Algorithmic Verification of Component-based Systems

THÈSE N^{O} 7753 (2017)

PRÉSENTÉE LE 6 JUIN 2017

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ANALYSE ET DE RAISONNEMENT AUTOMATISÉS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Qiang WANG

acceptée sur proposition du jury:

Prof. A. Lenstra, président du jury
Prof. V. Kuncak, Dr S. Bliudze, directeurs de thèse
Dr A. Cimatti, rapporteur
Prof. S. Bensalem, rapporteur
Prof. M. Odersky, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

The best way out is always through.
— Robert Frost

To my family…

Acknowledgements

I would like to thank my advisor Prof.Joseph Sifakis, for giving me the opportunity to join his research group in EPFL, and also his guidances and support on my graduate study. This work would not have been possible without his technical and moral supports. He introduced me to the field of formal methods, and opened doors for me in the research community. His enthusiastic supervision will long be a source of encouragement to me. I would also like to thank Dr.Simon Bliudze, my co-advisor, for his countless explanations and everything he did to guide me through the PhD. I am also grateful to Prof.Viktor Kuncak, for hosting me in the last few months and giving me the chance to finish my PhD. Many thanks to secretary Mrs Ariane Staudenmann and Mrs Sylvie Jankow for the organizations of my study. Thanks for being patient with my stubbornness!

I want to express my deepest gratitude to Dr. Alessandro Cimatti, who offered me the opportunity to study in his group for a few months. It was a great pleasure for me to work closely with Dr.Alessandro Cimatti, Dr.Marco Roveri and Dr.Sergio Mover. I have learned a lot from them and I could not have this work done without their help. Thanks to Alessandro again for taking time to review this dissertation. I also want to thank Prof.Helmut Veith, Dr.Igor Konnov and Dr.Tomer Kotek for sharing their thoughts and insights on formal verification. It was a wonderful experience and a very productive collaboration to work with them. I am also grateful to the other jury members Prof.Martin Odersky, Prof.Arjen Lenstra, Prof.Saddek Bensalem for taking time to serve my defense and review this dissertation.

I would like to take this opportunity to thank my family for their constant moral support. I want to give my special appreciations to Dr.Tongkai Zhao for providing me the opportunity to study abroad, to Prof.Chaojing Tang, Dr.Chao Feng, Dr.Xingtong Liu for helping me with the business in China, and to Prof.Mingsheng Ying for his kind recommandations. Last but not least, many thanks to the talented students in Prof.Sifakis's group Eduard Baranov, Anastasia Mavridou, Alina Zolotukhina, Stefanos Skalistis, Wajeb Saab, and my friends Lin Yuan, Xing Bi, Fengyun Liu, Weitian Zhao, Bin Zhang, Zhicong Huang, Shiming Ou, Mengjun Li, Hua Gao, Samantha Meylan, Arthur Meylan, Marguerite Delcourt, Yanguang Yang and many others with whom I have shared the good moments over the years. Thanks to Benjamin Wesolowski for proofreading the French abstract, and Sebastian Stich, Philippe Heer for proofreading the German abstract. To whom I may have forgotten to mention here, I owe you a sincere apology and thank you!

Lausanne, 27 March 2017 Wang Qiang

Abstract

This dissertation discusses algorithmic verification techniques for concurrent component-based systems modeled in the Behavior-Interaction-Priority (BIP) framework with both bounded and unbounded concurrency.

BIP is a component framework for mixed software/hardware system design in a rigorous and correct-by-construction manner. System design is defined as a formal, accountable and coherent process for deriving trustworthy and optimised implementations from high-level system models and the corresponding execution platform descriptions. The essential properties of a system model are guaranteed at the earliest possible design phase, and a correct implementation is then automatically generated from the validated high-level system model through a sequence of property preserving model transformations, which progressively refines the model with details specific to the target execution platform.

BIP comes with a well-defined formal modeling language and a toolchain to support the rigorous system design. The BIP modeling language offers a three-layered modeling mechanism, i.e. Behavior, Interaction, and Priority for constructing complex system behavior and architectures. Behavior is characterized by a set of components, which are formally defined as automata extended with local data variables. Interaction specifies the multiparty synchronization of components, among which data transfer may take place. Priority can be used to schedule the interactions or resolve conflicts when several interactions are enabled simultaneously. The key principle of this three-layered modeling mechanism is the separation of concerns, i.e. system behavior is captured by a set of components, and system coordination is modeled by interactions and priorities.

In BIP, algorithmic verification techniques are applied to ensure the essential safety properties of the system designs. The first major contribution of this dissertation is an efficient safety verification technique for BIP system models, where the number of participating components is fixed and the data variables can have infinite domains, but their manipulation is limited to linear arithmetic. The key insight of our technique is to take advantage of the structure features of the BIP system and handle the computation in the components and coordination between the components in the verification separately. On the computation level, we apply the state-of-the-art counterexample abstraction techniques to reason about the behavior of components and explore all the possible reachable states ; while on the coordination level, we exploit both partial order techniques and symmetry reduction techniques to handle the state space explosion problem due to concurrency, and reduce the redundant interleavings of concurrent interactions. We have implemented the proposed techniques in a prototype tool

and carried out a comprehensive performance evaluation on a set of BIP system models.
The second major contribution of this dissertation is a uniform design and verification framework for parameterized systems based on BIP. Parameterized systems are systems consisting of homogeneous processes, and the parameter indicates the number of such processes in the system. A parameterized system, therefore, describes an infinite family of systems, where instances of the family can be obtained by fixing the value of the parameter. Verification of correctness of such systems amounts to verifying the correctness of every member of the infinite family described by the system.
First of all, we propose the first order interaction logic (FOIL) as a formal language for parameterized system architectures and communication primitives. This logic is powerful enough to express architectures found in distributed systems, including the classical architectures : token-passing rings, rendezvous cliques, broadcast cliques, rendezvous stars. We also identify a fragment of FOIL that is well-suited for the specification of parameterized BIP systems and prove its decidability. Second, we provide a framework for the integration of mathematical models from the parameterized model checking literature in an automated way. With our new framework, we close the gap between the mathematical formalisms and algorithms from the parameterized verification research and the practice of parameterized verification, which is usually done by engineers who are not familiar with the details of the literature. Finally, we provide a preliminary prototype implementation of the proposed framework. Our prototype tool takes a parameterized BIP design as its input and identifies the classical model checking results which can be applies to this BIP design.

Keywords : Component-based design, Concurrent system, Model checking, Algorithmic verification, Parameterized verification, Predicate abstraction, Partial order reduction, Symmetry reduction, Well-structured transition system

Zusammenfassung

Diese Dissertation diskutiert algorithmische Verifikationstechniken für parallel laufende komponenten basierte Systeme, die im BIP-Framework (Behavior, Interaction, Priority) mit sowohl begrenzter als auch unbegrenzter Parallelität modelliert sind.

BIP ist ein Komponenten framework für gemischte Software/Hardware Systementwicklung welches mit einer rigorosen konstruktionsbegleitenden Korrektur ausgestattet ist. Systementwicklung ist definiert als ein formaler, rechenschaftspflichtiger und kohärenter Prozess zur Ableitung vertrauenswürdiger und optimierter Implementierungen aus hochrangigen Systemmodellen und den entsprechenden Ausführungsplattformbeschreibungen. Die wesentlichen Eigenschaften eines Systemmodells werden in der frühestmöglichen Entwicklungsphase garantiert und eine korrekte Implementierung erfolgt dann automatisch aus dem zertifizierten hochrangigem Systemmodell durch eine Sequenz von zielplattformspezifischen Transformationen, welche die Modelligenschaften bewahren und das Modell schrittweise verfeinern.

BIP ist mit einer klar definierten formalen Modelliersprache und einer Werkzeugkette zur Unterstützung der rigorosen Systementwicklung ausgestattet. Die BIP-Modellierungssprache bietet einen dreischichtigen Modellierungsmechanismus, d.h. Verhalten, Interaktion und Priorität, für den Aufbau komplexer Systemverhalten und Architekturen. Das Verhalten zeichnet sich durch einen Satz von Komponenten aus, welche formal als Automaten, erweitert mit linearer Arithmetik, definiert sind. Interaktion gibt die Multi-party Synchronisation von datenübertragender Komponenten an. Priorität kann verwendet werden um die Interaktionen zu planen oder Konflikte zu lösen, wenn mehrere Interaktionen gleichzeitig aktiviert werden. Das Hauptprinzip dieses dreischichtigen Modellierungsmechanismus ist die Trennung von Aufgaben, d.h. das Systemverhalten wird durch einen Satz von Komponenten erfasst und die Systemkoordination wird durch Interaktionen und Prioritäten modelliert.

Im BIP werden algorithmische Verifikationstechniken angewendet um die wesentlichen Sicherheitseigenschaften der Systementwicklung zu gewährleisten. Der erste wesentliche Beitrag dieser Dissertation ist eine effiziente Sicherheitsüberprüfungstechnik für BIP-Systemmodelle mit einer festen Anzahl an teilnehmenden Komponenten. Die Schlüsseleigenschaft unserer Technik ist, dass sie die Struktur BIP-Systeme nutzt um die Berechnung und Koordination bei der Überprüfung separat zu behandeln. Auf der Berechnungsstufe wenden wir die State-of-the-Art-Gegenbeispiel-Abstraktionstechniken an, um das Verhalten der Komponenten zu begründen und alle möglichen erreichbaren Zustände zu untersuchen. Auf der Koordinationsebene nutzen wir Halbordnungs- und Symmetriereduktionstechniken um das Zustandsraum-Explosions-Problem aufgrund von Parallelität zu behandeln und die redun-

danten Wechselwirkungen von gleichzeitigen Interaktionen zu reduzieren. Wir haben die vorgeschlagenen Techniken in einem Prototyp-Tool implementiert und eine umfassende Performanceevaluierung auf einem Satz von BIP-Systemmodellen vollzogen.
Der zweite Hauptbeitrag dieser Dissertation ist ein einheitliches Entwicklungs- und Verifizierungs-Tool für parametrisierte BIP-Systeme. Parametrisierte Systeme sind Systeme, die aus homogenen Prozessen bestehen, wobei der Parameter die Anzahl solcher Prozesse im System angibt. Ein parametrisiertes System beschreibt daher eine unendliche Familie von Systemen, in welcher Instanzen der Familie durch Festlegung des Parameters erhalten werden. Die Überprüfung der Fehlerfreiheit solcher Systeme beläuft sich auf das Überprüfen der Fehlerfreiheit jedes Mitglieds der unendlichen Familie, die durch das System beschrieben wird.
Zunächst schlagen wir die Interaktionslogik erster Ordnung als formale Sprache für parametrisierte Systemarchitekturen und Kommunikationsprimitive vor. Diese Logik ist leistungsfähig genug um Architekturen in verteilten Systemen auszudrücken, darunter die klassischen Architekturen wie Token-Passing Ringe, Rendezvous Cliquen, Broadcast Cliquen und Rendezvous Stars. Wir identifizieren auch ein Fragment der Interaktionslogik erster Ordnung welches gut geeignet ist für die Beschreibung von parametrisierten BIP-Modellen und beweisen seine Entscheidbarkeit. Zweitens stellen wir ein Framework für die automatische Integration von mathematischen Modelle aus der parametrisierten Modellprüfungsliteratur bereit. Mit unserem neuen Framework schliessen wir die Kluft zwischen den mathematischen Formalismen und Algorithmen aus der parametrisierten Verifikationsforschung und der Praxis der parametrisierten Verifikation, die in der Regel von Ingenieuren durchgeführt wird, die mit den Details der Literatur nicht vertraut sind. Schliesslich stellen wir eine vorläufige Prototypenimplementierung des vorgeschlagenen Frameworks zur Verfügung. Unser Prototyp-Tool nimmt ein parametrisiertes BIP-Design als Eingabe und identifiziert die klassischen Modellverifikationsresultate, welche auf dieses BIP-Design angewendet werden können.

Stichwörter: Komponentenbasiertes Design, Gleichzeitiges System, Modellprüfung, Algorithmische Verifikation, Parametrierte Verifikation, Eigenschafts Abstraktion, Partielle Auftragsreduktion, Symmetrieverkleinerung, Gut strukturiertes Übergangssystem

Résumé

Cette dissertation traite des techniques de vérification algorithmique pour les systèmes concurrents basés sur les composants, modélisés dans le cadre BIP (Behavior, Interaction, Priority) avec des concurrences bornées et non bornées.

BIP est un framework de composants pour la conception rigoureuse et correcte par construction de systèmes de systèmes logiciels/matériels mixtes. La conception du système est définie comme un processus formel, responsable et cohérent pour obtenir des implémentations fiables et optimisées à partir de modèles de systèmes de haut niveau et des descriptions des plates-formes d'exécution correspondantes. Les propriétés essentielles d'un modèle de système sont garanties à la phase de conception la plus précoce possible et une implémentation correcte est ensuite générée automatiquement à partir du modèle de système de haut niveau certifié par une suite de transformations préservant les propriétés du modèle, qui affine progressivement le modèle avec des détails spécifiques à la plate-forme d'exécution cible.

BIP est livré avec un langage de modélisation formel bien défini et une chaîne d'outils pour soutenir la conception rigoureuse du système. Le langage de modélisation BIP offre un mécanisme de modélisation à trois couches pour construire des comportements et des architectures de systèmes complexes, c'est-à-dire Comportement, Interaction et Priorité. 'Comportement' est caractérisé par un ensemble de composants qui sont formellement définis comme des automates étendus par des variables de données locales. 'Interaction' spécifie la synchronisation multipartite des composants parmi lesquels le transfert de données peut avoir lieu. 'Priorité' peut être utilisée pour planifier les interactions ou résoudre les conflits lorsque plusieurs interactions sont activées simultanément. Le principe clé de ce mécanisme de modélisation à trois couches est la séparation des préoccupations, c'est-à-dire que le comportement du système est capté par un ensemble de composantes, et la coordination du système est modélisée par des interactions et des priorités.

Dans BIP, des techniques de vérification algorithmique sont appliquées pour assurer les propriétés de sécurité essentielles des conceptions du système. La première contribution majeure de cette dissertation est une technique efficace de vérification de la sécurité pour les modèles de systèmes BIP avec un nombre fixe de composants participants. L'idée principale de notre technique est de profiter des fonctionnalités du système BIP et gérer le calcul et la coordination dans la vérification séparément. Au niveau du calcul, nous appliquons les techniques d'abstraction de contre-exemple pour raisonner sur le comportement des composants et explorer tous les états accessibles possibles ; alors qu'au niveau de la coordination, nous exploitons des

techniques d'ordre partiel et de réduction de symétrie pour gérer le problème de l'explosion de l'espace des états en raison de la simultanéité, et réduisons les interrelations redondantes des interactions simultanées. Nous avons implémenté les techniques proposées dans un outil prototype et évalué l'efficacité sur un ensemble de modèles de systèmes BIP.
La deuxième contribution majeure de cette dissertation est un cadre de conception et de vérification uniforme pour les systèmes paramétrés basés sur BIP. Les systèmes paramétrés sont des systèmes consistant en des processus homogènes, où le paramètre indique le nombre de tels processus dans le système. Un système paramétré décrit donc une famille infinie de systèmes où les instances de la famille peuvent être obtenues en fixant le paramètre. La vérification de l'exactitude de ces systèmes revient à vérifier l'exactitude de chaque membre de la famille infinie décrite par le système.
Tout d'abord, nous proposons la logique d'interaction du premier ordre comme langage formel pour les architectures système paramétrées et les primitives de communication. Cette logique est assez puissante pour exprimer les architectures trouvées dans les systèmes distribués, y compris les architectures classiques : anneaux de passage de jetons, cliques de rendez-vous, cliques de diffusion, étoiles de rendez-vous. Nous identifions un fragment de la logique d'interaction du premier ordre bien adapté à la specification des modèles BIP paramétrés et prouvons sa décidabilité. Deuxièmement, nous fournissons un framework pour l'intégration de modèles mathématiques issus de la littérature sur la vérification de modèles paramétrés de manière automatisée. Avec notre nouveau framework, nous comblons l'écart entre les formalismes mathématiques et les algorithmes de la littérature sur la vérification paramétrée et la pratique de la vérification paramétrer, ce qui est généralement fait par des ingénieurs qui ne sont pas familiers avec les détails de la littérature. Enfin, nous fournissons un prototype préliminaire d'une implémentation du framework proposé. Notre outil prototype prend une conception BIP paramétrée comme entrée et identifie les résultats classiques de vérification de modèle qui peuvent s'appliquer à cette conception BIP.

Mot Clef : Conception de composants, Système concurrent, Vérification de modèles, Vérification algorithmique, Vérification paramétrée, Abstraction de prédicat, Réduction d'ordre partiel, Réduction de symétrie, Système de transition bien structuré

Contents

List of Figures

List of Tables

1 Introduction

Computer technology has become ubiquitous in daily life. The past few decades witnessed a widespread deployment of embedded systems on controlling communication, transportation and medical systems. The consequences of system failure can transcend mere annoyance and may have profound negative effects on our lives, due to our ever-increasing reliance on embedded systems, both at the personal and the organizational level (e.g. the explosion of the first launch of Ariane 5 [1]). The correctness and robustness of embedded systems are ever more important. Paradoxically, as the embedded system complexity escalates tremendously, current design techniques and tools can hardly ensure sufficiently reliable systems at affordable costs. The development of reliable and robust embedded systems remains a grand challenge in both computer science and system engineering [91, 134]. The main culprit is understood as the lack of rigorous theories and techniques for embedded system design [92].

The design of embedded systems differs radically from pure software design. Embedded system design must account not only for functional properties but also for extra-functional requirements regarding the use of execution platform resources such as time and energy. However, the systems being currently built are based on empirical approaches. Designers use different frameworks, which are only loosely coupled to build sub-systems that are subsequently composed into complete systems. The lack of an underlying unifying semantic framework and rigorous theoretical foundations makes it difficult to ensure that the implicit assumptions made during the design of sub-systems are satisfied after integration.

Further, the predictability of the system behaviour is impossible to guarantee at design time and therefore, a costly posteriori validation remains the only means for ensuring the correctness of the design with respect to the functional or extra-functional properties. Despite its high complexity, this posteriori validation usually goes from the implementation level back to model level, which cannot take advantage of the original design and in most cases, would be computationally infeasible for large implementations. Therefore, we need a new design methodology to develop correct implementations of systems in a predictable manner.

1. https://en.wikipedia.org/wiki/Ariane_5

1.1 Rigorous system design

Rigorous system design [133, 135] has been proposed in response to the grand challenge of design, manufacture and validation of large scale reliable mixed hardware and software systems (e.g. cyber-physical systems). The main objective of the rigorous system design methodology is to develop the theories, methods and tools for building reliable systems in a predictable manner.

Rigorous system design follows the component-based approach, where complex system models are constructed by assembling simple atomic components with some composition entities. Atomic components are characterized by abstractions that ignore implementation details and only describe behavior relevant to their composition, e.g. transfer functions, interfaces. Composition entities are then used to build complex compound components from atomic ones. Component-based design allows to build large-scale systems in an incremental and predictable manner.

Rigorous system design can also be understood as a formal, accountable and coherent process for deriving trustworthy and optimised implementations from high-level system models and the corresponding execution platform descriptions. The essential properties of system models are guaranteed at the earliest possible design phase using formal verification techniques. Correct implementations are then automatically generated from validated high-level system models through a sequence of property preserving model transformations, which progressively refine the model with details specific to the target execution platform.

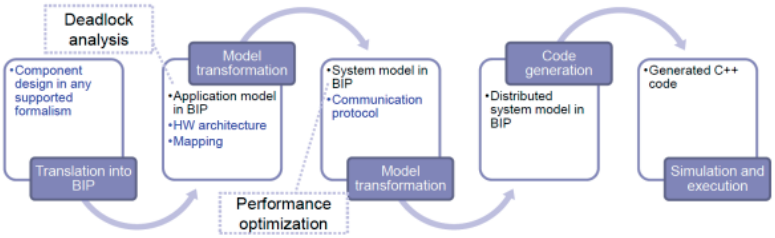

Figure 1.1 – The BIP instantiation of the rigorous system design flow

Figure 1.1 illustrates the rigorous system design flow, as it is instantiated in the Behaviour-Interaction-Priority (BIP) framework [19]. One starts by designing the application model, either directly in BIP or through a transformation from a domain specific language. The model consists of a set of atomic components and connectors. Atomic components model the application activities, from the control point of view, as finite state automaton. Each transition of an autamaton has an associated C function call, which realises functional computations and interaction with the environment (e.g. network communication protocols). This allows strict separation of concerns between control and functional behaviour. Connectors define all possible interactions between atomic components. Overall behaviour of the application is defined by the BIP operational semantics and enforced at run-time by the BIP Engine. This allows strict separation of concerns between stateful behaviour of individual components and

stateless coordination of their concurrent execution.

The individual components are verified to prove elementary safety properties, such as absence of local deadlock, and satisfaction of basic requirements. These elementary properties, serve as a basis for the proof of global properties, obtained by construction. Until recently, by-construction correctness provided by the BIP design flow illustrated in Figure 1.1 was limited to the fact that automatically generated executable code was guaranteed to satisfy the properties established on the corresponding BIP models. Correctness of the high-level application model was limited to deadlock freedom or had to be established by current model checking techniques.

The application model is then extended with additional components modeling the target platform to obtain the system model, which is used to perform platform specific analyses and the optimisation of performance through the exploration of the design space (memories, buses, mapping of software components to hardware elements etc.). Finally, the model is enriched with platform specific information (e.g. communication primitives) and, after removing components modelling hardware elements, executable code is automatically generated. Proving that the assumptions made at the modeling level to justify the separation of concerns hold, indeed, at the platform level, guarantees that all the properties established throughout the design process also hold for the generated code.

1.1.1 BIP component framework

BIP [19] is a component-based framework for rigorous system design. It addresses the following three main challenges to pursue essentials of the rigorous system design: 1) the development of a uniform modeling framework with well defined semantics for the incremental composition of heterogeneous components; 2) the development of verification methods for essential safety properties in order to guarantee the correctness of the high-level system designs, and 3) the development of automated support for component integration, validation and code generation, meeting the given requirements.

BIP comes with a well defined modeling language and an associated toolset (shown in Figure 1.3 [2]) to implement the rigorous design flow. BIP modeling language provides primitives for building composite components as the composition of simpler components, and it defines a common semantic model that can be used at all stages throughout the design flow. BIP also provides formal verification tools to check the deadlock-freedom of components, as well as advanced techniques to ensure by-construction correctness of the design. In BIP, the implementation (i.e. C++ code) can be automatically generated from the high-level system model using specific code generators by taking into account the specific execution platforms and environment.

