on model checking have been historically centered around state machines. Model checking techniques have only been applied to event-based formalisms to a limited extent. To our best knowledge, bounded model checking has not yet been applied to event-based languages like CSP [108] or CCS [155]. One of the reasons is that unlike in circuit verification [53] (where encoding the transition relation is rather straightforward), encoding the semantics of compositional processes using Boolean formulae is nontrivial. The number of system states for process algebra models is not statically known and exploring the full state space is computationally expensive. This chapter presents a compositional encoding of hierarchical processes as SAT problems. State-of-the-art SAT solvers are then applied for bounded model checking. The encoding avoids exploring the full state space for complex systems so as to avoid state space explosion. Based on the idea, we have implemented bounded model checking in PAT. The advantages of applying bounded model checking instead of symbolic model checking include that SAT tools usually need far less hand manipulation than BDDs. The experiment results show that our toolkit has a competitive performance for verifying systems with large number of states.

In this chapter, we only consider a subset of CSP# with no variables and channels (i.e., essentially the original CSP language). Because the newly added features in CSP#, e.g., variables, channels, etc., increase the complexity of the encoding process, which makes the bound model checking ineffective. This is also the reason that we do not pursue this direction further. With the rapid development of SAT solver, we believe this bottleneck can be resolved.

8.2 Encoding of Processes

This section is dedicated to a discussion on how to encode a given process P in CSP# as Boolean formulae for bounded model checking. We start with encoding simple processes by explicitly building their labeled transition system $\mathcal{L}^P$ and then discuss how to encode processes for which building $\mathcal{L}^P$ is not feasible. Note that variables and channels are ignored in this chapter. Hence system configurations have only the process component. We remark that the encoding technique is not restricted to CSP#.

8.2.1 Encoding Simple Processes

A process P can be encoded by firstly constructing $\mathcal{L}^P$ and then encoding $\mathcal{L}^P$. Given a LTS, a property to verify and a bound k, we need to translate the LTS and the negation of the property into a propositional formula which is satisfiable if and only if there is a trace of length k which violates the property (i.e., a counterexample). Thus, we need to find an efficient encoding of states, events, and the transition relation. Given $\mathcal{L} = (S, init, \rightarrow)$, we need $\lceil \log_2 \#S \rceil$ Boolean variables to encode the states. Let $\overrightarrow{xs}_i = \langle xs_i^1, xs_i^2, \cdots \rangle$ be a finite sequence of Boolean variables used to encode the states reached after $i - 1$ steps. The encoding of a state is a Boolean formula π over $\overrightarrow{xs}_i$ such that $\pi(\overrightarrow{xs}_i) = 1$ if and only if the valuation of the variables uniquely identifies the state. Or equivalently, a state is associated with a unique binary number and each Boolean variable represents one bit of the number. Similarly, we use $\lceil \log_2 \#\alpha\mathcal{L} \rceil$ Boolean variables to encode the alphabet of $\mathcal{L}$. Let $\overrightarrow{xe}_i$ be the variables used to encode the events. A transition is encoded as a Boolean formula of the following form,

$$\pi(\overrightarrow{xs}_i) \land \pi(\overrightarrow{xe}_i) \land \pi(\overrightarrow{xs}_{i+1})$$

where $\overrightarrow{xs}_{i+1}$ is a set of fresh variables used to encode the post-state. Let Π denote the encoding function which maps a state or event to a Boolean formula given a set of Boolean variables. $\Pi(P1, \overrightarrow{xs}_i) \land \Pi(e, \overrightarrow{xe}_i) \land \Pi(P_2, \overrightarrow{xs}_{i+1})$ where $(P_1, e, P_2) \in \rightarrow$ is the Boolean coding of the transition in the above form. Informally, this formula guarantees that if the transition is to be taken, the pre/post-state and event must be P_1/P_2 and e respectively. The transition relation $\rightarrow$ is encoded as the disjunction of all possible transitions, i.e., any encoded transition may be taken if it can be satisfied.

$$\mathcal{T}_i = \bigvee \{\Pi(P1, \overrightarrow{xs}_i) \land \Pi(e, \overrightarrow{xe}_i) \land \Pi(P_2, \overrightarrow{xs}_{i+1}) \mid (P_1, e, P_2) \in \rightarrow\}$$

Given a bound k for bounded model checking, the encoded transition relation must be applied k-times. Every time a fresh set of variables must be used to encode the engaged event as well as the target state. Thus, we need $(k+1) \times \lceil \log_2 \#S \rceil + k \times \lceil \log_2 \#\Sigma \rceil$ Boolean variables to represent state $s_1 \mathinner{..} s_{k+1}$ and $e_1 \mathinner{..} e_k$ where $s_1 = init$.

Definition 24 *An encoding of a LTS is 4-tuple $\mathcal{E} = (\mathcal{I}, \mathcal{T}_i, \overrightarrow{xs}_i, \overrightarrow{xe}_i)$ where $\mathcal{I} = \Pi(init, \overrightarrow{xs}_1)$ is the encoded initial state, $\mathcal{T}_i$ is the encoded transition relation as defined above, $\overrightarrow{xs}_i$ are the variables used to encode the source state of $\mathcal{T}_i$ and $\overrightarrow{xe}_i$ are the variables used to encode the labeling events of transitions of $\mathcal{T}_i$.*

Given a LTS $\mathcal{L}$ and its encoding $\mathcal{E}$, we say $\mathcal{E}$ is sound if and only if $\mathcal{E}$ and $\mathcal{L}$ are trace-equivalent, i.e., every trace allowed by $\mathcal{L}$ must be allowed by $\mathcal{E}$ and *vice versa*. The above encoding of a LTS is sound as we can show that the encoded transition relation conforms to $\rightarrow$ and the encoded initial condition conforms to $init$. Given an encoding of the system $\mathcal{E}$, a property ϕ to verify and a bound k, the propositional formula constructed is of the following form,

$$[\![\mathcal{E}, \phi]\!]_k \mathrel{\widehat{=}} \mathcal{I} \wedge \bigwedge_{i=1}^{k} \mathcal{T}_i \wedge [\![\neg\, \phi]\!]_k$$

where $[\![\neg\, \phi]\!]_k$ is the encoded negation of the given property (with regards to k). We leave it to Section 8.3 for detailed discussion. A satisfiability solution to the above formula gives a counterexample of the property, which satisfies the initial condition and the transition relation up to k-steps and violates the property.

In the following,we use $\mathcal{E}^P$ to denote the encoding of P. Explicitly constructing $\mathcal{L}^P = (S^P, init^P, \rightarrow^P)$ is however not always desirable for several reasons. Firstly, S^P (and therefore $\rightarrow^P$) may not be finite. For instance, processes like $P \mathrel{\widehat{=}} b \rightarrow Skip \;\Box\; (a \rightarrow P;\; c \rightarrow Skip)$ or $P \mathrel{\widehat{=}} a \rightarrow (P \mid\mid\mid P)$ allow unbounded recursion or replication and, thus, may result in infinite reachable process expressions. Our experiences, however, show that processes of the above forms are rather rare in practice. Without loss of generality, we assume that S^P is always finite. Optimally, the number of Boolean variables needed to encode S^P is $\lceil \log_2 \#S^P \rceil$. However, determining the exact size of S^P requires traversing through all reachable states, which is often undesirable due to state space explosion. For instance, assume $\#S^Q = n$, the interleaving of m copies of Q (say P) has n^m states. One remedy is to encode the $\mathcal{L}^Q$ (if its size is manageable) and then compose $\mathcal{E}^Q$ to generate $\mathcal{E}^P$ so as to avoid constructing $\mathcal{L}^P$.

8.2.2 Composing Encodings

A rich set of operators can be used to compose processes as illustrated in Section 3.1. Among all operators, the indexed parallel composition or indexed interleaving (which we refer to as indexed concurrency) causes state space explosion. Given P which contains indexed concurrency, instead of building $\mathcal{L}^P$ we shall deduce $\mathcal{E}^P$ from the encoding of its sub-components. In the following, we show how to compose the encoding of sub-components for various composition. In order to draw connections between transitions of different processes running in parallel, a global event-to-Boolean encoding is established beforehand. In the following, let $\Pi(e, \overrightarrow{xe})$ be the formula encoding e using variables $\overrightarrow{xe}$. Given $\overrightarrow{xs}_i = \langle xs_i^1, xs_i^2, \cdots, xs_i^n \rangle$ where $i \in \{1, 2\}$ as two sequences of Boolean variables of the same length, we write $\overrightarrow{xs}_1 \Leftrightarrow \overrightarrow{xs}_2$ to mean $xs_1^1 \Leftrightarrow xs_2^1 \wedge xs_1^2 \Leftrightarrow xs_2^2 \wedge \cdots \wedge xs_1^n \Leftrightarrow xs_2^n$. To further abuse notations, we write $\overrightarrow{xs}_1 \cup \overrightarrow{xs}_2$ to denote the sequence of variables which contains both variables in $\overrightarrow{xs}_1$ and $\overrightarrow{xs}_2$ and is sorted according to the unique variables ID.

Definition 25 *Let* $P = |||_{j=1}^n P_j$. *Let* $\mathcal{E}^{P_j} = (\mathcal{I}^{P_j}, \mathcal{T}_i^{P_j}, \overrightarrow{xs}_i^{P_j}, \overrightarrow{xe}_i^{P_j})$. *The encoding of* P *is* $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$, *where* $\mathcal{I}^P = \bigwedge_{j=1}^n \mathcal{I}^{P_j}$, $\overrightarrow{xs}_i^P = \bigcup_{j=1}^n \overrightarrow{xs}_i^{P_j}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^{P_j}$ *and* $\mathcal{T}_i^P = \bigvee_{j=1}^n (\mathcal{T}_i^{P_j} \wedge \bigwedge_{m \neq j} (\overrightarrow{xs}_i^{P_m} \Leftrightarrow \overrightarrow{xs}_{i+1}^{P_m}))$.

Note that the variables used to encode each P_j are disjoint. The encoded initial condition of P is the conjunction of the encoded initial conditions of each sub-component. Intuitively, this says that when the composition is initialized, all sub-components must be at its initial state. The predicate $\overrightarrow{xs}_i^{P_m} \Leftrightarrow \overrightarrow{xs}_{i+1}^{P_m}$ means that P_m remains in the same state. The encoded transition relation is the disjunction of a set of clauses, each of which states that a transition of P_j may be taken and the states of other sub-components are unchanged. Thus, any transition of a sub-component can be taken without affecting other sub-components. Indexed parallel composition is handled similarly. The complication is that the alphabet of a sub-component may actually contain more events than those constitute the process expression. The following definition shows that by manipulating the encoded transition relations of the sub-components, the encoded transition relation of the composition shall be exactly the conjunction of the encoded transition relations of the sub-components.

Definition 26 *Let* $P = \|_{j=1}^n P_i$. *Let* $\mathcal{E}^{P_j} = (\mathcal{I}^{P_j}, \mathcal{T}_i^{P_j}, \overrightarrow{xs}_i^{P_j}, \overrightarrow{xe}_i^{P_j})$. *The encoding of* P *is* $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ *where* $\mathcal{I}^P = \bigwedge_{j=1}^n \mathcal{I}^{P_j}$, $\overrightarrow{xs}_i^P = \bigcup_{j=1}^n \overrightarrow{xs}_i^{P_j}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^{P_j}$, *and* $\mathcal{T}_i^P$ *is defined as follows,*

$$\mathcal{T}_i^P = \bigwedge_{j=1}^n (\mathcal{T}_i^{P_j} \vee \bigvee \{\Pi(e, \overrightarrow{xe}_i) \wedge \overrightarrow{xs}_i \Leftrightarrow \overrightarrow{xs}_{i+1} \mid e \notin \alpha P_j \wedge e \in \alpha P \cup \{\tau\}\})$$

The transition relation $\mathcal{T}_i^{P_j}$ is extended with clauses to allow events not in αP_i but in αP (including τ) to occur freely without changing the status of this sub-component. Because the encoded transition relation of each sub-component is conjuncted and we use the same set of variables to encode the events, an event can be engaged if and only if every sub-component participates in it. This construction guarantees that the encoded transition relation of the composition allows only runs which conform to the semantics. It avoids constructing $\mathcal{L}^P$ by paying the price of extra transitions.

Example 8.2.1 The following specifies the classic dining philosophers problem [108],

$$\begin{array}{ll} \mathit{Phil}(i) & = \mathit{think}.i \rightarrow get.i.(i+1)\%N \rightarrow get.i.i \\ & \quad \rightarrow eat.i \rightarrow put.i.(i+1)\%N \rightarrow put.i.i \rightarrow \mathit{Phil}(i); \\ \mathit{Fork}(i) & = get.i.i \rightarrow put.i.i \rightarrow \mathit{Fork}(i) \;\Box \\ & \quad get.((i-1)\%N).i \rightarrow put.((i-1)\%N).i \rightarrow \mathit{Fork}(i); \\ \mathit{Phils}(N) & = \|_{i=0}^{N-1}(\mathit{Phil}(i) \parallel \mathit{Fork}(i)); \end{array}$$

where N is the number of philosophers, $get.i.j$ ($put.i.j$) is the action of the i-th philosopher picking up (putting down) the j-th fork. Assuming $N = 5$ and x_1, x_2, x_3 are used to encode the events, the event encoding is shown in the following table (and the rest are ignored for brevity).

Event	**Encoding**	**Event**	**Encoding**
$\mathit{think}.0$	$\neg x_1 \wedge \neg x_2 \wedge \neg x_3$	$put.0.1$	$x_1 \wedge \neg x_2 \wedge \neg x_3$
$get.0.1$	$\neg x_1 \wedge \neg x_2 \wedge x_3$	$put.0.0$	$x_1 \wedge \neg x_2 \wedge x_3$
$get.0.0$	$\neg x_1 \wedge x_2 \wedge \neg x_3$	$get.4.0$	$x_1 \wedge x_2 \wedge \neg x_3$
$eat.0$	$\neg x_1 \wedge x_2 \wedge x_3$	$put.4.0$	$x_1 \wedge x_2 \wedge x_3$

The following is the encoded transition relation $\mathcal{T}_1^{\mathit{Phil}(0) \| \mathit{Fork}(0)}$.

$$\begin{array}{l} \mathcal{T}_1^{\mathit{Phil}(0)} \wedge \mathcal{T}_1^{\mathit{Fork}(0)} \\ \vee\ (x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4 \Leftrightarrow x_7 \wedge x_5 \Leftrightarrow x_8 \wedge x_6 \Leftrightarrow x_9) \\ \vee\ (x_1 \wedge x_2 \wedge x_3 \wedge x_4 \Leftrightarrow x_7 \wedge x_5 \Leftrightarrow x_8 \wedge x_6 \Leftrightarrow x_9)) \\ \vee\ (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13}) \\ \vee\ (\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13})) \\ \vee\ (\neg x_1 \wedge x_2 \wedge x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13})) \\ \vee\ (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_{10} \Leftrightarrow x_{12} \wedge x_{11} \Leftrightarrow x_{13})) \end{array}$$

where x_4, x_5, x_6 (x_7, x_8, x_9) are used to encode the pre-state (post-state) of $\mathit{Phil}(0)$ and x_{10}, x_{11} (x_{12}, x_{13}) are used to encode the pre-state (post-state) of $\mathit{Fork}(0)$. **end**

A large class of systems can be specified as an indexed parallel composition or indexed interleaving of multiple sub-components which have relatively small number of states, e.g., Phils_N is specified as an indexed parallel composition of philosopher and fork fairs. For such systems, we encode each sub-component by explicitly constructing the LTS and then apply the above construction to build the composed transition relation. Nonetheless, if a process P which contains indexed concurrency is further composed with other processes using operators like $\triangle$, $\Box$ and ; , or amended using operators like $\backslash$, we shall be able to deduce the encoding of the composition from the encoding of P (and others). For instance, assuming the given process is $Phils(N);\ Q$, we must not explore all states of $Phils(N)$ in order to encode the given process.

Definition 27 *Let $P = M \,\Box\, N$. Let $\mathcal{E}^M = (\mathcal{I}^M, \mathcal{T}_i^M, \overrightarrow{xs}_i^M, \overrightarrow{xe}_i^M)$ be the encoding of M. Let $\mathcal{E}^N = (\mathcal{I}^N, \mathcal{T}_i^N, \overrightarrow{xs}_i^N, \overrightarrow{xe}_i^N)$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \mathcal{I}^M \wedge \mathcal{I}^N$, $\overrightarrow{xs}_i^P = \overrightarrow{xs}_i^M \cup \overrightarrow{xs}_i^N \cup \{xc_i\}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^M = \overrightarrow{xe}_i^N$, and $\mathcal{T}_i^P = (xc_i \wedge \mathcal{T}_i^M \wedge xc_{i+1}) \vee (\neg\, xc_i \wedge \mathcal{T}_i^N \wedge \neg\, xc_{i+1})$ where xc_i is a fresh control variable.*

The encoded initial condition has no constraints on xc_1 and thus xc_1 can be either true or false initially (which means transitions from P or Q can be taken). Once one of the choices has been taken, xc_i remains the same as xc_1 for all i and thus a later step must respect the choice made at the first step. This captures the semantics of choices. Note that $\sqcap$ is handled in the same way as $\Box$, and $\sqcap$ is equivalent to $\Box$ in the trace semantics [108].

Definition 28 *Let $P = M \triangle N$. Let $\mathcal{E}^M = (\mathcal{I}^M, \mathcal{T}_i^M, \overrightarrow{xs}_i^M, \overrightarrow{xe}_i^M)$ be the encoding of M. Let $\mathcal{E}^N = (\mathcal{I}^N, \mathcal{T}_i^N, \overrightarrow{xs}_i^N, \overrightarrow{xe}_i^N)$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \mathcal{I}^M \wedge \mathcal{I}^N$, $\overrightarrow{xs}_i^P = \overrightarrow{xs}_i^M \cup \overrightarrow{xs}_i^N \cup \{xc_i\}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^M = \overrightarrow{xe}_i^N$, and $\mathcal{T}_i^P = (\neg\, xc_i \wedge \mathcal{T}_i^M \wedge \neg\, xc_{i+1}) \vee (\mathcal{T}_i^N \wedge xc_{i+1})$ where xc_i is a fresh control variable.*

Interrupt can be viewed as a biased choice. Note that $\neg\, xc_i$ is true if and only if M has not yet been interrupted. If a transition of T_M is taken, xc_{i+1} remains false so that next transition can be taken from T_M or T_N. Whenever a transition of T_N is taken, xc_{i+1} must be true, which forbids all transitions from T_M.

Definition 29 *Let $P = M;\ N$. Let $\mathcal{E}^M = (\mathcal{I}^M, \mathcal{T}_i^M, \overrightarrow{xs}_i^M, \overrightarrow{xe}_i^M)$ be the encoding of M. Let $\mathcal{E}^N = (\mathcal{I}^N, \mathcal{T}_i^N, \overrightarrow{xs}_i^N, \overrightarrow{xe}_i^N)$. The encoding of P is $(\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i^P)$ where $\mathcal{I}^P = \mathcal{I}^M \wedge \neg\, xc_1$, $\overrightarrow{xs}_i^P = \overrightarrow{xs}_i^M \cup \overrightarrow{xs}_i^N \cup \{xc_i\}$, $\overrightarrow{xe}_i^P = \overrightarrow{xe}_i^M = \overrightarrow{xe}_i^N$, and $\mathcal{T}_i^P$ is defined as follows: where xc_i is a fresh variable,*

$$\neg\, xc_i \wedge \mathcal{T}_i^M \wedge (\neg\, \Pi(\checkmark, \overrightarrow{xe}_i) \wedge \neg\, xc_{i+1} \vee \Pi(\checkmark, \overrightarrow{xe}_i) \wedge xc_{i+1} \wedge \mathcal{I}^N) \vee xc_i \wedge \mathcal{T}_i^N$$

Initially, $\neg\, xc_1$ must be true. Note that $\neg\, xc_i$ is true if and only if M has not yet terminated. Intuitively, a sequential composition can be viewed a delayed choice whereby transitions from N can only be taken after a $\checkmark$ transition has been taken in M. xc_i and $\mathcal{I}^N$ is true once a transition of M labeled with $\checkmark$ has been taken. Because transitions from M are guarded with $\neg\, xc_i$, no transition from M can be taken afterwards.

Other compositional operators are handled similarly by manipulating the encoded transition relations of the sub-components and introducing control variables if necessary. For instance, if some events of an encoded process are to be hidden (i.e., $P \setminus A$), those events are renamed, i.e., the label e of a transition is encoded as $\Pi(\tau_e, \overrightarrow{xe_i})$ instead of $\Pi(e, \overrightarrow{xe_i})$. Note that hiding different events results in different τ transitions. This prevents synchronization between different hidden events.

Given a process P, if P contains no indexed concurrency, we construct $\mathcal{L}^P$ and then apply the encoding in Section 3.1. Otherwise, each sub-component of the indexed interleaving or parallel composition is encoded first. The encoding of P is then composed by applying the compositional encoding. If a sub-component of the indexed interleaving or parallel (say Q) contains indexed concurrency as well, the same procedure is repeated so as to encode Q. Note that for processes like $P = a \rightarrow P \;|||\; \cdots \;|||\; P$, this construction is not feasible (and thus we have to construct $\mathcal{L}^P$). Nonetheless, for most interesting systems in which there is no unbounded replication (or recursion), this construction not only terminates but results in Boolean formulae of manageable size, which can be efficiently solved by SAT-solvers.