2. This figure and the subsequent one are from the BIP website http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html.

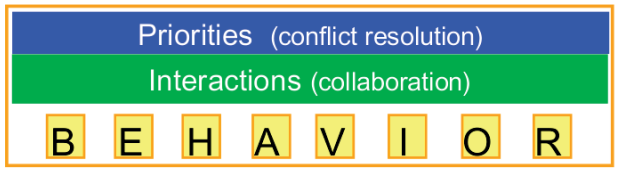

Figure 1.2 – BIP layered modeling framework

BIP language provides a three-layered modeling mechnism as shown in Figure 1.2. It allows building complex system models by coordinating three layers of modeling: 1) Behavior is descirbed by a set of components, each of which is formally specified as a finite state automaton extended with local data variables. Transition labels of the automaton are exported as ports, which are used to define the coordination between components. 2) Interaction specifies the coordination between components. An interaction is formally defined as a finite set of ports, and essentially it specifies a multiparty synchronization of the transitions, whose labels are the connected ports. 3) Priority is used to schedule the interactions or resolve conflicts when several interactions are enabled simultaneously.

BIP has clean operational semantics that describes the behavior of a composite component as the composition of the behaviors of its atomic components. A detailed introduction to BIP modeling language and its semantics is given in Chapter 2.

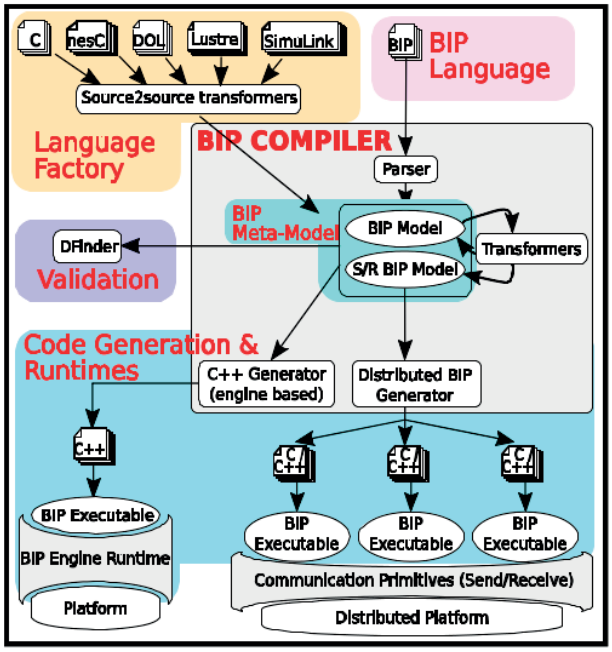

Figure 1.3 – BIP toolchain

BIP toolset includes the translators that translate various programming models, e.g. Simulink,

Lustre into BIP, and the source-to-source transformers that can transform one BIP model into another, e.g. a Send/Receive BIP model that is used in the distributed environment. It also includes compilers that generate executable code for various dedicated engines. The deadlock-freedom of the system model can be automatically checked, using the dedicated model checker DFinder. Currently, DFinder can only handle systems without data transfer among components. This limitation hampers the practical application of DFinder and of the BIP framework, since data transfer is necessary and common in the design of real-life systems.

1.1.2 The role of formal verification

Being able to check or assert correctness of the system under design using scalable formal verification techniques is an essential requirement in rigorous system design.

As opposed to the logic circuit synthesis and certified code generators for highly critical systems, such as SCADE Suite [3], most of the system and software design workflows do not combine verification of system models with guarantees that the final system satisfies the verified properties. This is due to the decoupling of modeling and verification tools. The most common workflow consists in verifying or simulating systems with dedicated modeling tools, such as MathLab/Simulink, then manually implementing the resulting solutions. As a result, the final executable code is not guaranteed to respect the verified properties, since errors may be introduced during the manual implementation phase. Another approach consists in extracting models from the implementation code for subsequent validation by existing or dedicated model checkers. This approach, on one hand, does not benefit from design-time analysis. On the other hand, the inevitable post-verification modifications of the system are costly, due to the difficulty of establishing the backward link between the automatically extracted model and the source code.

As discussed above, rigorous system design flow in BIP advocates correct-by-construction design. The system implementation, which is automatically generated executable is guaranteed to satisfy the properties established on the corresponding BIP models. While we still rely on correctness of the individual components to establish by-construction correctness of the global system model. Until recently, correctness of the high-level application model was limited to deadlock freedom. We still lack methods and tools to check general safety requirements of the high-level application model.

Differing from simulation or testing techniques, formal verification provides a rigorous way to prove or disprove that a system model meets the given requirements. The system being checked is usually modeled as a state machine and the property is specified as a formula in some temporal logic. In order to check if the system satisfies the given property, formal verification usually uses an exhaustive search procedure (either explicitly or symbolically) on the state space of the system model to check if the given property is satisfied on every reachable state. If the property is violated, a counterexample is generated as the diagnostics to

3. http://www.esterel-technologies.com/products/scade-suite/

help designers correct their designs.

Moreover, the growing power of formal verification tools makes the use of formal methods in complex embedded system design possible, as reported in [122]. Notably, formal verification has been successfully used in industry to help build reliable and secure systems. For instance, as reported in [123], formal methods have been successfully used at Amazon Web Services to solve difficult design problems and to build reliable web services. The authors reported that at Amazon seven teams have used TLA+ [4] to find subtle but serious bugs that they would not have found using other techniques, and also to devise optimized complex algorithms without sacrificing quality. Another example is the High-Assurance Cyber Military Systems(HACMS) program launched by Defense Advanced Research Projects Agency (DARPA) to create technologies to make networked embedded systems dramatically harder to attack [5]. Specifically, HACMS is pursuing a formal methods-based approach to the creation of high-assurance vehicles, where high assurance is defined to mean functionally correct and satisfying appropriate safety and security properties [67].

1.2 Evolution of formal verification

In this section, we give an overview of the development of formal verification. We postpone the elaboration of the relevant theories to Chapter 3.

In the early time, *Floyd-Hoare logic* [71, 93] and *Dijkstra's predicate transformer* [52] laid theoretical foundations of the modern (semi-)automated verification techniques. In [93], a formal framework for deducing the correctness of programs was introduced, also known as *Floyd-Hoare logic*. Given a piece of program C, and two assertions P, Q, Floyd-Hoare logic establishes the correctness proof in the form of $\{P\}\ C\ \{Q\}$ (i.e. *Hoare triple*), which intuitively means if the assertion P holds, then after the execution of C, the assertion Q must hold (if the execution of C terminates). Dijkstra's predicate transformer semantics of programs can be understood as a reformulation of Floyd-Hoare logic. It provides a way to reduce the problem of proving a Hoare triple to the problem of proving a first order formula.

In [48], a unifying framework, known as *abstract interpretation*, was proposed for automatic program analysis and verification. Since the computation of the concrete semantics of a program (i.e. the set of reachable states) is computationally infeasible in general, the idea of abstract interpretation is to map the concrete property (i.e. a set of concrete states) to an abstract property (i.e. an element in the abstract domain), and then computes the abstract semantics of the program (i.e. an over-approximation of the set of concrete reachable states). Abstract interpretation provides a disciplined way of building analysis over abstract domains.

Independently in [43] and [129], a technique, widely known as *model checking*, was proposed as an automated approach to check if a given mathematical structure satisfies a formal logic

4. http://lamport.azurewebsites.net/tla/tla.html
5. http://www.darpa.mil/program/high-assurance-cyber-military-systems

specification. In model checking, a system is formally described as a finite state machine, and the property being checked is specified as formulae in temporal logics [112]. Then model checking algorithmically enumerates all the states of the state machine to determine if it satisfies a temporal logic specification. Model checking has been successfully applied to hardware and protocol verification, which typically gives rise to relatively smaller state spaces. However, it does not apply to real programs, due to the large or even infinite-state spaces. Even for the hardware and protocol, the state space grows exponentially with the number of participating processes or components in the system, which makes automated model checking computationally infeasible. This problem is known as *state explosion problem*.

Over the last decades, a lot of effort have been made to tackle the state explosion problem, and numerous advances in model checking, abstract interpretation and constraint solving have pushed the frontiers of formal verification. We highlight the main achievements below.

Early attempts to deal with the state space explosion problem leverages on significant algorithmic advances that come in the form of symbolic techniques for succinctly representing large sets of states as formulas. In *symbolic model checking* [116], states and transition relations are symbolically represented as binary decision diagrams (BDD) that can be manipulated efficiently. While in *bounded model checking* [24], the unfolding of the transition system and the property being checked are encoded as a formula in propositional logic, whose satisfiability can be checked using SAT solvers [18]. The capability of such techniques is still limited by the underlying routines that manipulate the symbolic data structures.

One prevalent way to address the state explosion problem is to employ *abstraction* [109]. Informally speaking, abstraction aims at minimising the system model to be verified in such a way that automated verification of the abstract model becomes computationally feasible, while the desired properties are still preserved by the abstraction. Abstraction relies on the observation that in most cases the system model contains information irrelevant to the desired properties. Discarding such information reduces the verification burden dramatically. In the past decades, various abstraction techniques have been developed. In [79], *predicate abstraction* was proposed as a specific technique that over-approximates the semantics of a program and constructs a finite state abstraction of the program, where each abstract state represents possibly infinitely many concrete program states. This technique enables direct application of finite state model checking approaches to programs which have large or infinite state spaces. Since then, predicate abstraction has been widely investigated in research. In [106, 107, 138], efficient SMT based symbolic techniques for constructing predicate abstraction were studied. It has also been successfully applied in practice [50, 15, 16, 69].

Generally abstraction results in an over-approximation, which may introduce false positives. In other words, verification of the abstract system may conclude that the property is violated, which is not the case for the concrete system. The *counterexample guided abstraction refinement* (CEGAR) [39] approach offers a solution to this problem. Specifically, given a counterexample (a faulty execution) found by analyzing the finite-state abstraction, CEGAR either

confirms that the counterexample is real, i.e. it corresponds to a concrete execution, or proposes a refined abstraction in which this counterexample is eliminated. Advanced abstraction refinement techniques based on Craig interpolant have been popularized in [117, 118, 120]. Predicate abstraction and Craig interpolant abstraction refinement have been successfully applied in practice. Notably, The SLAM project [17], initiated by Microsoft Research, applied such techniques to build an industrial toolchain for verifying Windows device driver APIs, and inspired a large interests in automated software verification research.

Alternatively to the model checking and abstraction techniques, a proof rule for invariance properties of transition systems was proposed in [113], which is also known as *deductive verification* approach. In order to prove an invariance property, deductive verification approach aims at finding a stronger assertion that entails the invariance property, and then proves that the assertion is inductive, which is done by first checking that all the initial states satisfy the invariance property, and then checking that from the set of states satisfying the assertion, one cannot reach a state that violates the assertion in one step. Deductive verification provides a partial solution to the verification of invariance property, and it leaves open the questions of how to find the auxiliary predicate. More recently, a novel technique for constructing the inductive invariant incrementally, called IC3 in [32] (and also called PDR in [55]), has been proposed.

The attention of the above mentioned techniques are mainly focused on the verification problem for systems of fixed size, i.e. the number of participating processes or components is fixed. There are systems where the number of participating processes is not fixed a priori, but given as a parameter. Such systems can be widely found in the distributed context, e.g. consensus protocols, where the number of participating processes could be arbitrary large. The verification problem for such systems, known as *parameterized verification*, asks whether the desired properties hold on system of all sizes.

Though being undecidable in general [11, 136], many interesting results and decidable fragments have been obtained. One technique to prove that a fragment of the parameterized verification problem is decidable is by reduction to the *coverability problem* of *well-structured transition system*, whose decidability is known [1, 66]. Well-known well-structured transition systems include Petri net, vector addition systems. Another technique is by reduction to a finite collection of classical verification problems, known as *cutoff techniques* [60, 74, 58, 42, 10]. That is, in order to prove a property holds on system of all sizes, it is sufficient to prove the property holds for system instances up to a fixed size, i.e. cutoff. However, cutoff does not always exist. If it does exist, cutoff varies according to the property and the state machine of the process being checked.

There is also a wide range of techniques that aim at solving the parameterized verification problem automatically, instread of obtaining the decidability results. *Counter abstraction* [127, 74] is one of such widely used techniques. The idea is that, for systems consisting of an unbounded number of components, where each component is modeled as a finite state

automaton, we only keep track of the number of components in each control location, instead of tracking the exact control locations of all components. It abstracts a parameterized system into a finite state system, which can be checked by using either classical model checking techniques, or well-structured transition system based techniques [5, 6].

In [126], the authors extend the deductive verification approach to parameterized systems. The key insight is to compute a quantified inductive invariant, which can prove properties for all system sizes. The proposed way to compute such a quantified invariant is to first construct an invariant for a system of fixed size, and then generalize this invariant to the parameterized case. However, it is not guaranteed that the obtained invariant is inductive, or strong enough to prove the desired property. In [30, 3], *regular model checking* is proposed as general framework for algorithmic verification of infinite-state systems. In this approach, sets of system states are represented via regular languages and automata. Symbolic procedures based on automata manipulation can be applied to perform traversals of the infinite search space induced by a parameterized system. In [75, 76], the authors propose using array-based systems to model parameterized systems, and then apply a backward reachability analysis procedure, which symbolically computes pre-images of the set of unsafe states, and checks safety and fixpoint by using SMT solving. In [104], the authors propose the method of *network invariant* for verifying temporal properties of parameterized systems. The idea is to find a single finite state automaton (network invariant) that soundly abstracts the parallel composition of n processes. The soundness is obtained by showing a simulation relation between the network invariant and the concrete systems.

Automated verification and parameterized verification still remain very active areas of research, particularly for concurrent and distributed systems. This brief survey is biased towards the focus of this dissertation. We remark that there is a wide range of reduction techniques for algorithmic verification of concurrent systems, called *partial order reduction* [77, 124, 46, 139], that rely on the partial order semantics of concurrent systems. We postpone the elaborations to Chapter 3. For detailed explanations of each verification technique, we refer to the various books on formal verification [47, 13, 81, 28].

1.3 Challenges and contributions

The high-level contributions of this dissertation comprise new modeling framework and verification algorithms that push the frontiers of algorithmic verification of component-based systems modeled in BIP framework with both bounded and unbounded concurrency. By systems with bounded concurrency, we mean the systems that consist of a fixed number of components, and with unbounded concurrency, we mean the systems that consist of a parameterized number of components. These contributions have been published in the following articles:

1. *Formal verification of infinite-state BIP models*, Bliudze, Simon and Cimatti, Alessandro

and Jaber, Mohamad and Mover, Sergio and Roveri, Marco and Saab, Wajeb and **Wang, Qiang**, International Symposium on Automated Technology for Verification and Analysis (ATVA 2015), pages 326–343, 2015, Springer.

2. *Verification of component-based systems via predicate abstraction and simultaneous set reduction*, **Qiang, Wang** and Bliudze, Simon, International Symposium on Trustworthy Global Computing (TGC 2015), pages 147–162, 2015, Springer.
3. *Parameterized systems in BIP: design and model checking*, Konnov, Igor and Kotek, Tomer and **Wang, Qiang** and Veith, Helmut and Bliudze, Simon and Sifakis, Joseph, Proceedings of the 27th International Conference on Concurrency Theory (CONCUR 2016), pages 30–1, 2016, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
4. *Exploiting Symmetry for Efficient Verification of Infinite-State Component-Based Systems*, **Wang, Qiang**, International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA 2016), pages 246–263, 2016, Springer.

1.3.1 Algorithmic verification of systems with bounded concurrency

As we discussed in the previous section, the BIP modeling language offers a three-layered modeling mechanism, i.e. Behavior, Interaction, and Priority, for constructing complex system behavior and architectures. Behavior is characterized by a set of components, which are formally defined as automata extended with linear arithmetic. Interaction specifies the multiparty synchronization of components, among which data transfer may take place. Priority can be used to schedule the interactions or resolve conflicts when several interactions are enabled simultaneously. The key insight underlying this three-layered modeling mechanism is the principle of separation of concerns, that is, system computation is captured by a set of components, and system coordination is modeled by interaction and priority.

Our approach is inspired by the *Explicit Scheduler Symbolic Thread* (ESST) approach to efficient verification of SystemC programs [130]. In brief, we aim at decomposing the verification of infinite-state component-based systems into two levels by taking advantage of the structural features of such systems, and, thus, we palliate the state space explosion by handling the computation in the components and the coordination among components separately.

On the computation level, we exploit the state-of-the-art counterexample guided abstraction refinement technique, to deal with the sequential computations and explore the reachable states of each individual component; while on the coordination level, we resolve the redundant interleavings of concurrent interactions by applying explicit state partial order reduction techniques [77, 78, 124, 125, 46, 139, 140]. Specifically, we combine the lazy abstraction with interpolant based abstraction refinement [90, 88, 119] and the persistent set partial order reduction for BIP. We have implemented the proposed verification techniques based on the Kratos model checker [36]. We also propose two further techniques to improve the reductions of redundant interleavings. The first technique aims at exploring as many independent interactions as possible simultaneously in one step, and the second technique exploits the system

symmetry to improve the persistent set reduction.

These contributions are elaborated, respectively, in Chapters 4 and 5.

We remark that in the BIP framework, DFinder [22, 21] is a dedicated tool for invariant generation and deadlock detection. DFinder computes the system invariant in a compositional manner: it first computes a component invariant over-approximating the reachable states of each component and then computes an interaction invariant over-approximating the global reachable states. The system invariant is then the conjunction of all component invariants and the interaction invariant. Though being scalable for large system models, DFinder does not handle system models with data transfer, which hampers the practical application of DFinder and of the BIP framework, since data transfer is necessary and common in the design of real-life systems (e.g. message passing). Besides, when the inferred invariant fails to prove the property, DFinder produces a single state as the counterexample other than an execution path. By the time DFinder was developed, it was not clear how to efficiently refine the abstraction automatically from the single state. In [95], the authors present an encoding of a subset of BIP models into Horn Clauses, which are solved by the model checker ELDARICA [94]. However, the encoding does not handle data transfer on interactions. As the current stage of their work, the encoding still requires massive manual work.

An efficient instantiation of the ESST framework [130] for BIP has been presented in [26]. This ESST based technique encodes the components as preemptive threads with predefined primitive functions and utilizes a dedicated stateful BIP scheduler to orchestrate the abstract reachability analysis of the components. The scheduler interacts with components via primitive functions, and also respects BIP operational semantics. Moreover, partial order reduction techniques [78] are applied in the scheduler to reduce its state space.

1.3.2 Modeling and verifying systems with unbounded concurrency

Parameterized systems are systems consisting of homogeneous processes, where the parameter indicates the number of such processes in the system. A parameterized system, therefore, describes an infinite family of systems where instances of the family can be obtained by fixing the parameter value. Verification of the correctness of such systems amounts to verifying the correctness of every member of the infinite family described by the system. This problem is undecidable in general [136]. However, many efforts have been invested into extending of classic model checking to the parameterized case, leading to numerous parameterized model checking echniques (see [28] for a recent survey).

Unfortunately, often parameterized model checking techniques come with their own mathematical models, which makes their practical application difficult. To perform parameterized model checking, the user needs to apply deep knowledge from the literature. First, the user needs to manually inspect the parameterized models and match them with the mathematical formalisms from the relevant available parameterized verification techniques. Using the

match, the users would then apply the decidability results (if any) for the parameterized models, e.g. by computing a cutoff or translating the parameterized model into the language of a particular tool for the specific architecture.

Thus, there is a gap between the mathematical formalisms and algorithms from the parameterized verification research and the verification in practice, which is usually done by engineers who are not familiar with the details of the literature. We aim at closing this gap by introducing a framework for design and verification of parameterized systems in BIP. With this framework, we make the following specific contributions:

1. We propose the *first-order interaction logic* (FOIL) within BIP framework as a formal language for architectures of parameterized systems, i.e. system topologies and communication mechanisms. FOIL is powerful enough to express architectures found in parameterized systems, including the classical architectures: token-passing rings, rendezvous cliques, broadcast cliques, rendezvous stars. We also identify a decidable fragment of FOIL, which is important for practical applicability..

2. We investigate the decidability of the verification of parameterized BIP models, where components are descirbed by finite state automata, and the system architecture is specified by a FOIL formula. We prove that this problem is undecidable in general, and also identify certain decidable fragments, relying on the well-structured transition system theory [1, 66].

3. We provide a framework for the integration of mathematical models from the parameterized model checking literature in an automated way: given a parameterized BIP design, our framework detects parameterized model checking techniques that are applicable to this design. We present how to identify the system architecture automatically by the use of SMT solvers and standard (non-parameterized) model checkers.

4. We provide a preliminary prototype implementation of the proposed framework. Our prototype tool takes a parameterized BIP design as its input and detects whether one of the following classical results applies to this BIP design: the cut-off results for token-passing rings by Emerson & Namjoshi [60], the VASS-based algorithms by German & Sistla [74], and the undecidability and decidability results for broadcast systems by Abdulla et al. [1] and Esparza et al. [64]. More importantly, our framework is not specifically tailored to the mentioned techniques.

We remark that our framework builds on the notions of BIP, which allows us to express complex notions in a terminology understood by engineers. Moreover, our framework allows an expert in parameterized model checking to capture seminal mathematical models found in the verification literature, e.g. [74, 64, 60, 42].

These contributions are elaborated in Chapter 6.

1.4 Organization of this dissertation

The rest of the dissertation is organized as follows:

- In Chapter 2, we present some preliminaries of safety property verification, and introduce the BIP modelling language, which we use in this dissertation as the formal system model, and its operational semantics. We also present a symbolic encoding of BIP system models as symbolic transition systems.
- In Chapter 3, we review the most relevant verification techniques for concurrent systems, in particular, abstraction and partial order reduction technqiues.
- In Chapter 4, we present the main verification techniques for the class of BIP models with a fixed number of components. In particular, we present an instantiation of the lazy predicate abstraction technique and a partial order reduction for BIP, and also their combination. We also present comprehensive experimental evaluations of the proposed techniques against the state-of-the-art verification techniques in the end of this chapter.
- In Chapter 5, we present two further techniques for improving partial order reductions. First, we investigate how to explore independent interactions simultaneously, instead of postponing them as in the other classical partial order reduction approaches. Second, we study how to exploit system symmetries to improve the reductions. We also present their combinations with lazy abstraction and the experimental evaluations.
- In Chapter 6, we present the design and uniform verification framework for parameterized systems in BIP, that is the systems with unbounded number of participating components. We first present an extension of the current BIP framework to enable the modelling of a wide range of parameterized systems. Then we present an automated verification framework that can incorporate the existing parameterized verification techniques. We also present some decidability results for certain fragments of parameterized BIP models.
- In Chapter 7, we summarize this dissertation and also present some perspectives and future work.

2 Preliminary and system model

In this chapter, we first present some preliminaries of formal invariant verification. Then we present the BIP modeling language for systems that consist of a fixed number of components. In the end, we present an encoding of BIP system model as symbolic transition system.

2.1 Labeled transition system

We denote by $\mathbb{V}$ a set of integer variables, and the symbol $\mathbf{V}$ ranges over all possible valuations of variables. We also denote by $\mathscr{E}_{\mathbb{V}}$ the set of expressions, and $\mathscr{F}_{\mathbb{V}}$ the set of formulae in the theory of linear arithmetic over $\mathbb{V}$. We denote by $\mathbf{V} \models \phi$ the statement that a valuation $\mathbf{V}$ satisfies a formula $\phi \in \mathscr{F}_{\mathbb{V}}$. We denote by $\mathbf{V}[x := e]$ the substitution of variable x by expression e in the valuation $\mathbf{V}$. As usual, we use primed variables to represent the state of the system after one step. The priming notation is extended to formulae and assignments in the standard way.

In this dissertation, we use labeled transition systems to define the operational semantics of computing systems.

Definition 2.1.1 (Labeled transition system) *A labeled transition system (LTS) is defined by a tuple $\mathscr{T} = \langle C, \Sigma, R, C_0 \rangle$, which consists of*

1. *a set of states C;*
2. *a set of transition labels Σ;*
3. *a set of transition relations $R \subseteq C \times \Sigma \times C$;*
4. *a set of initial states $C_0 \subseteq C$.*

For simplicity, we denote a transition $\langle c, t, c' \rangle \in R$ by $c \xrightarrow{t} c'$. A transition t is enabled in the state c, if $c \xrightarrow{t} c'$, for some $c' \in C$. An LTS is deterministic if $c \xrightarrow{t} c_1$ and $c \xrightarrow{t} c_2$ implies $c_1 = c_2$, for any $c \in C$ and $t \in \Sigma$. In this dissertation, we focus on deterministic transition systems.

A trace (or an execution) of a transition system is a sequence of transitions from a given state. We denote a trace by the sequence of transition labels. For instance, the sequence of transitions $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} c'$ is represented as $c \xrightarrow{t_1 t_2 \ldots t_n} c'$.

A state c is reachable if there is a trace $c_0 \xrightarrow{t_1 \ldots t_n} c$, where $c_0 \in C_0$ and $t_i \in \Sigma$, for each $i \in [1, n]$. Given a state c, we denote by $en(c) \subset \Sigma$ the set of transitions enabled in state c. A state c is a deadlock state if there is no such $t \in \Sigma$, and $c' \in C$ that $c \xrightarrow{t} c'$. We denote by RS the set of all reachable states.

A set of states can also be represented by its characteristic predicate, that is, a predicate represents all the states that satisfy it. Given a predicate p, we define the *post* operator as follows:

$$post(p, R) = \{c' \in C \mid \exists c \in C, p(c) \wedge (c, t, c') \in R\}$$

In other words, $post(p, R)$ characterises the set of states that are reachable from the states satisfying the predicate p in one step by taking a transition in R. For instance, $post(C_0, R)$ represents the set of states that are reachable from the initial states in one step. More generally, the set of reachable states within i steps can be defined as follows using the *post* operator:

$$RS_i = C_0 \vee post(C_0, R) \vee \ldots \vee post^i(C_0, R)$$

where $post^i$ represents the ith applications of *post*.

2.2 Invariant verification

An invariant is a safety property, which requires that 'something bad' should never happen in all possible executions of the system. An invariant is often given by a condition ϕ for the system states and requires that ϕ holds for all reachable states. Formally it is defined as follows.

Definition 2.2.1 (Invariant) *Given a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$, a formula ϕ is an invariant of $\mathcal{T}$ if $\forall c \in RS$, state c satisfies ϕ.*

Problem 2.2.2 (Invariant verification) *Given a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$ and an invariant property ϕ, the invariant verification problem asks whether for every state c that is reachable from an initial state $c_0 \in C_0$, i.e. $c \in RS$, holds $c \models \phi$.*

One simple way to verify an invariant of a given labeled transition system $\mathcal{T}$ is to compute the set of reachable states RS by repeatedly applying the *post* operator until a fixpoint is reached. The existence of least fixpoint is guaranteed by the monotonic property of the $post$ operator.

We refer to [132] for more details. The invariant verification problem can be solved by checking whether all the states RS satisfy the invariant or not. However, this approach does not work well in practise: the state space is frequently much too large to be exhaustively explored, and the fixpoint computation hardly converge, due to either data space explosion or concurrent interleavings.

Another approach to prove that a formula ϕ is an invariant is to construct another formula ϕ_i and prove that ϕ_i is inductive invariant such that $\phi_i \implies \phi$.

Definition 2.2.3 (Inductive invariant) *Given a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$, a formula ϕ is an inductive invariant if the following two conditions hold:*

1. $\forall c \in C_0, c \models \phi$, *and*
2. $\forall c \in C, (c \models \phi) \wedge (c, t, c') \in R \implies (c' \models \phi')$.

where ϕ' is the formula obtained by replacing all the variables in ϕ by the corresponding primed ones.

Using the *post* operator, the second condition of inductive invariant can also be specified as $post(p, R) \implies p$, or equally $p = p \vee post(p, R)$. Thus, the strongest inductive invariant can be expressed as the least fixpoint of the $post$ operator, which characterizes exactly the set of reachable states RS. However, computing the least fixpoint is computationally expensive as stated above. We remark that finding inductive invariants automatically is a difficult task, and it remains an active research area in formal verification. We refer to [113] for more information about invariant verification.

An invariant property can also be specified dually by a set of error states C_{error} that violate the invariant. We say $\mathcal{T}$ is safe with respect to C_{error}, if no states in C_{error} are reachable, i.e. the intersection of C_{error} and RS is empty. Suppose the characterizing predicate of the set of error states is p_{error}. Equivalently, we say that $\mathcal{T}$ is safe with respect to C_{error}, if $\neg p_{error}$ is an invaraint of $\mathcal{T}$. Thus, the error-states reachability problem can be viewed as an invariant verification problem and vice versa. In this thesis, we do not differentiate them and focus on devising efficient techniques to solve the invariant verification problem for concurrent systems, in particular the component-based systems.

2.3 BIP modeling framework

In this section, we present the fragment of the BIP language with multiparty synchronization and priority. The BIP language only allows describing systems with fixed structure and interaction topology. First, we present the syntactic BIP model [1].

1. There are two versions of BIP. We present the new version in this dissertation. We omit the langauge differences between these two versions, which are minor. For more information we refer to http://www-verimag.imag.fr/New-BIP-tools.html.

2.3.1 Syntactic BIP model

A BIP model contains a finite set of components. Each component is an instantiation of a component type, which is formally defined as a finite state automaton extended with data.

Definition 2.3.1 (Component type) *A* BIP *component type is defined as a tuple* $\mathbb{B} = \langle \mathbb{V}, \mathbb{L}, \mathbb{P}, \mathbb{E}, \ell \rangle$, *where*

1. $\mathbb{V}$ *is a finite set of variables;*
2. $\mathbb{L}$ *is a finite set of control locations;*
3. $\mathbb{P}$ *is a finite set of communication port types;*
4. $\mathbb{E} \subseteq \mathbb{L} \times \mathbb{P} \times \mathscr{F}_{\mathbb{V}} \times \mathscr{E}_{\mathbb{V}} \times \mathbb{L}$ *is a finite set of transition edges extended with guards in* $\mathscr{F}_{\mathbb{V}}$ *and operations in* $\mathscr{E}_{\mathbb{V}}$*;*
5. $\ell \in \mathbb{L}$ *is an initial control location.*

Given a tuple of component types $\bar{\mathbb{B}} = \mathbb{B}_0 \cup \cdots \cup \mathbb{B}_{k-1}$, and a tuple of natural numbers $\bar{n} = \langle n_0, \ldots, n_{k-1} \rangle$, where $n_i, i \in [0, k-1]$ represents the number of instantiations of component type $\mathbb{B}_i$, For each $i \in [0, k-1]$, we denote by $B_i[j]$, $j \in [1, n_i]$ instantiations of the component type $\mathbb{B}_i$, where every element of $\mathbb{B}_i$ has a local copy in $B_i[j]$. We denote by P_i the set of ports instantiated from the type $\mathbb{P}_i$.

Since we have a finite number of components, we can refactor the index of components and for the presentation simplicity, we denote by $\{B_i\}_{i=1}^{n}$ the set of components instantiated from all types $\bar{\mathbb{B}}$, and we do not distinguish the variables, control locations and the transitions of component type $\mathbb{B}_i$ from its local copies, when it is clear from the context.

Transition edges in a component are labeled by ports, which form the interface of the component. Ports are used for communication or synchronization with other components. We assume that, from each control location, every pair of outgoing transitions have different ports, and the ports of different components are disjoint. Thus, transitions with the same ports are not enabled simultaneously. Given a component violating such assumptions, we can easily transform it into the required form by renaming the ports, while still retaining the BIP expressiveness power. For the simplicity of presentation, we denote in the sequel the identity of the unique component where port type p is defined by $id(p)$.

Component coordination is realised by defining the set of allowed interactions, which synchronise transitions of different components. In BIP systems with a fixed number of component, an interaction is represented as a finite set of ports.

Definition 2.3.2 (Interaction) *A BIP interaction is defined as a tuple* $\gamma = \langle g, P, f \rangle$, *where* $g \in \mathscr{F}_{\mathbb{V}}$, $f \in \mathscr{E}_{\mathbb{V}}$ *and* $P \subseteq \bigcup_{i=1}^{n} P_i$, $P \neq \emptyset$, *and for all* $i \in [1, n]$, $|P \cap P_i| \leq 1$.

An interaction consists of a guard condition, a set of connected ports and an operation on the variables, which are defined in the connected components. Condition $|P \cap P_i| \leq 1$ imposes the restriction that an interaction can connect at most one port from each component.

Intuitively, an interaction defines a guarded multiparty synchronization with data transfer: an interaction γ is enabled only if the guard g is enabled, and when γ is executed, the data transfer specified by f is executed first, and then the transitions labeled by the ports in P are taken simultaneously. We denote by Γ a finite set of interactions.

Priority can be used to resolve the conflicts among interactions.

Definition 2.3.3 (Priority) *Given a set of interactions* Γ*, a priority model* Π *is a strict partial order on* Γ*. For* $\gamma, \gamma' \in \Gamma$*, we write* $\gamma < \gamma'$ *if and only if* $(\gamma, \gamma') \in \Pi$*, which means that interaction* γ' *has a higher priority than* γ*.*

We remark that priority restricts the coordination of the system. Thus, ignoring the priority would be a safe over-approximation in terms of the invariant verification.

In BIP, we can construct a compound component by composing a finite number of components with interactions, and then use this compound component as a building block to construct a hierarchical model. However, in this dissertation, we do not consider hierarchical models. A BIP model is a single flat compound component, constructed by composing atomic components with interactions.

Definition 2.3.4 (BIP Model) *A* BIP *model is a tuple* $\mathcal{M}_{\text{BIP}} = \langle \{B_i\}_{i=1}^n, \Gamma, \Pi \rangle$*, where* $\{B_i\}_{i=1}^n$ *is a finite set of components,* Γ *is a finite set of interactions for all components, and* Π *is a priority model on* Γ*.*

In the rest of this section, we use two examples to illustrate the BIP modeling framework.

Example 2.3.5 (Ticket mutual exclusion protocol [110]) *Figure 2.1 depicts a* BIP *model of the ticket mutual exclusion protocol with two processes. The protocol works as follows. Upon entering the critical section, each process requests a fresh ticket from the controller, then the process waits until its ticket equals to the number to be served next. When leaving the critical section, the process resets its ticket and the controller increases the number to be served by one. Notice that all the variables are local to the component where they are defined.*

We model the process by a component with one integer variable $ticket_i$*, and three control locations* I_i*,* W_i*, and* C_i*,* $i \in \{1,2\}$*, where* C_i *represents the critical section. Each component also defines three ports* $request_i$*,* $enter_i$ *and* $leave_i$*, representing the transitions of requesting the ticket, entering and leaving the critical section respectively.*

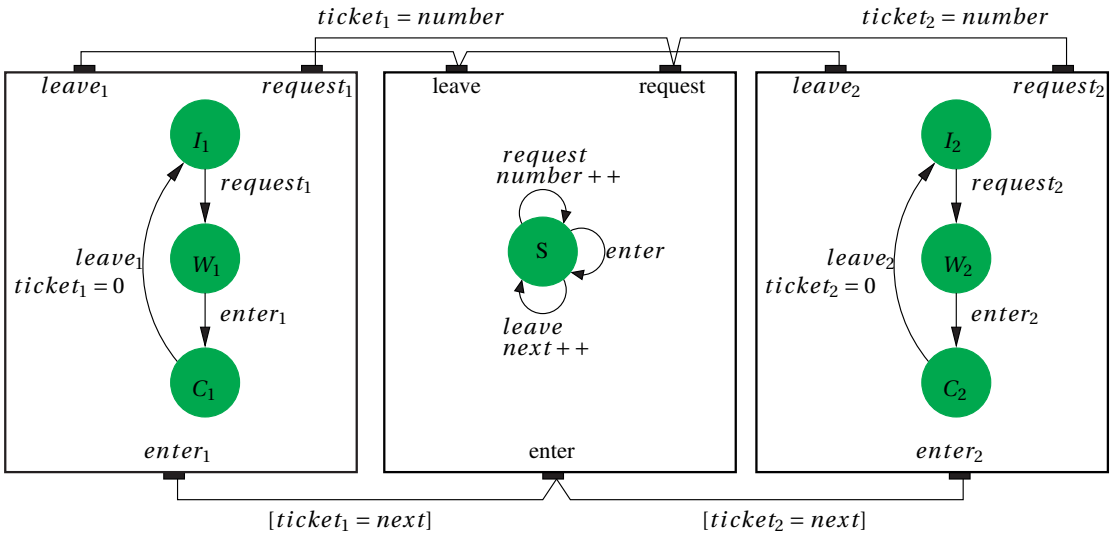

Figure 2.1 – Ticket mutual exclusion protocol

The synchronisations between the controller component and the processes are defined by six interactions. Each interaction is depicted as a wire in Figure 2.1. The ports connected by a wire are synchronized. When a port belongs to several interactions (e.g. the leave port of the central component), it must be synchronised through exactly one of them each time that it is fired. An interaction may also have a guard, e.g. the interaction $\{enter_1, enter\}$ *is guarded by* $[ticket_1 = next]$*, and an operation, e.g. upon firing of the interaction* $\{request_1, request\}$*, the operation* $ticket_1 = number$ *updates the variable* $ticket_1$ *to* $number$*. For simplicity, the constant guard* $true$ *and the empty assignment are omitted.*

To request a ticket number, the process i *(*$i \in [1,2]$*) synchronizes its transition* $request_i$ *with the controller's transition request, whereby the process copies the value of number to* $ticket_i$*. This is achieved by interactions (true,* $\{request_i$*, request},* $ticket_i$ *= number), where* $i \in [1,2]$*. To enter the critical section, a process synchronizes its transition enter with the controller's transition enter. This is achieved by interactions (* $[ticket_i$ *= next],* $\{enter_i$*, enter}, skip),* $i \in [1,2]$*. To leave the critical section, a process synchronizes its transition* leave *with the controller's transition* leave*. It is denoted by interactions (true,* $\{leave_i$*, leave}, skip),* $i \in [1,2]$*.*

Initially the controller sets both $number$ *and* $next$ *to be 1, and the local variable of each process is 0. The mutual execlusion property requires that the two processes cannot enter the critical location at the same time.*

This model is specified in the BIP language as shown in Figure 2.2.

Example 2.3.6 (Temperature control system [9]) *In Figure 2.3, we show a graphical representation of the* BIP *model of a coolant temperature control system in a reactor tank. There are three atomic components: controller in the middle, and two rods on left and right side. These three*

```
package TicketBIP

    port type NullPort()
    port type DataPort(int x)

    atom type controller()
        data int number
        data int next

        export port DataPort request(number)
        export port DataPort enter(next)
        export port NullPort leave()

        place S

        initial to S do {number = 1; next = 1;}
        on request from S to S do {number++;}
        on enter from S to S
        on leave from S to S do {next++;}
    end

    atom type process()
        data type ticket

        export port DataPort request(ticket)
        export port DataPort enter(ticket)
        export port NullPort leave()

        place I, W, C

        initial to I
        on request from I to W
        on enter from W to C
        on leave from C to I do {ticket = 0;}
    end

    connector type NullPortConnector(NullPort p1, NullPort p2)
        define [p1 p2]
    end

    connector type DataPortConnector(DataPort p1, DataPort p2)
        define [p1 p2]
        on p1 p2 provided(p1.x == p2.x)
    end

    connector type DataPortMsgConnector(DataPort p1, DataPort p2)
        define [p1 p2]
        on p1 p2 down {p2.x = p1.x;}
    end

    compound type System()
        component controller c()
        component process a1()
        component process a2()

        connector DataPortMsgConnector a1requet(c.request1, a1.request)
        connector DataPortMsgConnector a2requet(c.request2, a2.request)
        connector DataPortConnector a1enter(a1.enter, c.enter1)
        connector DataPortConnector a2enter(a2.enter, c.enter2)
        connector NullPortConnector a1leave(a1.leave, c.leave1)
        connector NullPortConnector a2leave(a2.leave, c.leave2)
    end
end
```

Figure 2.2 – Ticket mutual exclusion protocol in BIP language

component are composed by five interactions: $(true, \{tick, tick1, tick2\}, skip)$, $(true, \{cool, cool1\}, skip)$, $(true, \{cool, cool2\}, skip)$, $(true, \{reset, reset1\}, skip)$, *and* $(true, \{reset, reset2\}, skip)$. *No guards or actions are defined in these interactions.*

The temperature of the tank (denoted by the variable t *in the controller) rises with the rate of* $v_r = 1°/s$, *modeled by the transition* $(S3, tick, true, t := t+1, S3)$. *When the temperature reaches the upper bound of* $100°$, *the controller will refrigerate the tank by moving one of the two rods (i.e. firing one of the interactions* $\{cool, cool1\}, \{cool, cool2\}$*). The temperature will then decrease with the rate of* $v_d = 1°/s$. *When the temperature the lower bound of* $50°$, *the controller removes the rod from the coolant (i.e. firing the corresponding interaction* $\{reset, reset1\}$ *or* $\{reset, reset2\}$*).*

A rod can be moved again only when 100 *time units have elapsed after the last movement, e.g. transition* $(S1, cool1, [c_1 >= 100], skip, S2)$

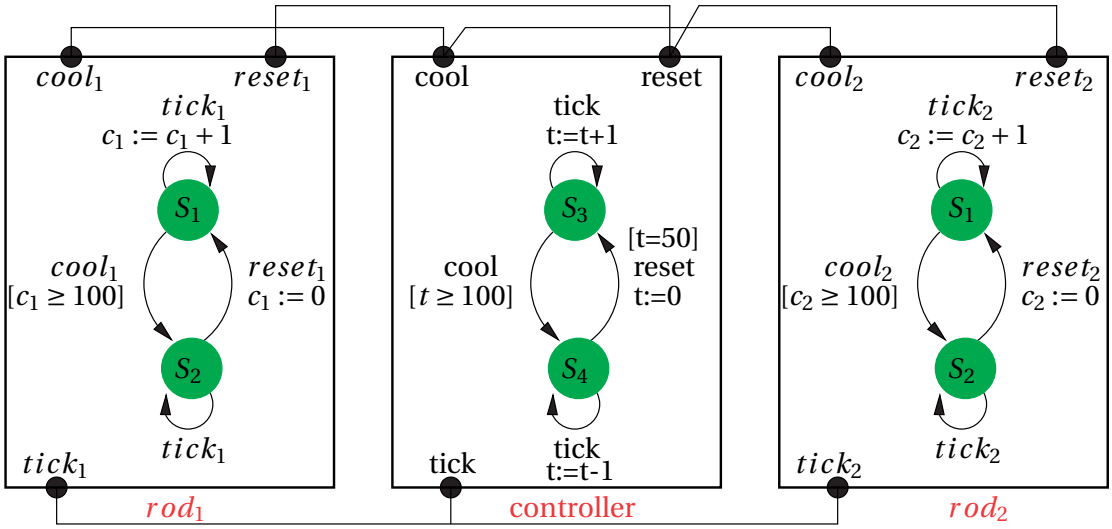

Figure 2.3 – Temperature Control System in BIP

This model is specified in the BIP language as shown in Figure 2.4.

2.3.2 BIP operational semantics

A state of a BIP model comprising the components $\{B_i\}_{i=1}^n$, with each $\mathbb{B}_i = \langle \mathbb{V}_i, \mathbb{L}_i, \mathbb{P}_i, \mathbb{E}_i, \ell_i \rangle$, is a tuple $c = \langle \langle l_1, \mathbf{V}_1 \rangle, \ldots, \langle l_n, \mathbf{V}_n \rangle \rangle$, where for all $i \in [1, n]$, $l_i \in \mathbb{L}_i$ and $\mathbf{V}_i$ is a valuation of $\mathbb{V}_i$. A state c_0 is initial if for all $i \in [1, n]$, $l_i = \ell_i$ and $\mathbf{V}_i$ is the initial valuation of $\mathbb{V}_i$. A state c is an error if for some $i \in [1, n]$, l_i is an error location. We say an interaction $\gamma = \langle g, P, f \rangle$ is enabled in a state $c = \langle \langle l_1, \mathbf{V}_1 \rangle, \ldots, \langle l_n, \mathbf{V}_n \rangle \rangle$, if $\bigcup_{i=1}^n \mathbf{V}_i \models g$ and for every component B_i, such that $P \cap P_i \neq \emptyset$, there is an edge $\langle l_i, P \cap P_i, g_i, f_i, l_i' \rangle \in \mathbb{E}_i$ and $\mathbf{V}_i \models g_i$.