Theorem 8.2.2 *Let P be a process. $\mathcal{E}^P$ is the encoding of P as defined above. $\mathcal{E}^P$ is trace-equivalent to $\mathcal{L}^P$.*

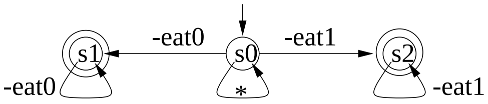
This theorem states that our encoding is sound. It is proved by structural induction. The base case is when a process contains no indexed concurrency, i.e., the encoding in Definition 24 is sound. Then we prove the induction step by showing the compositional encoding preserves the equivalence. We skipped the proof for brevity.

8.3 LTL Properties Encoding and Verification

In this section, we focus on LTL verification using bounded model checking techniques.

Let $B^{\neg\phi}$ be the Büchi automaton constructed from a LTL property $\neg\phi$. In the explicit model checking approach, the product of $B^{\neg\phi}$ and P is generated (same as in SPIN). Explicit model checking is to determine the emptiness of $\mathcal{L}^P \times B$, i.e., explore on-the-fly whether the product contains a loop which is composed of at least one accepting state. Finite traces are extended to infinite ones in a standard way. In the presence of a counterexample, on-the-fly model checking usually produces a trace leading to a bad state or a loop quickly (refer to Section 8.4). However, the counterexample produced may be extremely long because it relies on a depth first search. Bounded model checking may then be used to produce a shorter trace which leads to the same bad state or loop. Though because of the Büchi automata, the generated counterexample may not be the shortest. Nonetheless, our bounded model checker can be used as a separate model checker.

Example 8.3.1 The following is the Büchi Automaton generated from the negation of the formula $\Box\Diamond eat_0 \wedge \Box\Diamond eat_1$,



where $s0$ is the initial state, $s1$ and $s2$ are two accepting states and $*$ means the transition is unguarded. $\neg\, e$ means the transition can be labeled with any event but e. **end**

For bounded model checking, because $\mathcal{L}^P$ may not be built, we encode the Büchi automaton and then compose it with the encoding of the model (i.e., $\mathcal{E}^P$). Given a guard g labeled with a transition of B, let $\Pi(g, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B)$ be the encoded guard, e.g., $\Pi(e, \overrightarrow{xe}_i^B)$ if g is an event e or $\neg\,\Pi(e, \overrightarrow{xe}_i^B)$ if g is an event $\neg\, e$ or $\Pi(g_1, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B) \wedge \Pi(g_2, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B)$ if g is $g_1 \wedge g_2$.

Definition 30 *Let P be a process. Let $\mathcal{E}^P = (\mathcal{I}^P, \mathcal{T}_i^P, \overrightarrow{xs}_i^P, \overrightarrow{xe}_i)$. Let B be a Büchi automaton (S, I, T, F) . $\mathcal{E}^B = (\mathcal{I}^B, \mathcal{T}_i^B, \overrightarrow{xs}_i^B, \overrightarrow{xe}_i)$ where*

$$\mathcal{T}_i^B = \bigvee\{\Pi^B(s, \overrightarrow{xs}_i^B) \wedge \Pi(g, \overrightarrow{xe}_i^B, \overrightarrow{xs}_{i+1}^B) \wedge \Pi^B(s', \overrightarrow{xs}_{i+1}^B) \mid (s, g, s') \in T\}$$

The encoding of the product of P and B is $(\mathcal{I}, \mathcal{T}_i, \overrightarrow{xs}_i, \overrightarrow{xe}_i)$ where $\mathcal{I} = \mathcal{I}^B \land \mathcal{I}^P$, $\overrightarrow{xs}_i = \overrightarrow{xs}_i^P \cup \overrightarrow{xs}_i^B$ and $\mathcal{T}_i = \mathcal{T}_i^P \land \mathcal{T}_i^B$.

Because P and B share the same alphabet as well as the variables to encode the events, transitions of P and B are always synchronized. P violates the property if and only if the language of $P \times B$ is not empty. Let $\mathcal{F}^B(\overrightarrow{xs_i}) = \bigvee\{\Pi^B(s, \overrightarrow{xs_i}) \mid s$ is an accepting state of B$\}$ be the encoded accepting states. The following theorem states the correctness of our bounded model checking.

Theorem 8.3.2 *Given a process P, and a Büchi automaton B constructed from $\neg\,\phi$, let $\mathcal{E}^{P \times B} = (\mathcal{I}, \mathcal{T}_i, \overrightarrow{xs}_i, \overrightarrow{xe}_i)$ be the encoding of $P \times B$. Let k to be a bound. The following formula is satisfiable iff there is a counterexample of size k.*

$$[\![P, \phi]\!]_k = \mathcal{I} \land \bigwedge_{i=1}^{k} \mathcal{T}_i \land [\![\neg\,\phi]\!]_k,\ where\ [\![\neg\,\phi]\!]_k \textit{ is } \bigvee_{i=1}^{k-1}\{\overrightarrow{xs_k} \Leftrightarrow \overrightarrow{xs_i} \land \bigvee_{j=i}^{k}\{\mathcal{F}^B(\overrightarrow{xs_j})\}\}$$

The proof is sketched in the following. A solution to $[\![P, \phi]\!]_k$ is an assignment of *true* or *false* to $\bigcup_{i=1}^{k+1} \overrightarrow{xs}_i$ and $\bigcup_{i=1}^{k} \overrightarrow{xe}_i$ as well as the control variables (if any), from which we can identify a finite run $\langle s_1, e_1, s_2, e_2, \cdots, s_k, e_k, s_{k+1}\rangle$. Because $\mathcal{I}$ must be true, s_1 is an initial state. Because $\mathcal{T}_i$ must be true, by Theorem 8.2.2, $s_i \overset{e_i}{\Rightarrow} s_{i+1}$ for all $1 \leq i \leq k$. Thus, the sequence of states/events identified must be a run of P (as well as a finite prefix of a trace of ϕ). The constraint $[\![\neg\,\phi]\!]_k$ states that the finite run must contain a loop, i.e., $xs_k \Leftrightarrow xs_i$ for some i, and the loop must contain at least one accepting state, i.e., there exists some j satisfying $j \geq i \land j \leq k$ such that s_j is accepting. Therefore, the finite run identifies an infinite trace which is allowed by P and violates ϕ.

8.4 Experiments

In this section, we present a number of experiments to show the feasibility of applying SAT-based model checking to process algebras. The effectiveness of our compositional encoding is straightforward. If compositional encoding is applied, the encoding time is often negligible and the size of the formula is comparable to that of the formula generated by constructing the LTS[1].

[1] Depends on whether the processes are strongly coupled or not.

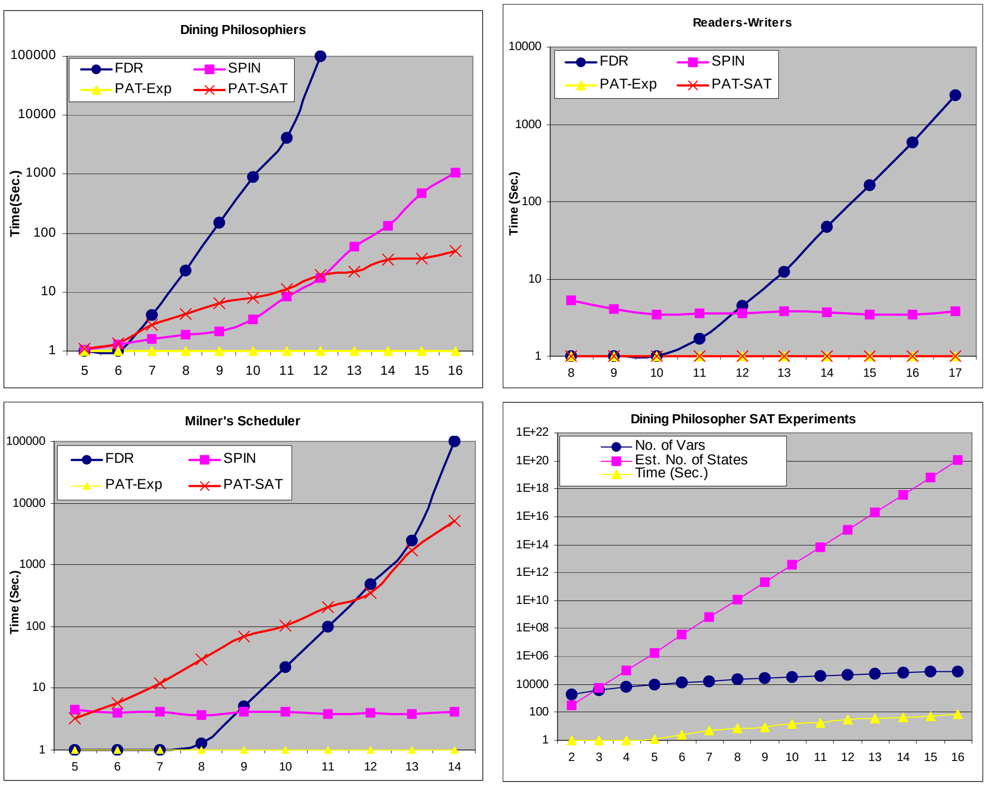


Figure 8.1: Performance evaluation with a 2.0 GHz Intel Core Duo CPU and 1 GB memory

For timely efficiency, we compare PAT with FDR and SPIN. We choose SPIN over others because it is the most established explicit model checker and its input language is loosely based on CSP. Note that partial order reduction, which partly makes SPIN very successful, has been implemented in PAT. We choose not to compare our bounded model checker with NuSMV [53] because it focuses a different application domain (i.e., circuit verification), in which often the transition relation is known statically. Our bounded model checker has been evaluated with two award wining SAT solvers, i.e., MiniSAT and RSAT [2].

Figure 8.1 summarizes the performance using three benchmark models, i.e., the dining philosopher problem as in Example 8.2.1 (against the property $\Box\Diamond eat_0 \wedge \Box\Diamond eat_1 \cdots \Box\Diamond eat_{N-1}$), the classic readers/writers problem and Milner's cyclic scheduler. This model describes a protocol for coordination of N readers and N writers accessing a shared resource. The property to verify is reachability of an erroneous situation (i.e., wrong readers/writers coordination). Milner's cyclic scheduler describes a scheduler for N concurrent processes. The processes are scheduled in cyclic fashion so that the first process is reactivated after the N-th process has been activated. The property to verify is that a process must eventually be scheduled. Details of the models and more experiments can be found at [1]. Since FDR supports refinement checking only, to prove a liveness property written in LTL, a property model needs to be constructed. For example, we use $Prop = eat_0 \rightarrow Prop \;\Box\; eat_1 \rightarrow Prop \ldots \;\Box\; eat_{N-1}$ to express the property mention above. The results are obtained by showing a (failure) refinement relationship between the system model and a process capturing the property to verify. For the dining philosopher example (the left-upper chart), our on-the-fly explicit model checker (referred as PAT-Exp) performs best to produce a counterexample. Our bounded model checker (referred as PAT-SAT) outperforms SPIN for 13 or more philosophers. The main reason is that the LTL to Büchi automata conversion in SPIN suffers from large LTL formulae, i.e., takes more time and produces bigger automata. All verifiers outperforms FDR (except PAT-SAT for small number of philosophers because of the encoding overhead), which is not feasible for more than 12 philosophers. For the readers/writers example, all verifiers except FDR produces a counterexample efficiently. Note that for every experiment SPIN takes less than a few seconds to build model-specific executables. For the Milner's example, the full state space (which is exponentially increasing without partial order reduction) must be explored because the property to verify is true. SAT outperforms FDR for 12 or more processes. This suggests that SAT-based model checking has the potential to handle large state space. Moreover, the current implementation of PAT-SAT may be improved by orders of magnitude should we incorporate recently development on incremental bounded model checking and more [191, 226]. Nonetheless, bounded model checking currently is mainly for falsification (if without a proper threshold bound). The time taken by SPIN and PAT-Exp remains constant. This should be credited to the partial order reduction. The right-bottom chart summarizes the performance our SAT-based verifier in terms of the size of the generated formula, the time needed for encoding and solving against the number of states of the model. The estimated number of states increase exponentially whereas the number of Boolean variables and the time needed increase much slower.

Note that the experimental results presented in this section on FDR can be improved significantly with a slightly different modeling, which is subject to FDR's hierarchical compression [176].

8.5 Summary

In summary, we have developed a way to encode compositional system models without explicitly exploring all reachable states. Experiment results show that our approach does verification rather efficiently. Though presented in the framework of CSP#, our encoding of compositional processes may be applied to other formal specification languages and notations.

In literature, there have been a number of bounded model checkers dedicated to different specification languages. Some noticeable ones include the first bounded model checker [34], NuSMV [53] and UCLID [43]. However, as far as the authors know, there has not yet been a bounded model checker dedicated to process algebras, which have a compositional nature.

Chapter 9

Verification of Real-time Systems

Ensuring the correctness of computer system used in life-critical systems is crucial and challenge. This is especially true when the correctness of a real-world system depend on quantitative timing, e.g., the pacemaker system. Specification and verification of real-time systems are important research topics which have practical implications.

Recall that the real-time system modeling language proposed in Chapter 3 supports a rich set of concurrent operators as well as hierarchical timed constructs. It is nontrivial to offer efficient mechanical verification support for this modeling language because of the infinite domains of the rational clock values. In this chapter, we develop a fully automated abstraction technique to build an abstract finite state machine from the model. The idea is to dynamically create clocks to capture constraints introduced by the timed process constructs. A clock may be shared for many constructs in order to reduce the number of clocks. During system exploration, a constraint on the active clocks is maintained and solved using Difference Bound Matrix (DBM [68]). We show that the abstraction is finite state and is subject to model checking. Further, it weakly bi-simulates the concrete model and, therefore, we may perform sound and complete LTL-X (i.e. LTL without the next operator) model checking, refinement checking or even timed refinement checking upon the abstraction.

The remainder of the chapter is organized as follows. Section 9.1 presents the zone abstraction using dynamical clocks. Section 9.2 discusses the soundness of the abstraction and its implication on model checking. Section 9.3 presents the verification algorithms for LTL checking, refinement checking and timed refinement checking. Section 9.4 concludes this chapter.

9.1 Zone Abstraction

Model checking is applicable to finite state systems. Nonetheless, the number of concrete configurations (and hence the concrete transition system) is infinite because of the time transitions. In the following, we apply zone abstraction to build an abstract configuration system. Different from zone abstraction applied to Timed Automata [68, 224], we dynamically create/delete a set of clocks to precisely encode the timing requirements. We show that the abstract transition system is finite state and subject to model checking.

9.1.1 Clock Activation and De-activation

A clock is a variable ranging from 0 to some bounded natural number. Given a configuration (V, P), a clock is necessary to measure time elapsing if, and only if, a timed process (e.g. $Wait[d]$, $P\ timeout[d]\ Q$, $P\ interrupt[d]\ Q$, or $P\ deadline[d]$) has been enabled. If a timed process (say $Wait[d]$) is enabled, we associate a clock (say tm) with the process to record time elapsing (written as $Wait[d]_{tm}$). The timing requirements can be captured using a constraint on the valuation of the clock. During system execution, multiple clocks may be used to capture quantitative timing constraints. A clock may become irrelevant as soon as the related process takes a transition. For instance, if P in $P\ timeout[d]_{tm}\ Q$ engages in an observable event, then the process transforms to P' and clock tm becomes irrelevant. It is known that model checking of real-time systems is exponential in the number of clocks. Therefore, it is desirable to use clocks only necessary and discharge them as early as possible.

Definition 31 (Abstract system configuration) *An abstract system configuration is a triple* (V, P, D)*, where* V *is a variable valuation,* P *is a process and* D *is a zone.*

A *zone* is the maximal set of clock valuations satisfying a set of primitive clock constraints. A primitive constraint on a clock is of the form $tm \sim d$ where tm is a timer, d is a constant and $\sim$ is $\geq$, $=$ or $\leq$. Since clocks are implicit, clock readings cannot be compared directly. A zone is not empty if and only if the constraint is true. We write $D[t]$ to be the constraints on clock t in zone D.

Next, we show how to systematically activate and de-activate clocks using process $Wait[d]$ and $P\ timeout[d]\ Q$ as examples. Let t be a fresh clock. Given an abstract configuration, we define function $\mathcal{A}(P, t)$ to recursively determine whether a clock is necessary and associate the clock with the relevant process constructs. A clock is necessary if and only if one (or more) timed pattern has just been enabled. For instance,

$$\begin{aligned} \mathcal{A}(Wait[d]_{t'}, t) &= Wait[d]_{t'} \\ \mathcal{A}(Wait[d], t) &= Wait[d]_t \end{aligned}$$

where $Wait[d]_{t'}$ denotes that the timed process is associated with a clock t', whereas $Wait[d]$ denotes that it has not been associated with a clock. The intuition is for the former case, $\mathcal{A}$ does nothing and t is not used (since it is not necessary to introduce another clock); for the latter case, $\mathcal{A}$ associates t the the timer process. The following shows how to apply $\mathcal{A}$ to process $P\ timeout[d]\ Q$.

$$\begin{aligned} \mathcal{A}(P\ timeout[d]_{t'}, t) &= P\ timeout[d]_{t'}\ Q \\ \mathcal{A}(P\ timeout[d]\ Q, t) &= \mathcal{A}(P)\ timeout[d]_t\ \mathcal{A}(Q) \end{aligned}$$

If a clock t' has already been associated with $P\ timeout[d]\ Q$, then function $\mathcal{A}$ simply returns the process. Otherwise, it is associated with t and further $\mathcal{A}$ is applied to the sub-processes P and Q recursively. The complete definition of function $\mathcal{A}$ is presented in Figure 9.1. In an abuse of notation, given an abstract configuration $c = (V, P, D)$, we write $\mathcal{A}(c)$ to be $(V, \mathcal{A}(P), D \wedge t = 0)$ if t is used; otherwise $\mathcal{A}(c)$ is simply c.

A runtime clock may later be discarded when the time-related process has evolved such that the reading of the clock is no longer relevant. For instance, the clock associated with $P\ timeout[d]\ Q$ can be discarded when P engages in an observable event. It should be clear that we can identify the set of active runtime clocks by a similar procedure. To minimize clocks, all in-active runtime clocks, and the associated timing constraints, shall be pruned from D. We assume a function $\mathcal{D}$ which performs clock de-activation in a sound and complete way.

$$\begin{array}{lll}
\mathcal{A}(P \,\Box\, (or \parallel or \,|||)\; Q, t) & = \mathcal{A}(P, t) \,\Box\, (or \parallel or \,|||)\; \mathcal{A}(Q, t) & \\
\mathcal{A}(P;\; Q, t) & = \mathcal{A}(P, t);\; Q & \\
\mathcal{A}(P \setminus X, t) & = \mathcal{A}(P, t) \setminus X & \\
\mathcal{A}(P, t) & = \mathcal{A}(Q, t) & \text{– if } P \text{ is defined as } Q \\
\mathcal{A}(\mathit{Wait}[d], t) & = \mathcal{A}(\mathit{Wait}[d]_t) & \\
\mathcal{A}(P \; \mathit{timeout}[d] \; Q, t) & = \mathcal{A}(P, t) \; \mathit{timeout}[d]_t \; \mathcal{A}(Q, t) & \\
\mathcal{A}(P \; \mathit{interrupt}[d] \; Q, t) & = \mathcal{A}(P, t) \; \mathit{interrupt}[d]_t \; \mathcal{A}(Q, t) & \\
\mathcal{A}(P \; \mathit{deadline}[d], t) & = \mathcal{A}(P, t) \; \mathit{deadline}[d]_t &
\end{array}$$

Figure 9.1: Clock activation: $\mathcal{A}(P, t)$ is P except the above cases

9.1.2 Zone Abstraction

We define $D^{\uparrow} = \{t + d \mid t \in D \wedge d \in \mathbb{R}_+\}$, i.e. the zone obtained by delaying arbitrary amount of time. Notice that all clocks take the same pace. Next, we define function ι to compute the zone which can be reached by idling from a given abstract system configuration [224], presented in Figure 9.2. Given the current zone D, process $P \; \mathit{timeout}[d]_{tm} \; Q$ may keep idling as long as P may keep idling and the reading of clock tm is less or equal to d (so that $\mathit{timeout}$ does not occur). The rest are similarly defined.

In the following, we define the firing rules based on the abstract system configurations. The idea is to eliminate time transitions altogether and use the timing constraint to ensure that the time-related process constructs behave correctly. An abstract transition is of the form $(V, P, D) \overset{x}{\hookrightarrow} (V', P', D')$, where $x \in \Sigma \cup \{\checkmark, \tau\}$.

$$\frac{}{(V, \mathit{Wait}[d]_{tm}, D) \overset{\tau}{\hookrightarrow} (V, \mathit{Skip}, D^{\uparrow} \wedge tm = d)} \; [\; ade \;]$$

Process $\mathit{Wait}(d)$ idles for exactly d time units and then engages in event τ and the process transforms to Skip. Intuitively, it should be clear that this is ‘equivalent’ to the concrete firing rules (see Section 3.2.2). We will define what equivalence means later in this section.

$$\frac{(V, P, D) \overset{\tau}{\hookrightarrow} (V', P', D')}{(V, P \; \mathit{timeout}[d]_{tm} \; Q, D) \overset{\tau}{\hookrightarrow} (V', P' \; \mathit{timeout}[d]_{tm} \; Q, D' \wedge tm \leq d)} \; [\; ato1 \;]$$

$$\frac{(V, P, D) \overset{x}{\hookrightarrow} (V', P', D')}{(V, P \; \mathit{timeout}[d]_{tm} \; Q, D) \overset{x}{\hookrightarrow} (V', P', D' \wedge tm \leq d)} \; [\; ato2 \;]$$

$$\begin{array}{lll}
\iota(V, Stop, D) & = D^{\uparrow} & \\
\iota(V, Skip, D) & = D^{\uparrow} & \\
\iota(V, e \rightarrow P, D) & = D^{\uparrow} & \\
\iota(V, [b]P, D) & = D^{\uparrow} & \\
\iota(V, P \ \Box \ (or \ \| \ or \ |||) \ Q, D) & = \iota(V, P, D) \wedge \iota(V, Q, D) & \\
\iota(V, P; \ Q, D) & = \iota(V, P, D) & \\
\iota(V, P \setminus X, D) & = \iota(V, P, D) & \\
\iota(V, Wait[d]_{tm}, D) & = D^{\uparrow} \wedge tm \leq d & \\
\iota(V, P \ timeout[d]_{tm} \ Q, D) & = \iota(V, P, D) \wedge tm \leq d & \\
\iota(V, P \ interrupt[d]_{tm} \ Q, D) & = \iota(V, P, D) \wedge tm \leq d & \\
\iota(V, P \ deadline[d]_{tm}, D) & = \iota(V, P, D) \wedge tm \leq d & \\
\iota(V, P, D) & = \iota(V, Q, D) & \text{– if } P \widehat{=} Q
\end{array}$$

Figure 9.2: Idling calculation

$$\frac{}{(V, P \ timeout[d]_{tm} \ Q, D) \overset{\tau}{\hookrightarrow} (V, Q, tm = d \wedge \iota(V, P, D))} \ [\ ato3\ ]$$

Depending on when the first event of P takes place and whether it is observable, $P \ timeout[d] \ Q$ behaves differently in three ways. An observable transition of P must occur no later than d time units since the process is enabled (rule $ato1$ and $ato2$). If the first transition is observable, then the *choice* is resolved (rule $ato2$). If it is silent, then the it transforms to $P' \ timeout[d] \ Q$. If P may delay more than d time units (captured by the constraint $\iota(V, P, D)$), then it times out after exactly d time units (rule $ato3$). The constraint $tm = d \wedge \iota(V, P, D)$ means that the delay is exactly d time units and P must be idling during the period.

$$\frac{(V, P, D) \overset{x}{\hookrightarrow} (V', P', D')}{(V, P \ interrupt[d]_{tm} \ Q, D) \overset{x}{\hookrightarrow} (V', P' \ interrupt[d]_{tm} \ Q, D' \wedge tm \leq d)} \ [\ ait1\ ]$$

$$\frac{}{(V, P \ interrupt[d]_{tm} \ Q, D) \overset{\tau}{\hookrightarrow} (V, Q, tm = d \wedge \iota(V, P, D))} \ [\ ait2\ ]$$

Process $P \ interrupt[d] \ Q$ behaves differently in two ways. Transitions of P must take place no later than d time units since the process is enabled (rule $ait1$). If P delay more than d time units (captured by the constraint $\iota(V, P, D)$), then it is interrupted after exactly d time units (rule $ait2$).

$$\frac{(V, P, D) \overset{x}{\hookrightarrow} (V', P', D'), x \neq \checkmark}{(V, P \ deadline[d]_{tm}, D) \overset{x}{\hookrightarrow} (V', P' \ deadline[d]_{tm}, D' \wedge tm \leq d)} \ [\ adl\ ]$$

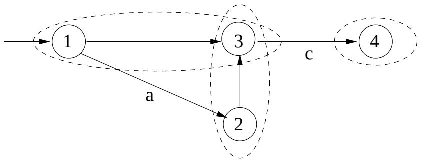


Figure 9.3: An example of abstract timed transition system

$P\ deadline[d]$ behaves exactly as P except that any transition must occur before d time units.

The rest of the firing rules is present in Appendix C. A transition is valid if, and only if, it conforms to the firing rules and the resultant zone is not empty. Intuitively, this means that a transition must be allowed by the untimed system and at the same time satisfy the additional timing requirement.

Definition 32 (Abstract transition system) *Let $\mathcal{S} = (Var, init, P)$ be a system model. The abstract transition system corresponding to $\mathcal{S}$ is a LTS $\mathcal{L}_a^{\mathcal{S}} = (C_a, init_a, \hookrightarrow)$ where C_a is the set of reachable valid abstract system configurations, $init_a$ is the initial configuration $(init, P, true)$ and $\hookrightarrow$ is the smallest transition relation satisfying $\forall\, c, c' : C_a,\ c \overset{e}{\hookrightarrow} c' \Leftrightarrow \mathcal{A}(c) \overset{e}{\hookrightarrow} \mathcal{D}(c')$.*

Example 9.1.1 (A simple example) Assume a model $(\varnothing, \varnothing, P)$ with no variable and P is $(a \rightarrow Wait[5];\ b \rightarrow Stop)\ interrupt[3]\ c \rightarrow Stop$. The abstract transition system is shown in Figure 9.3, where transition label τ is skipped for simplicity. Let $\langle t_1, t_2 \rangle$ be a sequence of clocks. The following illustrates how to construct the abstract transition system. Let s_0 be $(\varnothing, P, true)$.

- Step 1: apply $\mathcal{A}$ to s_0 to get

  $$s_1 = (\varnothing, (a \rightarrow Wait[5];\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow Stop, t_1 = 0)$$

- Step 2: apply rule $ait1$ to s_1 to get

  $$s_2 = (\varnothing, (Wait[5];\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow Stop, 0 \leq t_1 \leq 3)$$

  Notice that $(t_1 = 0)^{\uparrow}$ equals to $t_1 \geq 0$.

- Step 3: apply $\mathcal{D}$ to s_2. The result is exactly s_2. We obtain the transition from state 1 to state 2.

- Step 4: apply rule $ait2$ to s_1 to get

  $$s_3 = (\varnothing, (c \rightarrow Stop), t_1 \geq 0 \wedge t_1 = 3)$$

  Notice that $\iota(\varnothing, a \rightarrow Wait[5];\ b \rightarrow Stop, t_1 = 0)$ is $t_1 \geq 0$.

- Step 5: apply $\mathcal{D}$ to s_3 to get $s_4 = (\varnothing, (c \rightarrow Stop), true)$. We remark that because t_1 becomes inactive, it is pruned from the constraint. This generates the transition from state 1 to state 3.
- Step 6: apply $\mathcal{A}$ to s_2 to get

  $$s_5 = (\varnothing, (Wait[5]_{t_2};\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow Stop, 0 \leq t_1 \leq 3 \wedge t_2 = 0)$$

- Step 7: apply rule $ait1$ to s_5, we get

  $$s_6 = (\varnothing, (Skip;\ b \rightarrow Stop)\ interrupt[3]_{t_1}\ c \rightarrow Stop, 0 \leq t_1 \leq 3 \wedge t_2 = 5)$$

  Notice that the timing constraint is false given that all timers take the same pace. Refer to next section on how this is discovered systematically.
- Step 8: apply rule $ait2$ to s_5 to get

  $$s_7 = (\varnothing, c \rightarrow Stop, t_1 \geq 0 \wedge t_2 \geq 0 \wedge t_2 \leq 5 \wedge t_1 = 3)$$

- Step 9: apply $\mathcal{D}$ to s_7 to get s_4. Notice that both clocks are inactive and therefore pruned. This generates the transition from state 2 to state 3.
- Lastly, we generate the transition from state 3 to state 4. Notice that this transition involves no quantitative timing.

end

9.1.3 Zone Operations

In order to construct and verify the abstract transition system, we need efficient and sound procedures to manipulate zones. For instance, we need to determine whether a zone is empty or not. The procedure must be sound (so that invalid configurations are ruled out) and complete (so that a valid configuration is not missed).

A zone D can be equally represented as a difference bound matrices (DBM). Let $\{t_1, t_2, \cdots, t_n\}$ be a set of n clocks. Let t_0 be a dummy clock whose value is always 0. A DBM representing a constraint on the clocks contains $n+1$ rows, each of which contains $n+1$ elements. Let D^i_j represent entry (i, j) in the matrix. A DBM represents the constraint: $\forall\, i : 0 \mathinner{..} n,\ \forall\, j : 0 \mathinner{..} n,\ t_i - t_j \leq D^i_j$. The most important property of DBM is that there is a relatively efficient procedure to compute a unique canonical form. Given a DBM in canonical form, checking whether the zone is empty or not is as easy as looking up an entry in the matrix. DBM has been well studied [68, 29, 30]. In the following, we introduce the relevant DBM operations/properties. We skip the discussion on rest of the zone operations (e.g. $D^{\uparrow}$, adding a constraint, etc.) as they resemble the discussion in [30].

Calculate canonical form In theory, there are infinite different timing constraints which represent the zone. For instance, $0 \leq t_1 \leq 3 \wedge 0 \leq t_1 - t_2 \leq 3$ is equivalent to $0 \leq t_1 \leq 3 \wedge 0 \leq t_1 - t_2 \leq 3 \wedge t_2 \leq 1000$. In order to systematically compare two zones, we compute their unique canonical forms. In other words, we compute the tightest bound on each clock difference. If the clocks are viewed as vertices in a weighted graph and the clock difference as the label on the edge connecting two clocks, the tightest clock difference is the shortest path between the respective vertices. The Floyd-Warshall algorithm [90] thus can be used to compute the canonical from. Given that this algorithm is cubic in the number of clocks, it is desirable to reduce the number of clocks. Besides, the algorithm must be invoked if necessary and ideally (if possible) the result of performing an operation on a canonical DBM should be canonical.

Check satisfiability In order to construct the abstract transition system, it is essential to check whether a zone is empty. Given the DBM representing a zone, it is unsatisfiable if, and only if, there is a clock which has a negative difference from itself, i.e. $t_k - t_k < 0$ for some k so that the constraint is false. If the DBM is in canonical form, then there exists at least one D^i_i which is negative. Further, it can be shown that the DBM is false if, and only if, D^0_0 is negative. Therefore, we compute the canonical form whenever it is necessary to check for satisfiability.

Add clocks In our setting, clocks may be introduced during system exploration. Assume the new clock is t_k and the given DBM is canonical. The following shows how the DBM is updated with entries for t_k. For all i, $D^i_k = D^i_0$ and $D^k_i = D^0_i$ as the new clock always starts with value 0. By a simple argument, it can be shown the resultant DBM is canonical.

	t_0	t_1	$\cdots$	t_i	$\cdots$	t_{k-1}	$\mathbf{t_k}$
t_0	0	d_1^0	$\cdots$	d_i^0	$\cdots$	d_{k-1}^0	$\mathbf{0}$
t_1	d_0^1	$*$	$\cdots$	$*$	$\cdots$	$*$	$\mathbf{d_0^1}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
t_i	d_0^i	$*$	$\cdots$	$*$	$\cdots$	$*$	$\mathbf{d_0^i}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
t_{k-1}	d_0^{k-1}	$*$	$\cdots$	$*$	$\cdots$	$*$	$\mathbf{d_0^{k-1}}$
$\mathbf{t_k}$	$\mathbf{0}$	$\mathbf{d_1^0}$	$\cdots$	$\mathbf{d_i^0}$	$\cdots$	$\mathbf{d_{k-1}^0}$	$\mathbf{0}$

Prune clocks Because entries in a canonical DBM represent the tightest bound on clock differences, pruning clocks is to remove the relevant row and column in the table. It should be clear that the remaining DBM is canonical, i.e. the bounds can not be possibly tightened with less constraints.

Notice that the number of reachable timing constraints in canonical form are finite as proved in [68]. As a result, the abstraction system is finite state and therefore subject to model checking[1].

Example 9.1.2 (DBM manipulation example) The following illustrates how the DBM is transformed through system exploration in Example 9.1.1.

$$\begin{tikzcd}[ampersand replacement=\&]
\begin{array}{c|c|} & t_0 \\ \hline t_0 & 0 \\ \hline \end{array} \arrow[r, "Step1"] \& \begin{array}{c|c|c|} & t_0 & t_1 \\ \hline t_0 & 0 & 0 \\ \hline t_1 & 0 & 0 \\ \hline \end{array} \arrow[r, "Step2"] \arrow[d, "Step4"] \& \begin{array}{c|c|c|} & t_0 & t_1 \\ \hline t_0 & 0 & 0 \\ \hline t_1 & 3 & 0 \\ \hline \end{array} \arrow[r, "Step6"] \& \begin{array}{c|c|c|c|} & t_0 & t_1 & t_2 \\ \hline t_0 & 0 & 0 & 0 \\ \hline t_1 & 3 & 0 & 3 \\ \hline t_2 & 0 & 0 & 0 \\ \hline \end{array} \arrow[r, "Step7"] \arrow[d, "Step8"] \& \begin{array}{c|c|c|c|} & t_0 & t_1 & t_2 \\ \hline t_0 & 0 & 0 & \text{-}5 \\ \hline t_1 & 3 & 0 & 3 \\ \hline t_2 & 5 & 0 & 0 \\ \hline \end{array} \arrow[r, phantom, "\equiv"] \& \begin{array}{c|c|c|c|} & t_0 & t_1 & t_2 \\ \hline t_0 & \text{-}2 & \text{-}5 & \text{-}7 \\ \hline t_1 & 1 & \text{-}2 & \text{-}4 \\ \hline t_2 & 1 & \text{-}2 & \text{-}4 \\ \hline \end{array} \\
\& \begin{array}{c|c|c|} & t_0 & t_1 \\ \hline t_0 & 0 & \text{-}3 \\ \hline t_1 & 3 & 0 \\ \hline \end{array} \arrow[r, "Step5"] \& \begin{array}{c|c|} & 0 \\ \hline 0 & 0 \\ \hline \end{array} \& \begin{array}{c|c|c|c|} & t_0 & t_1 & t_2 \\ \hline t_0 & 0 & \text{-}3 & 0 \\ \hline t_1 & 3 & 0 & 3 \\ \hline t_2 & \infty & 0 & 0 \\ \hline \end{array} \arrow[r, "Step9"] \& \begin{array}{c|c|} & t_0 \\ \hline t_0 & 0 \\ \hline \end{array} \&
\end{tikzcd}$$

The DBM obtained after Step 7 is indeed false, i.e. D_0^0 is -2 after applying the Floyd-Warshall algorithm. **end**

[1] assume that the variable domains are finite and the reachable process expressions are finite.

9.2 Verification of Real-time Systems

In this section, we prove that our abstraction is sound and complete with respect to a number of properties. The abstract transition system is shown to be equivalent to the concrete transition system using a specialized bi-simulation relationship [136]. We then show that three different system verification methods are sound.

In the concrete transition system, if a configuration (V', P') can be reached from (V, P) by idling only, we write $(V, P) \rightsquigarrow (V', P')$. By a simple argument, it can be shown that if $(V, P) \rightsquigarrow (V', P')$, then $V = V'$. We write $(V, P) \overset{x}{\rightsquigarrow} (V', P')$ if, and only if, there exists (V, P_1), (V', P_2) such that $(V, P) \rightsquigarrow (V, P_1)$ and $(V, P_1) \xrightarrow{x} (V', P_2)$ and $(V', P_2) \rightsquigarrow (V', P')$.

Definition 33 (Time abstract bi-simulation) *Let $\mathcal{S} = (Var, init, P)$ be a model. Let $\mathcal{L}_c^{\mathcal{S}} = (C_c, init_c, \rightarrow)$ and $\mathcal{L}_a^{\mathcal{S}} = (C_a, init_a, \hookrightarrow)$ be the concrete and abstract transition systems. $\mathcal{L}_c$ and $\mathcal{L}_a$ are time abstract bi-similar (hereafter bi-similar) if, and only if, there exists a binary relation $\mathcal{R} : C_c \rightarrow C_a$ such that $(init_c, init_a) \in \mathcal{R}$ and $\forall\, x : \Sigma \cup \{\checkmark, \tau\}$; $c = (V_c, P_c)$; $a = (V_a, P_a, D_a)$ such that $(c, a) \in \mathcal{R}$ implies,*

- *$V_c = V_a$,*
- *if $c \overset{x}{\rightsquigarrow} c'$, then for some a', $a \overset{x}{\hookrightarrow} a'$ and $(c', a') \in \mathcal{R}$.*
- *if $a \overset{x}{\hookrightarrow} a'$, then for some c', $c \overset{x}{\rightsquigarrow} c'$ and $(c', a') \in \mathcal{R}$.*

We say that c and a are bi-similar, written as $c \sim a$, if, and only if, there exists $\mathcal{R}$ such that the transition systems are bi-similar. Notice that $\mathcal{L}_c$ and $\mathcal{L}_a$ are bi-similar if, and only if, $init_c \sim init_a$.

Theorem 9.2.1 *Let $\mathcal{S} = (Var, init, P)$ be a system model. $\mathcal{L}_c^{\mathcal{S}}$ and $\mathcal{L}_a^{\mathcal{S}}$ are time abstract bi-similar.*

Proof: Let $\mathcal{S} = (Var, i, P)$ be the model; $\mathcal{L}_c$ and $\mathcal{L}_a$ be the concrete and abstract transition system respectively. By definition, it suffices to construct a binary relation which satisfies the condition. The theorem is proved by structural induction on the all types of process expressions. The following are the base cases.

- $Stop$: $\mathcal{R} = \{(i, Stop) \mapsto (i, Stop, true)\}$. Trivially true.
- $Skip$: $\mathcal{R} = \{(i, Skip) \mapsto (i, Skip, true), (i, Stop) \mapsto (init, Stop, true)\}$. Trivially true.
- $Wait[d]$: $\mathcal{R} = \{(i, Wait[d]) \mapsto (i, Wait[d], true), (i, Skip) \mapsto (i, Skip, true), (i, Stop) \mapsto (i, Stop, true)\}$. The transition $(i, Wait[d]) \overset{\tau}{\rightsquigarrow} (i, Skip)$ of $\mathcal{L}_c$ corresponds to the transition $(i, Waid[d], true) \overset{\tau}{\hookrightarrow} (i, Skip, true)$. Notice that the clock introduced by function $\mathcal{A}$ would be pruned by $\mathcal{D}$. The rest is trivial.

Next, we prove the induction step.

- $e\{prg\} \rightarrow P$: $(i, e\{prg\} \rightarrow P)$ and $(i, e\{prg\} \rightarrow P, true)$ are bi-similar since $(i, e\{prg\} \rightarrow P) \overset{e}{\rightsquigarrow} (prg(i), P)$ (by rule $as1$ and $as2$) and $(i, e\{prg\} \rightarrow P, true) \overset{e}{\hookrightarrow} (prg(i), P, true)$ (by rule aev), and $(prg(i), P) \sim (prg(i), P, true)$ (by hypothesis).
- $[b]P$: if $i \models b$, then $[b]P$ behaves exactly as P (rule $gu2$ and rule agu), hence by hypothesis, $(i, [b]P) \sim (i, [b]P, true)$. If $i \not\models b$, then $[b]P$ behaves exactly as $Stop$ (rule $gu1$ and no abstract firing rule), hence $(i, [b]P) \sim (i, [b]P, true)$.
- $P \ \Box \ Q$: $P \ \Box \ Q$ behaves either as P or Q, in both cases, by hypothesis $(i, P \ \Box \ Q) \sim (i, P \ \Box \ Q, true)$.
- $P \sqcap Q$. Similarly as above.
- $P \parallel Q$: there is one-to-one correspondence on the concrete firing rules (rule $pa1$, $pa2$ and $pa3$) and the abstract firing rules ((rule $apa1$, $apa2$ and $apa3$)). It is clear that by hypothesis $(i, P \parallel Q) \sim (i, P \parallel Q, true)$.
- $P \ ||| \ Q$. Similarly as above.
- $P; \ Q$. Similarly as above.
- $P \ timeout[d] \ Q$: let the associated clock be tm. We show that each abstract transition is possible if and only if there is a corresponding concrete transition $(i, P) \rightsquigarrow (i', P')$. Rule $ato1$ is applicable if and only if $tm \leq d$ and (i, P, D) may perform a τ-transition. By

hypothesis, (i, P, D) may perform a τ-transition if and only if (i, P) does. By rule $to2$, $to3$ and $to4$, a τ of P may happen if and only if $tm \leq d$. Therefore, we conclude rule $ato1$ is applicable if and only if there is a corresponding concrete transition. Similarly, we argue that rule $ato2$ and $ato3$ are applicable if and only if there is a corresponding concrete transition. This concludes that $(i, P\ timeout[d]\ Q) \sim (i, P\ timeout\ Q, true)$.

- $P\ interrupt[d]\ Q$: Similarly as above.
- $P\ waituntil[d]$: rule $awu1$ is applicable if and only if x is not ✓ or (V, P, D) is capable of performing x. By hypothesis, (V, P) must be able to perform x. Rule $awu2$ is applicable if and only if $tm \geq d$, this is implied by rule $wu3$ and hypothesis (and vice versa). Lastly, rule $awu3$ is implied by rule $wu2$ and $wu4$.
- Similarly as above.

□

By definition, it suffices to construct a binary relation which satisfies the condition. Time abstract bisimulation is strong enough to guarantee soundness on verification of a number of useful properties.

9.2.1 LTL-X Model Checking

In this section, we study the verification of LTL formulae (see Section 2.3.2) without the next operator (i.e. LTL-X), constituted by propositions on global variables. Notice that no clocks are allowed in the property. The philosophy is that a critical property may often be independent of the speed of the hardware on which the system is deployed, whereas the model of the implementation shall incorporate known hardware limitations.