The labeled transition system semantics of a BIP model is defined as follows.

```
package ReactorBIP
    port type Sync()

    atom type Rode()
        data int c

        export port Sync cool()
        export port Sync rest()
        export port Sync tick()

        place S1, S2

        initial to S1 do {c = 0;}
        on tick from S1 to S1 do {c = c + 1;}
        on tick from S2 to S2
        on cool from S1 to S2 provided (c >= 100)
        on rest from S2 to S1 do {c = 0;}
    end

    atom type Controller()
        data int t

        export port Sync cool()
        export port Sync heat()
        export port Sync tick()

        place S3, S4

        initial to S3 do{ t = 10;}
        on cool from S3 to S4 provided (t >= 100)
        on tick from S3 to S3 do {t = t + 1;}
        on heat from S4 to S3 provided (t == 50) do {t = 0;}
        on tick from S4 to S4 do {t = t - 1;}
    end

    connector type Sync2(Sync cc, Sync rc)
        define [ cc rc ]
    end

    connector type SyncAll(Sync r1, Sync r2, Sync ctrl)
        define[r1 r2 ctrl]
    end

    compound type Reactor()
        component Rode rode1()
        component Rode rode2()
        component Controller controller()

        connector Sync2 c1(controller.cool, rode1.cool)
        connector Sync2 c2(controller.cool, rode2.cool)
        connector Sync2 h1(controller.heat, rode1.rest)
        connector Sync2 h2(controller.heat, rode2.rest)
        connector SyncAll tick(rode1.tick, rode2.tick, controller.tick)
    end
end
```

Figure 2.4 – Temperature control system in BIP language

Definition 2.3.7 (BIP operational semantics) *Given a* BIP *model* $\mathscr{M}_{\text{BIP}} = \langle \{B_i\}_{i=1}^{n}, \Gamma, \Pi \rangle$, *its operational semantics is defined by a labeled transition system* $\mathscr{T}_{\text{BIP}} = \langle C_{BIP}, \Sigma_{BIP}, R_{BIP}, C_{0_{BIP}} \rangle$, *where*

1. C_{BIP} *is the set of states as defined above,*
2. $\Sigma_{BIP} = \Gamma$,
3. R_{BIP} *is the set of transitions, such that there is a transition from a state* c *to another* c', *if and only if there is an interaction* $\gamma = \langle g, P, f \rangle$ *such that,*
   (a) γ *is enabled in* c;
   (b) *for each component* B_i *such that* $P \cap P_i \neq \emptyset$, *there is an edge* $\langle l_i, P \cap P_i, g_i, f_i, l_i' \rangle \in \mathbb{E}_i$, *then* $\mathbf{V}_i' = \mathbf{V}_i[f_i(f(\mathbb{V}))]$;
   (c) *for each component* B_i *such that* $P \cap P_i = \emptyset$, $l_i' = l_i$ *and* $\mathbf{V}_i' = \mathbf{V}_i$;
   (d) *there does not exist an interaction* γ', *such that* γ' *is enabled in* c *and* $\gamma' > \gamma$.
4. $C_{0_{BIP}}$ *is the set of initial states.*

Notation $\mathbf{V}_i[f(\mathbb{V})]$ represents the update of variable evaluation $\mathbf{V}_i$ by the function application $f(\mathbb{V})$. For instance, suppose $\mathbf{V}_i = x = 1$, and $f(\mathbb{V}) = x{+}{+}$, then $\mathbf{V}_i[f(\mathbb{V})] = x = 2$. Notation $f_i(f(\mathbb{V}))$ denotes the sequential applications of function f and f_i. For simplicity, we denote by $c \xrightarrow{\gamma} c'$ that there is a transition from state c to state c', following the interaction γ.

For the invariant verification of BIP models with a fixed number of components, we can encode into the reachability of a set of locations, i.e. error locations. A BIP model is safe if no error states are reachable. Notice that any safety property can be encoded as a reachability problem by adding additional components.

2.4 Encoding BIP into Symbolic Transition System

In this section, we briefly present the encoding of BIP into symbolic transition systems, originally introduced in [26], which enables a direct application of the state-of-the-art model checkers for infinite-state systems, such as the NUXMV [33] symbolic model checker, to verify BIP models.

Definition 2.4.1 (Symbolic transition system) *A symbolic transition system is defined as a tuple* $STS = \langle \mathbb{V}, \phi_{C_0}, \phi_R \rangle$, *where*

1. $\mathbb{V}$ *is a finite set of variables;*
2. $\phi_{C_0}(\mathbb{V})$ *is a first-order formula over* $\mathbb{V}$ *defining the set of initial states;*
3. $\phi_R(\mathbb{V}, \mathbb{V}')$ *is a first-order formula over* $\mathbb{V} \cup \mathbb{V}'$ *defining the transition relation.*

The semantic of a symbolic transition system can be given in terms of an LTS (see for example [112]).

Given a BIP model $\mathscr{M}_{\text{BIP}} = \langle \{B_i\}_{i=1}^n, \Gamma, \Pi \rangle$, the encoding as a symbolic transition system $STS_{\mathscr{M}_{\text{BIP}}} = \langle \mathbb{V}, \phi_{C_0}, \phi_R \rangle$ is the following.

Variables. First, the set of variables is defined as:

$$\mathbb{V} = \bigcup_{i=1}^n \{loc_i\} \cup \bigcup_{i=1}^n \{v | v \in \mathbb{V}_i\} \cup \bigcup_{i=1}^n \{v_p | p \in P_i\} \cup \{v_\Gamma\}$$

where for all $i \in [1, n]$, $\mathbb{V}_i$ is the set of variables in component B_i and we preserve the domain of each variable $v \in \mathbb{V}_i$ in the encoding. We introduce a variable loc_i for each component B_i to encode its control locations, and for each port $p \in P_i$, we also introduce a boolean variable v_p, representing the status of the port, being enabled or disabled. Besides, we introduce an enumerative variable v_Γ, which represents the set of interactions Γ.

Initial condition. The initial condition is defined as:

$$\phi_{C_0} = \bigwedge_i^n (loc_i = l_{0_i} \wedge \bigwedge_{v \in \mathbb{V}_i} v = v_0)$$

The initial valuations of port variables and the interaction variable v_Γ are arbitrary.

Transition relation. The transition relation is the following:

$$\phi_R = \bigwedge_{i=1}^n (Tr_{e_i} \wedge Tr_{p_i}) \wedge \phi_\Gamma \wedge \phi_\Pi$$

where Tr_{e_i} encodes the edges of the component B_i, Tr_{p_i} encodes the conditions when the port p is enabled in component B_i, ϕ_Γ encodes the interaction, and ϕ_Π encodes the priorities. In the following, let Γ_{B_i} be the set of all the interactions in which the component B_i participates and Γ_e be the set of interactions that involve the port that labels the edge e.

The encoding of the edges in component B_i is defined as:

$$Tr_{e_i} = \bigvee_{e=\langle l_i, p_e, g_e, f_e, l_i' \rangle \in \mathbb{E}_i} loc_i = l_i \wedge loc_i' = l_i' \wedge g_e \wedge \bigvee_{\gamma \in \Gamma_{B_i}} v_\Gamma = \gamma \wedge$$
$$\bigwedge_{\gamma \in \Gamma_{B_i}} \left(v_\Gamma = \gamma \rightarrow f_e(f_\gamma(\mathbb{V}', \mathbb{V}))\right) \wedge \bigwedge_{\gamma \notin \Gamma_{B_i}} \left(v_\Gamma = \gamma \rightarrow \bigwedge_{x \in \mathbb{V}_i} x' = x\right)$$

where the expression $f_e(f_\gamma(\mathbb{V}', \mathbb{V}))$ is a symbolic encoding of function application f_e of the transition e and the function application f_γ of the interaction $\gamma = \langle g_\gamma, P_\gamma, f_\gamma \rangle$ [2].

The encoding of the port enablement Tr_{p_i} is defined as:

2. Note that, while in our definition f_γ is a single assignment, the approach can be easily generalized to sequential programs by applying a *single-static assignment (SSA)* transformation.

$$Tr_{p_i} = \bigwedge_{p \in P_i} \left(v_p \leftrightarrow \bigvee_{\langle l,p,g,op,l' \rangle \in \mathbb{E}_i} \left(loc_i = l \wedge g \right) \right)$$

That is, a port p is enabled if one of the transitions labeled by p is enabled.

Finally, the conditions that constrain the interactions to their ports and the priorities among the interactions are defined as:

$$\phi_\Gamma = \bigwedge_{\gamma = \langle g_\gamma, P_\gamma, op_\gamma \rangle \in \Gamma} \bigwedge_{p \in P_\gamma} v_\Gamma = \gamma \rightarrow (v_p \wedge g_\gamma)$$

$$\phi_\Pi = \bigwedge_{(\gamma_2,\gamma_1) \in \Pi, \gamma_1 = \langle g_{\gamma_1}, P_{\gamma_1}, op_{\gamma_1} \rangle} (g_{\gamma_1} \wedge \bigwedge_{p \in P_{\gamma_1}} v_p) \rightarrow v_\Gamma \neq \gamma_2$$

The encoding preserves the BIP semantics. It is not hard to prove the correctness of the encoding. The initial configuration is precisely characterised by the formula ϕ_{C_0}, where loc_i is constrained to the initial locations of the corresponding component, and each component variable is also assigned to the initial value. The transition relation is also characterised precisely, since the variable v_Γ can be assigned to the value representing a single interaction γ at a time, which will enable the corresponding transitions of the components. The encoding of ϕ_Γ ensures that v_Γ gets the value only if γ is enabled in the corresponding state of the BIP model. The valuations of the additional variables v_p and v_Γ do not alter the state space: their valuations are constrained by the formulae Tr_{p_i} and ϕ_Γ to reflect the BIP semantics.

3 Verification of concurrent systems

In this chapter, we review the most relevant techniques for algorithmic verification of infinite-state concurrent systems. We present these techniques in the following two categories: 1) abstraction techniques for resolving the data state explosion problem, and 2) partial order techniques for reducing the number of redundant interleavings due to concurrency.

3.1 Abstraction techniques

Verification of the transition system with an infinite or large state space is computationally hard. A prevalent way to reduce the state space size is to employ abstraction, as discussed in Chapter 1. The basic idea of abstraction is to construct a smaller abstract transition system, that soundly over-approximates the concrete system, such that safety of the abstract system entails the safety of the concrete one. However, the inverse does not hold in general: unsafety of the abstract system does not entail the unsafety of the concrete system. A general framework to formalize abstraction and its soundness is abstract interpretation [48].

3.1.1 Abstract interpretation

Given a transition system $\mathscr{T} = \langle C, \Sigma, R, C_0 \rangle$, we denote the the concrete property domain by $\mathscr{D}^\natural = (2^C, \subseteq)$. As we discussed in Section 2.2, invariant verification boils down to the fixpoint computation of the *post* operator. In order to avoid the expensive fixpoint computation in the concrete domain, abstract interpretation works in an abstract property domain, denoted by $\mathscr{D}^\sharp$, performs the fixpoint computation in this abstract domain, and then maps the result back to the concrete domain. The correctness of the abstract analysis can be established by a correspondence between the concrete and abstract domains, called Galois connection.

Definition 3.1.1 (Galois connection) *Let $(\mathscr{D}^\natural, \subseteq)$ and $(\mathscr{D}^\sharp, \sqsubseteq)$ be the concrete and abstract property domain respectively, a pair of (monotone) functions (α, β) defines a Galois connection between these two domains, where $\alpha : \mathscr{D}^\natural \mapsto \mathscr{D}^\sharp$ and $\beta : \mathscr{D}^\sharp \mapsto \mathscr{D}^\natural$, iff for all $a \in \mathscr{D}^\natural$ and $b \in \mathscr{D}^\sharp$,*

the following holds:

$$\alpha(a) \sqsubseteq b \;\Leftrightarrow\; a \subseteq \beta(b)$$

Usually, α is called abstraction function, and β is called the concretisation function. Galois connection preserves the order in the corresponding domain: if the abstraction $\alpha(a)$ of an element $a \in \mathscr{D}^\natural$ is smaller than $b \in \mathscr{D}^\sharp$ in the abstract domain, then $a \in \mathscr{D}^\natural$ is smaller than the concretisation $\beta(b)$ of $b \in \mathscr{D}^\sharp$ in the concrete domain. Intuitively, order preserving ensures the soundess of the analysis: computation in the abstract domain will always be a safe over-approximation.

The abstraction of the computation in the concrete domain is then captured by the following definition.

Definition 3.1.2 (Function abstraction) *Given a concrete domain* $(\mathscr{D}^\natural, \subseteq)$*, an abstract domain* $(\mathscr{D}^\sharp, \sqsubseteq)$ *and a Galois connection* (α, β)*, a function* $f^\sharp \in \mathscr{D}^\sharp \mapsto \mathscr{D}^\sharp$ *is an abstraction of a function* $f^\natural \in \mathscr{D}^\natural \mapsto \mathscr{D}^\natural$ *iff*

$$\alpha \circ f^\natural \circ \beta \sqsubseteq f^\sharp$$

where $\circ$ *denotes functional composition.*

$f^\sharp$ is the exact function abstraction of $f^\natural$ when $f^\sharp = \alpha \circ f^\natural \circ \beta$. In this case, we also say $f^\sharp$ is the induced abstraction of $f^\natural$ by the Galois connection (α, β). In particular, if $f = post$, then function $f^\sharp$ is the abstract *post* predicate transformer, which can be used to approximate the concrete reachable states.

Abstract interpretation provides a general framework to automate program analysis, where the crux is to design a suitable abstract property domain.

3.1.2 Predicate abstraction

As an instantiation of the abstract interpretation framework, predicate abstraction [79] uses a finite set of predicates to construct the abstract domain. The predicates usually denote properties of the state and are expressed as formulae, modulo some background theory, over the state variables. The abstraction is defined by the value of these predicates in any concrete state of the system. The fundamental operation in predicate abstraction can be described as the following. Given a formula ϕ and a set of predicates $P = \{p_1, ..., p_n\}$, generate the most precise approximation (either under-approximation or over-approximation) of ϕ using P. Over the decades, many techniques have been proposed to compute predicate abstraction efficiently [50, 69, 106, 107, 138].

We describe the predicate abstraction as it was presented in the seminal paper [79] and in the framework of abstract interpretation. Suppose the set of predicates is $P = \{p_1, ..., p_n\}$, the abstract domain is $\mathscr{D}^\sharp = (2^P, \sqsubseteq)$, where the order $\sqsubseteq$ is subset inclusion. Given an abstract state $P_1 \sqsubseteq P$, and a transition relation R, the successor abstract state P_1' is then defined as follows:

$$P_1' = \{p \in P \mid post(\bigwedge P_1, R) \implies p \text{ is valid}\}$$

where $\bigwedge P_1$ represent the conjunction of all the predicates in P_1.

The abstraction function of predicate abstraction $\alpha : 2^C \rightarrowtail 2^P$ is defined as follows:

$$\alpha(C') = \{p \in P \mid C' \cap \{c | c \models p\} \neq \emptyset\}.$$

and the concretization function $\beta : 2^P \rightarrowtail 2^C$ is defined by:

$$\beta(P') = \{c \mid c \models \bigwedge P'\}.$$

the induced abstraction of the $post$ operator under predicate abstraction is defined by:

$$post^\sharp = \alpha \circ post \circ \beta$$

The reachable states of the concrete transition system can be approximated by computing the least fixpoint of $post^\sharp$ in the abstract domain. However, since predicate abstraction computes an over-approximation, the analysis may report a false positive, i.e. a spurious counterexample.

To refine the precision of the abstract analysis and eliminate the spurious counterexamples, the counterexample guided abstraction refinement (CEGAR) technique was proposed in [39, 40].

3.1.3 Counterexample guided abstraction refinement

The process of CEGAR reasoning is depicted in Figure 3.1. Starting with an initial abstraction (possibly an empty set of predicates in predicate abstraction), we check if the current abstraction is able to prove the correctness or not. If no error is reported, then we can conclude the safety of the system. Otherwise, we check if the reported error is real or not. If the error is real, then we conclude the unsafety of the system and report a counterexample. If the error is unreal, then we eliminate this spurious error and refine the abstraction. After a successful refinement, we will repeat the above analysis until we either prove the correctness or a real counterexample is found.

We now elaborate two important subroutines of CEGAR techniques, i.e. counterexample analysis and abstraction refinement.

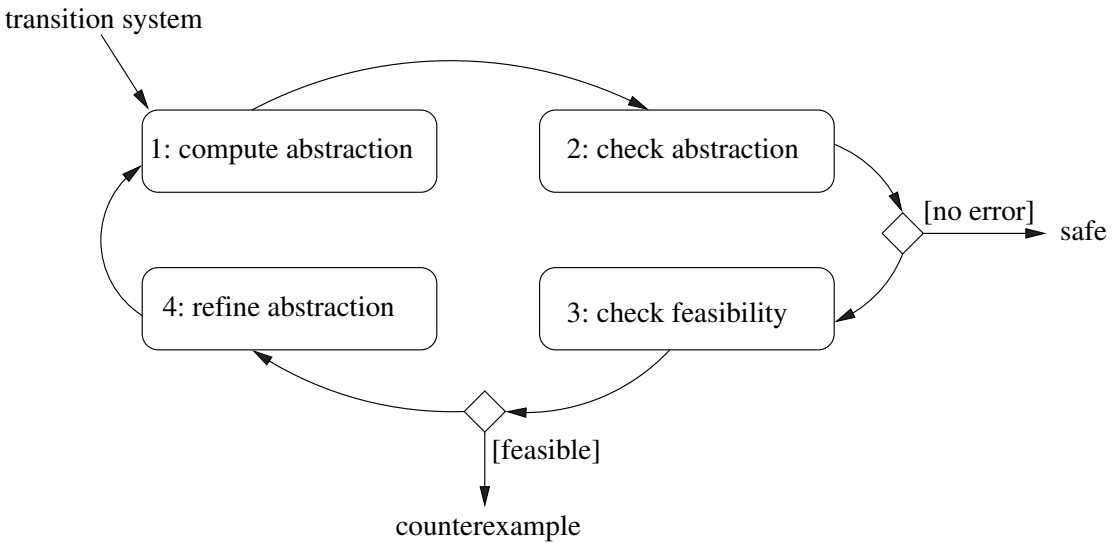

Figure 3.1 – Counterexample guided abstraction refinement loop

Suppose the abstract analysis produces a counterexample as shown in Figure 3.2, where the node in blue represents an error state, one way to check if it is a real counterexample is to build a trace formula $\phi_{\gamma_1} \wedge \phi_{\gamma_2} \wedge \ldots \wedge \phi_{\gamma_n}$ corresponding to the single static assignment form of the statements in the trace, and check its satisfiability by using an SMT solver. If the constructed trace formula is satisfiable, then the counterexample is real. Otherwise, it is spurious.

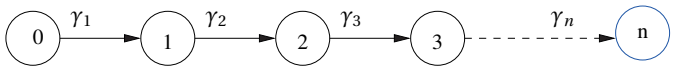

Figure 3.2 – A counterexample trace

When a spurious counterexample is found, it must be eliminated by refining the precision of the abstract analysis. In predicate abstraction, refining the abstraction boils down to discovering new important predicates to enrich the abstract property domain. One advanced technique to solve this problem relies on using Craig interpolation [49, 118, 120].

Definition 3.1.3 (Craig interpolant) *Given a pair of formulae* (ϕ_A, ϕ_B)*, such that the conjunction* $\phi_A \wedge \phi_B$ *is unsatisfiable, a Craig interpolant for* (ϕ_A, ϕ_B) *is a formula* ϕ'_A *with the following properties:*

1. $\phi_A \implies \phi'_A$,
2. $\phi'_A \wedge \phi_B$ *is unsatisfiable and*
3. ϕ'_A *only refers to the common non-logic symbols of* ϕ_A *and* ϕ_B.

Given a spurious counterexample trace $c_1 \xrightarrow{\gamma_1\gamma_2\ldots\gamma_n} c_n$, there are $n-1$ possible ways of splitting the unsatisfiable formula $\phi_{\gamma_1} \wedge \phi_{\gamma_2} \wedge \ldots \phi_{\gamma_n}$ into two formulas $(\phi_{\gamma_1} \wedge \cdots \wedge \phi_{\gamma_i}, \phi_{\gamma_{i+1}} \wedge \cdots \wedge \phi_{\gamma_n})$, for

$i \in [1, n-1]$, preserving the order of interactions. For a given splitting, e.g. $(\phi_{\gamma_1}, \phi_{\gamma_2} \wedge \ldots \wedge \phi_{\gamma_n})$ we can compute an interpolant ϕ and derive a predicate from it. This predicate is then added to the abstract domain to refine the abstraction. The chief advantage of interpolants derived from refutations of unsatisfiable formulae is that they capture the facts that the prover derived about ϕ_{γ_1} in showing that ϕ_{γ_1} is inconsistent with $\phi_{\gamma_2} \wedge \ldots \wedge \phi_{\gamma_n}$. Thus, if the prover tends to ignore irrelevant facts and focus on relevant ones, we can think of interpolation as a way of filtering out irrelevant information from ϕ_{γ_1}.

The CEGAR approach (based on predicate abstraction and interpolation) has been widely investigated in literature and successfully applied in practice in program and software verification. An incomplete list of available tools that perform predicate abstraction includes the SLAM toolkit [15, 16, 14], BLAST toolkit [90, 88], SATABS model checker [41], and CPAchecker [23].

3.1.4 Lazy abstraction

In the traditional predicate abstraction, e.g. [15], an abstract transition system is first constructed eagerly and then used for model checking. A bottleneck of this approach is the construction of the abstract transition system, which may be very inefficient. Lazy abstraction [90, 88, 119] avoids the expensive construction of the abstract transition system and performs the abstraction only when necessary.

In combination with counterexample guided refinement, lazy abstraction provides a powerful technique to address the data state explosion problem for the verification of sequential programs. It also relies on predicate abstract domain to approximate the concrete states. In lazy abstraction, programs are represented as control flow graphs.

Definition 3.1.4 (Control flow graph) *A control flow graph is a tuple* $CFG = (\mathbb{V}, \mathbb{L}, \mathbb{E}, l_0)$*, consisting of a set of variables* $\mathbb{V}$*, a set of control locations* $\mathbb{L}$*, a set of transition edges* $\mathbb{E} \subseteq \mathbb{L} \times (\mathscr{E}_{\mathbb{V}} \cup \mathscr{F}_{\mathbb{V}}) \times \mathbb{L}$ *and a initial control location* l_0*.*

Lazy abstraction performs a forward reachability analysis and constructs an abstract reachability tree (ART) to approximate the concrete reachable states. The construction of ART proceeds by expanding the ART nodes, starting with the initial one. To expand a node, we first check if it represents an error location. If an error location is reached, we then check if the path from the root to this error node represents a real counterexample or not. One way to conduct this check is to symbolically simulate this path from the initial states. If the path is feasible, then a real counterexample is reported. Otherwise, we will identify the location from which the path becomes infeasible, and restart the construction from this location using a refined abstract domain.