Example 9.2.2 Given the Fischer's algorithm in Section 3.2.3, the following are some critical properties.

$\Box ct \leq 1$ – safety property
$\Box(x = i \Rightarrow \Diamond cs.i)$ – liveness property

where $\Box$ and $\Diamond$ read as 'always' and 'eventually'. The first property precisely states mutual exclusion, i.e., at all time, there must not be 2 or more processes in the critical section. The second states that if process i is attempting to access the shared resource, it must eventually do so. **end**

In order to reflect model checking results on the abstract transition system to the original system, we need to establish that the abstract transition system is equivalent to the concrete one with respect to LTL-X formulae. The idea is to show stutter equivalence between traces of the abstract system and the concrete system. Given two traces $tr_1 = \langle V_0, V_1, \cdots \rangle$ and $tr_2 = \langle V_0', V_1', \cdots \rangle$, tr_1 and tr_2 are stutter equivalent if, and only if, tr_1 and tr_2 can be partitioned into blocks, so that the variable valuation in the k-th block in tr_1 is the same as those in the k-th block of tr_2. Formally, tr_1 is stutter equivalent to tr_2 if, and only if, there are two infinite sequences of integers $0 < i_0 < i_1 < \cdots$ and $0 < j_0 < j_1 < \cdots$ such that for every block $k \geq 0$ holds $V_{s_{i_k}} = V_{s_{i_k+1}} = \cdots = V_{s_{i_{k+1}-1}} = V'_{s_{j_k}} = V'_{s_{j_k+1}} = \cdots = V'_{s_{j_{k+1}-1}}$. It is known that tr_1 satisfies a LTL-X property if, and only if, tr_2 does.

Let ϕ be such a property, we write $\mathcal{L} \vdash \phi$ to denote that the labeled transition system $\mathcal{L}$ satisfies ϕ, i.e. every trace of $\mathcal{L}$ satisfies ϕ.

Lemma 9.2.3 *Let $\mathcal{S} = (Var, init_G, P)$ be a system model. For every trace of the concrete transition system $\mathcal{L}_c$, there is a stutter equivalent trace of the abstract transition system $\mathcal{L}_a$ and vice versa.*

The above lemma can be proved by structural induction or implied from Theorem 9.2.1. Consequently, the following theorem can be proved straightforwardly.

Theorem 9.2.4 *Let $\mathcal{S} = (Var, init_G, P)$ be a system model. Let ϕ be a LTL-X formula constituted by propositions on Var. $\mathcal{L}_c^{\mathcal{S}} \vDash \phi$ if, and only if, $\mathcal{L}_a^{\mathcal{S}} \vDash \phi$.*

9.2.2 Refinement Checking

In this section, we investigate an alternative verification schema for finite system executions. That is, to verify whether the system satisfies the property by showing a refinement relationship between the system and a model which models the property. Chapter 6 presents a variety of refinement relationships, e.g. trace-refinement, stable failures refinement and failures/divergence refinement [108]. In order to check refinement between two (timed) models, time abstraction must be applied to both models.

Example 9.2.5 Given the Fischer's algorithm in Section 3.2.3, a natural question is whether ϵ and δ are necessary or their values would make a difference. Equivalently, the former is to ask whether $(init, uProcotol)$ where $init = \{x \mapsto -1, ct \mapsto 0\}$ and $uProcotol$ defined as follows, trace-refines the original one $(init, Procotol)$.

```
$uProc(i)$    $= [x == -1]uActive(i);$
$uActive(i)$  $= update.i\{x = i\} \rightarrow$
                  **if** $(x == i)$ $\{$
                      $cs.i\{ct = ct + 1\} \rightarrow exit.i\{ct = ct - 1;\ x = -1\} \rightarrow uProc(i)$
                  $\}$ **else** $\{$
                      $uProc(i)$
                  $\}$;
$uProtocol$   $= uProc(0) \parallel uProc(1) \parallel uProc(2);$
```

By showing trace refinement in both directions, we may establish trace equivalence. Or, the users may change the value of ϵ and δ check for equivalence. **end**

We have defined refinement and equivalence relations between two concrete models in Definition 22 in Section 6.1. In the following, we argue that it is sound and complete to show stable failures refinement (i.e. assuming both models are divergence-free) between the abstraction transition systems in order to show failures refinement between the concrete models.

Theorem 9.2.6 *Let $\mathcal{S}_i$ where $i \in \{1, 2\}$ be two models. $\mathcal{S}_1$ refines $\mathcal{S}_2$ in stable failures semantics iff $traces(\mathcal{L}_a^{\mathcal{S}_1}) \subseteq traces(\mathcal{L}_a^{\mathcal{S}_2})$ and $failures(\mathcal{L}_a^{\mathcal{S}_1}) \subseteq failures(\mathcal{L}_a^{\mathcal{S}_2})$.*

By Theorem 9.2.1, it should be clear that our abstraction preserves failures. Intuitively, this is because not only observable transitions but also τ-transitions are preserved by the abstraction. The theorem can then be proved straightforwardly. We remark that it is clear the failures refinement subsumes trace-refinement and, therefore, it too can be supported by only checking the abstract transition systems.

9.2.3 Timed Refinement Checking

We have looked at the refinement checking without the consideration of timed transitions. To include timed transitions, we need to include the time stamps in the traces.

We assume a global clock t_G which starts (with reading 0) whenever the system starts. We equip a system configuration (V, P) withe a clock value t, written as $(V, P)_t$ where t is the current reading of t_G. The changes of t in the firing rules presented in Section 3.2.2 can be calculated easily by adding the elapsed time in the transition. Assume that $\epsilon \in \mathbb{R}_+$ is a real number denoting the event of time elapsing. Given a transition $(V, P)_t \rightarrow (V', P')_{t'}$, we have $t' = t + \epsilon$ in rule $de1$, $to3$, $it2$ and $de2$, and $t' = t$ in rest rules.

A timed event is an event associated with a time stamp, written as $x@t$ where t is the reading of t_G when x occurs, i.e. a timestamp. A run of a model $\mathcal{S} = (Var, init_G, P)$ is a *finite* sequence of alternating configurations and timed events, i.e.

$$\langle (V_0, P_0)_{t_0}, x_1@t_1, (V_1, P_1)_{t_1}, x_2@t_2, \cdots, x_n@t_n, (V_n, P_n)_{t_n} \rangle$$

such that $V_0 = init_G$, $P_0 = P$, $t_0 = 0$ and $(V_i, P_i)_{t_i} \overset{x_{i+1}}{\rightarrow} (V_{i+1}, P_{i+1})_{t_{i+1}}$ for all i. An execution of $\mathcal{S}$ is a finite sequence of timed events $\langle x_1@t_1, x_2@t_2, \cdots, x_n@t_n \rangle$ such that there exists a corresponding run $\langle (V_0, P_0)_{t_0}, x_1@t_1, \cdots, x_n@t_n, (V_n, P_n)_{t_n} \rangle$. Given an execution E, let $E \upharpoonright X$ where X is a set of event names be the sequence generated by removing events with a name in X from the sequence. *E is divergent if and only if E can be extended with infinite τ-events and possibly ϵ-events. E is timed divergent if and only if E can be extended with infinite ϵ-events and τ-events.* A model is (timed) divergence-free if and only it contains no (timed) divergent executions.

Example 9.2.7 The following illustrates a model which is not timed divergence-free. Assume there are no variables and the process is defined as follows: $P = Wait[5];\ P$. The empty execution is timed divergent since it can be extended with the sequence $\langle 5, \tau, 5, \tau, \cdots \rangle$. **end**

A sequence of observable timed events tr is a trace of $\mathcal{S}$ if and only if there exists an execution E such that $tr = E \upharpoonright \mathbb{R}_+ \upharpoonright \{\tau\}$. Let $traces(\mathcal{S})$ denote the set of all traces of model $\mathcal{S}$. Let $\mathcal{I}$ and $\mathcal{S}$

be two system models. $\mathcal{I}$ trace-refines $\mathcal{S}$, written as $\mathcal{I} \sqsupseteq_T \mathcal{S}$, if and only if $traces(\mathcal{I}) \subseteq traces(\mathcal{S})$. A timed safety property can be proved by showing a timed trace refinement relationship from an implementation to a hand-crafted specification which captures the property.

Example 9.2.8 Assume a model $\mathcal{I}$ which contains two events $start$ and end. Further, the property is that end must occur within 5 seconds since $start$ occurs. In order to show the $\mathcal{I}$ satisfies the property, we can show that $\mathcal{I}$ refines (in timed traces semantics) the following specification: $\mathcal{S} = start \rightarrow (end \rightarrow \mathcal{S})\ within[5]$. **end**

An abstract timed event is written as $e@D$, which denotes that e may occur at any time point in D. An execution of $\mathcal{L}_a^{\mathcal{S}}$ is a finite sequence of abstract timed events $\langle e_0@D_0, e_1@D_1, \cdots, e_n@D_n\rangle$ such that $c_i \xrightarrow{e_i}_a c_{i+1}$ and $c_i = (V_i, P_i, D_i)$ for all i and $c_0 = init_a$.

Because a clock is associated with a process construct and we assume the reachable process expressions are finite, only finitely many runtime clocks are necessary at the same time. Further, because there exists a clock ceiling for all runtime clocks, we can apply zone normalization [30] on runtime clocks so that there are finitely many zones with respects to runtime clocks only (i.e. excluding t_G from each zone).

Theorem 9.2.9 *Let $\mathcal{S} = (Var, init_G, P)$ be a model. $\langle a_1@t_1, a_2@t_2, \cdots, a_n@t_n\rangle$ is a trace of $\mathcal{S}$ if and only if there exists a trace of $\mathcal{L}_a^{\mathcal{S}}$, $\langle a_1@D_1, a_2@D_2, \cdots, a_n@D_n\rangle$, such that $t_i \in D_i[t_G]$ for all i.*

In the theorem above, $t_i \in D_i[t_G]$ means that value $t_G = t_i$ is a solution of the constraint represented by $D_i[t_G]$. The theorem states that abstraction is sound and complete with respect to timed traces semantics. We prove this theorem using the following auxiliary theorem, which subsumes Theorem 9.2.9. Theorem 9.2.10 states that not only observable timed event sequences but all timed event sequences are preserved.

Theorem 9.2.10 *Let $\mathcal{S} = (Var, init, P)$ be a model. $\langle e_1@t_1, e_2@t_2, \cdots, e_n@t_n\rangle$ is an execution of $\mathcal{S}$ if and only if there exists an execution of $\mathcal{L}_a^{\mathcal{S}}$, $\langle e_1@D_1, e_2@D_2, \cdots, e_n@D_n\rangle$, such that $t_i \in D_i[t_G]$ for all i,.*

Proof: The theorem is proved by structural induction, on process expression types. The base cases are that P is $Stop$ or $Wait[d]$. Notice that $Skip = \checkmark \rightarrow Stop$. The base cases are straightforward. Assume that P is enabled at time t_0, the clock associated with P is t, the current valuation is V and the current zone is D. The following proves the induction step for the case that P is $P\ timeout[d]\ Q$.

- **only-if**: There are three cases according to rule $to1$ to $to4$. If e_1 is a τ event generated by rule $to2$, we get $t_1 - t_0 \leq d$ (since rule $to3$ is applicable only when d is positive). By hypothesis, let $e_1@D_1'$ be the abstract timed event generated by rule $ato1$. Since $t_1 \in D_1'[t_G]$ (by hypothesis) and $t \leq d$ (by $t_1 - t_0 \leq d$), we conclude that $t_1 \in D_1[t_G]$. The same argument applied to the case where e_1 is an observable event generated by rule to_1. If e_1 is a τ event generated by rule $to4$, then $t_1 - t_0 = d$. Let $e_1@D_1'$ be the abstract timed event generated by rule $ato3$. Since $t_1 \in D_1'[t_G]$ (by hypothesis) and $t = d$ (since $t_1 - t_0 = d$), we conclude that $t_1 \in D_1[t_G]$. Therefore, we prove the only-if part.
- **if**: There are three cases according to rule $ato1$ to $ato3$. If $e_1@D_1$ is generated by rule $ato1$, $D_1[t_G] \Rightarrow t \leq d$. Let $e_1@D'$ be the abstract timed event generated by P. By hypothesis, there exists $e_1@t_1$, a timed event generated by P such that $t_1 \in D'$. By rule $to1$ and $to2$, t_1' satisfies $t_1' \leq d$, and hence $t_1 \in D_1[t_G]$. Similarly, we prove the other cases.

By similar argument on each and every types of process expression, we prove that the theorem holds. Further, it is straightforward to conclude that Theorem 9.2.9 holds given the above theorem. □

By Theorem 9.2.9, given two models $\mathcal{I}$ and $\mathcal{S}$, it is sufficient to show that $\mathcal{L}_a^{\mathcal{I}}$ trace-refines $\mathcal{L}_a^{\mathcal{S}}$ in order to verify that $\mathcal{I}$ trace-refines $\mathcal{S}$. Nonetheless, because the reading of t_G is unbounded, $\mathcal{L}_a^{\mathcal{S}}$ and $\mathcal{L}_a^{\mathcal{I}}$ have infinite number of states[2]. In this section, we show how to overcome this problem.

Normalization To verify that $\mathcal{L}_a^{\mathcal{I}}$ trace-refines $\mathcal{L}_a^{\mathcal{S}}$, we need to normalize (or equivalently determinize) $\mathcal{L}_a^{\mathcal{S}}$. Unlike normalization Timed Automata (which is infeasible [10]), normalization in

[2] Zone normalization on t_G does not work.

this setting follows the standard subset construction. A state of the normalized transition system (referred as normalized state) is a subset of states which are connected by τ-transitions. The intuition is that given a trace only one normalized state is reached. Given an abstract configuration c_a, let $\tau^*(c_a)$ is the set of abstraction configurations which can be reached from c_a via 0 or more τ-transitions in $\hookrightarrow$.

Definition 34 *Let $\mathcal{S} = (Var, init_G, P)$ be a model; $\mathcal{L}_a^{\mathcal{S}} = (C_a, init_a, \hookrightarrow)$ be the abstract transition system. The normalized abstract transition system is $(C_n, init_n, \hookrightarrow_n)$ where C_n is a set of normalized states, $init_n = \tau^*(init_a)$ and $\hookrightarrow_n$ is a labeled transition relation satisfying the following condition: $P \overset{e}{\hookrightarrow}_n Q$ if and only if $Q = \{c_a : C_a \mid \exists\, c_1 : P,\ c_1 \overset{e}{\hookrightarrow} c_2' \wedge c_a \in \tau^*(c_2')\}$.*

We define the traces based on the normalized transition system in the standard way. It can be shown that the above normalization is timed trace preserving. We remark that normalization is not a pre-request for our refinement checking algorithm, instead the normalized transition system is constructed on-the-fly. Figure 9.3 in Example 9.1.1 illustrates how the transition system is normalized using dotted ellipses. The initial normalized state contains state 1 and 3 (since state 3 can be reached from state 1 via a τ-transition).

A refinement checking algorithm (e.g. the one for un-timed refinement checking in the FDR refinement checker [178]) works by normalizing $\mathcal{L}_a^{\mathcal{S}}$ and then comparing states of $\mathcal{L}_a^{\mathcal{I}}$ with the normalized states. It reports a counterexample if and only if a state of $\mathcal{L}_a^{\mathcal{I}}$ enables *more* than the normalized state of $\mathcal{S}$ (which is reached via the same trace) does. The following defines what is *enabled* in the timed setting.

Given an abstract configuration, the set of enabled events is a set of abstract timed events. That is, $enabled(c)$ is a set of pairs $(a, D[t_G])$ such that a is an observable event and D is a zone which tells when a is enabled, i.e. there exists an abstract configuration (V, P, D) such that $c \overset{a}{\hookrightarrow} (V, P, D)$. Notice that only reading of t_G is interested. Multiple abstract timed events with the same event name may be present in $enabled(c)$. In the following, we assume that abstract timed events with the same event are always grouped together in a set of abstract timed events, by applying the following rule:

$\{\cdots, e@D_1[t_G], e@D_2[t_G], \cdots\} = \{\cdots, e@D_1[t_G] \cup D_2[t_G], \cdots\}$[3]. Given two sets of abstract timed events E_1 and E_2, $E_1 \subseteq E_2$ if and only if for all $e@D$ in E_1, there exists $e@D'$ in E_2 such that $D \subseteq D'$. Given a normalized state c_n, we write $enabled(c_n)$ to mean the set $\{x \mid \exists\, c_a : c_n \cdot x \in enabled(c_a)\}$. Similarly, we assume that abstract timed events with the same event are grouped together.

The algorithm Let $\mathcal{I}$ and $\mathcal{S}$ be two system models. Let the corresponding abstract transition systems be $\mathcal{L}_a^{\mathcal{I}} = (C_a^{\mathcal{I}}, init_a^{\mathcal{I}}, \hookrightarrow^{\mathcal{I}})$ and $\mathcal{L}_a^{\mathcal{S}} = (C_a^{\mathcal{S}}, init_a^{\mathcal{S}}, \hookrightarrow^{\mathcal{S}})$ respectively. Further, let $(C_n^{\mathcal{S}}, init_n^{\mathcal{S}}, \hookrightarrow_n^{\mathcal{S}})$ be the normalized transition system of $\mathcal{L}_a^{\mathcal{S}}$. In an abuse of notations, given an abstraction configuration $c_a = (V, P, D)$, we write $c_a[t := 0]$ to mean (V, P, D') where $D' = D_{t:=0}$, i.e. the configuration with clock t being reset.

The algorithm presented in Figure 9.4 verifies whether $\mathcal{I}$ refines $\mathcal{S}$. The idea is to construct the synchronous product of $\mathcal{L}_a^{\mathcal{I}}$ and $\mathcal{L}_n^{\mathcal{S}}$ on-the-fly whilst searching for a state-pair $(i, s_n) : C_a^{\mathcal{I}} \times C_n^{\mathcal{S}}$ of the product such that i enables more events than s_n does. The algorithm is inspired by the one used in FDR and follows its soundness/completeness argument. In order to guarantee termination, the reading of t_G must be bounded. This is achieved by resetting t_G when synchronizing $\mathcal{I}$ and $\mathcal{S}$. Notice that t_G is never pruned during the clock de-activation.

As in standard reachability testing, two data structures are maintained, i.e. a stack named $working$ to store all reachable state-pairs which are yet to be explored and a set named $visited$ to record all visited state-pairs. At line 2, the initial state of the product is pushed into the $working$ stack. If there is a state-pair (i, s_n) yet to explored (so that the condition at line 5 is satisfied), and if i enables more events than s_n (i.e. satisfying the condition at line 7), the algorithm returns false (and reports a counterexample). Otherwise, we generate state-pairs from (i, s_n) and push them into stack $working$. Notice that if i' can be reached from i by a τ-transition in $\mathcal{L}_a^{\mathcal{I}}$, then (i', s_n) is a state of the product. We remark that all visible events must be engaged synchronously by $\mathcal{L}_a^{\mathcal{I}}$ and $\mathcal{L}_a^{\mathcal{S}}$. That is, if i' can be reached from i by a and s_n' can be reached from s_n by a, then (i', s_n') is a state of the product. Further, clock t_G is reset whenever a visible event is synchronously engaged (line 14 and

[3] DBM is not closed under union. Nonetheless, it does not matter here.

```
**procedure** $\mathit{refines}(\mathcal{I}, \mathcal{S})$
1. **Stack** $\mathit{working} := \langle\rangle$; **Set** $\mathit{visited} := \varnothing$;
2. $\mathit{working.push}(\mathit{init}_a^{\mathcal{I}}, \mathit{init}_n^{\mathcal{S}})$;
3. **while** $\mathit{working} \neq \langle\rangle$
4.       $(i, s_n) := \mathit{working.pop}()$;
5.       **if** $(i, s_n) \notin \mathit{visited}$
6.             $\mathit{visited} := \mathit{visited} \cup (i, s_n)$;
7.             **if** $\mathit{enabled}(i) \not\subseteq \mathit{enabled}(s_n)$
8.                   **return** $\mathit{false}$;
9.             **endif**
10.             **foreach** $i'$ **s.t.** $(i, \tau, i') \in \hookrightarrow^{\mathcal{I}}$
11.                   $\mathit{working.push}(i', s_n)$;
12.             **endfor**
13.             **foreach** $a, i'$ **s.t.** $(i, a, i') \in \hookrightarrow^{\mathcal{I}}$
14.                   **foreach** $s_n'$ **s.t.** $s_n' = \{x[t_G := 0] \mid \exists\, c : s_n.\ (c, a, x) \in \hookrightarrow^{\mathcal{S}}\}$
15.                         $\mathit{working.push}(i'[t_G := 0], \tau^*(s_n'))$
16.                   **endfor**
17.             **endfor**
18.       **endif**
19. **endwhile**
20. **return** $\mathit{true}$;
```

Figure 9.4: Algorithm: $\mathit{refines}(\mathit{Impl}, \mathit{Spec})$

15). The soundness of the algorithm is stated in the following theorem.

Theorem 9.2.11 *Let $\mathcal{I}$ and $\mathcal{S}$ be two models. Algorithm $\mathit{refines}(\mathcal{I}, \mathcal{S})$ returns true if and only if $\mathcal{I}$ refines $\mathcal{S}$. It terminates if both are $\mathcal{I}$ and $\mathcal{S}$ are timed divergent-free.*

Proof: The theorem has two parts. The first is that the result is sound. The second is that the algorithm terminates under some condition.