Then we check if the current node can be covered by some other nodes. If it is covered, we can stop the exploration from the covered node, since it represents a subset of states of the covering

node. Otherwise, we expand the current node and compute all the abstract successors. We then add all the successors in the tree as the children nodes, and process them later.

The ART node expansion looks at the control location of the node, and for each outgoing edge in the control flow graph, a new successor node is created. The computation in [90] is slightly different from that of [119]. In [90], predicate abstraction is used to approximate the *post* operator, while in [119], no actual *post* operator is used. The state formula of the successor node is obtained from the interpolation.

The ART construction terminates when all leaf nodes are either covered or fully expanded. An ART is safe if no errors states are found. If the ART construction terminates, lazy abstraction returns either a safe ART as the safety proof for the given program, or a counterexample. However, the termination is not guaranteed in general.

3.2 Partial order reduction techniques

Abstraction techniques presented in the previous sections can resolve the data explosion problem. There is, however, another source of state explosion that cannot be handled by abstraction techniques. In concurrent systems, the state space also grows exponentially to the number of components in the systems. This is due to interleavings of concurrent transitions. Dedicated techniques for handling concurrency and for reducing the number of interleavings are generally called partial order reduction (POR) techniques [77, 78, 124, 125, 46, 139, 140, 141]. The observation is that many interleavings in concurrent systems are equivalent in the sense that they lead to the same final state, though in different execution order. The basic idea of POR is to explore only one representative interleaving out of all equivalent ones.

In order to select the equivalent interleavings, POR makes use of the independence property of concurrent transitions, i.e. when concurrent transitions are independent, their executions do not interfere with each other, and changing the order of interleaving does not change the final state. Formally, the concept of transition independence is defined as follows.

Definition 3.2.1 (Transition independence) *Given a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$, two transitions $t_1, t_2 \in \Sigma$ are independent, if in every state $c \in C$, the following two conditions hold:*

1. *if t_1 is enabled in c, and $c \xrightarrow{t_1} c'$, then t_2 is enabled in c iff t_2 is enabled in c'; and symmetrically for the case of t_2.*
2. *if t_1, t_2 are both enabled in c, and $c \xrightarrow{t_1 t_2} c'_1$, $c \xrightarrow{t_2 t_1} c'_2$, then $c'_1 = c'_2$.*

Independent transitions neither disable nor enable each other. Simultaneously enabled independent transitions commute with each other and different execution orders result in the same final state. It is worthy of notice that the above definition of independence is uncondi-

tional, i.e. for all states in C. Independence is also a global property. One has to look at every possible reachable state in order to obtain the precise independence relation.

Similarly, given a sequence of transitions $t_1 \dots t_i t_j \dots t_n$, where t_i and t_j are independent, permutating t_i and t_j will result in an equivalent transition sequence in the sense that they both lead to the same final state. More generally, given a state $c \in C$, if $c \xrightarrow{t_1 \dots t_n} c_1$ and $c \xrightarrow{t'_1 \dots t'_n} c_2$, and $t'_1 \dots t'_n$ can be obtained from $t_1 \dots t_n$ by successively permutating adjacent independent transitions, then we have $c_1 = c_2$.

In the literature, traces that can be obtained from each other by successively permuting adjacent commutable transitions are called Mazurkiewicz's traces [115]. If the intermediate states of the traces are irrelavant to the property of interest, e.g. deadlocks, only one interleaving trace out of all Mazurkiewics's traces need be explored, thus saving the verification effort by exploring a reduced reachable state space.

In algorithmic verification, POR essentially performs a selective search to explore a subset of the whole state space, as shown in Algorithm 1. A selective search takes as inputs a transition system $\mathcal{T}$ to be explored, and a reduction function $f^{POR} : C \mapsto 2^{\Sigma}$, which is used to select the subset of explored transitions on each state. It basically operates as a classical state space search, e.g. DFS, except that, at each state reached during the search, it computes and explores a subset of all the enabled transitions on this state, using a reduction function f^{POR}. The other enabled transitions are postponed to be explored in the future or possibly ignored.

Algorithm 1 Basic selective search

```
1:  procedure SELECTIVESEARCH(T, f^POR)
2:      Stack, History initially empty
3:      push C_0 into Stack
4:      while Stack ≠ ∅ do
5:          pop c from Stack
6:          if c ∉ History then
7:              push c into History
8:              T = f^POR(c)
9:              for t ∈ T do
10:                 c' = post(c, t)
11:                 push c' into Stack
```

Clearly, a selective search only reaches a subset of all the reachable states, thus, constructs a reduced transition system defined as follows.

Definition 3.2.2 (Reduced labeled transition system) *Given a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$, a reduced transition system constructed by using a reduction function f^{POR} is a tuple $\mathcal{T}_r = \langle C_r, \Sigma_r, R_r, C_0 \rangle$, where $C_0 \subseteq C_r \subseteq C$, and if $c \in C_r$, $t \in f^{POR}(c)$, and $c \xrightarrow{t} c'$, then $c' \in C_r$ and $(c, t, c') \in R_r$.*

If the set of enabled transitions to be explored, i.e. $f^{POR}(c)$ is chosen properly, the reduced reachable state space C_r may be significantly smaller than C, while still preserving the same properties as the full reachable state space. It is important to notice that POR avoids generating the full reachable state space, and constructs the reduced one directly. Depending on the reduction function f^{POR}, POR approaches can be roughly classified into three categories: the ample set approach [124, 125, 46, 47], the stubborn set approach [139, 140, 141, 143], and the persistent set approach [77, 78, 68, 147]. In the following, we review the key insights of these POR approaches, and refer to the various research articles for the specific algorithms for computing such sets.

3.2.1 Ample set

Ample-set-based partial order reduction makes use of the property of transition independence in Definition 3.2.1. We denote the reduction function by f^{ample}.

Definition 3.2.3 (Ample set) *For a state c, a set of transitions $f^{ample}(c)$ is called an ample set iff it satisfies the following two conditions:*

A0. *if $en(c) \neq \emptyset$, then $f^{ample}(c) \neq \emptyset$;*

A1. *if $c \xrightarrow{t_1 \ldots t_n}$ and $t_i \notin f^{ample}(c), \forall i \in [1,n]$, then $t_i, \forall i \in [1,n]$ is independent with all transitions in $f^{ample}(c)$.*

It has been proved that POR based on an ample set reduction function preserves all the deadlocks of the full state space [47].

Theorem 3.2.4 ([47]) *Given a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$, and a deadlock state $c_d \in C$, $c_0 \xrightarrow{t_1 \ldots t_n} c_d$, $c_0 \in C_0$, if the reduction function $f^{POR}(c_0)$ obeys the conditions **A0** and **A1** in Definition 3.2.3, then there is a permutation $t'_1 \ldots t'_n$ of $t_1 \ldots t_n$ sucht that $c_0 \xrightarrow{t'_1 \ldots t'_n} c_d$ in the reduced transition system $\mathcal{T}_r$.*

Proof 3.2.5 *The proof proceeds by induction on the length of the trace. The conclusion holds trivally when $n = 0$.*

*If $n > 0$, then $f^{ample}(c_0)$ contains an enabled transition t. If none of $t_1 \ldots t_n$ is in $f^{ample}(c_0)$, then $c_d \xrightarrow{t}$ by **A1**, contradicting the assumption that c_d is a deadlock. So there must be a smallest index i such that $t_i \in f^{ample}(c_0)$. Let c_{i-1} and c_i be the states such that $c_0 \xrightarrow{t_1 \ldots t_{i-1}} c_{i-1} \xrightarrow{t_i} c_i$. Furthermore, let c'_1 be the state such that $c_0 \xrightarrow{t_i} c'_1$. Then by **A1**, applying independence $i-1$ times, we have that $c'_1 \xrightarrow{t_1 \ldots t_{i-1}} c'_i$. Since the transition system is deterministic, $c'_i = c_i$. So $c_0 \xrightarrow{t_i} c'_1 \xrightarrow{t_1 \ldots t_{i-1}} c'_i \xrightarrow{t_{i+1} \ldots t_n} c_d$. Hence, by the fact that c'_1 is in the reduced transition system, and the induction hypothesis that the conclusion holds for traces of length $n-1$, we conclude that it also holds for traces of length n.*

3.2.2 Stubborn set

With the stubborn set approach, transition independence is not explicitly assumed.

Definition 3.2.6 (Stubborn set) *For a state c, a set of transitions $f^{stubb}(c)$ is called a stubborn set iff the following conditions hold:*

- ***S0***. *if $en(c) \neq \emptyset$, then $f^{stubb}(c) \neq \emptyset$;*
- ***S1***. *for each transition $t \in f^{stubb}(c)$, which is is disabled in state c, if $c \xrightarrow{t_1 \ldots t_n} c_n$ and $t_i \notin f^{stubb}(c)$, for all $i \in [1,n]$, then t is also disabled in state c_n;*
- ***S2***. *for each transition $t \in f^{stubb}(c)$, which is enabled in state c, and $c \xrightarrow{t} c'$, if $c \xrightarrow{t_1 \ldots t_n} c_n$ and $t_i \notin f^{stubb}(c)$, for all $i \in [1,n]$, then there is another state c'_n, such that $c_n \xrightarrow{t} c'_n$ and $c' \xrightarrow{t_1 \ldots t_n} c'_n$.*

A stubborn set may contain both enabled and disabled transitions. Condition **S1** says that disabled transitions in the stubborn set remain disabled, while outside transitions take place. Condition **S2** says that enabled transitions in the stubborn set commute with sequences of outside transitions. Sets satisfying the above conditions are also called strong stubborn sets.

It is possible to change the third condition **S3** such that instead of requiring all enabled transitions in a stubborn set remain enabled while outside transitions occur, we only require one of them exists and remains enabled. Sets in this case are often called weak stubborn sets.

The following theorem states that every ample set is also a strong stubborn set when the transition system is deterministic [143].

Theorem 3.2.7 ([143]) *Assume that transition system is deterministic, then every ample set $f^{ample}(c)$ is also a strong stubborn set in state c.*

Proof 3.2.8 *Condition **A0** implies **S0**. Furthermore, **S2** follows directly from **A1** and **S1** follows from the fact that $f^{ample}(c) \subseteq en(c)$.*

However, the opposite does not hold in general.

3.2.3 Persistent set

Persistent-set-based partial order reduction relies on the conditional transition independence in a single state, instead of the unconditional one as in Definition 3.2.1.

Definition 3.2.9 (Conditional transition independence) *Given a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$, two transitions $t_1, t_2 \in \Sigma$ are independent in a state $c \in C$, iff the following two conditions hold:*

1. *if t_1 is enabled in c, and $c \xrightarrow{t_1} c'$, then t_2 is enabled in c iff t_2 is enabled in c'; and symmetrically for the case of t_2.*
2. *if t_1, t_2 are both enabled in c, and $c \xrightarrow{t_1 t_2} c'_1$, $c \xrightarrow{t_2 t_1} c'_2$, then $c'_1 = c'_2$.*

The only difference from Definition 3.2.1 is that the independence is considered in a single state, instead of the whole state space.

Definition 3.2.10 (Persistent set) *For a state c, a set of transitions $f^{pers}(c)$ is called a persistent set iff the following two conditions hold,*

P0. *$f^{pers}(c) \subseteq en(c)$;*

P1. *for every trace $c = c_0 \xrightarrow{t_1 \ldots t_n} c_n$, with $t_i \notin f^{pers}(c)$, $i \in [1, n]$, all transitions in $f^{pers}(c)$ are independent of t_i in state c_{i-1}.*

Intuitively, a set of transitions is called persistent in a state if whatever transition one takes from this state, while remaining outside of the set, does not interfere with all the transitions in the set.

The following theorem says that persistent set and strong stubborn set are equivalent when the transition system is deterministic, as stated in [78] and in [143].

Theorem 3.2.11 ([143, 78]) *Assume that the transition system is deterministic, then every nonempty persistent set is also a strong stubborn set and all the enabled transitions of a strong stubborn set give a persistent set.*

Proof 3.2.12 *Condition **P0** implies **S1** and if a persistent set is nonempty, then it implies **S0**. Further, condition **P1** implies **S2**.*

*Assume a stubborn set $f^{stubb}(c)$, let $f^{pers}(c) = f^{stubb}(c) \cap en(c)$. Let $t \in f^{pers}(c)$, $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} c_2 \ldots \xrightarrow{t_n} c_n$, and $t_i \notin f^{pers}(c)$, for $i \in [1, n]$. Then **S2** implies that there are states $c'_1, \ldots, c'_{n-1}$ such that $c' \xrightarrow{t_1} c'_1 \xrightarrow{t_2} c'_2 \ldots \xrightarrow{t_n} c_n$. By giving $t_1 \ldots t_i$ instead of $t_1 \ldots t_n$, we can conclude that $c_i \xrightarrow{t} c'_i$, for $i \in [1, n]$. Thus, **S2** implies that for all $i \in [1, n]$, t is independent of t_i in c_{i-1}, which means that the set $f^{pers}(c)$ is persistent.*

To summarize, for deterministic transition systems, the relation between the notions of ample set, stubborn set and persistent set is shown in the Figure 3.3. That is, ample set implies strong stubborn set and strong stubborn set is equivalent to persistent set.

3.2.4 Partial order reduction for safety properties

All the reduction functions above only guarantee the preservation of deadlocks in the reduced system. However, in general, preservation of safety properties cannot be guaranteed. This is

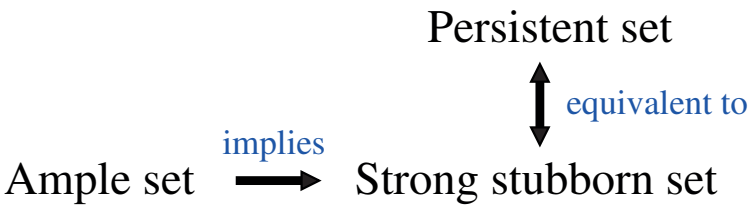

Figure 3.3 – Relations between ample, stubborn and persistent sets

due to the *ignoring problem* [139, 140, 46, 13], that is, when cycles are encountered during the state space exploration, the behaviour (local reachable states) of some component may be completely ignored on the states visited in the selective search. An example illustrating the ignoring problem is shown in Figure 3.4. We only show the automata of the two components. No interactions are defined to synchronize the transitions. Transitions of the two components can only interleave.

In the intial state (c, c_1), set $\{t\}$ satisfies the ample set conditions **A0** and **A1**, however, if we only explore the transition t, we would ignore all the reachable states of the component on the right-hand side. Thus, if the property being verified concerns the reachable states of that component, the above set would not be sufficient.

Figure 3.4 – An example for illustrating the ignoring problem

In order to avoid the ignoring problem, partial order reduction techniques usually introduce an additional condition **S**:

S. For every transition $t \in en(c)$, there exists a trace $c = c_0 \xrightarrow{t_1} c_1 \xrightarrow{t_2} c_2 \ldots \xrightarrow{t_n} c_n$, where $t_i \in f^{POR}(c_{i-1})$, such that $t \in f^{POR}(c_n)$.

This condition ensures that every enabled transition will occur at least once in the reduced state space. In the above example, both $\{t_1\}$ and $\{t, t_1\}$ satisfy this additional condition. Thus, either can be selected in the initial state.

4 Verification of BIP with bounded concurrency

In this chapter, we focuse on the algorithmic verification of component-based systems modeled in BIP with bounded concurrency, i.e. systems with a fixed number of components. Given the significant progress on algorithmic verification of concurrent infinite-state systems in the past decade, as discussed in the previous chapter, we leverage on the state-of-the-art abstraction techniques to analyze the behavior of infinite-state BIP components. Meanwhile, inspired by the ESST approach [130], in order to handle the concurrent interactions between components, we incorporate partial order reduction techniques to reduce the redundant interaction interleavings in the abstract reachability analysis.

In the sequel, we first present a lazy abstraction with interpolant-based abstraction refinement algorithm for BIP. Then we present a persistent set based partial order reduction for BIP and show how to combine it with lazy abstraction succinctly. Last but not the least, we present a comprehensive experimental evaluations of the proposed technique against the state-of-the-art.

This chapter is based on the following publication:

– *Formal verification of infinite-state BIP models*, Bliudze, Simon and Cimatti, Alessandro and Jaber, Mohamad and Mover, Sergio and Roveri, Marco and Saab, Wajeb and **Wang, Qiang**, International Symposium on Automated Technology for Verification and Analysis (ATVA 2015), pages 326–343, 2015, Springer.

The author proposed the algorithm of lazy abstraction with reduction for BIP and did the correctness proofs with the help of Dr. Sergio Mover, Dr. Marco Roveri and Dr. Alessandro Cimatti. The implementation is based on the Kratos model checker [36]. In the above article, there is also the symbolic encoding of BIP into nuXmv (presented in Section 2.4), which was initiated by Dr. Simon Bliudze, and implemented by Wajeb Saab, Dr. Mohamad Jaber, separately from mine. The author formalized the encoding.

4.1 Lazy abstraction of BIP

4.1.1 Data structures for verification

As in lazy abstraction, we compute an abstract reachability tree (ART) to over approximate all the reachable states. The ART for the verification of BIP is defined as follows.

Definition 4.1.1 (Abstract reachability tree) *An abstract reachability tree for a* BIP *model* $\mathscr{M}_{\text{BIP}} = \langle \{B_i\}_{i=1}^n, \Gamma, \Pi \rangle$ *is defined as a tuple* $\mathbb{T} = (\mathit{Nodes}, \mathit{Root}, \mathit{Edges}, \mathit{Covering})$*, where*

1. *Nodes is a set of tree nodes;*
2. *Root* $\in$ *Nodes is the unique root node;*
3. *Edges* $\subseteq$ *Nodes* $\times \Gamma \times$ *Nodes is a set of tree edges;*
4. *Covering* $\subseteq$ *Nodes* $\times$ *Nodes is the covering relation between tree nodes.*

An ART node represents an over-approximation of a set of BIP system states. Edges of the ART are labeled by interactions. They model both the branches of the control flow of each component, and the interleavings of concurrent interactions in different components. A path in an ART is then a sequence of interactions.

For a BIP model, an ART node is defined as follows.

Definition 4.1.2 (ART node) *An ART node for a BIP model* $\mathscr{M}_{\text{BIP}} = \langle \{B_i\}_{i=1}^n, \Gamma, \Pi \rangle$ *is defined as a tuple* $\eta = \langle \langle l_1, \phi_1 \rangle, \ldots, \langle l_n, \phi_n \rangle, \phi \rangle$*, where for each* $i \in [1, n]$*,* $\langle l_i, \phi_i \rangle$ *is the local region of component* B_i *with the control location* l_i *and the abstract data region* ϕ_i*, and* ϕ *is the global data region.*

The abstract data region ϕ_i over-approximates the valuations of variables in the control location l_i. We also maintain a global data region ϕ to keep track of the relations of the variables that are used in data transfer. The reason is that when data transfer is present, we may use predicates containing variables from different components in the abstraction structure, which cannot be expressed by using predicates only containing variables from the same component. It is worthy of noticing that the presence of data transfer prevents us from discovering modular proofs for component-based systems in the way similar to [83, 84].

A node is consistent if the conjunction of all data regions, i.e. $\phi \wedge \bigwedge_{i=1}^n \phi_i$ is satisfiable. An inconsistent node does not represent any concrete states. An ART node is an error node if some control location l_i is an error location and the data regions of the node are consistent, i.e. $\phi \wedge \bigwedge_{i=1}^n \phi_i$ is satisfiable.

A state $c = \langle \langle l_1, \mathbf{V}_1 \rangle, \ldots, \langle l_n, \mathbf{V}_n \rangle \rangle$ is covered by an ART node $\eta = \langle \langle l'_1, \phi_1 \rangle, \ldots, \langle l'_n, \phi_n \rangle, \phi \rangle$, denoted by $c \models \eta$, if for all $i \in [1, n]$, $l_i = l'_i$ and $\mathbf{V}_i \models \phi_i$ and $\wedge_{i=1}^n \mathbf{V}_i \models \phi$. A node can also be covered by another one, as defined in the following.

Definition 4.1.3 (Node covering) *Given two ART nodes* $\eta = \langle\langle l_1, \phi_1\rangle, ..., \langle l_n, \phi_n\rangle, \phi\rangle$, $\eta' = \langle\langle l'_1, \phi'_1\rangle, ..., \langle l'_n, \phi'_n\rangle, \phi'\rangle$, *we say* η *is covered by* η', *if the following conditions hold:*

1. $l_i = l'_i$,
2. *the implication* $\phi_i \Rightarrow \phi'_i$ *is valid for all* $i \in [1, n]$, *and*
3. *the implication* $\phi \Rightarrow \phi'$ *is valid.*

Intuitively speaking, a node η is covered by another one η', if the set of states approximated by η is a subset of the states approximated by η'. Moreover, the possible successors of η are also reachable from η'. Then it is safe to stop the exploration from η. An ART is complete if all the nodes are either fully expanded or covered. An ART is safe if it is complete and contains no error nodes.

4.1.2 Main verification algorithm

The main verification algorithm is shown in Algorithm 2. It takes a BIP model as input, and explores an over-approximation of all the possible reachable states by constructing an ART. When it terminates, it either returns a safe ART, concluding the model is safe, or reports a counterexample. The termination is not guaranteed in general [1, 56].

The ART construction proceeds with expanding the ART nodes. Upon expanding a node, it first checks if this node indicates an error. If an error node is detected, it generates a counterexample path by invoking function *BuildCEX* and checks if the counterexample path represents a real trace or not. If the counterexample is real, then the algorithms reports the counterexample and stops the analysis. Otherwise, it refines the abstraction by function *Refine* and continues the ART construction after refining the abstraction successfully. Then if the node is not an error not, it checks if this node can be covered by a previous explored node. If it can be covered, the algorithm stops the expansion from this node and marks it as covered, and proceeds some other uncovered nodes. Finally, a node is expanded if it is neither an error nor covered by another one. To expand a node, it first computes all the possible enabled interactions by function *EnabledInteraction*, and then computes the successor nodes by function *Expand*. All the consistent successor nodes are inserted in the ART to be expanded later.

We elaborate the node expansion and counterexample guided abstraction refinement in details in the subsequent sections.