Partial correctness If we never reset T_G, algorithm *refine* resembles the refinement checking algorithm used in FDR, then the theorem follows the soundness and completeness argument presented in [176]. Next, we argue that resetting t_G as in the algorithm is sound and complete as follows. $\mathcal{I}$ does not refine $\mathcal{S}$ if and only if there exists $tr = \langle a_1@t_1, a_2@t_2, \cdots\rangle$ such that $tr \in \mathit{traces}(\mathcal{I}) \wedge tr \notin \mathit{traces}(\mathcal{S})$. Let tr^i be the prefix of tr, i.e., $tr^i = \langle a_1@t_1, \cdots, a_i@t_i\rangle$. By a simple argument, it can be shown that $\mathcal{I}$ does not refine $\mathcal{S}$ if and only if there exists i such that $tr^i \frown \langle a_{i+1}@t_{i+1}\rangle$ is a trace of $\mathcal{I}$ but not $\mathcal{S}$. Next, we prove the theorem using an induction on i.

- **Base case**: if $i = 0$, no resetting of t_G is necessary, we conclude the theorem holds.
- **Induction step**: Assume that $tr^i \in traces(\mathcal{I}) \wedge tr^i \in traces(\mathcal{S})$ and there exists $a_{i+1}@t_{i+1}$ such that $tr^i \frown \langle a_{i+1}@t_{i+1}\rangle$ is a trace of $\mathcal{I}$ but not $\mathcal{S}$. Now, assume t_G starts with $-t_i$. There exists $a_{i+1}@t_{i+1} - t_i$ such that $tr^i \frown \langle a_{i+1}@t_{i+1} - t_i\rangle$ is a trace of $\mathcal{I}$ but not $\mathcal{S}$, where $t_{i+1} - t_i$ is the reading of t_G after resetting at step i. This justifies that resetting t_G is sound and complete.

Terminating By assumption, all variables have finite domains and there are only finite process expressions, and therefore, it remains to show that the number of zones are finite. Further by assumption, both $\mathcal{I}$ and $\mathcal{S}$ are timed divergent-free, this implies that through τ-transitions only, there are only finitely many partitioned on t_G's reading. Because we reset t_G every time an observable event is engaged, we have only finitely many zones since all run-time clocks are bounded (and have finitely partitions since only integers are allowed as time constants). □

Notice that in order to guarantee that the algorithm can terminate, both $\mathcal{I}$ and $\mathcal{S}$ must be timed divergence-free. This assumption is reasonable for two reasons. Firstly, it is relatively straightforward to check whether a model is timed divergent-free or not. A simple approach is to apply zone abstraction without using the global clock t_G. Assume the abstract transition system is $\mathcal{L}_a$. The model is timed divergence-free if and only if $\mathcal{L}_a$ does not contain a loop which contains only τ-transitions and time elapsing. Existence of such a loop can be checked using well-studied algorithms like nested Depth-First-Search or Tarjan's algorithm for strongly connected components[4]. Secondly, without explicit hiding, a model which is not timed divergence-free is often problematic. Furthermore, timed divergence due to hiding of observable events can be avoided by carefully crafting the specification model. This is illustrated in the next section using a pacemaker case study.

[4] The latter has been implemented in PAT.

Model	Size	Property	States/Transitions	PAT (sec.)
Fischer	4	$\Box ct \leq 1$	3452/8305	0.22
Fischer	5	$\Box ct \leq 1$	26496/73628	2.49
Fischer	6	$\Box ct \leq 1$	207856/654776	27.7
Fischer	7	$\Box ct \leq 1$	1620194/5725100	303
Fischer	4	$\Box(x = i \Rightarrow \Diamond cs.i)$	5835/16776	0.53
Fischer	5	$\Box(x = i \Rightarrow \Diamond cs.i)$	49907/169081	5.83
Fischer	6	$\Box(x = i \Rightarrow \Diamond cs.i)$	384763/1502480	70.5
Fischer	4	$Protocol\ refines\ uProtocol$	7741/18616	5.22
Fischer	5	$Protocol\ refines\ uProtocol$	72140/201292	126.3
Fischer	6	$Protocol\ refines\ uProtocol$	705171/2237880	3146
Railway Control	5	deadlock-free	4551/6115	0.42
Railway Control	6	deadlock-free	27787/37482	3.07
Railway Control	7	deadlock-free	195259/263641	24.2
Railway Control	8	deadlock-free	1563177/2111032	223.1
Railway Control	5	$\Box(appr.1 \rightarrow \Diamond leave.1)$	8137/10862	0.95
Railway Control	6	$\Box(appr.1 \rightarrow \Diamond leave.1)$	50458/67639	6.58
Railway Control	7	$\Box(appr.1 \rightarrow \Diamond leave.1)$	359335/482498	58.63

Table 9.1: Experiment results of LTL and refinement checking

9.3 Experiments

The techniques presented in this chapter have been implemented in PAT. We separate the experiments on LTL and refinement checking with timed refinement checking. The data are obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 2GB memory.

LTL and Refinement Checking

In the following, we present the experiment results on two benchmark models. Table 9.1 shows the experiment results on the Fischer's mutual exclusion algorithm and a railway control system [224].

In both examples, PAT performs reasonably well. It handles 10^7 states/transition in a few hours, which is comparable to existing model checkers [111, 179]. Further, a simple experiment shows that the computational overhead of calculating clocks/DBMs is around one third of the overall time.

The data on UPPAAL [135] or RED [217] verifying the same models has been omitted from the table. Because UPPAAL and PAT are based on a different modeling language, the results must be taken with a grain of salt. The state graph generated from a PAT model may contain unnecessary τ-transitions introduced by the compositional process constructs, e.g. the τ in rule $ato3$. In handcrafted UPPAAL models, however, the τ-transitions may be removed by carefully manipulating the clock guards and grouping clock guards and events on the same transition. In such a setting, verification of the UPPAAL is faster (by a factor related to the number of such τ-transitions). However, our experiment show that if we manually construct a PAT model and a UPPAAL model with the same state graph, then PAT and UPPAAL have a similar performance.

Timed Refinement Checking

The refinement checking algorithm is computationally complex, specially when the specification model is highly non-deterministic (i.e. normalization is EXP-time). In this section, we show that our approach is still practically useful.

Case study A pacemaker is an electronic implanted device which functions to regulate the heart beat by electrically stimulating the heart to contract and thus to pump blood throughout the body. Quantitative timing is crucial to pacemaker. Common pacemakers are designed to correct bradycardia, i.e. slow heart beats. A pacemaker mainly performs two functions, i.e. pacing and sensing. Sensing is to monitor the heart's natural electrical activity, helping the pacemaker to gather information on the heart beats and react accordingly. Pacing is when a pacemaker sends electrical stimuli, i.e. tiny electrical signals, to heart through a pacing lead, which start a heartbeat. A model of the pacemaker is of the following form: $Heart \parallel Sensing \parallel Pacing$ where process $Heart$ models normal or abnormal heart condition; process $Sensing$ and $Pacing$ model the two functions. A pacemaker can operate in many different modes, according to the implanted patient's heart problem. All three components above may be different in different modes. For instance, the following models one of the simple models, namely the AAT mode.

$$\begin{aligned}
AAT &= \mathit{Heart} \mid\mid \mathit{Sensing} \mid\mid \mathit{Pacing}(LRI)\\
\mathit{Sensing} &= \textbf{if}\ (SA == 1)\{\textbf{atomic}\{\mathit{pulseA} \rightarrow \mathit{senseA} \rightarrow \mathit{Skip}\};\ \mathit{Sensing}\}\\
&\ \textbf{else}\ \{\mathit{pulseA} \rightarrow \mathit{Sensing}\};\\
\mathit{Pacing}(X) &= (\textbf{atomic}\{\mathit{senseA} \rightarrow \mathit{paceA}\{SA = 0\} \rightarrow \mathit{Skip}\}\\
&\qquad \mathit{timeout}[X]\ (\mathit{paceA}\{SA = 0\} \rightarrow \mathit{Skip})\ \mathit{within}[0]);\ \mathit{Wait}[URI];\\
&\ (\mathit{enableSA}\{SA = 1\} \rightarrow \mathit{Pacing}(LRI - URI))\ \mathit{within}[0];
\end{aligned}$$

where URI and LRI are constants representing upper rate interval (i.e. the fastest a normal heart can beat) and lower rate interval (i.e. the slowest a normal heart can beat). Process $heart$ generates two events $pluseA$ (i.e. atrial does a pulse) and $pluseV$ (i.e. ventral does a pulse), periodically for a normal heart or within one of them missing once a while for an abnormal heart. Process $Sensing$ synchronizes with $heart$ on $pluseA$ and engages in event $senseA$ immediately. Process $atomic\{P\}$, once started, continues to execute without interleaving until blocked. SA is flag telling whether it is necessary to monitor atria (1 for necessary). Process $Pacing$ synchronizes with $Sensing$ on event $senseA$ and paces a heart (captured by event paceA) if a heart pace is missing (captured by $\mathit{timeout}$). We skip the details here. Interested readers can refer to [28].

We model all 16 different modes and verify that the pacemaker satisfies multiple specification. An essential property is that the pacemaker must restore normal heart condition, which can be modeled by the following process.

$$\begin{aligned}
\mathit{Spec} &= \mathit{paceA} \rightarrow \mathit{Started}\\
\mathit{Started} &= (\mathit{paceA} \rightarrow \mathit{Started})\ \mathit{within}[URI, LRI]
\end{aligned}$$

where $P\ within[m, n]$ requires that P must react within m to n time units. Intuitively, Spec means that following one paceA, the next paceA must occur within URI to LRI time units. Because the specification only concerns event paceA, other events must be ignored. One solution is to hide the rest of the events using the hiding operator. For instance, we may verify that $AAT \setminus \{\mathit{pulseA}, \mathit{senseA}, \mathit{enableSA}\}$ refines Spec in order to show that the pacemaker satisfies the property. In general, hiding events may introduce timed divergent traces, which then makes the refinement checking algorithm non-terminating[5]. An alternative way is to verify AAT refines $\mathit{Spec} \mid\mid\mid \mathit{Dummy}$

[5] Also the maximal progress assumption on hidden events may change system behaviors.

Model	Size	Property	States/Transitions	Result	Time (s)
Pacemaker	-	deadlock-free	302442/2405850	true	92.1
Pacemaker	-	correctness	986342/2608226	true	122
Fischer	4	mutual exclusion	9941/34244	true	0.78
Fischer	5	mutual exclusion	141963/599315	true	17.2
Fischer	6	mutual exclusion	2144610/10795380	true	401
Fischer	6	bounded bypass	2429/8065	false	0.36
Fischer	7	bounded bypass	9213/34611	false	1.47
Fischer	8	bounded bypass	32785/137417	false	6.16
Fischer	9	bounded bypass	91665/425966	false	21.1
Fischer	10	bounded bypass	300129/1542020	false	79.8
Fischer	11	bounded bypass	693606/3880577	false	214
Railway Control	4	bounded waiting	918/1359	true	0.45
Railway Control	5	bounded waiting	4764/7199	true	3.21
Railway Control	6	bounded waiting	28782/43795	true	26.2
Railway Control	7	bounded waiting	201444/307071	true	238

Table 9.2: Experiment results of timed refinement checking

where process $dummy$ is defined as follows.

$$Dummy = pulseA \rightarrow Dummy \Box senseA \rightarrow Dummy \Box enableSA \rightarrow Dummy$$

As a result, $Dummy$ will synchronize the irrelevant observable events with AAT. By a simple argument, it can be shown that this is sound and complete.

Table 9.2 summarizes part of our experiments on demonstrating the scalability of our method using the pacemaker system and benchmark systems. The pacemaker contains little concurrency and hence is verified efficiently. Using refinement relationship, we can encode a variety of different properties [178], including mutual exclusion, bounded by-pass, etc. The experiment on Fischer's mutual exclusion algorithm shows that PAT finds a counterexample efficiently. It is time consuming if a system contains multiple parallel processes and the property is true. Nonetheless, PAT handle 10^7 states in a few hours which is comparable to model checkers like SPIN and UPPAAL.

9.4 Summary

In this chapter, we studied model checking of hierarchical real-time systems. Based on the real-time modeling language proposed in Section 3.2, we developed a fully automated abstraction technique to build an abstract finite state machine from the real-time model. We showed that the abstraction has finite state and is subject to model checking. Further, it weakly bi-simulates the concrete model and, therefore, we may perform sound and complete LTL-X model checking, refinement checking and even timed refinement checking upon the abstraction.

This chapter is related to verification of real-time systems. Verification support has been developed for hierarchical specification based on process algebras (e.g. the algebra of timed processes ATP [187, 159], CCS + real-time [223], Timed CSP [183], etc). A preliminary PVS encoding of Timed CSP was presented in [40], which rely heavily on user interactions. In [224], a constraint solving method was proposed to verify CCS + real time. Several model checkers have been developed with Timed Automata [10] being the core of their input languages [135, 35, 207]. The zone abstraction is closely related to works presented in [224]. The difference is that we use implicit clocks and support the hierarchical specification. The soundness discussion of our abstraction is inspired by [136]. There are few verification support for Timed CSP, e.g. the theorem proving approach documented in [40, 101], the translation to UPPAAL models [70, 71] and the approach based on constraint solving [72]. The PAT model checker is the first dedicated verification tool support for Timed CSP models. In addition, PAT complements UPPAAL with the ability to check full LTL-X property and check refinement relationship.

To the best of our knowledge, there are few tool support for timed refinement checking. One of the reasons is that Timed Automata, which extended Büchi Automata with clocks [10], is designed to capture infinite languages. The refinement checking problem is undecidable in the setting of Timed Automata, because the language of Timed Automata is not closed under complement. Our approach is, however, decidable because we are based on finite timed trace semantics. As a price to pay, our method is limited to verify timed safety properties or bounded liveness properties. This is justified by the fact that most of the verified properties are safety properties [103].

Chapter 10

Tool Implementation: Process Analysis Toolkit

Concurrent systems exhibit complex behaviors. System simulation and verification become more and more demanding as the complexity grows. It is highly desirable to have automatic tool support for the system analysis. In this chapter, we present Process Analysis Toolkit (PAT) [1], which is a self-contained framework to support composing, simulating and reasoning of concurrent systems. PAT provides user friendly interfaces for system modeling and simulation. Most importantly, PAT implements various model checking algorithm and optimization techniques developed in the previous chapters. PAT is designed to be a general framework, which can be easily extended to support systems with new languages syntax and verification algorithms. Currently, three modules have been developed in PAT: Communicating Sequential Processes (CSP) module, Real-time System (RTS) module and Web Service (WS) module.

The remainder of the chapter is organized as follows. Section 10.1 provides an architecture overview of PAT. Section 10.2 presents PAT's system design. Section 10.3 discusses the three modules currently developed in PAT in details. Section 10.4 reviews related work and concludes.

10.1 Overview of PAT

Critical system requirements like safety, liveness and fairness play important roles in software specification, development and testing. It is desirable to have handy tools to simulate the system behaviors and verify critical properties. Process Analysis Toolkit (PAT) [1] was initially designed to investigate system verification under fairness assumptions [202]. Later, we have successfully demonstrated PAT as an analyzer for process algebras in [146]. Since then, PAT has been evolved to be a self-contained framework to support analysis of concurrent and real-time systems.

PAT provides simple installation, wizard like guidance and users friendly interfaces. System models can be easily composed with the help of featured editing functions. The models can then be simulated using automatic animations. Most importantly, PAT implements various model checking techniques (proposed in the previous chapters) catering for checking deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions (see Chapter 4), refinement relation checking (see Chapter 6) and etc. To achieve good performance, advanced techniques are implemented in PAT, e.g. partial order reduction (see Section 6.2.2), process counter abstraction (see Section 5.4), bounded model checking (see Chapter 8), parallel model checking (see Section 4.6), etc. We have used PAT to model and verify a variety of systems, ranging from recently proposed distributed algorithms, security protocols to real-world systems like the pacemaker system [28]. Previously unknown bugs have been discovered (see Section 5.2.3). The experiment results show that PAT is capable of verifying systems with large number of states and complements the state-of-the-art model checkers in some cases.

Starting from PAT 2.0, we have applied a layered design to support the analysis of the different system/languages by implementing them as plug-in modules. Figure 10.1 shows the architecture design of PAT. For each supported system (e.g., distributed system, real-time system, service oriented computing, bio-system, security protocols and sensor network), a dedicated module is created in PAT, which identifies the (specialized) language syntax, well-formness rules as well as (operational) formal semantics. For instance, the CSP module is developed for the analysis of concurrent system modeled in CSP#. The operational semantics of the target language translates the behaviors

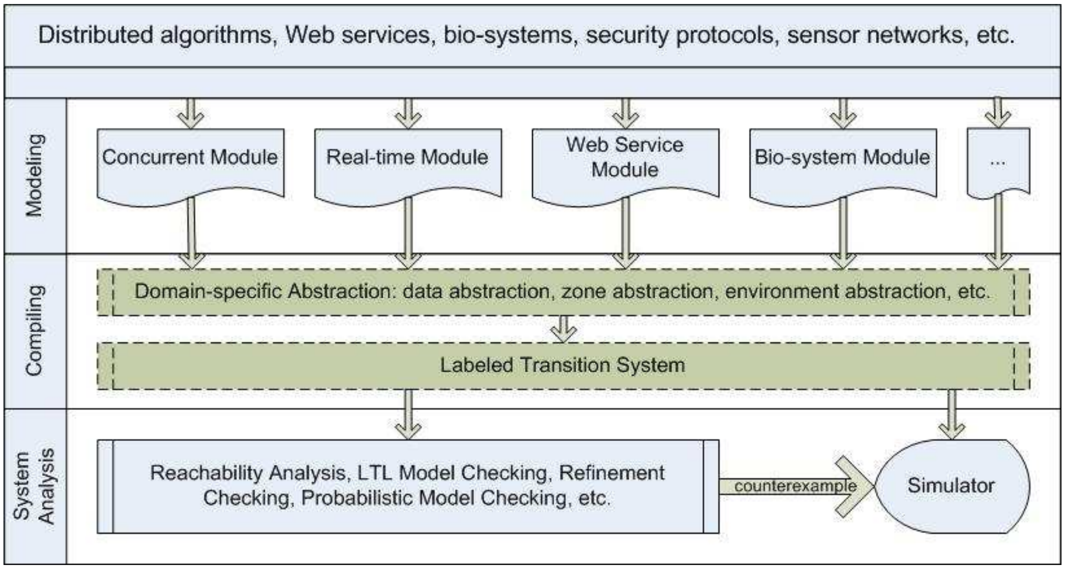

Distributed algorithms, Web services, bio-systems, security protocols, sensor networks, etc.
Modeling
Concurrent Module
Real-time Module
Web Service Module
Bio-system Module
...
Compiling
Domain-specific Abstraction: data abstraction, zone abstraction, environment abstraction, etc.
Labeled Transition System
System Analysis
Reachability Analysis, LTL Model Checking, Refinement Checking, Probabilistic Model Checking, etc.
counterexample
Simulator

Figure 10.1: PAT architecture

of a model into Labeled Transition Systems (LTS). LTS serves as the internal representations of the input models, which can be automatically explored by the verification algorithms or used for simulation. To perform model checking on LTSs, the number of states in the LTSs needs to be finite. For systems with infinite behaviors (e.g., real time clocks or infinite number of processes), abstraction techniques are needed. Examples of abstraction techniques include data abstraction, process counter abstraction (see Section 5.4), clock zone abstraction (see Section 9.1), environment abstraction, etc. The verification algorithms perform on-the-fly exploration of the LTSs. If any counterexample is identified during the exploration, then it can be animated in the simulator. The advantage of this design allows the developed model checking algorithms to be shared by all modules. To create a new module in PAT, users simply need to develop a parser for the target modeling language and language construct classes which define their operational semantics. With the help of predefined APIs, examples and tools (e.g. automatic parser generator), developing a module for a new language becomes relatively easy and requires less expertise[1].

[1] Experiences suggest that a new module can be developed in months or even weeks.

Till now, three modules have been developed. CSP module supports modeling and verification of general concurrent systems, especially under a variety of fairness assumptions. RTS module provides analysis for real-time systems which are specified using hierarchical timed processes. WS module supports modeling and verification of Web service orchestration and choreography. In the future, we plan to develop modules supporting sensor network, UML (state chart and sequence diagrams), security domain (security protocols) and so on.

Starting from 2007, PAT (current version 2.7) has come to a stable stage with solid testing and various applications. More than 50 build-in examples (including all examples and case studies in this thesis) are embedded in PAT. PAT has been used by a number of institutions as a research or educational tool. It has attracted more than 400 downloads from 93 organizations in 23 countries and regions. Currently, there are 1213 classes with more than 110K LOC in PAT's source code. We continue the development with the aim of developing an easy-to-use, powerful and efficient analysis toolkit for multiple domains. A complete PAT history and user information can be found in Appendix D.

10.2 System Design

PAT is implemented in C# 2.0 for the benefits of Object-Oriented design and competitive performance. PAT adopts a hierarchical design. The class diagram in Figure 10.2 shows the horizontal view of the system design. The system consists of two basic packages: $PAT.GUI$ and $PAT.Common$. $PAT.GUI$ contains all graphical user interface classes. $PAT.Common$ contains all basic entities and associations that other language modules can use and follow. Each module is packed into a package and implements necessary classes by following the design interfaces.