4.1.3 Node expansion

To expand an ART node, the set of structurally enabled interactions is first computed by the function *EnabledInteraction* in Algorithm 2. We say that an interaction $\gamma = \langle g, P, f\rangle$ is structurally enabled in an ART node $\eta = \langle\langle l_1, \phi_1\rangle, ..., \langle l_n, \phi_n\rangle, \phi\rangle$, if for each component B_i such that $P \cap P_i = \{p_i\}$, there is a transition $\langle l_i, g_i, p_i, f_i, l'_i\rangle \in \mathbb{E}_i$, which starts from l_i and is labeled by the involving port p_i. Basically this computation extracts the control location of each

Algorithm 2 Lazy abstraction of BIP

Input: a BIP model $\mathscr{M}_{\text{BIP}}$ and an error state
Output: either $\mathscr{M}_{\text{BIP}}$ is safe, or a counterexample *cex*

```
1: create an ART node η_0 for the initial state
2: create an ART 𝕋 with the root η_0
3: create a worklist wl of ART nodes
4: push η_0 into wl
5: while wl ≠ ∅ do
6:     η ← pop(wl)
7:     if η is an error node then
8:         cex ← BuildCEX(η)
9:         if cex is real then
10:            return cex
11:        else
12:            Refine(𝕋, cex)
13:    else if η is covered then
14:        mark η as covered
15:    else
16:        Γ_enab ← EnabledInteraction(η)
17:        Expand(η, Γ_enab)
18:        push all children of η into wl
19: return M_BIP is safe
```

component from the given node, and then looks up the possible transitions starting from these control locations, and marks them as structurally enabled transitions. Then if all the participating transitions of an interaction are structurally enabled, we say this interaction is structurally enabled.

Notice that the structural enabledness on an ART node is different from the interaction enabledness on a BIP state in Section 2.3. We do not check the satisfiability of the interaction or transition guards on the ART node. It is safe to do so when we are doing lazy abstraction: if an interaction is disabled on the ART node due to the unsatisfiability of guards, then the successor node will be inconsistent, i.e. the conjunction $\wedge_{i=1}^{n}\phi'_i \wedge \phi'$ is unsatisfiable in the successor node. Thus, a disabled interaction will lead to an inconsistent successor node, which will be later discarded.

Suppose the set of enabled interactions on node η is $\Gamma_{enab} = \{\gamma_1, \ldots, \gamma_k\}$, then for each $i \in [1,k]$, $\gamma_i = \langle g_i, P_i, f_i\rangle$, $P_i = \{p_1, ..., p_l\}$, and for each $p \in P_i$, $\langle l, g, p, f, l'\rangle \in \mathbb{E}_{id(p)}$, we denote g, f by g_p and f_p respectively. We expand the node η and create a successor node $\eta' = \langle\langle l'_1, \phi'_1\rangle, ..., \langle l'_n, \phi'_n\rangle, \phi'\rangle$ for interaction γ_i, according to the following rule:

1. $\phi'_j = \widehat{post}(\phi_i \wedge \phi, o\hat{p}_j)$, for each $j \in [1,n]$, such that $P_i \cap P_j = \{p_j\}$ and $o\hat{p}_j$ = if $g_i \wedge g_{p_j}$ then $f_i; f_{p_j}$,
2. $\phi'_j = \phi_j$, for $j \in [1,n]$, such that $P_i \cap P_j = \emptyset$,

3. $\phi' = \widehat{post}(\phi, o\hat{p})$, $o\hat{p}$ = if $g_i \wedge \bigwedge_{p \in P_i} g_p$ then $f_i; f_{p_1}; ...; f_{p_l}$,

Given a set of enabled interactions Γ_{enab}, the pseudo-code of the node expansion is shown in Algorithm 3. For each enabled interaction $\gamma \in \Gamma_{enab}$, we create a new ART node η' as the successor of current node η. For each component B_i, that participates in γ, we invoke function *ExtractTransition*$(\mathbb{E}_i, l_i, p_i)$ to extract the participating transition starting from l_i and labelled by port p_i from the set of transitions $\mathbb{E}_i$. We omit the details of this function, since it is simply a membership check. Then we compute the new abstract data region ϕ'_i by applying the abstract strongest post-condition $\widehat{post}(\phi_i \wedge \phi, o\hat{p}_j)$. For other components, which do not participate in this interaction, their abstract data regions and control locations remain the same. To update the global region, we need to consider all the participating transitions, since they may modify variables in data transfer and change the global data region. We create two temporary variables g' and op', where variable g' is the conjunction of the interaction guard and all the participating transition guards, and op' is the sequential composition of the data transfer and all the participating transitions. Notice that data transfer should always be executed before all the participating transitions, but the execution order of component transitions does not matter, since they only access the local variables in the components. The new global region ϕ is then updated by applying the abstract strongest post-condition $\widehat{post}(\phi, o\hat{p})$, where $o\hat{p}$ is the guarded operation composed of g' and op'. Finally, if all the abstract strongest post-condition computations succeed, the new ART node η' is inserted and the edge is labeled by interaction γ. Otherwise, this new successor node η' is inconsistent and discarded.

Algorithm 3 Node expansion

```
1:  procedure EXPAND(η = ⟨⟨l_1, φ_1⟩, ..., ⟨l_n, φ_n⟩, φ⟩, Γ)
2:      for γ = ⟨g, P, f⟩ ∈ Γ do
3:          η' ← ⟨⟨l''_1, φ'_1⟩, ..., ⟨l''_n, φ'_n⟩, φ'⟩
4:          g' ← g
5:          op' ← f
6:          for each B_i in M_BIP do
7:              if P_i ∩ P = {p_i} then
8:                  ⟨l_i, g_i, p_i, f_i, l'_i⟩ ← ExtractTransition(E_i, l_i, p_i)
9:                  g' ← g' ∧ g_i
10:                 op' ← op'; f_i
11:                 ôp_i ← g_i; f_i
12:                 ⟨l''_i, φ'_i⟩ = ⟨l'_i, post^(φ ∧ φ_i, ôp_i)⟩
13:                 if ¬φ'_i then
14:                     break
15:             else if P_i ∩ P = ∅ then
16:                 ⟨l''_i, φ'_i⟩ = ⟨l_i, φ_i⟩
17:         ôp ← g'; op'
18:         φ' = post^(φ, ôp)
19:         if ¬φ' then
20:             break
21:         AddChild(η, γ, η')
```

4.1.4 Abstraction refinement

If an error node is encountered during the ART construction in Algorithm 2, function *BuildCEX* is called to construct a counterexample path by backtracking the ART from the current error node to the root node. In BIP, we denote a counterexample *cex* by the sequence of interactions, labeling the path in the ART from the root to the error node.

To check if the counterexample *cex* is real or not, we first construct a sequential execution trace tr_{cex}. Suppose the counterexample is $cex = \gamma_1\gamma_2\ldots\gamma_k$, where for each $i \in [1,k]$, interaction $\gamma_i = \langle g_i, P_i, f_i\rangle$, $P_i = \{p_1^i,\ldots,p_t^i\}$, $tr_{\gamma_i} = g_i; f_i; f^i_{\pi(1)};\ldots; f^i_{\pi(t)}$, where π is an arbitrary permutation in $\{1,\ldots,t\}$, and $f^i_{\pi(j)}$ is the operation of transition labeled by port $p^i_{\pi(j)}$. The trace of counterexample is the sequential composition of all tr_{γ_i}, i.e. $tr_{cex} = tr_{\gamma_1};\ldots; tr_{\gamma_k}$. We say that the counterexample *cex* is real if and only if the first order encoding of tr_{cex} is satisfiable, or equivalently $post(true, trace_{cex}) \neq false$. Otherwise, the counterexample is spurious.

If a spurious counterexample is found, we must eliminate the spurious counterexample and refine the abstraction. Our computes a sequent interpolant from the first order encoding of tr_{cex} [88, 119], and extract the predicates from the interpolant and use them to refine the abstraction (function *Refine* in Algorithm 2).

4.1.5 Correctness proof

In order to prove the correctness of lazy abstraction algorithm for BIP, we need to relate the construction of ART with BIP operational semantics. We first show that the node expansion in Algorithm 3 creates successor nodes, which safely over-approximate the corresponding reachable states in BIP operational semantics.

Lemma 4.1.4 *Let η be an ART node of a BIP model $\mathcal{M}_{\text{BIP}}$ and η' be a successor of η following interaction γ, and let c be a concrete state such that $c \models \eta$, then for every concrete state c' such that $c \xrightarrow{\gamma} c'$, we have $c' \models \eta'$.*

Proof 4.1.5 *Assume $c = \langle\langle l_1, \mathbf{V}_1\rangle,\ldots,\langle l_n, \mathbf{V}_n\rangle\rangle$, and $\eta = \langle\langle l_1,\phi_1\rangle,\ldots,\langle l_n,\phi_n\rangle,\phi\rangle$, we have $\mathbf{V}_i \models \phi_i$, for each $i \in [1,n]$, and $\bigwedge_{i=1}^n \mathbf{V}_i \models \phi$, since $c \models \eta$. Assume also the successor of c following $\gamma = \langle g, P, f\rangle$ is $c' = \langle\langle l'_1, \mathbf{V}'_1\rangle,\ldots,\langle l'_n, \mathbf{V}'_n\rangle\rangle$, and the successor of node η is $\eta' = \langle\langle l''_1,\phi'_1\rangle,\ldots,\langle l''_n,\phi'_n\rangle,\phi'\rangle$. In order to prove $c' \models n'$, we have to show that $l'_i = l''_i$ and $\mathbf{V}'_i \models \phi'_i$, for all $i \in [1,n]$, and $\bigwedge_{i=1}^n \mathbf{V}'_i \models \phi'$.*

Consider the component B_i, such that $P \cap P_i = \{p_i\}$, i.e. component B_i participates the interaction γ, and let the participating transition in $\mathbb{E}_i$ be $\langle l_i, g_i, p_i, f_i, l'_i\rangle$. Then we have $\mathbf{V}_i \models g_i$ and $\mathbf{V}'_i = f_i(f(\mathbf{V}_i))$ according to the operational semantics of BIP.

Then according to Algorithm 3, we have $l''_i = l'_i$ and $\phi'_i = \widehat{post}(\phi_i \wedge \phi, o\hat{p}_i)$, where $o\hat{p}_i$ denotes the sequential composition $g_i; f; f_i$. Based on the semantics of abstract strongest post-condition, and

the fact that $\mathbf{V}_i \models \phi_i$ *and* $\phi_i \wedge g_i$ *is satisfiable, we have* $\mathbf{V}'_i \models \phi'_i$. *Following a similar argument, we can prove* $\bigwedge_{i=1}^{n} \mathbf{V}'_i \models \phi'$.

For each component B_i *such that* $P \cap P_i = \emptyset$, *since it does not participate the interaction, its state is unchanged. Thus, the satisfaction relation trivially holds.*

Then the following theorem states the correctness of our lazy abstraction algorithm for BIP.

Theorem 4.1.6 *Given a BIP model* $\mathcal{M}_{\text{BIP}}$ *and an error state encoding the invariant property, for every terminating execution of Algorithm 2, we have the following properties:*

1. *if Algorithm 2 returns a counterexample, then there is a concrete trace from an initial state to an error state in* $\mathcal{M}_{\text{BIP}}$;
2. *if Algorithm 2 returns a safe ART, then for every reachable state of* $\mathcal{M}_{\text{BIP}}$, *there is an ART node that covers it.*

Proof 4.1.7 *Suppose Algorithm 2 returns a counterexample* $cex = \gamma_1\gamma_2\ldots\gamma_k$, *then according to the counterexample analysis presented in Section 4.1.4, we know that* $post(true, trace_{cex})$ *is satisfiable, which means the counterexample cex represents a concrete trace in* $\mathcal{M}_{\text{BIP}}$.

Suppose Algorithm 2 returns a safe ART, we prove that for every reachable state c, there is an ART node η *that covers c. The proof is by induction on the length of trace from the initial state to c. The base case holds trivially since we create the initial node from the initial state.*

Assume the conclusion holds for all traces of length n, i.e. if $c_0 \xrightarrow{\gamma_1\cdots\gamma_n} c_n$, *then there is an ART node* η *that covers state* c_n, *Now suppose state c is reachable by a trace of length* $n+1$, *i.e.* $c_0 \xrightarrow{\gamma_1\cdots\gamma_n} c_n \xrightarrow{\gamma_{n+1}} c$, *because* $c_n \models \eta$, *and based on Lemma 4.1.4, we conclude that the successor node* η' *of* η *following interaction* γ_{n+1} *also covers c. This concludes the proof of the theorem.*

4.2 Persistent set reduction for BIP

In lazy abstraction, all interleavings of concurrent interactions are explored, which may result in visiting some redundant states. Considering the example in Figure 4.1, in the initial state (S, I_1, I_2) (only show control locations for simplicity), both interactions $\{try_1\}$ and $\{try_2\}$ are enabled, and explored in lazy abstraction. However, both interleavings $\{try_1\};\{try_2\}$ and $\{try_2\};\{try_1\}$ lead to the same final state (S, W_1, W_2). These interleavings can be seen as equivalent, since the intermediate state is of no interest to our verification (i.e. wether (C_1, C_2) is reachable or not). It is thus preferable to explore only one interleaving of the two.

In this section, we present the persistent-set-based partial order reduction for BIP, that aims at selecting one representative interaction interleaving out of all equivalent ones in state space exploration. First of all, we introduce the following definition of interaction independence.

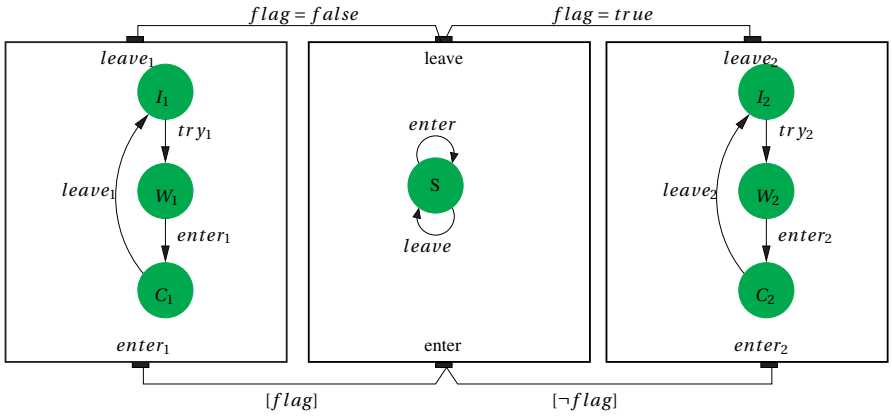

Figure 4.1 – Example for illustrating partial order reduction for BIP

Definition 4.2.1 (Interaction independence) *Two interactions γ_1 and γ_2 are independent in state c, if the following condition hold:*

1. *if γ_1 is enabled in c, then γ_2 is enabled in c iff γ_2 is enabled in c', where $c \xrightarrow{\gamma_1} c'$; and symmetrically for the case if γ_2 is enabled in c.*
2. *if γ_1 and γ_2 are both enabled in c, and $c \xrightarrow{\gamma_1\gamma_2} c'_1$, $c \xrightarrow{\gamma_2\gamma_1} c'_2$, then $c'_1 = c'_2$.*

Independence relation in the above definition is a global property, and in order to check if two interactions are independent or not, we have to look at every possible state in the state space of the transition system. Hence, computing the precise independence relation may be as hard as the invariant verification problem. Static analysis of the system model is usually used to obtian an under-approximation of the independence relation.

We remark that the above defines a conditional independence relation, as in [78, 147], i.e. two interactions are defined as independent with respect to a single state. This definition works well with explicit-state model checking, where individual concrete states are visited and checked. However, in this dissertation, we aim at applying partial order reductions to abstraction structures (e.g. ART), where the conditional independence may not hold. Consider the interactions with this two actions $x := z + y$ and $x := z$, they are independent on state $(x = 1, y = 0, z = 1)$ (actually any state with $y = 0$). Now suppose the predicate we use for abstraction is $x = z$, the current abstract state then $x = z$, on which it is unclear whether $y = 0$ or not. Thus, it is unable to conclude whether the two actions are independent or not.

Furthermore, independent interactions do not commute on abstract states. For instance, consider a BIP model with only two components and the following two simple interactions $\gamma_1 = \langle true, \{p_1\}, x_1 = x_1 + 1\rangle$ and $\gamma_2 = \langle true, \{p_2\}, x_2 = x_2 + 1\rangle$, that is, each of the two components increment a local integer variable by 1. It is obvious they are independent in the concrete transition system, and from the initial state $(x_1 = 0, x_2 = 0)$, the two interleavings $\gamma_1;\gamma_2$ and $\gamma_2;\gamma_1$ will lead to the same state $(x_1 = 1, x_2 = 1)$. Now suppose the predicate language of the abstraction structure is given by $b_1 = (x_1 > x_2)$, $b_2 = (x_1 = x_2)$ and $b_3 = (x_1 = x_2 + 1)$ and their

negations. Then starting from the initial abstract state $\neg b_1 \wedge b_2 \wedge \neg b_3$, interleaving $\gamma_1;\gamma_2$ leads to an abstract state $\neg b_1 \wedge b_2 \neg b_3$ (with an intermediate state $b_1 \wedge \neg b_2 \wedge b_3$), while another interleaving $\gamma_2;\gamma_1$ leads to a different abstract state $\neg b_1 \wedge \neg b_3$ (with an intermediate state $\neg b_1 \wedge \neg b_2 \wedge \neg b_3$).

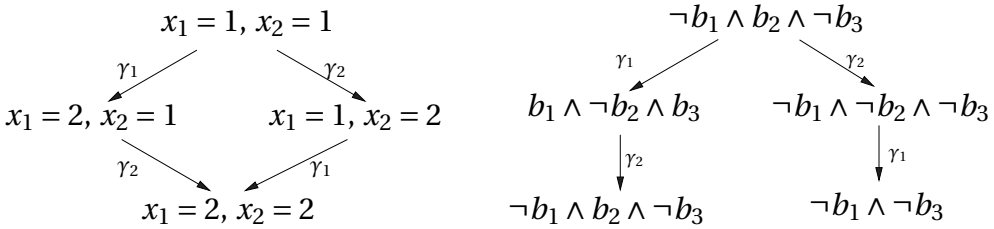

Figure 4.2 – Example showing independent interactions don't commute on abstract states

Thus, we cannot simply lift the Deifinition 4.2.1 to abstraction structures. These concerns motivate the following definition of abstract independence. Recall that β is the concretisation function that maps an abstract state to the concrete ones.

Definition 4.2.2 (Abstract independence) *Two interactions γ_1 and γ_2 are independent in an abstract state η, if for all states $c \in \beta(\eta)$, the following conditions hold:*

1. *if γ_1 is enabled in c, then γ_2 is enabled in c iff γ_2 is enabled in c', where $c \xrightarrow{\gamma_1} c'$; and symmetrically for the case if γ_2 is enabled in c.*
2. *if γ_1 and γ_2 are both enabled in c, and $c \xrightarrow{\gamma_1\gamma_2} c'_1$, $c \xrightarrow{\gamma_2\gamma_1} c'_2$, then $c'_1 = c'_2$.*

We can see that independence in all states in Definition 3.2.1 implies abstract independence, and it is also preferable to use the independence in Definition 3.2.1, given the fact that it is easier to compute. In the sequel, by independent interactions we mean the interactions that are independent in all states.

The question is whether it is still sound if we only explore one interaction sequence out of two interleavings in the abstract analysis, given two independent interactions. Our first observation is that in lazy abstraction, since we keep track of the control locations, the set of outgoing interactions in each ART node will always contain the set of interactions in each concrete system state represented by this node. The second observation is that though independent interactions do not commute on abstract states, they still commute on the reachable concrete states represented by the abstract states. The following lemma formalizes this observation and ensures that exploiting independence on abstraction structures is still sound. Recall that $\widehat{post}$ is the abstract *post* operator.

Lemma 4.2.3 *Let γ_1 and γ_2 be two independent interactions, and let η be an abstract state, then for all $c \in \beta(\widehat{post}(\widehat{post}(\eta,\gamma_1)),\gamma_2)$, we have that if there is a state $c_1 \in \beta(\eta)$, such that $c = post(post(c_1,\gamma_1),\gamma_2)$, then $c \in \beta(\widehat{post}(\widehat{post}(\eta,\gamma_2)),\gamma_1)$.*

Proof 4.2.4 *Assume we have* $c = post(post(c_1, \gamma_1), \gamma_2)$*, since* γ_1 *and* γ_2 *are independent, then we also have* $c = post(post(c_1, \gamma_2), \gamma_1)$*. According to the semantics of* $\widehat{post}$*, it holds that* $c \in \beta(\widehat{post}(\widehat{post}(\eta, \gamma_2), \gamma_1))$.

Similarly, we can conclude that sequences of interaction interleavings that can be obtained from each other by permuting adjcent independent interactions are also equivalent. Exploring one interleaving sequence out of all equivalent ones would not miss any reachable concrete states. Thus, it is still sound to perform selective search on abstraction structures.

Then the question is how we can select the set of interactions to be explored on an abstract state. For this purpose, we define the persistent set as in Definition 3.2.10, but on an abstract state.

Definition 4.2.5 *A set of interactions* Γ_{per} *in an abstract state* η *is persistent if the following conditions hold:*

1. $\Gamma_{per} \subseteq en(\eta)$ *and* $\Gamma_{per} = \emptyset$ *if and only if* $en(\eta) = \emptyset$*;*
2. *for every execution* $\eta \xrightarrow{\gamma_1 \dots \gamma_n} \eta_n$*, where* $\gamma_i \notin \Gamma_{per}$, $i \in [1, n]$, γ_n *is abstractly independent with all interactions in* Γ_{per}*;*
3. *for every execution* $\eta = \eta_0 \xrightarrow{\gamma_1 \dots \gamma_n} \eta_n$*, where* η_n *is implied by some* η_i, $i \in [0, n-1]$*, and for every* $\gamma \in en(\eta_j)$, $j \in [1, n]$*, there is* $k \in [1, n]$ *such that* $\gamma \in \Gamma_{per}(\eta_k)$.

The first two conditions ensure the deadlocks are preserved, and the third one is required when reasoning about the general safety properties.

A persistent set in an abstract state may not be persistent on the represented concrete states, since an interaction enabled on an abstract state may not be enabled on some represented concrete states. However, the persistent set in an abstract state is a safe over-approximation.

4.2.1 Combining persistent set reduction with lazy abstraction

In order to combine the persistent set reudction with lazy abstraction for BIP, we incorporate the selective search in the abstract reachability analysis of lazy abstraction. The new algorithm is listed in Algorithm 4. It constructs a reduced ART in a similar way to Algorithm 2. More specifically, when we expand an ART node in the abstract reachability analysis, instead of computing successor nodes for all possible enabled interactions, we only compute the ones that follow the interactions in the persistent set. The exploration of the interactions outside of the persistent set are postponed.

To solve the ignoring problem illustrated in Section 3.2.4, the new algorithm will also have to detect if a cycle occurs, before expanding an ART node. We say a cycle occurs, if the control locations of a node have been visited before in the ART path to this node. In case a cycle is

Algorithm 4 Lazy abstraction with persistent set reudction for BIP

Input: a BIP model $\mathscr{M}_{\text{BIP}}$ and an error state
Output: either $\mathscr{M}_{\text{BIP}}$ is safe, or a counterexample *cex*

```
1: create an ART 𝕋 with initial node η0
2: create a worklist wl and push η0 into wl
3: while wl ≠ ∅ do
4:     η ← pop(wl)
5:     if IsError(η) then
6:         cex ← BuildCEX(η)
7:         if cex is real then
8:             return cex
9:         else
10:            Refine(𝕋, cex)
11:    else if Cycle(η) then
12:        FullyExpand(Predecessor(η)), and add all successors into wl
13:        mark η as covered
14:    else if Covering(η) then
15:        mark η as covered
16:    else
17:        Γ_P ← PersistentSet(η)
18:        Expand(η, Γ_P), and add all successors into wl
19: return M_BIP is safe
```

detected, the predecessor of this node will be fully expanded by function *FullyExpand* to avoid some interactions are postponed forever. Basically, in the new algorithm, we expand the set of interactions, which are outside of the persistent set of this node. This is a stronger guarantee that implies the second condition of persistent set in Definition 4.2.5. Detailed elaboration of techniques for solving the ignoring problem can be found in [65].

If no cycle is detected, then the new algorithm computes the set of selected interactions using the function *PersistentSet*. We elaborate the implementation details of this function in the next subsection. Then the node is expanded according to interactions in the persistent set and all the consistent successors are added into the ART. We remark that the actual implementation of persistent set computation does not affect the integration. Any optimization or new persistent set computation implementations can be easily incorporated without jeopardizing the correctness of the algorithm.

The following theorem states the correctness of the new algorithm.

Theorem 4.2.6 *Given a* BIP *model* $\mathscr{M}_{\text{BIP}}$ *and an error state encoding the invariant property, for every terminating execution of Algorithm 4, we have the following properties:*

1. *If a counterexample is returned, then it is concrete counterexample in* $\mathscr{M}_{\text{BIP}}$*;*
2. *If a safe ART is returned, then for every reachable state* c *in* $\mathscr{M}_{\text{BIP}}$*, there is an ART node* η *such that* $c \models \eta$*.*

Proof 4.2.7 *Item 1 holds for the same argument with Theorem 4.1.6. In the following, we prove the second item.*

According to Theorem 4.1.6, for every reachable state c in $\mathscr{M}_{\mathrm{BIP}}$, there is a node η^w in the ART returned by Algorithm 2 such that $c \models \eta^w$. Suppose the path to node η^w is $\eta_0 \xrightarrow{\gamma_1 \dots \gamma_n} \eta_n$, where η_0 is the root and $\eta_n = \eta^w$.

Now we prove that there is another path in the ART returned by Algorithm 4, $\eta_0 \xrightarrow{\gamma'_1 \dots \gamma'_n} \eta'_n$, such that γ'_1 is in the persistent set $\Gamma_{per}(\eta_0)$ and $c \models \eta'_n$.

Case 1 : if η^w is a deadlock node, then we show that at least one interaction γ_i, $i \in [1,n]$ in the path $\eta_0 \xrightarrow{\gamma_1 \dots \gamma_n} \eta_n$ is in the persistent set $\Gamma_{per}(\eta_0)$. Otherwise, suppose that none of interactions $\gamma_i, i \in [1,n]$ is in the persistent set $\Gamma_{per}(\eta_0)$, then by the second condition of persistent set in Definition 4.2.5, the interactions in persistent set $\Gamma_{per}(\eta_0)$ will still be enabled in η_n, which controdicts the assumption that η^w is a deadlock node.

Thus, there is at least one interaction $\gamma_i, i \in [1,n]$ in the persistent set $\Gamma_{per}(\eta_0)$. Assume the first such interaction is $\gamma_j, j \in [1,n]$, then for all interactions $\gamma_k, k < j$, we have γ_j is independent with γ_k. Thus, γ_j can be moved to the beginning of the path. The new path would be the same one with $\gamma_1 \dots \gamma_n$, except γ_j has been moved to the first.

Then applying 4.2.3, we can conclude that $c \models \eta'_n$.

Case 2 : if η^w is not a deadlock node, but covered by some other node in the same path, assume the covering node is η_i, $i \in [0, n-1]$. Assume also no interactions in the persistent set $\Gamma_{per}(\eta_0)$ occur in γ_j, $j \in [1,i]$. Then we know that interactions in $\Gamma_{per}(\eta_0)$ are also enabled in η_i and η_n as well. Then according to the third condition of persistent set in Definition 4.2.5, we know that at least one interaction in $\Gamma_{per}(\eta_0)$ occurs in γ_j, $j \in [i+1,n]$. Let γ_k, $k \in [i+1,n]$ be the first such interaction, then for the same reason as the case 1, γ_k can be shifted to the beginning of the path. The new path would be the same one with $\gamma_1 \dots \gamma_n$, except γ_k has been moved to the first.

Then applying 4.2.3, we can conclude that $c \models \eta'_n$.

Case 3 : if η^w is covered by some node in another path. It is equivalent to prove the conclusion for another path. Since our models are finite branching, we are guaranteed that there is a path such that argument in case 2 applies.

Case 4 : if η^w is not covered, then it is possible to extend the path, where case 2 or case 3 applies.

4.2.2 Computing persistent set

In this section, we present an algorithm to compute a persistent set in an ART node by means of static analysis of BIP model. This algorithm is an actural implementation of the function *PersistentSet* in Algorithm 4.

The basic idea is to find static criteria for selecting persistent set that can be checked efficiently by a syntactic analysis of the high-level formal description of the system. It is static in the sense

the persistent set is constructed on the basis of the current state, without knowing its future states. This is important, since the future states are not known when the state is expanded.

First, we elaborate how to obtain the dependence relation $\mathscr{D}$ of interactions. We compute the dependence relation statically from the control flow of the system model a priori: two interactions are *dependent* if they share a common component. This will give us an over-approximation of the dependency relation $\mathscr{D}$. That is, if $(\gamma_1, \gamma_2) \in \mathscr{D}$, then γ_1 and γ_2 are considered as dependent, though they may be independent in the reachable state space. We take the complement of $\mathscr{D}$ as an under-approximation of the independence relation.

For simplicity, given an interaction γ, we denote in the sequel by $\mathscr{D}_\gamma$ the set of interactions that are dependent with γ, and we denote by $\mathscr{I}_\gamma$ the compliment of $\mathscr{D}_\gamma$, i.e. the set of interactions that are independent with γ.

We now introduce the definition of an enabling set for a disabled interaction in an ART node.

Definition 4.2.8 (Enabling set) *Let γ be a disabled interaction in an ART node η, an enabling set for γ in η is a set of interactions $\mathscr{N}_\gamma$, such that for all sequences of interactions $\eta \xrightarrow{\gamma_1 \cdots \gamma_n} \eta' \xrightarrow{\gamma}$, there is at least one interaction $\gamma_i \in \mathscr{N}_\gamma$, $i \in [1, n]$.*

An enabling set of a disabled interaction in an ART node characterizes the interactions that may interfere with the disabled interaction in the control flow. A disabled interaction can be taken only if some interactions in its enabling set are taken first.

To compute an enabling set for a disabled transition, we use a fine-grained static analysis. Formally, given a disabled interaction $\gamma = \langle g, P, f \rangle$ in an ART node $\eta = \langle \langle l_1, \phi_1 \rangle, ..., \langle l_n, \phi_n \rangle, \phi \rangle$, for each component B_i such that $P \cap P_i = \{p_i\}$, and there is no such an outgoing transition $(l_i, g_i, p'_i, f_i, l'_i) \in \mathbb{E}_i$ that $p'_i = p_i$, then we say another interaction $\gamma' = \langle g', P', f' \rangle$ is in the enabling set $\mathscr{N}_\gamma$ of γ, if $P' \cap P_i = \{p'_i\}$, and there is a path in B_i from l_i to a control location, where p_i is an outgoing transition.

Finally, we present the algorithm for computing a persistent set in Algorithm 5. The algorithm builds a persistent set incrementally by making sure the following conditions hold:

1. Γ_{stub} contains at least one enabled interaction if the set of enabled interactions on η is non-empty;
2. for each disabled interaction $\gamma \in \Gamma_{stub}$, then there is an enabling set $\mathscr{N}_\gamma$, such that $\mathscr{N}_\gamma \subseteq \Gamma_{stub}$;
3. for each enabled interaction $\gamma \in \Gamma_{stub}$, then $\mathscr{D}_\gamma \subseteq \Gamma_{stub}$.

The following theorem states that the set of enabled interactions in the returned set is indeed a persistent set.

Theorem 4.2.9 *Let $\Gamma_{stub}(\eta)$ be a set returned by Algorithm 5, and let Γ' be the set of all enabled interactions in $\Gamma_{stub}(\eta)$, then Γ' is a persitent set in the given ART node η.*

Algorithm 5 Persistent set computation

```
1:  procedure PERSISTENTSET(η, M_BIP)
2:      Γ_work = {γ} such that γ is enabled on η
3:      Γ_stub = ∅
4:      while Γ_work ≠ ∅ do
5:          pick some γ ∈ Γ_work
6:          Γ_work = Γ_work − γ, Γ_stub = Γ_stub ∪ {γ}
7:          if γ is enabled then
8:              Γ_work = Γ_work ∪ D_γ \ Γ_stub
9:          else
10:             N_γ = EnablingSet(γ, η, M_BIP)
11:             Γ_work = Γ_work ∪ N_γ \ Γ_stub
12:     return Γ_stub
```

Proof 4.2.10 *Suppose* Γ' *is not a persistent set on the ART node* η*, then there is a path* $\eta \xrightarrow{\gamma_1} \eta_1 \xrightarrow{\gamma_2} \eta_2 \ldots \xrightarrow{\gamma_n} \eta_n \xrightarrow{\gamma}$*, such that for all* $i \in [1,n]$, $\gamma_i \notin \Gamma'$*, and* γ *depends on some interaction* γ' *in* Γ'*.*

Assume γ *is enabled on* η*, then* γ *is also enabled on* η *and should be included in the set* Γ*, and* Γ' *as well, since it depends on* γ'*, Contradicting the assumption.*

Assume γ *is disabled on* η*, however, since it is enabled on* η_n*, there must be a nonempty enabling set for* γ *on the node* η*. Moreover, there is at least one interaction* $\gamma_j, 1 \le j \le n$ *in this enabling set, and according to the assumption,* γ_j *is disabled on* η*, otherwise* γ_j *should be in* Γ'*. Then by repeating the same reasoning, there is an interaction* $\gamma_{j'}, 1 \le j' < j$ *in the enabling set for* γ_j *and* $\gamma_{j'}$ *is diabled on* η*. In the end, we can conclude that* γ_1 *is in some enabling set and is disabled in* η*, which contradicts the assumption.*

Thus, Γ' *is indeed a persistent set.*

4.3 Experimental evaluation

We have implemented the proposed verification techniques for BIP based on the Kratos software model checker [36], the symbolic model checker nuXmv [33] and the SMT solver MathSAT5 [38]. To evaluate the performance of the proposed techniques, we carried out a comprehensive experimental evaluation, where we took a set of benchmarks from the literature [26], and modeled them in the BIP framework. The benchmarks include the ticket mutual exclusion protocol, the ATM transaction model, the leader election algorithm, and a Quorum consensus algorithm and the reactor temperature control system. For each benchmark, we also create a unsafe version with manually injected faults. All these benchmarks are infinite-state and scalable with respect to the number of components. In the experiments we create ten instantiations for each benchmark model. In total we have 120 models. The details of all the benchmark models are provided in the Appendix. All the experiments have been run on a

64-bit Linux PC with a 2.8 GHz Intel i7-2640M CPU, with a memory limit of 4Gb and a time limit of 300 seconds per benchmark.

In the experiments, we run the following two configurations of our prototype tool: 1) plain lazy abstraction, denoted by 'plain' in the plots; 2) lazy abstraction with persistent set reduction, denoted by 'pset' in the plots. For simplicity, we call lazy abstraction with persistent set reduction as persistent set reduction in the sequel. We compare them to two other infinite-state verification techniques implemented in nuXmv [33]: the state-of-the-art IC3 algorithm for software model checking [37] (IC3 in the sequel) and the implicit predicate abstraction model checking [138] (IPA in the sequel). For the translation from BIP to nuXmv, we refer to the encoding in 2. We measure both the running time of solving both safe and unsafe benchmarks, and the memory consumption for the three configurations of our prototype tool in terms of the number of created ART nodes. We do not compare the performance of our tool to DFinder [21] or the work [95], since they do not handle data transfer in interaction, or infinite-state models.

We present the evaluations in the following subsections. The detailed statistics data is attached in the Appendix A.5, and Appendix A.6.

4.3.1 Comparing lazy abstraction to persistent set reduction

In Figure 4.3, we compare the two configurations of our prototype tool, and show the scatter plots of time for solving each benchmark. [1] In the plots (and all the subsequent scatter plots), symbol × represents a safe benchmark, and ○ represents an unsafe benchmark. A point in the plots indicates the analysis time taken by the algorithms represented by x-axis and y-axis. From the plot, we can see that combining persistent set reduction improves the performance of lazy abstraction for both safe and unsafe benchmarks. However, the improvement is not significant.

In order to understand the performance of lazy abstraction and the impact of persistent set reduction, we collect the time used by each subroutine of the algorithms and compare them in the bar plots in Figure 4.4 and Figure 4.5. Each bar in the plot represents the total analysis time for a benchmark model, with different colors showing the time used by different subroutines. We only depict the results with total runtime greater than 1 second. The one in the left depicts the results with runtime greater than 10 seconds, and the one in the right depicts the results with runtime from 1 second to 10 seconds.

In plain lazy abstraction, the total runtime consists of the time of transfer function computation i.e. the abstract post image computation, the time of coverage check, and the time of counterexample analysis and refinement. The results are shown in Figure 4.4, from which we can see that the most expensive routines of lazy abstraction are the computation of transfer function and the coverage check.

1. Red diagonal guides provide a reference for comparison, each indicating shift of one order of magnitude.

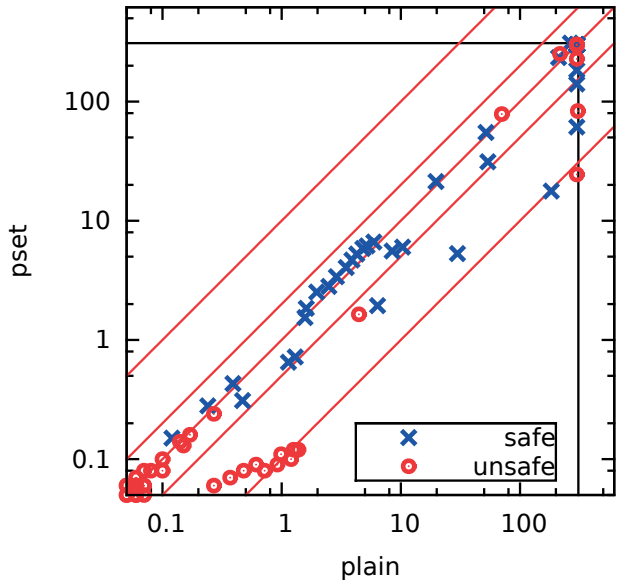

Figure 4.3 – Lazy abstraction vs. lazy abstraction with persistent set reduction

In lazy abstraction with persistent set reduction, it contains all the subroutines of lazy abstraction, and has additionally the time of persistent set computation and the time of cycle detection. The results are shown in Figure 4.5. We can see that in addition to the expensive subroutines of transfer function computation and coverage check, the cycle detection also contributes significantly to the total runtime. This subroutine is necessary for persistent set reduction to solve the ignoring problem. In fact, as also noticed in [142] recently, solving the ignoring problem using cycle detection is computationally expensive, since whenever a cycle is found, we have to fully expand all the postponed interactions, and visit a large number of redundant states. Mostly, it is the bottleneck and harm the power of partial order reduction.

We also measure the effect of partial order reduction as the percentage of successful reductions over the total number of attempts. That is, one ART node expansion accounts for a reduction attempt, and a successful reduction would explore only a strict subset of all the enabled interactions, while a failed reduction would explore all the enabled interactions. We only list the results for the solvable benchmarks, and for the simplicity of presentation, we abbreviate the list of entries that have 0 percentage from the same benchmark with only one entry. The result is shown in Table 4.1.

The result in Table 4.1 shows that persistent set reduction as presented in this dissertation has limited reduction power on the set of benchmarks we have. There are a number of benchmarks, where no successful reductions are achieved, i.e. the ticket mutual exclusion protocol and the reactor temperature control system, and the unsafe version of the railway control system. Except for the reason that the static persistent set is nonoptimal [2], another main reason

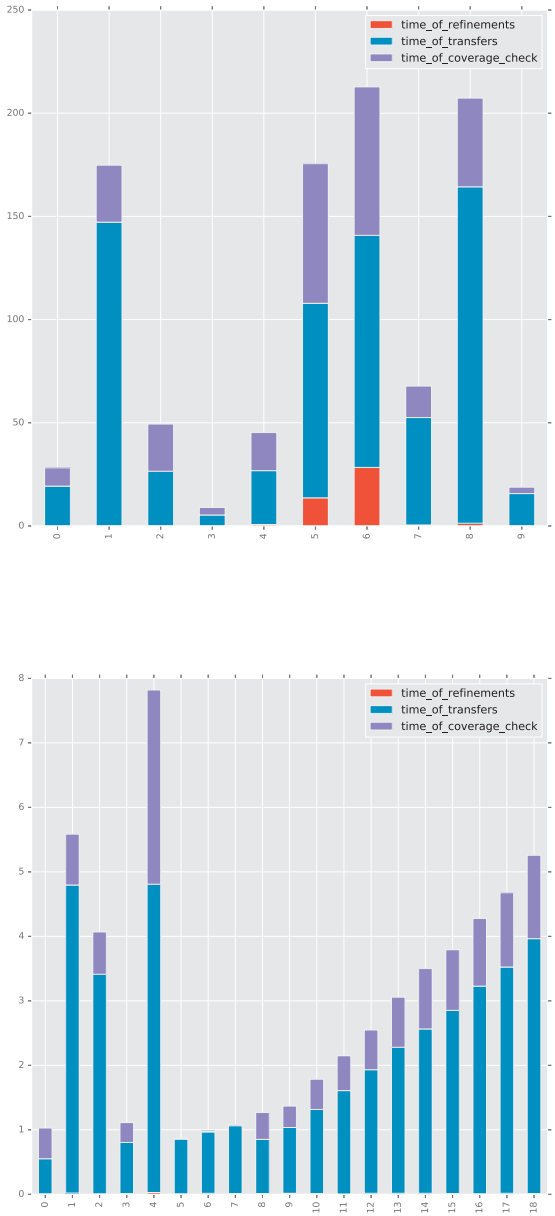

Figure 4.4 – Runtime of plain lazy abstraction subroutines

Figure 4.5 – Runtime of lazy abstraction with persistent set reduction subroutines

is that these models are highly synchronized, i.e. every two interactions have one common component involved. Thus, our interaction dependence analysis (we say two interactions are dependent if they share one common component) reports that all interaction are dependent, and no reductions are achieved on these benchmarks. This may also explain why the overall improvements of persistent set reduction are less significant, as shown in the scatter plot in Figure 4.3.

To summarize, we mainly evaluate the performance of lazy abstraction and the impact of persistent set reduction in this subsection. We find that persistent set reduction improves the abstract analysis, but at the same time, it brings the task of solving the ignoring problem, which harms the overall improvements. And the power of persistent set reduction is largely affected by the precision of the dependence analysis. A coarse dependence analysis is computationally cheap, but may give little reduction achievement. In our measurements, it can only reduce models that exhibit a high degree of concurrency and interleaving.

model	percentage	model	percentage
atm_safe_02	0.500000	atm_safe_03	0.492674
atm_safe_04	0.442058	atm_unsafe_02	0.402583
leader_election_safe_02	0.400000	leader_election_safe_03	0.543478
leader_election_safe_04	0.481553	leader_election_safe_05	0.410788
leader_election_unsafe_02	0.466667	leader_election_unsafe_03	0.555556
leader_election_unsafe_04	0.465347	leader_election_unsafe_05	0.395445
quorum_safe_02	0.297872	quorum_safe_03	0.327189
quorum_safe_04	0.320191	quorum_safe_05	0.285627
quorum_unsafe_02	0.400000	quorum_unsafe_03	0.400000
quorum_unsafe_04	0.400000	quorum_unsafe_05	0.400000
quorum_unsafe_06	0.400000	quorum_unsafe_07	0.400000
quorum_unsafe_08	0.400000	quorum_unsafe_09	0.400000
quorum_unsafe_10	0.400000	quorum_unsafe_11	0.400000
railway_control_safe_02	0.250000	railway_control_safe_03	0.380282
railway_control_safe_04	0.404762	railway_control_safe_05	0.368159
railway_control_safe_06	0.208363	railway_control_safe_07	0.166628
railway_control_safe_08	0.253109	railway_control_unsafe	0.000000
temperature_safe	0.000000	temperature_unsafe	0.000000
ticket_safe	0.000000	ticket_unsafe	0.000000

Table 4.1 – Percentage of persistent set reduction

4.3.2 Comparing to IC3 and IPA

In this subsection, we compare each of our configurations to both IC3 and IPA in terms of the running time for solving each benchmark. The results are shown in the following scatter plots.