Abstract class $ModuleFacadeBase$ in $PAT.Common$ package defines the module interface by adopting the Façade design pattern. It has three public methods to do the parsing, show simulator window and model checker window. $SimulatorGUI$ and $ModelCheckerGUI$ are the graphic user interface classes for simulation and model checking, which can be used by all modules. In addition, the two classes can be overridden according to new display requirements. The internal

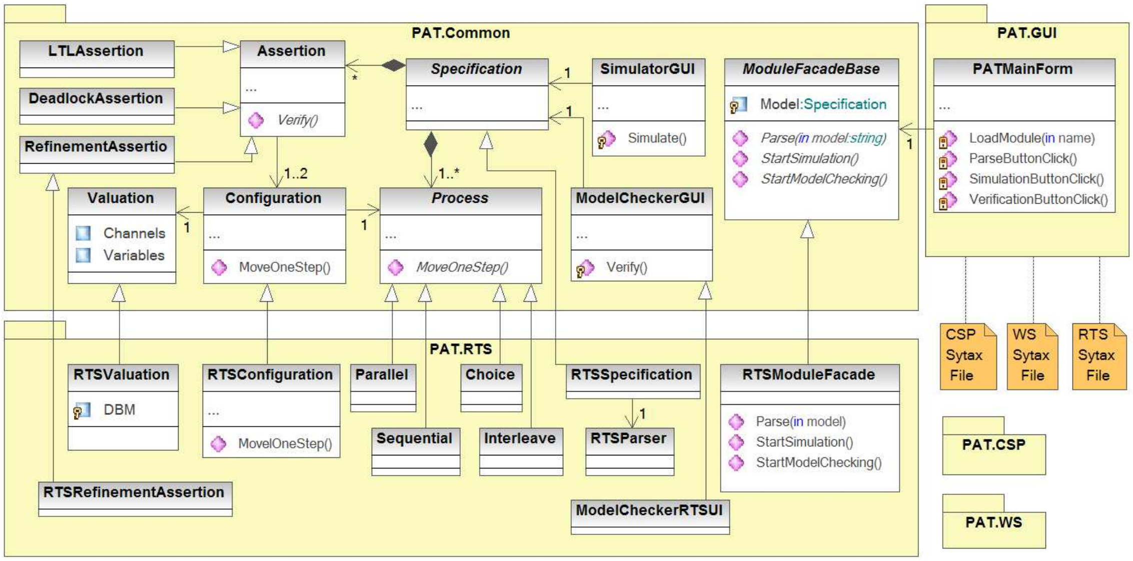


Figure 10.2: Class diagram of PAT

representation of system models are stored as $Specification$ objects, which are composed by a collection of $Process$es and $Assertion$s. $Process$ is the base class for all language constructs with an abstract method $MakeOneMove$, which should be overridden in all sub-classes according to operational semantics of the language constructs. For example, $MakeOneMove$ method in *Skip* class returns a $Stop$ process and unchanged valuation together with the termination event $\checkmark$ (by following *skip* rule in Section 3.1.2). $Assertion$ is the base class of all assertions with an abstract method $Verify$, which should be overridden in the sub-classes to implement the actual verification algorithms. In $PAT.Common$ package, several basic model checking algorithms have been implemented and can be shared by all modules. Each assertion has one $Configuration$ that represents the initial configuration of the LTS to be verified (refinement assertion has two initial configurations for the implementation and specification respectively). Each configuration has a $Process$ and a $Valuation$ of the global variables and channels, which conforms to our definition of the configuration in Section 3.1.2. The LTS of a model to be checked is generated dynamically in the $Assertion$s by keeping on invoking the $MoveOneStep$ method, starting from the initial configuration.

Every module needs to implement $ModuleFacade$ interface (by inheriting $ModuleFacadeBase$

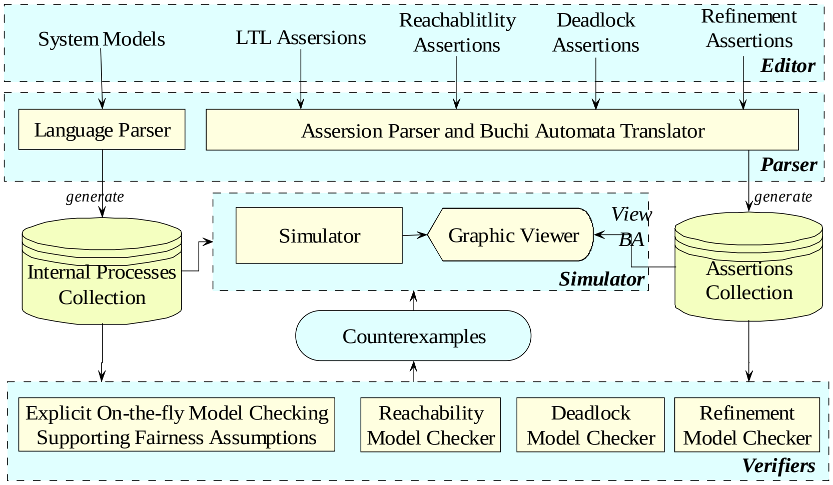


Figure 10.3: Workflow of CSP module

class) in order to communicate with $PAT.GUI$ package. The internal process constructs (e.g. parallel composition, sequential compositions, choice process and so on) need to inherit $Process$ class. New assertions can also be implemented by inheriting $Assertion$ class or its subclasses. Parser class needs implemented in each module according to its language syntax. $Configuration$ and $Valuation$ classes can be customized in each module. For instance, RTS module redefines the two classes in order to store the DBM data structure (see Section 9.1.3) and manipulate clocks in $MoveOneStep$ method. Each module is compiled into a Dynamic Linked Library (DLL), which can be loaded at run-time by the $PAT.GUI$ package.

$PAT.GUI$ package loads the syntax files of different modules at the initialization, which stores the syntax color and DLL linking information. When users want to parse, simulate or verify an input model, the linked DLL is loaded into the system dynamically and the corresponding interface method is invoked.

In the vertical view, four components constitute to PAT, namely the editor, the parser, the simulator and the verifiers. Figure 10.3 demonstrates the design of CSP Module. The editor is featured with powerful text editing, syntax highlighting, multi-documents environment, etc. The parser transforms the system models and the properties into internal representation as $Process$es and $Assertion$s. The simulator allows users to perform various simulation tasks on the input models: complete states generation of execution graph, automatic simulation, user interactive simulation, trace replay and etc. The simulator is also used to visualize Büchi automata generated from the negation of LTL assertions and counterexamples generated by the verifiers. Common verifiers are used to deal with different properties efficiently.

10.3 PAT Modules

In this section, we introduce the three modules and their major functionalities developed so far. We focus on the unique features for each module. The common functions like model editing, simulation, deadlock and reachability analysis are omitted.

10.3.1 CSP Module

CSP module is designed for analyzing general concurrent systems. CSP module supports a rich modeling language CSP# (see Section 3.1 and PAT user manual [1]). Distinguished from existing model checkers, CSP module found its strength in two unique aspects.

Firstly, the LTL model checking algorithm in CSP module is designed to handle a variety of fairness constraints efficiently. Two different approaches for verification under fairness are supported in PAT, targeting different users. For ordinary users, one of the following options may be chosen and applied to the whole system: *weak fairness or strong local/global fairness*. The model checking algorithm works by identifying one bundle of fair executions (within a SCC) at a time and checks whether the desirable property is satisfied. In general, however, system level fairness may sometimes be overwhelming. The worst case complexity is high and, worse, partial order reduction is not feasible for model checking under strong local/global fairness. A typical scenario for network protocols is that fairness constraints are associated with only messaging but not local actions. We thus support an alternative approach, which allows users annotate individual actions with fairness. Notice that this option is only for advanced users who know exactly which part of the system needs fairness constraints. Nevertheless this approach is much more flexible, i.e., different parts of the system may have different fairness. Furthermore, it allows partial order reduction over actions which are irrelevant to the fairness constraints, which allows us to handle much larger systems. Other effective reduction techniques supported by CSP module including process counter abstraction (see Section 5.4 for more details), which reduces state space dramatically by grouping similar processes in interleaving composition. Furthermore, CSP module provides a parallel verification option (see Section 4.6) for LTL properties to make best use of multi-core CPU.

Secondly, CSP module allows users to reason about behaviors of a system as a whole by refinement checking. Refinement checking (see Chapter 6) is to verify whether an implementation's behaviors follow the specification's. CSP module supports six notions of refinements based on different semantics, namely trace refinement/equivalence, stable failures refinement/equivalence, failures divergence refinement/equivalence. A refinement checking algorithm (see Section 6.2) is used to perform refinement checking on-the-fly.

10.3.2 Real-time System Module

Real-time system (RTS) module supports analysis of real-time systems. In RTS module, a system is modeled using a hierarchical timed process with mutable data (see Section 3.2). Additional behavioral patterns which are useful in modeling and analyzing real-time systems are introduced. Examples are $deadline$ (which constrains a process to terminate within some time units), $timed$ $interrupt$, etc. Instead of explicitly manipulating clock variables (as in Timed Automata), the time related process constructs are designed to build on implicit clocks. Based on the clock zone abstraction (see Chapter 9), RTS module is designed to support dense-time semantic model (in contrast to discrete-time or continuous-time), i.e., all clock values are rational numbers. RTS module supports only integer numbers, since a set of rational numbers can be converted an 'equivalent' set of integer numbers by multiplying the least common multiple.

RTS module provides efficient mechanical verification support for a number of properties, deadlock-freeness, (timed) divergence-freeness, reachablity, etc. LTL-X (i.e. LTL without next operator) model checking or trace refinement checking (without time transitions in the specification) are supported based on the clock abstraction techniques developed in Chapter 9.

For timed refinement checking, RTS module uses a timed trace semantics (i.e. a mapping from a model to a set of finite timed event sequences) and a timed trace refinement relationship (i.e. a model satisfies a specification if and only if the timed traces of the models are a subset of those of the specification). The verification algorithm developed for timed refinement checking (see Section 9.2.3) will verify that a system model is consistent with a specification by showing a refinement relationship. A timed event sequence is presented as a counterexample if there is no such refinement relationship. For the timed refinement checking, PAT requires that the implementation or specifications are not divergent, otherwise the shared clock will not be bounded.

10.3.3 Web Service Module

The Web Services paradigm promises to enable rich, dynamic, and flexible interoperability of highly heterogeneous and distributed web-based platform. There are two different viewpoints in the area of Web Service composition. Web Service *choreography* describes collaboration protocols of cooperating Web Service participants from a global view. Web Service *orchestration* describes collaboration of the Web Services in predefined patterns based on local decision about their interaction with one another at the message/execution level, which is a local view.

WS module is developed to offer practical solutions to for two important issues in Web Services paradigm. First, if both the choreography and orchestration are given, it is important to guarantee that the two views are consistent, by showing that the orchestration conforms to the choreography. Second, given only a choreography, it is necessary to check whether it is implementable and synthesize a prototype implementation (if possible). In WS module, conformance is verified by showing weak simulation relationship using an on-the-fly model checking algorithm. A scalable lightweight approach is used to solve the synthesis problem.

Figure 10.4 shows the workflow of WS module. Given a choreography or an orchestration, a preprocessing component is used to extract relevant information and build a simplified model in inter-

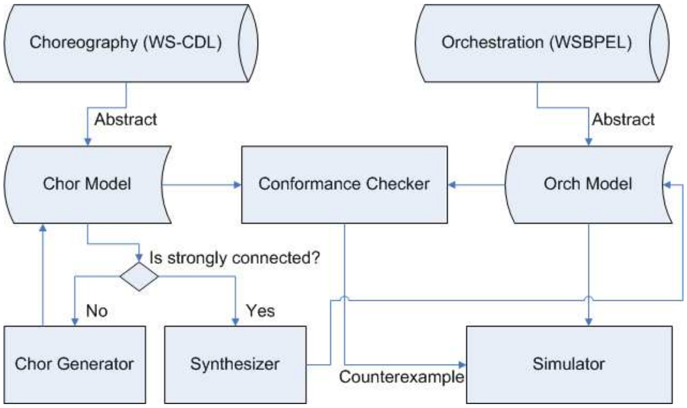


Figure 10.4: WS module workflow

mediate languages (see Section 7.5), which are designed to capture behaviors of choreography and orchestration. Both languages (for choreography and orchestration) have their own parsers, compilers as well as formal operational semantics. Therefore, users can quickly write a Web Service model and analyze it using our visualized simulator, verifier and synthesizer.

Given a choreography, WS module can statically analyze whether it is well-formed, for instance whether it can be implemented in a distributed setting without introducing unexpected behaviors. If the choreography is not implementable, WS module generates an implementable one, by injecting extra message passing into the choreography. Otherwise, WS module may be used to automatically generate a prototype orchestration (which may later be refined and translated to a WS-BPEL document). If an orchestration is provided, the conformance checker allows users to verify whether the orchestration is valid with respect to the choreography. Choreography may contain free variables (for environment inputs), which must be instantiated during execution time. This is achieved by synchronizing the environments (of the choreography and the orchestration) whenever a free variable is used. WS module offers other verification options as well, in particular, deadlock-freeness checking, LTL model checking, etc.

10.4 Summary

In summary, we have developed a self-contained framework PAT for specification, simulation and verification of concurrent and real-time systems. PAT adopts an extensible design, which allows new languages and verification algorithms to be supported easily. Three modules have been developed to support the analysis of different systems. Experiment results show that PAT does verification rather efficiently.

As a temporal logic model checker, PAT is related to the model checking tools like NuSMV [53], SPIN [111], mCRL2 [102] and so on. Compared to these tools, PAT adopts event-based modeling language with the emphasis of fairness verification. Better usability and extensibility are the advantages of PAT. Bogor [77] is another extensible model checker developed as a plugin of Eclipse. It allows user to extend the base language to support new language features, but can not be fully customized with desired syntax and semantic model.

For refinement checking, PAT is related to tools on equivalence/refinement checking (or language containment checking), e.g. FDR [176], ARC [162] and ProBE [3]. Compared with FDR, PAT performs an on-the-fly verification with partial order reduction. ARC (Adelaide Refinement Checker) is a refinement checker based on ordered binary decision diagrams (BDD). It has been shown that ARC outperforms FDR in a few cases [162]. PAT adapts an explicit approach for model checking. It has long been known there are pros and cons choosing an explicit approach or a BDD approach (refer to comparisons between SPIN and NuSMV). Nonetheless, we may incorporate BDD in the future. ProBE is a simulator developed by Formal Method Europe to interactively explore traces of a given process. The simulator embedded in PAT has the full functionality of ProBE.

In terms of real-time verification, PAT is related to a number of automatic verification support for Timed Automata, including UPPAAL [135], KRONOS [35], RED [217], Timed COSPAN [207], Rabbit [33]. Different from the Timed Automata approach, PAT model checker is the first dedicated verification tool support for hierarchical real-time modeling languages (like Timed CSP) by adapting advanced verification techniques. In addition, PAT complements UPPAAL with the ability to check full LTL-X properties and refinement relationship. To the best of our knowledge, there are few verification support for Timed CSP, e.g. the theorem proving approach documented in [40, 101], the translation to UPPAAL models [70, 71] and the approach based on constraint solving [72]. Regarding to the timed refinement checking, there is no tool support to the best of our knowledge. One of the reasons is that it has been proved that the refinement checking (or equivalently the language inclusion) problem in the setting of Timed Automata [10] is undecidable. Other negative results include that Timed Automata cannot be determinized. It has been proved [160] that language inclusion checking against a Timed Automaton with one clock, however, is decidable. We show that timed refinement checking (under an assumption on τ-transitions) is decidable in our setting. The timed systems we tackle correspond to a special subclass of Timed Automata (i.e. with one clock only and τ-transitions, and all states are accepting). As a price to pay, direct comparison of clocks' values are disallowed and our method is limited to verify timed safety properties (or bounded liveness properties). The latter is justified by the fact that most of the verified properties are safety properties [103].

WS module is related to works on verifying Web Services. In particular, it is closely related to LTSA-WS [95], which translates Web Service model into Finite State Processes (i.e., a simple modeling language) and then verifies conformance by showing a bi-simulation relationship. Different from their approach, WS module is based on languages specially designed for Web Services and supports features like channel passing, shared variables/arrays, service invocation with service replication, etc.

Chapter 11

Conclusion

This chapter concludes the thesis. Section 11.1 summarizes the contribution of this thesis and Section 11.2 discusses some on-going and future directions.

11.1 Summary of the Thesis

In this thesis, we focused on the verification of concurrent and real-time systems using model checking approach. The main outcome is Process Analysis Toolkit (PAT), which is a self-contained framework to support composing, simulating and reasoning of concurrent and real-time systems. We used PAT to model and verify a variety of systems, ranging from recently proposed distributed algorithms, concurrent systems to real-world applications. In the following, we summarize the contributions of the thesis, which are all implemented in PAT.

First of all, we designed an event-based modeling language for concurrent and real-time systems. This language integrates high-level specification languages with mutable data variables and low-level procedural codes. Timing requirements for real-time systems are captured using behavior patterns. With the formally defined syntax and operational semantics, the system models can be translated into labeled transition systems, which are suitable for model checking.

One of the main focuses of this thesis is LTL verification with fairness assumptions. We developed an on-the-fly LTL model checking algorithm for fairness enhanced systems based on SCC searching. This algorithm gives a unified solution that handles a variety of fairness, e.g. process-level weak/strong fairness, event-level weak/strong fairness and strong global fairness. To achieve better performance, the algorithm was further changed to support parallel verification in multi-core architecture with shared memory. We applied the proposed fairness model checking algorithms on a set of self-stabilizing population protocols for ring networks, which only work under global fairness. One previously unknown bug in a leader election protocol [118] was discovered using PAT. Population protocols are designed for network with large or even unbounded number of nodes, which raises the space explosion problem. To solve this problem, a process counter abstraction technique was developed for model checking parameterized systems under fairness. We showed that model checking under fairness is feasible, even without the knowledge of process identifiers.

The second focus of this thesis is refinement checking. Our modeling language is an event-based formalism, whose behaviors can be captured using event traces. Following the trace semantics in CSP [108], we developed a trace refinement verification algorithm to verify complex properties beyond the expressiveness power of LTL. Advanced model checking techniques, like partial order reduction was incorporated into the proposed algorithm. To demonstrate the usefulness of refinement checking, we presented two applications. First, we verified linearizability based on refinement relations from concrete implementations to linearizable abstract specifications. We have checked a variety of implementations of concurrent objects, including the first algorithms for the mailbox problem [19] and scalable NonZero indicators [78]. Second, we applied the refinement checking algorithm to automatically check consistency between Web Service choreography and Web Service orchestration by showing conformance relationship between them.

As an attempt to handle large state space, we used bounded model checking technique [54] to verify LTL properties using compositional encoding of hierarchical systems as SAT problems. The encoding avoids state space explosion by exploring only the partial state space. The experiment results showed that our approach has a competitive performance for verifying systems with large number of states. However, this approach was limited by the performance of the SAT solver and hard to scale up for encoding of variables. Therefore, we did not continue in this direction.

We remark that in theory we can encode the property model as temporal logic formulae (as temporal logic is typically more expressive than LTS) and then apply temporal-logic based model checking to verify the property (e.g. SCC-based LTL verification). It is, however, impractical. For instance, LTL model checking is exponential in the size of the formulae and therefore it cannot handle formulae which encode non-trivial property model. In short, refinement checking allows users to verify a different class of properties from temporal logic formulae. Comparing the three verification support, SCC-based LTL verification is more efficient than refinement checking and bounded model checking when the LTL formula is small. Bounded model checking is good for the verification with limited search depth or counterexamples.

Real-time systems are not subject to model checking directly because of the infinite clock values. To support the automatic verification of real-time systems, we proposed a clock zone abstraction technique to build an abstract finite state machine from the real-time model, which makes the model checking feasible. In our approach, clocks are created dynamically to capture constraints introduced by the timed process constructs, and deleted if they are not used by any process. Clocks may be shared for many constructs so that the number of clocks used is minimum. We proved that this abstraction has finite state and it weakly bi-simulates the concrete model, which allows us to perform sound and complete LTL model checking or refinement checking upon the abstraction. To reason about behaviors involving time, we formally defined a timed trace semantics and a timed trace refinement relationship. We extended the zone abstraction technique to preserve timed event traces, hence timed refinement checking is possible.

11.2 On-going and Future Works

We are actively developing PAT. In this section, we discuss some on-going and future works surrounding the PAT development.

11.2.1 Tool Development

Starting with modeling languages, CSP# and its real-time extension are quite expressive for modeling concurrent and real-time systems. We are planning to introduce new language features, including user defined data structures, variables of real values, higher order processes. Adding new features is not a matter of arithmetic, each of these features requires substantial effort in both research and implementation. User defined data structures can be implemented using external C# code with proper interfaces interacting with PAT. To support real values, data abstraction is needed to reduce the continuous domains into discrete ones. For the real-time systems, we would like to add the syntax for explicit clocks and the notion of urgent events [62], which can considerably simplify the modeling process. Another aspect related to the modeling is the model conversion from existing languages. We are working on the automatic conversion from Promela models [111] (the input language for SPIN) to CSP# models. This is feasible because Promela is a subset of precess algebra, and CSP# covers most syntax of Promela. The benefit of this conversion is to attract more users of other tools to start using PAT. Another targeting language is Petri-net [165] for its simple structure based on labeled transition systems.

To improve the usability of PAT, we are working of two directions. First, we are providing the advanced editing features for system modeling like refactoring functions and IntelliSense (code auto completion). Examples of refactoring functions are “go to definition for selected variable or process", “find usage of selected variable and process", “extract selected text as another process", etc. IntelliSense is an auto completion technique based on the user input, which is handy when there are more libraries. Second, graphical system input as supported by Uppaal [135] and TINA [32] is extremely helpful for starting users. Our plan is to implement the drag-and-drop model creation in the future.

To improve the reliability of PAT, there are two possible strategies. The first strategy is to create more testing cases. We are using unit testing tool (e.g. NUnit) to perform unit testing and integration testing. With the rapid increasing of the functionalities, more testing cases are needed. The second strategy is to use verification tools like Spec# [27] and Contracts [26] to conduct both static and dynamic verifications on pre-conditions, post-conditions, loop invariants, assertions and so on.

11.2.2 Model Checking Techniques

To develop new model checking algorithms and related techniques has top priority in PAT. We have identified five possible directions.

First, we plan to investigate methods to combine well-known state space reduction techniques. we know that systems that accept an infinite number of threads or unbound data structures make model checking impossible. Symmetric properties among threads can reduce infinite number of threads to a small number. Data abstraction for infinite domain data variables can also be incorporated into the model checking to handle unbounded data size. These solutions are valuable for our model checking algorithms and refinement checking algorithms.

Second, we are interested in adopting symbolic representation techniques, like Binary Decision Diagrams (BDD) [42]. Our bounded model checking solution is limited by the performance of SAT solvers. Since BDD has been used to handle extremely large number of states (many orders of magnitude larger than could be handled by the explicit-state algorithms), implementation based on BDD may make bound model checking scalable for compositional models.

Third, we are interested in automatic detection of symmetry relations. Reductions based on symmetry relations have been investigated during the last decade. Most of these approaches requires users to provide the symmetry relations, which makes this technique impractical. This is the reason why SPIN does not support symmetry reduction. Therefore, automatical discover of symmetry relations from the model is worth investigating. One possible solution is to detect symmetry relations by analyzing program structures statically.

Last, we plan to look in to probabilistic model checking techniques, which can help to analyze systems which exhibit random or probabilistic behavior. Particular, the integration of probabilistic model checking with fairness assumptions and refinement checking techniques is interesting to us.

11.2.3 Module Development

Since PAT offers a flexible design, implementing new modules is particularly interesting. Our target domains include orchestration language, security protocol, sensor network (particular NesC language [98]). Orc language [156] is a modeling language for distributed and concurrent programming, which provides uniform access to computational services, including distributed communication and data manipulation. The only verification support for Orc language is the approach by converting Orc models to Timed Automata, hence using UPPAAL [73] to do the verification. Direct support of Orc language in PAT is possible, because Orc is an algebra like language extended with clocks. CSP has been used to model and verify security protocol with great success [185, 180]. The common approach is to convert security model into CSP, and use FDR to do the verification. Direct supporting security modeling language will make the verification simpler and more effective. The verification of sensor network model is done by converting NesC language into Promela, which is a subset of CSP#. Therefore, developing a module to support NesC language is straightforward.

Bibliography

[1] PAT: An Enhanced Simulator, Model Checker and Refinement Checker for Concurrent and Real-time Systems. http://pat.comp.nus.edu.sg/.

[2] SAT Competition. http://www.satcompetition.org/.

[3] Process Behaviour Explorer (ProBE), 2003. http://www.fsel.com/probe_download.html.

[4] P. A. Abdulla, A. Bouajjani, and M. Müller-Olm. Abstracts collection – software verification: Infinite-state model checking and static program analysis. In *Software Verification: Infinite-State Model Checking and Static Program Analysis*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006.

[5] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. Ramakrishna, and D. White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 1999)*, pages 207–222, 1999.

[6] K. Alagarsamy. Some Myths About Famous Mutual Exclusion Algorithms. *SIGACT News*, 34(3):94–103, 2003.

[7] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 1997.

[8] R. Alur and D. Dill. Model-Checking for Real-Time Systems. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science (LICS 1990)*, pages 414–425, 1990.

[9] R. Alur and D. L. Dill. Automata for Modeling Real-time Systems. In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming (ICALP 1990)*, pages 322–335, 1990.

[10] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[11] R. Alur and T. A. Henzinger. A Really Temporal Logic. *Journal of the ACM*, 41:181–204, 1994.

[12] R. Alur, K. Mcmillan, and D. Peled. Model-checking of Correctness Conditions for Concurrent Objects. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 219–228. IEEE, 1996.

[13] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under Abstraction for Verifying Linearizability. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

[14] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing Population Protocols. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, volume 3974 of *LNCS*, pages 103–117, 2005.

[15] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing Population Protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4):643–644, 2008.

[16] D. Angluin, M. J. Fischer, and H. Jiang. Stabilizing Consensus in Mobile Networks. In *Proceedings of the 2006 International Conference on Distributed Computing in Sensor Systems (DCOSS 2006)*, volume 4026 of *LNCS*, pages 37–50, 2006.

[17] K. Apt and D. Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Information Processing Letters*, 22(6):307–309, 1986.

[18] K. R. Apt, N. Francez, and S. Katz. Appraising Fairness in Languages for Distributed Programming. *Distributed Computing*, 2(4):226–241, 1988.

[19] M. K. Arguilera, E. Gafni, and L. Lamport. The Mailbox Problem. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC 2008)*, pages 1–15. Springer, 2008.

[20] J. Aspnes and E. Ruppert. An Introduction to Population Protocols. *Bulletin of the European Association for Theoretical Computer Science*, 93:98–117, 2007.

[21] P. C. Attie, N. Francez, and O. Grumberg. Fairness and Hyperfairness in Multi-Party Interactions. *Distributed Computing*, 6(4):245–254, 1993.

[22] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, Inc., Publication, 2004.

[23] J. Barnat, J. Chaloupka, and J. V. D. Pol. Improved Distributed Algorithms for SCC Decomposition. *ENTCS*, 198(1):63–77, 2008.

[24] J. Barnat, J. Chaloupka, and J. V. D. Pol. Distributed Algorithms for SCC Decomposition. *To appear in Journal of Logic and Computation*, 2009.

[25] J. Barnat and P. Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 316–330, 2006.

[26] M. Barnett, M. Fähndrich, P. de Halleux, F. Logozzo, and N. Tillmann. Exploiting the Synergy between Automated-Test-Generation and Programming-by-Contract. In *Proceedings of the 31th International Conference on Software Engineering (ICSE 2009) Companion*, pages 401–402, 2009.

[27] M. Barnett, K. R. M. Leino, K. Rustan, M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Proceedings of the International Workshop of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, pages 49–69. Springer, 2004.

[28] S. S. Barold, R. X. Stroopbandt, and A. F. Sinnaeve. *Cardiac Pacemakers Step by Step: an Illustrated Guide*. Blachwell Publishing, 2004.

[29] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 341–353. Springer, 1999.

[30] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[31] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread Quantification for Concurrent Shape Analysis. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, pages 399–413. Springer, 2008.

[32] B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In *Proceedings of the 3rd International Conference on the Quantitative Evaluaiton of Systems (QEST 2006)*, pages 123–124, 2006.

[33] D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, pages 122–125. Springer-Verlag, 2003.

[34] A. Biere, A. Cimatti, E. M. Clarke, and Y. S. Zhu. Symbolic Model Checking without BDDs. In *PProceedings of the 5th International Conference of Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, pages 193–207. Springer, 1999.

[35] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.

[36] L. Brim, I. Cerna, P. Krcal, and R. Pelanek. Distributed LTL Model Checking Based on Negative Cycle Detection. In *Proceedings of the 21st International Conference of Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2001)*, pages 96–107, 2001.

[37] L. Brim, I. Cerna, P. Moravec, and J. Simsa. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *Proceedings of the 5th International Conference of Formal Methods in Computer-Aided Design (FMCAD 2004)*, pages 352–366, 2004.

[38] L. Brim, I. Cerna, P. Moravec, and J. Simsa. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *Proceedings of the 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2005)*, pages 1–12, 2005.

[39] L. Brim, I. Cerná, P. Moravec, and J. Simsa. On Combining Partial Order Reduction with Fairness Assumptions. In *Proceedings of the 11th International Workshop Formal Methods: Applications and Technology (FMICS 2006)*, volume 4346 of *LNCS*, pages 84–99, 2006.

[40] P. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, 1999.

[41] S. D. Brookes, A. W. Roscoe, and D. J. Walker. An Operational Semantics for CSP. Technical report, 1986.

[42] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[43] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence Testing in Term-Level Bounded Model Checking. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, pages 348–362, 2003.

[44] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[45] M. J. Butler and M. Leuschel. Combining CSP and B for Specification and Property Verification. In *Proceedings of the 12th International Symposium on Formal Methods (FM 2005)*, pages 221–236, 2005.

[46] S. V. Campos and E. M. Clarke. Real-time Symbolic Model Checking for Discrete Time Models. *Theories and experiences for real-time system development*, pages 129–145, 1994.

[47] D. Canepa and M. Potop-Butucaru. Stabilizing Token Schemes for Population Protocols. *Computing Research Repository (CoRR)*, abs/0806.3471, 2008.

[48] M. Carbone, K. Honda, N. Yoshiba, R. Milner, G. Brown, and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. *WCD-Working Note*, 2006.

[49] I. Cerná and R. Pelánek. Distributed Explicit Fair Cycle Detection: Set Based Approach. In *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN 2002)*, 2002.

[50] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*, pages 128–147, 2004.

[51] C. Q. Chen, J. S. Dong, and J. Sun. A Verification System for Timed Interval Calculus. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 271–280. ACM, 2008.

[52] C. Q. Chen, J. S. Dong, J. Sun, and A. Martin. A Verification System for Interval-based Specification Languages. *ACM Transactions on Software Engineering and Methodology*, 2009. Accepted.

[53] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, pages 359–364, 2002.

[54] E. M. Clarke, A. Biere, R. Raimi, and Y. S. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[55] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[56] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages (POPL 1983)*, pages 117–126, 1983.

[57] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, volume 697 of *LNCS*, pages 450–462. Springer, 1993.

[58] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[59] R. Colvin, S. Doherty, and L. Groves. Verifying Concurrent Data Structures by Simulation. *Electronic Notes in Theoretical Computer Science*, 137(2):93–110, 2005.

[60] R. Colvin and L. Groves. Formal Verification of an Array-Based Nonblocking Queue. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 507–516. IEEE, 2005.

[61] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[62] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[63] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, pages 53–68, 2000.

[64] Y. Deng and J.-F. Monin. Verifying Self-stabilizing Population Protocols with Coq. In *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008)*, pages 201–208. IEEE Computer Society, 2009.

[65] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving Linearizability Via Non-atomic Refinement. In *Proceedings of the 5th International Conference on integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 195–214. Springer, 2007.

[66] E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[67] E. W. Dijkstra. Programming: From Craft to Scientific Discipline. In *International Computing Symposium 1977*, pages 23–30, 1977.

[68] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

[69] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal Verification of a Practical Lock-free Queue Algorithm. In *Proceedings of the 24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004)*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

[70] J. S. Dong, P. Hao, S. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *LNCS*, pages 483–498. Springer, 2004.

[71] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and W. Yi. Timed Automata Patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008.

[72] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *LNCS*, pages 342–359. Springer, 2006.

[73] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *LNCS*, pages 226–245. Springer, 2006.

[74] J. S. Dong, B. P. Mahony, and N. Fulton. Modeling Aircraft Mission Computer Task Rates. In *Proceedings of the 6th International Symposium on Formal Methods (FM 1999)*, page 1855, 1999.

[75] R. Duke, G. Rose, and G. Smith. Object-Z: a specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17(5-6):511–533, 1995.

[76] B. Dutertre and S. Schneider. Using a PVS Embedding of CSP to Verify Authentication Protocols. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOL 1997)*, pages 121–136. Springer-Verlag, 1997.

[77] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*, pages 148–152, 2005.

[78] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC 2007)*, pages 13 – 22. ACM, 2007.

[79] E. A. Emerson and V. Kahlon. Reducing Model Checking of the Many to the Few. In *Proceedings of the 13th International Conference on Automated Deduction (CADE 2000)*, pages 236–254, London, UK, 2000. Springer-Verlag.

[80] E. A. Emerson and C. L. Lei. Modalities for Model Checking: Branching Time Logic Strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[81] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. In *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV 1991)*, pages 136–145, London, UK, 1991. Springer-Verlag.

[82] E. A. Emerson and A. P. Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):617–638, 1997.

[83] E. A. Emerson and R. J. Trefler. From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking. In *Proceedings of the 8th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 1999)*, pages 142–156, 1999.

[84] J. Esparza. Verification of Systems with an Infinite State Space. In *4th Summer School Modeling and Verification of Parallel Processes*, pages 183–186, 2000.

[85] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhorst. A Set-Theoretic Model for Real-Time Specification and Reasoning. In *Proceedings of the 4th International Conference on Mathematics of Program Construction (MPC 1999)*, pages 188–206. Springer-Verlag, 1999.

[86] C. Fischer. CSP-OZ: a combination of object-Z and CSP. In *Proceedings of the 1st International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 1997)*, pages 423–438. Chapman & Hall, Ltd., 1997.

[87] M. J. Fischer. (personal communication with leslie lamport), June 1985.

[88] M. J. Fischer and H. Jiang. Self-stabilizing Leader Election in Networks of Finite-state Anonymous Agents. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS 2006)*, volume 4305 of *LNCS*, pages 395–409. Springer, 2006.

[89] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is There a Best Symbolic Cycle-Detection Algorithm? In *Proceedings of the 7th International Conference of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, pages 420–434. Springer, 2001.

[90] R. W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962.

[91] H. Foster. Tool Support for Safety Analysis of Service Composition and Deployment Models. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2008)*, pages 716–723, 2008.

[92] H. Foster. WS-Engineer 2008. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC 2008)*, volume 5364 of *LNCS*, pages 728–729, 2008.

[93] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model Checking Service Compositions under Resource Constraints. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE 2007)*, pages 225–234, 2007.

[94] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 152–163, 2003.

[95] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2008)*, pages 771–774, 2006.

[96] N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[97] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, pages 53–65. Springer, 2001.

[98] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The NesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI 2003)*, pages 1–11. ACM, 2003.

[99] J. Geldenhuys and A. Valmari. More Efficient On-the-fly LTL Verification with Tarjan's Algorithm. *Theoretical Computer Science*, 345(1):60–82, 2005.

[100] D. Giannakopoulou, J. Magee, and J. Kramer. Checking Progress with Action Priority: Is it Fair? In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 1999)*, volume 1687 of *LNCS*, pages 511–527, 1999.

[101] T. Göthel and S. Glesner. Machine Checkable Timed CSP. In *Proceedings of the First NASA Formal Methods Symposium (NFM 2009)*. NASA Conference Publication, 2009.

[102] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. *The Formal Specification Language mCRL2*. IBFI, 2007.

[103] N. Halbwachs. Delay Analysis in Synchronous Programs. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, pages 333–346. Springer, 1993.

[104] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[105] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modeling and Analysis of an Audio/video Protocol: an Industrial Case Study using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS 1997)*, pages 2–13, 1997.

[106] M. R. Henzinger and J. A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996)*, pages 16–27, 1996.

[107] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[108] C. A. R. Hoare. *Communicating Sequential Processes*. International Series on Computer Science. Prentice-Hall, 1985.

[109] G. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth-first Search. In *Proceedings of the 2nd International SPIN Workshop on Model Checking Software (SPIN 1996)*, pages 23–32, 1996.

[110] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engeering*, 23(5):279–295, 1997.

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

[112] G. J. Holzmann and D. Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[113] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, 1977.

[114] C. P. Inggs and H. Barringer. CTL* model checking on a shared-memory architecture. *Formal Methods in System Design*, 29(2):135–155, 2006.

[115] C. N. Ip and D. L. Dill. Verifying Systems with Replicated Components in Murphi. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV 1996)*, pages 147–158, London, UK, 1996. Springer-Verlag.

[116] Y. Isobe and M. Roggenbach. A Generic Theorem Prover of CSP Refinement. In *Proceedings of the 11th International Conference of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2005)*, pages 108–123, 2005.

[117] F. Jahanian and A. K. Mok. Modechart: A Specification Language for Real-Time Systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, 1994.

[118] H. Jiang. *Distributed Systems of Simple Interacting Agents*. PhD thesis, Yale University, 2007.

[119] H. Jiang. Personal Communications, 2008.

[120] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International(UK) Ltd.

[121] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0. http://www.oasis-open.org/specs/#wsbpelv2.0, Apr 2007.

[122] M. B. Josephs. A State-based Approach to Communicating Processes. *Distributed Computing*, V3(1):9–18, March 1988.

[123] R. Kazhamiakin, P. K. Pandya, and M. Pistore. Representation, Verification, and Computation of Timed Properties in Web. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2006)*, pages 497–504. IEEE Computer Society, 2006.

[124] Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model Checking with Strong Fairness. *Formal Methods and System Design*, 28(1):57–84, 2006.

[125] S. Kundu, S. Lerner, and R. Gupta. Automated Refinement Checking of Concurrent Systems. In *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2007)*, pages 318–325, Piscataway, NJ, USA, 2007. IEEE Press.

[126] O. Kupferman and M. Y. Vardi. An Automata-Theoretic Approach to Reasoning about Infinite-State Systems. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 36–52. Springer, 2000.

[127] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton university press, 1995.

[128] M. Z. Kwiatkowska. Event Fairness and Non-interleaving Concurrency. *Formal Aspects of Computing*, 1(3):213–228, 1989.

[129] A. L. Lafuente. Simplified Distributed LTL Model Checking by Localizing Cycles. Technical report, Institute of Computer Science, Albert-Ludwings Universität Freiburg, 2002.

[130] L. M. Lai and P. Watson. A Case Study in Timed CSP: The Railroad Crossing Problem. In *Proceedings of the International Workshop of Hybrid and Real-Time Systems (HART 1997)*, pages 69–74, 1997.

[131] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[132] L. Lamport. Fairness and Hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.

[133] L. Lamport. Real-Time Model Checking Is Really Simple. In *Proceedings of the 14th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, pages 162–175, 2005.

[134] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing Real-time Embedded Software using UPPAAL-TRON: an Industrial Case Study. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2005)*, pages 299–306, 2005.

[135] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[136] K. G. Larsen and W. Yi. Time-abstracted Bisimulation: Implicit Specifications and Decidability. *Information and Computation*, 134(2):75–101, 1997.

[137] T. Latvala and K. Heljanko. Coping with Strong Fairness. *Fundamenta Informaticae*, 43(1-4):175–193, 2000.

[138] D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming (ICALP 1981)*, volume 115 of *LNCS*, pages 264–277, 1981.

[139] M. Leuschel and M. J. Butler. Automatic Refinement Checking for B. In *Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, pages 345–359, 2005.

[140] M. Leuschel and T. Massart. Infinite State Model Checking by Abstract Interpretation and Program Specialisation. In *Proceedings of the 9th International Workshop on Logic Programming Synthesis and Transformation*, pages 62–81, 1999.

[141] M. Lindahl, P. Pettersson, and Y. Wang. Formal Design and Analysis of a Gearbox Controller. *International Journal on Software Tools for Technlogy Transfer (STTT)*, 3(3):353–368, 2001.

[142] S. Y. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engeering*, 24(1):24–45, 1998.

[143] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model Checking Lineariability via Refinement. In *Proceedings of the 16th International Symposium on Formal Methods (FM 2009)*, 2009. Accepted.

[144] Y. Liu, J. Pang, J. Sun, and J. Zhao. Efficient Verification of Population Ring Protocols in PAT. In *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009)*, pages 81–89, 2009.

[145] Y. Liu and J. Sun. Algorithmic Design Using Object-Z for Twig XML Queries Evaluation. *Electronic Notes in Theoretical Computer Science*, 151(2):107–124, 2006.

[146] Y. Liu, J. Sun, and J. S. Dong. An Analyzer for Extended Compositional Process Algebras. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008) Companion Volume*, pages 919–920. ACM, 2008.

[147] Y. Liu, J. Sun, and J. S. Dong. Scalable Multi-Core Model Checking Fairness Enhanced Systems. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM 2009)*, Dec 2009. Accepted.

[148] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[149] N. A. Lynch and F. W. Vaandrager. Action Transducers and Timed Automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[150] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 95–104, Kyoto, Japan, 1998.

[151] B. P. Mahony and J. S. Dong. Network Topology and a Case Study in TCOZ. In *Proceedings of the 11th International Conference of Z Users (ZUM 1998)*, volume 1493 of *LNCS*, pages 308–327, 1998.

[152] B. P. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[153] R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap Decomposition for Concurrent Shape Analysis. In *Proceedings of the 15th International Static Analysis Symposium (SAS 2008)*, pages 363–377. Springer, 2008.

[154] M. M. Michael and M. L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.

[155] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[156] J. Misra and W. Cook. Computation Orchestration: A Basis for Wide-area Computing. *Software and Systems Modeling (SoSyM)*, 6(1):83–110, March 2007.

[157] L. Momtahan, A. Martin, and A. W. Roscoe. A Taxonomy of Web Services Using CSP. In *Proceedings of the International Workshop on Web Languages and Formal Methods (WLFM 2005)*, pages 71–87, 2005.

[158] M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 362–371. ACM, 2008.

[159] X. Nicollin and J. Sifakis. The Algebra of Timed Processes, ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[160] J. Ouaknine and J. Worrell. On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 54–63, 2004.

[161] J. Pang, Z. Q. Luo, and Y. X. Deng. On Automatic Verification of Self-stabilizing Population Protocols. In *Proceedings of the 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008)*, pages 185–192. IEEE, 2008.

[162] A. Parashkevov and J. Yantchev. ARC - a Tool for Efficient Refinement and Equivalence Checking for CSP. In *Proceedings of the IEEE International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 1996)*, pages 68–75, 1996.

[163] D. Peled. All from One, One for All: on Model Checking Using Representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, volume 697 of *LNCS*, pages 409–423, 1993.

[164] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV 1994)*, pages 377–390, 1994.

[165] C. A. Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of IFIP Congress*, pages 386–390, 1963.

[166] A. Pnueli and Y. Sa'ar. All You Need Is Compassion. In *Proceedings of the 9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2008)*, volume 4905 of *LNCS*, pages 233–247, 2008.

[167] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, volume 2204 of *LNCS*, pages 107–122, 2002.

[168] F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

[169] G. Pu, J. Shi, Z. Wang, L. Jin, J. Liu, and J. He. The Validation and Verification of WSCDL. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007)*, pages 81–88. IEEE Computer Society, 2007.

[170] A. Puhakka and A. Valmari. Liveness and Fairness in Process-Algebraic Verification. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, pages 202–217, 2001.

[171] S. C. Qin, J. S. Dong, and W.-N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *Proceedings of International Symposium of Formal Methods Europe (FME 2003)*, pages 321–340, 2003.

[172] Z. Y. Qiu, X. P. Zhao, C. Cai, and H. L. Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International World Wide Web Conference (WWW 2007)*, pages 973–982, 2007.

[173] J.-P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems - A Temporal Logic to Deal with Fairness. *Acta Informaticae*, 19:195–220, 1983.

[174] G. M. Reed and A. W. Roscoe. A Timed Model for Communicating Sequential Processes. In *Proceedings of the 13th Colloquium on Automata, Languages and Programming (ICALP 1986)*, volume 226 of *LNCS*, pages 314–323. Springer, 1986.

[175] A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.

[176] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[177] A. W. Roscoe. Compiling Shared Variable Programs into CSP. In *Proceedings of PROGRESS workshop 2001*, 2001.

[178] A. W. Roscoe. On the Expressive Power of CSP Refinement. *Formal Aspects of Computing*, 17(2):93–112, 2005.

[179] A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *Proceedings of the 1st International Conference of Tools and Algorithms for Construction and Analysis of Systems (TACAS 1995)*, pages 133–152, 1995.

[180] P. Y. A. Ryan and S. A. Schneider. An Attack on a Recursive Authentication Protocol. A Cautionary Tale. *Information Processing Letters*, 65(1):7–10, 1998.

[181] S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116(2):193–213, 1995.

[182] S. Schneider. *Concurrent and Real-time Systems: the CSP Approach*. John Wiley and Sons, 2000.

[183] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and Practice. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 640–675, London, UK, 1992. Springer-Verlag.

[184] S. Schneider and H. Treharne. Communicating B Machines. In *Proceedings of the 2nd International Conference of B and Z Users (ZB 2002)*, pages 416–435. Springer, 2002.

[185] S. A. Schneider and R. Delicata. Verifying Security Protocols: An Application of CSP. In *25 Years Communicating Sequential Processes*, pages 243–263, 2004.

[186] C. H. Shann, T. L. Huang, and C. Chen. A Practical Nonblocking Queue Algorithm Using Compare-and-Swap. In *Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS 2000)*, pages 470–475. IEEE, 2000.

[187] J. Sifakis. The Compositional Specification of Timed Systems - A Tutorial. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 2–7. Springer, 1999.

[188] A. P. Sistla and E. Clarke. The Complexity of Propositional Temporal Logics. *The Journal of ACM*, 32:733–749, 1986.

[189] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[190] G. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.

[191] O. Strichman. Accelerating Bounded Model Checking of Safety Properties. *Formal Methods in System Design*, 24(1):5–24, 2004.

[192] J. Sun and J. S. Dong. Design Synthesis from Interaction and State-Based Specifications. *IEEE Transactions on Software Engineering*, 32(6):349–364, 2006.

[193] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, pages 307–322. Springer, 2008.

[194] J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009)*, pages 127–135, 2009.

[195] J. Sun, Y. Liu, J. S. Dong, and J. Pang. A Unified Framework for Model Checking under Fairness. Submitted for review.

[196] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Towards a Toolkit for Flexible and Efficient Verification under Fairness. Technical Report TRB2/09, National Univ. of Singapore, Dec 2008. http://www.comp.nus.edu.sg/~pat/report.ps.

[197] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *Proceedings of the 21th International Conference on Computer Aided Verification (CAV 2009)*, pages 702–708, Grenoble, France, June 2009.

[198] J. Sun, Y. Liu, J. S. Dong, and G. G. Pu. Model-based Methods for Linking Web Service Choreography and Orchestration. Submitted for review.

[199] J. Sun, Y. Liu, J. S. Dong, and J. Sun. Bounded Model Checking of Compositional Processes. In *Proceedings of the Second IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008)*, pages 23–30. IEEE Computer Society, 2008.

[200] J. Sun, Y. Liu, J. S. Dong, and J. Sun. Compositional Encoding for Bounded Model Checking. *Frontiers of Computer Science in China*, 2(4):368–379, November 2008.

[201] J. Sun, Y. Liu, J. S. Dong, F. Wang, L. A. Tuan, and M. Zheng. Verifying Safety Critical Compositional Real-time Systems by Refinement Checking. Submitted for review.

[202] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM 2008)*, pages 318–337. Springer, Oct 2008.

[203] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Verifying Stateful Timed CSP using Implicit Clocks and Zone Abstraction. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM 2009)*, Dec 2009. Accepted.

[204] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair Model Checking of Parameterized Systems. In *Proceedings of the 16th International Symposium on Formal Methods (FM 2009)*, 2009. Accepted.

[205] K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In *ICFEM*, pages 283–292, 1997.

[206] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 2:146–160, 1972.

[207] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying Abstractions of Timed Systems. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR 1996)*, volume 1119 of *LNCS*, pages 546–562, 1996.

[208] H. Tej and B. Wolff. A Corrected Failure-Divergence Model for CSP in Isabelle/HOL. In *Proceedings of the 4th International Symposium on Formal Methods (FM 1997)*, 1997.

[209] W. Thomas. Automata on Infinite Objects. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 133–191, 1990.

[210] R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[211] V. Vafeiadis. Shape-Value Abstraction for Verifying Linearizability. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, pages 335–348. Springer, 2009.

[212] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving Correctness of Highly-concurrent Linearisable Objects. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006)*, pages 129–136. ACM, 2006.

[213] A. Valmari. A Stubborn Attack On State Explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV 1990)*, pages 156–165, 1991.

[214] A. Valmari. Stubborn Set Methods for Process Algebras. In *Proceedings of the Workshop on Parital Order Methods in Verification (PMIV 1996)*, pages 213–231, 1996.

[215] M. Vechev and E. Yahav. Deriving Linearizable Fine-grained Concurrent Objects. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 125–135. ACM, 2008.

[216] H. Völzer, D. Varacca, and E. Kindler. Defining Fairness. In *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR 2005)*, pages 458–472. Springer, 2005.

[217] F. Wang, R. Wu, and G. Huang. Verifying Timed and Linear Hybrid Rule-Systems with RED. In *Proceedings of the 17st International Conference on Software Engineering & Knowledge Engineering (SEKE 2005)*, pages 448–454, 2005.

[218] L. Wang and S. Stoller. Static Analysis of Atomicity for Programs with Non-blocking Synchronization. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2005)*, pages 61–71. ACM, 2005.

[219] H. Wehrheim. Partial Order Reductions for Failures Refinement. *Electronic Notes in Theoretical Computer Science*, 27, 1999.

[220] J. Woodcock. Formal Specification of the Lift Problem. In M. Harandi, editor, *Proceedings of the 4th IEEE International Workshop on Software Specification and Design (IWSSD 1987)*. IEEE Press, 1987.

[221] J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *Proceedings of the 2nd International Conference of B and Z Users (ZB 2002)*, pages 184–203. Springer, 2002.

[222] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International, 1996.

[223] W. Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *Proceedings of the 18th Colloquium on Automata, Languages and Programming (ICALP 1991)*, volume 510 of *LNCS*, pages 217–228. Springer, 1991.

[224] W. Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-time Communicating Systems by Constraint-Solving. In *Proceedings of the 14th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 1994)*, pages 243–258. Chapman & Hall, 1994.

[225] S. J. Zhang, Y. Liu, J. Sun, J. S. Dong, W. Chen, and Y. A. Liu. Formal Verification of Scalable NonZero Indicators. In *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE 2009)*, pages 406–411, 2009.

[226] W. Zhang. SAT-Based Verification of LTL Formulas. In *Proceedings of the 11th International Workshop FMICS 2006*, pages 277–292, 2006.

Appendix A

Operational Semantics of CSP#

The following are firing rules associated with process constructs other than those discussed in Section 3.1.2. Let $e \in \Sigma$, $e_\tau \in \Sigma \cup \{\tau\}$, $x \in \Sigma \cup \{\checkmark\}$ and $* \in \Sigma \cup \{\tau, \checkmark\}$.

$$\frac{(V, P) \xrightarrow{e} (V', P'), e \in X}{(V, P \setminus X) \xrightarrow{\tau} (V', P')} [\ hide1\ ] \qquad \frac{(V, P) \xrightarrow{x} (V', P'), x \notin X}{(V, P \setminus X) \xrightarrow{x} (V', P' \setminus X)} [\ hide2\ ]$$

$$\frac{(V, P) \xrightarrow{e_\tau} (V', P')}{(V, P;\ Q) \xrightarrow{e_\tau} (V', P';\ Q)} [\ seq1\ ] \qquad \frac{(V, P) \xrightarrow{\checkmark} (V', P')}{(V, P;\ Q) \xrightarrow{\tau} (V', Q)} [\ seq2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \Box Q) \xrightarrow{x} (V', P')} [\ ch1\ ] \qquad \frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \Box Q) \xrightarrow{x} (V', Q')} [\ ch2\ ]$$

$$\frac{(V, P) \xrightarrow{\tau} (V', P')}{(V, P \Box Q) \xrightarrow{x} (V', P' \Box Q)} [\ ch3\ ] \qquad \frac{(V, Q) \xrightarrow{\tau} (V', Q')}{(V, P \Box Q) \xrightarrow{\tau} (V', P \Box Q')} [\ ch4\ ]$$

$$\frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, P)} [\ non1\ ] \qquad \frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, Q)} [\ non2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \mid\mid\mid Q) \xrightarrow{x} (V', P' \mid\mid\mid Q)} [\ int1\ ] \qquad \frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \mid\mid\mid Q) \xrightarrow{x} (V', P \mid\mid\mid Q')} [\ int2\ ]$$

$$\frac{(V, P) \xrightarrow{\checkmark} (V', P'), (V, Q) \xrightarrow{\checkmark} (V', Q')}{(V, P \mathbin{|||} Q) \xrightarrow{\checkmark} (V', P' \mathbin{|||} Q')} \; [\ int3\ ]$$

$$\frac{(V, P) \xrightarrow{*} (V', P')}{(V, P \bigtriangleup Q) \xrightarrow{*} (V', P' \bigtriangleup Q)} \; [\ inter1\ ] \qquad \frac{(V, Q) \xrightarrow{e} (V', Q')}{(V, P \bigtriangleup Q) \xrightarrow{e} (V', Q')} \; [\ inter2\ ]$$

$$\frac{(V, Q) \xrightarrow{\tau} (V', Q')}{(V, P \bigtriangleup Q) \xrightarrow{\tau} (V', P \bigtriangleup Q')} \; [\ inter3\ ]$$

Appendix B

CSP# Models of Population Protocols

```
#define N 3; #define C 3;
var color[N]; var precolor[N]; var succolor[N];
Interaction(u, v) = if (color[v] == precolor[u] ∧ color[v] ≠ succolor[u]){
                            act1.u.v{succolor[v] = mycolor[u]} → Interaction(u, v)
                        } else if (color[v] == succolor[u] ∧ color[v] ≠ precolor[u]){
                            act2.u.v{precolor[v] = color[u]; } → Interaction(u, v)
                        } else {
                            act3.u.v{precolor[u] = color[v]; succolor[v] = color[u]}
                             → Interaction(u, v)
                        };
Init() = ...
OrientingUndirected() = Init(); ||| x : {0..N − 1}@(Interaction(x, (x + 1)%N)
                             ||| Interaction((x + 1)%N, x));
#define property1 (x : {0..N − 1}@precolor[x] ≠ succolor[x]));
#define property2 (...);
#assert OrientingUndirected() ⊨ ◇□property1;
#assert OrientingUndirected() ⊨ ◇□property2;
```

Figure B.1: CSP# model for orienting undirected ring protocol

```
#define $N$ $3$;
var $leader[N]$; var $label[N]$; var $probe[N]$; var $phase[N]$; var $bullet[N]$;
$Interact(u, v) =$
    $[label[u] == label[v] \wedge probe[u] == 1 \wedge phase[u] == 0]$
        $act1.u.v\{leader[u] = 1;\ probe[u] = 0;\ bullet[v] = 0;\ phase[u] = 1;$
        $probe[v] = 1;\} \rightarrow Interact(u, v)$
    $\Box\ [label[u] == label[v] \wedge probe[u] == 1 \wedge phase[u] == 1 \wedge probe[v] == 0]$
        $act2.u.v\{leader[u] = 1;\ probe[u] = 0;\ bullet[v] = 0;$
        $label[v] = 1 - label[v];\ phase[v] = 0;\} \rightarrow Interact(u, v)$
    $\Box$ ...
    $\Box\ [label[u] \neq label[v] \wedge leader[v] == 0 \wedge bullet[v] == 1 \wedge probe[v] == 0]$
        $act11.u.v\{bullet[u] = 1;\ bulllet[v] = 0;\} \rightarrow Interact(u, v)$
$Init() = ...$
$LeaderElection() = Init();\ (|||\ x : 0..N - 1@Interaction(x, (x + 1)\%N));$
#define $leaderelection$ $(leader[0] + leader[1] + leader[2] == 1);$
#assert $LeaderElection() \vDash \diamond\Box leaderelection;$
```

Figure B.2: CSP# model for leader election protocol in odd rings

```
#define $N$ $3$;
var $leader[N]$; var $label[N]$; var $token[N]$;
$Rule1(u, v) = [!leader[u] \wedge leader[v] \wedge label[u] == label[v]]$
                    $(rule1.u.v\{token[u] = 0;\ token[v] = 1;\ label[v] = 1 - label[u];\}$
                    $\rightarrow Rule1(u, v));$
$Rule2(u, v) = [!leader[v] \wedge label[u] \neq label[v]]$
                    $(rule2.u.v\{token[u] = 0;\ token[v] = 1;\ label[v] = label[u];\}$
                    $\rightarrow Rule2(u, v));$
$Init() = ...$
$TokenCirculation() = Init();\ (|||\ x : 0..N - 1@(Rule1(x, (x + 1)\%N)$
                        $|||\ (Rule2(x, (x + 1)\%N));$
#define $onetoken$ $(token[0] + token[1] + token[2] == 1);$
#assert $TokenCirculation() \vDash \diamond\Box onetoken;$
```

Figure B.3: CSP# model for token circulation protocol

Appendix C

Operational Semantics of Abstract Real-Time System

The following are abstract firing rules associated with process constructs other than those discussed in Section 9.1.2. Let $e \in \Sigma$ and $x \in \Sigma \cup \{\checkmark\}$.

$$\frac{}{(V, \mathit{Skip}, D) \overset{\checkmark}{\hookrightarrow} (V, \mathit{Stop}, D^{\uparrow})}\ [\ \mathit{aki}\ ] \qquad \frac{V \vDash b}{(V, [b]P, D) \overset{\tau}{\hookrightarrow} (V, P, D^{\uparrow})}\ [\ \mathit{agu}\ ]$$

$$\frac{}{(V, e\{\mathit{prg}\} \rightarrow P, D) \overset{e}{\hookrightarrow} (\mathit{prg}(V), P, D^{\uparrow})}\ [\ \mathit{aev}\ ]$$

$$\frac{(V, P, D) \overset{x}{\hookrightarrow} (V', P', D')}{(V, P \mid Q, D) \overset{x}{\hookrightarrow} (V', P', D' \wedge \iota(V, Q, D))}\ [\ \mathit{aex}1\ ]$$

$$\frac{(V, Q, D) \overset{x}{\hookrightarrow} (V', Q', D)}{(V, P \mid Q, D) \overset{x}{\hookrightarrow} (V', Q', D' \wedge \iota(V, P, D))}\ [\ \mathit{aex}2\ ]$$

$$\frac{(V, P, D) \overset{e}{\hookrightarrow} (V', P', D'), e \notin \alpha Q}{(V, P \parallel Q, D) \overset{e}{\hookrightarrow} (V', P' \parallel Q, D' \wedge \iota(V, Q, D))}\ [\ \mathit{apa}1\ ]$$

$$\frac{(V, Q, D) \overset{e}{\hookrightarrow} (V', Q', D'), e \notin \alpha P}{(V, P \parallel Q, D) \overset{e}{\hookrightarrow} (V', P \parallel Q', D' \wedge \iota(V, P, D))} \; [\; apa2 \;]$$

$$\frac{(V, P, D) \overset{e}{\hookrightarrow} (V, P', D'), (V, Q, D) \overset{e}{\hookrightarrow} (V, Q', D''), e \in \alpha P \cap \alpha Q}{(V, P \parallel Q, D) \overset{e}{\hookrightarrow} (V, P' \parallel Q', D' \wedge D'')} \; [\; apa3 \;]$$

$$\frac{(V, P, D) \overset{x}{\hookrightarrow} (V', P', D'), x \neq \checkmark}{(V, P;\ Q, D) \overset{x}{\hookrightarrow} (V', P';\ Q, D' \wedge (\checkmark \notin init(V, P) \vee D))} \; [\; ase1 \;]$$

$$\frac{(V, P, D) \overset{\checkmark}{\hookrightarrow} (V', P', D')}{(V, P;\ Q, D) \overset{\tau}{\hookrightarrow} (V, Q, D \wedge D')} \qquad \frac{(V, P, D) \overset{x}{\hookrightarrow} (V', P', D'), Q \mathrel{\widehat{=}} P}{(V, Q, D) \overset{x}{\hookrightarrow} (V', P', D')}$$

Appendix D

PAT History

PAT project started from July, 2007 in National University of Singapore. PAT was named Libra originally for its emphasis on the fairness model checking. Soon, it was renamed to PAT because of the conflict with Microsoft search engine. After finishing LTL verification under fairness assumption, we looked at the bounded model checking, which resulted a bounded model checker for CSP. However, we found that bounded model checking was difficult to apply for variables. Since we were expending the modeling languages quickly, we decided to stop the development of the bounded model checker. At the same time, the on-the-fly refinement checking algorithm was quickly implemented in PAT by following the ideas in FDR.

In year 2008, we started to look for applications of the model checking algorithms developed. Our first application is to apply fairness model checking on population protocols, which gave a successful result with a bug discovered. The second application was to verify linearizability. After several attempts, we found refinement checking can be applied to it directly. In ICSE 2008, we successfully demonstrated PAT as an analysis toolkit for CSP [146]. After that we started the development of Web Service module with an architecture redesign.

Starting from 2009, we looked at the real-time verification since there is very few tool support for Timed CSP. RTS module was completely finished in September 2009. We also looked reduction and optimizations for fairness model checking, for example the multi-core support and process counter abstraction.

Currently, PAT version 2.7.0 is public available at our web site [1]. We keep in working on the improvements of every aspect of the system. Our aim is to develop an easy-to-user, powerful and efficient analysis toolkit for concurrent systems.

PAT Users

As a research tool, PAT has been used by quite a number of institutions for various purposes. Till now, there are more than 400 downloads from 93 organizations in 23 countries and regions.

PAT has also been used for teaching two courses (CS4211 Advanced Software Engineering and CS5232 Formal Methods) in National University of Singapore. More than 300 students are using it as an educational tool for learning process algebra and model checking. PAT's development involved with collaboration of Microsoft Research Asia. We worked with theory group in Microsoft Research Asia to model checking distributed algorithms with successful results. PAT has been used as a model checker for web service choreography verification at Peking University in China.

Maturity and Robustness

After two years' development, PAT has come to a stable stage with solid testing. The toolkit has developed as a self-contained application with user friendly design. The editing functions are complete. The detailed debugging message will be popped up for syntax errors. A rich set of simulation and model checking options are also provided for different requirements.

Currently the system contains 1213 classes with more than 110K LOC. We have conducted heavy testing to guarantee the correctness. Internally, we have a complete set of unit testing for the whole system. For the black-box testing, PAT has been used to model hundreds of systems with different properties. Currently, there are 50 built-in examples in PAT ranging from classical concurrent algorithms, math puzzles, real world problems (e.g., pace maker), population protocols, security protocols and recently published distributed algorithms (e.g., mailbox problem).

For scalability, the model checker in PAT is capable of handling tens of millions states within several hours, which is compatible to SPIN. The simulator can display up to several hundreds of states within the readability. The whole system has gone through syntax changes twice and once system redesign. The Object-Oriented design is incorporated maximally, which makes the language and properties extension easily and independently.