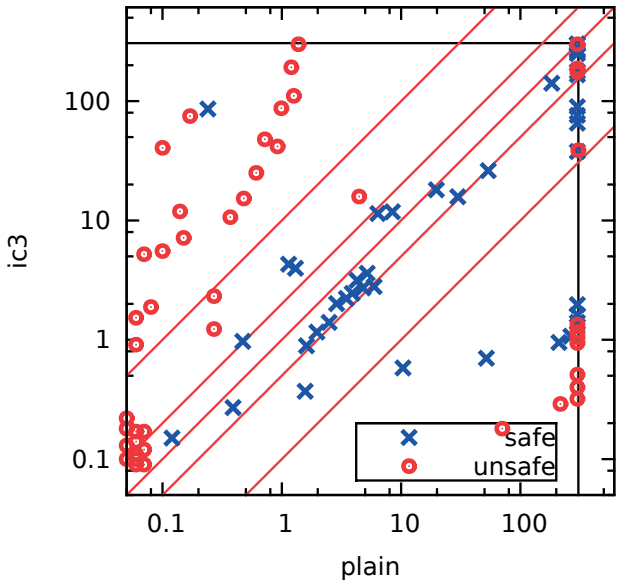

Figure 4.6 – Lazy abstraction vs. IC3

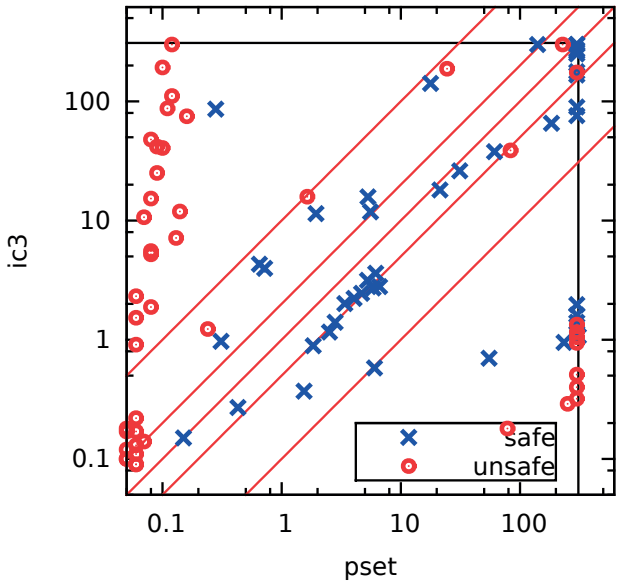

Figure 4.7 – Lazy abstraction with persistent set reduction vs. IC3

In Figure 4.6, and Figure 4.7, we show the comparisons with IC3. We can see that for unsafe benchmarks, persistent set reduction is faster than IC3 for most unsafe benchmarks, though there are some exceptions that persistent set reduction runs out of time. For safe benchmarks, our techniques are comparable to IC3, while there are a number of models that can be solved by IC3, but the other techniques run out of time.

In Figure 4.8, and Figure 4.9, we compare each of our techniques to IPA. For safe benchmarks, plain lazy abstraction is comparable to IPA and persistent set reduction perform slightly better. For unsafe benchmarks, our techniques always perform better. IPA is unable to solve any unsafe benchmarks.

4.3.3 Cumulative plots

In this subsection, we present the cumulative plots that indicate the number of benchmark models that can be solved by each technique (y-axis) within the given time (x-axis). In Figure 4.10, we plot the cumulative time of solving the benchmarks for all techniques. The plot shows that overall IC3 can solve more benchmark models within the time limits. In total, IC3 has solved 84 benchmark models, IPA has solved 25 models and plain lazy abstraction, persistent set reduction have solved 63, 68 models respectively.

In Figure 4.11 and Figure 4.12, we plot the cumulative time of solving safe and unsafe benchmarks respectively. The plot in Figure 4.11 shows that for safe benchmarks, IC3 is able to solve more benchmark models. The other techniques are comparable, while persistent set reduction performs slightly better. For most unsafe benchmarks, our techniques are faster than IC3. There are still some unsafe models that can be solved by IC3, while our techniques run out of time or memory. There is no data in Figure 4.12 for IPA, since it fails to solve all the unsafe benchmarks.

We also measure the memory usage of our techniques in terms of the size of ART. We collect the number of nodes that are needed to solve each benchmark model. The cumulative plots are shown in Figure 4.13, Figure 4.14 and Figure 4.15. They show that for solving the same amount benchmarks, plain lazy abstraction needs to create more ART nodes than the other one, thus consuming more memory usage.

4.4 Related work

Many work on algorithmic verification of safety properties can be found in literature. Below, we review the most closely related ones to this dissertation.

In a seminal paper on verification of infinite-state systems [1], the authors prove some general decidability results and propose a backward reachability analysis technique for safety verification, relying on the well-structured transition system framework [66]. In [56], the authors present a uniform forward reachability analysis procedure for infinite-state systems based on

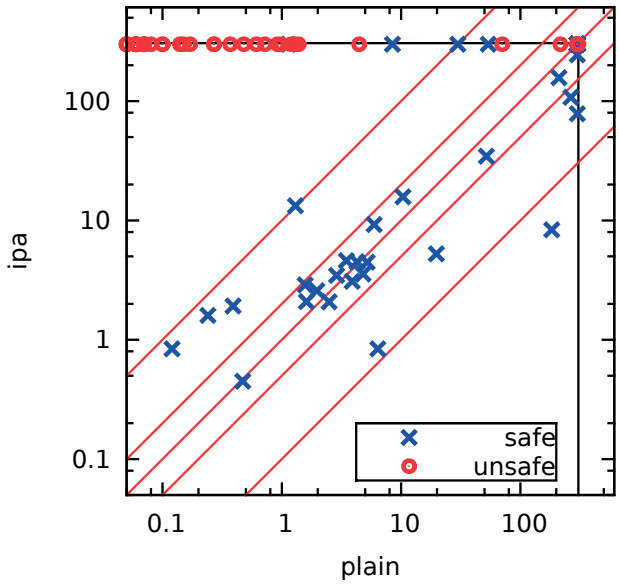

Figure 4.8 – Lazy abstraction vs. IPA

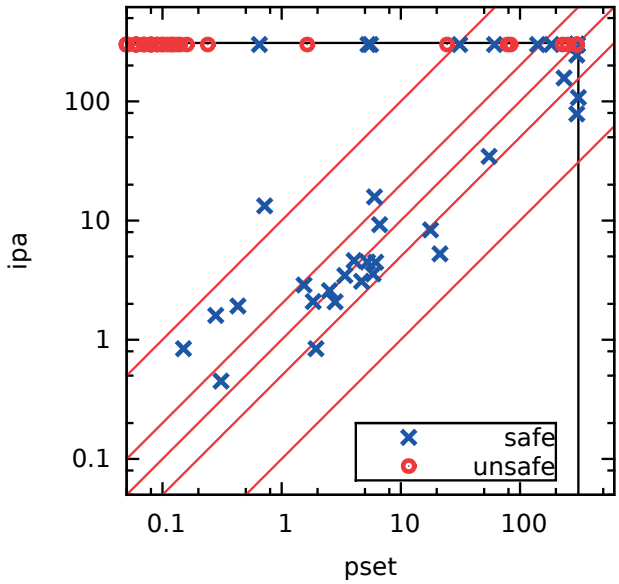

Figure 4.9 – Lazy abstraction with persistent set reduction vs. IPA

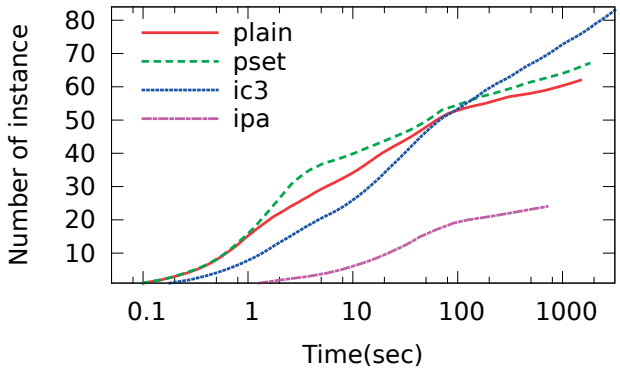

Figure 4.10 – Cumulative plot of time for all benchmarks

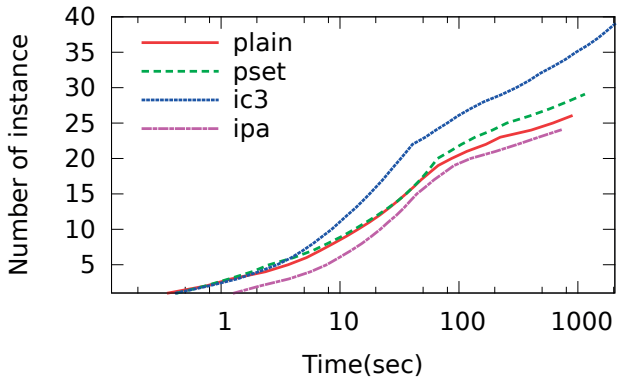

Figure 4.11 – Cumulative plot of time for safe benchmarks

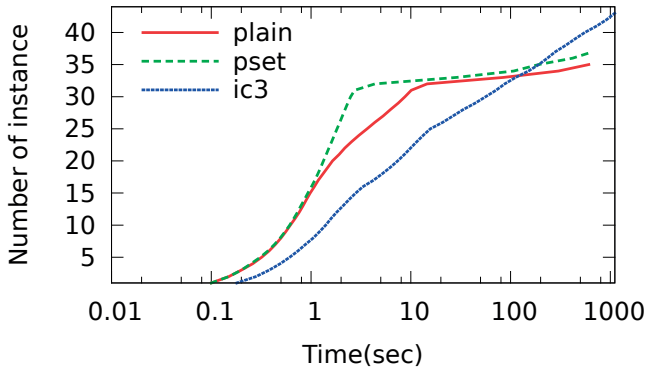

Figure 4.12 – Cumulative plot of time for unsafe benchmarks

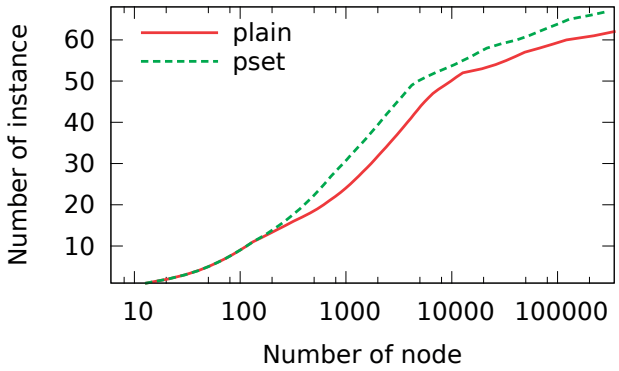

Figure 4.13 – Cumulative plot of ART size

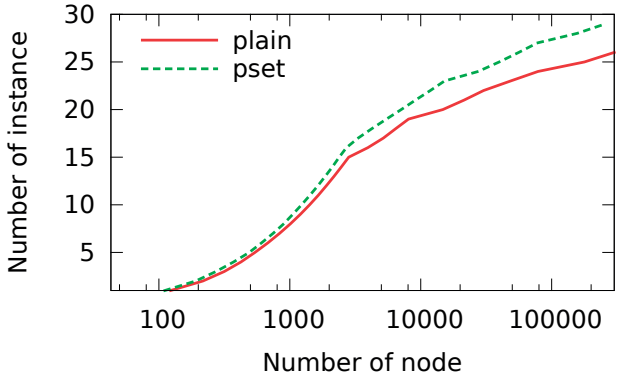

Figure 4.14 – Cumulative plot of ART size for safe benchmarks

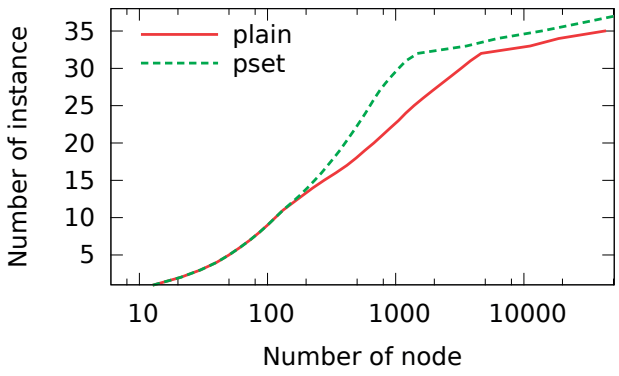

Figure 4.15 – Cumulative plot of ART size for unsafe benchmarks

the construction of a covering graph. Later in [90, 88], the authors present an abstract forward reachability analysis technique for sequential programs based on predicate abstraction and interpolation-driven abstraction refinement. However, these techniques can hardly scale for the concurrent system models we consider in this dissertation. The main reason is that they do not resolve the state explosion problem resulting from concurrency, which is one of the major obstacles for the verification of component-based systems. On the other hand, partial order reduction is the dedicated technique to deal with concurrency. Though several partial order reduction techniques are adopted to software verification [96, 68, 147, 102], they still suffer from the inefficiency of reasoning about arithmetic.

In the following, we roughly classify the techniques for efficient verification of infinite-state concurrent systems into the following categories: 1) combination of symbolic reasoning and explicit reductions; 2) compositional reasoning.

Attempts to combine partial order techniques with abstraction techniques for efficient verification of concurrent software have also been made in [36, 145]. In [36], the authors propose the Explicit Scheduler, Symbolic Thread (ESST) verification framework for multi-threaded programs with a preemptive and stateful scheduler. However, atomic synchronization among transitions is not supported. The work in [145] combines lazy abstraction with interpolant algorithm for sequential programs [119] and dynamic partial order reduction [102] for the verification of generic multi-threaded programs with pointers. They put the emphasis on shared-variable concurrency, and do not leverage the separation between coordination and computation, which is the core of our approach. Recently, in [86] the author considers combining abstraction technique with stubborn set reduction for CSP models. In [8], the authors combine ample-set-based partial order reduction with BDD-based symbolic model checking. But no abstraction is involved. In [87], the authors extends stubborn-set-based partial order reduction to real-time models with zone abstraction.

With respect to compositional reasoning, the most relevant works are [70, 89, 111, 29, 34, 83, 128]. In [70], the authors present a thread modular verification technique for multi-threaded programs, relying on the assume-guarantee style reasoning [101]. In this approach, each thread is verified against its environment assumption, which is the disjunction of the guarantees of all the other threads. The guarantee of each thread models all the possible global state update performed by this thread. Initially the guarantee is the empty relation, and is iteratively extended during the verification process. Later in [89], the authors extend the thread modular verification with counterexample-guided predicate abstraction refinement and apply their resuts to the data race detection. In [111], the authors formalize thread modular verification in the abstract interpretation framework, and prove that thread modular verification essentially is Cartesian abstract interpretation. In [29], the authors also present an assume-guarantee abstraction refinement technique for compositional verification of component-based systems. However, the system models being verified are finite-state and without data transfer. In [34], the authors present a modular verification technique for software components in C. Their approach consists in, first, abstracting each component as a finite-state automaton by using

predicate abstraction, then checking whether the finite-state automaton specification simulates the obtained abstractions. The applied abstraction technique is eager in the sense that an abstract transition system is construcuted prior to the acual analysis, as opposed to the lazy abstraction we apply, where the abstract transition system is constructed on the fly and as only far as necessary. In [83, 84], the authors propose a compositional verification technique for multi-threaded programs, where proof rules are encoded as recursion-free Horn clauses and auxiliary assertions are automatically computed and refined using predicate abstraction and interpolation. In [80], the authors propose to use horn clauses as a general representation of various proof rules, e.g. deductive proof rule, rely guarantee and assume guarantee proof rule. It allows us to automatically synthesis program verification tools, using horn clause solver as backend engine. Later in [128], the authors combine this compositional verification technique with a reduction technique based on Lipton's theory of reduction [108]. Reduction is applied as a program transformation that inserts atomic section based on a lockset analysis. At present, their tool still requires manual transformations. Moreover, the programming model is quite different from ours. They handle shared-variable concurrency, whereas in BIP we consider multiparty synchronisation and data transfer.

5 Further techniques for improving reductions

In this chapter, we present two further techniques for improving the performance of partial order reductions for BIP. The first technique is called simultaneous set reduction. As opposed to persistent set reduction, where the explorations of independent interactions are possibly postponed, simultaneous set reduction tries to explore the independent interactions in a single step. Since no interactions are postponed to explore, simultaneous set reduction does not need to solve the ignoring problem, thus, avoiding the expensive cycle detection.

Secondly, we present an advanced reduction technique for a particular class of BIP models, which have certain symmetric structure features, e.g. component symmetries. Symmetries are very common in most component-based designs. For instance, a system model consisting of one server and several identical users is symmetric with respect to the users. Permutating the indexes of the users would not affect the satisfaction of certain safety properties, e.g. deadlocks, that is, if one state is a deadlock state, then permutating the indexes of the symmetric components in the state does not change the deadlock. Thus, we can view the set of states that are identical under certain permuations as equivalent. In the state space exploration, we can select and visit only one representative of these equivalent states ideally, if the properties to be verified are invariant under symmetries. In the second part of this chapter, we investigate how to exploit symmetries of component-based systems to improve the efficiency of partial order reductions, the persistent set reduction in particular.

This chapter is based on the following publications:

– *Verification of component-based systems via predicate abstraction and simultaneous set reduction*, **Qiang, Wang** and Bliudze, Simon, International Symposium on Trustworthy Global Computing (TGC 2015), pages 147–162, 2015, Springer.
– *Exploiting Symmetry for Efficient Verification of Infinite-State Component-Based Systems*, **Wang, Qiang**, International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA 2016), pages 246–263, 2016, Springer.

The author proposed the verification algorithms, and did the implementations as well.

5.1 Simultaneous set reduction for BIP

In this section, we present a new reduction technique for BIP, called simultaneous set reduction. It also makes use of the interaction independence in Definition 3.2.1. However, differing from the persistent set reduction, which aims at avoiding the redundant interleavings of independent interactions, simultaneous set reduction executes as many independent interactions as possible simultaneously in one step.

In the sequel, we illustrate the idea by using an example, and then formalize the conditions imposed on the set of interactions, which can be executed simultaneously, and prove that no deadlocks are missed in the reduced reachable state space. In the end, we present how to combine it with lazy predicate abstraction, and how to compute the simultaneous set.

5.1.1 Motivating example

Example 5.1.1 *In Figure 5.1, we show a simple* BIP *model with two components B_1 and B_2. Each component defines three local integer variables and may enter the deadlock state S_5 by taking transition $error_1$ or $error_2$ when the guard $[x \neq y]$ holds. One binary interaction $\langle true, \{error_1, error_2\}, skip\rangle$ is defined to synchronize the two transitions labeled by ports $error_1$ and $error_2$ to take the system to an error state. Besides, all the other transitions form singleton interactions, e.g. $\langle true, \{invalid_1\}, x = 0; y = 0\rangle$. No data transfer is defined in this model.*

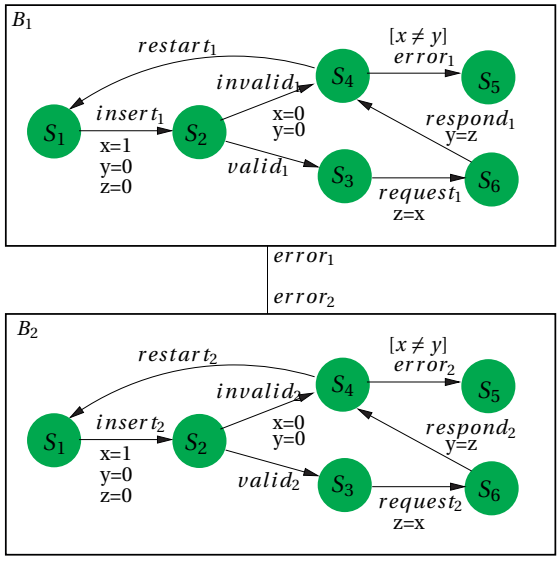

Figure 5.1 – The first example for illustrating simultaneous set

On the initial state $c_0 = \langle\langle S_1, x = 0, y = 0, z = 0\rangle, \langle S_1, x = 0, y = 0, z = 0\rangle\rangle$, There are two enabled interactions $\gamma_1 = \langle true, \{insert_1\}, x = 1; y = 0; z = 0\rangle$ and $\gamma_2 = \langle true, \{insert_2\}, x = 1; y = 0; z = 0\rangle$. It is easy to see that they are independent interactions, thus, the two interleavings $\gamma_1;\gamma_2$

and $\gamma_2;\gamma_1$ will lead to the same state $c = \langle\langle S_2, x=1, y=0, z=0\rangle, \langle S_2, x=1, y=0, z=0\rangle\rangle$.

In persistent set reduction, one interleaving out of the two is selected and explored. While in this simultaneous set reduction, we consider executing the two interactions simultaneously in one step in the abstract reachability analysis. The first question is that in order to preserve all the deadlock states in the reduced state space, what conditions should we impose on the set of interactions to be executed simultaneously?

The very first condition is independece. However, independence is not sufficient. Consider the model in Figure 5.1, suppose we want to expand the node $\eta = \langle\langle S_3, \phi_A\rangle, \langle S_4, \phi_B\rangle, \phi\rangle$, where component B_1 is at control location S_3 and component B_2 is at control location S_4, we first compute the set of enabled interactions $\Gamma_{enab} = \{\{request_1\}, \{restart_2\}\}$. Notice that interaction $\{error_1, error_2\}$ is disabled, because port $error_1$ is disabled in component B_1. Since the two interactions $\{request_1\}$ and $\{restart_2\}$ are independent, we may execute then simultaneously, however, in case of doing so, we would miss the following (fragment) counterexample from this node: $\{request_1\};\{respond_1\};\{error_1, error_2\}$. The reason is that although interaction $\{error_1, error_2\}$ is disabled on node η, it becomes enabled when interactions $\{request_1\};\{respond_1\}$ are executed.

Thus, when firing interactions simultaneously in one step, we have to make sure that no counterexample traces would be ignored in the future executions.

5.1.2 Combining simultaneous set reduction with lazy abstraction

Formally, we define the set of interactions that can be safely executed in one step as a simultaneous set.

Definition 5.1.2 *A set of enabled interactions Γ_{sim} on a state c is a simultaneous set, iff the following two conditions hold:*

1. *all the interactions in Γ_{sim} are independent in c;*
2. *for each interaction $\gamma \in \Gamma_{sim}$ and each execution $c \xrightarrow{\gamma} c_0 \xrightarrow{\gamma_1 \ldots \gamma_n} c_n$, if there is $\gamma' \in \Gamma_{sim} \setminus \{\gamma\}$ such that $\gamma' \notin \{\gamma_1, \ldots, \gamma_n\}$, then γ_n is independent of all interactions in $\Gamma_{sim} \setminus \{\gamma\}$.*

Notice the difference between simultaneous set the persistent set in Definition 3.2.10: interactions in a persistent set are inter-dependent, and their interleavings should be taken into account, while in a simultaneous set, interactions are independent and their interleavings can be avoided.

The second condition in the above definition means that in each execution starting from an interaction in a simultaneous set, the interactions appearing in the execution should be independent of all the interactions in the simultaneous set, unless all the interactions in the simultaneous set have been executed.

In order to prove the correctness of simultaneous set reduction, we introduce the following definitions. Given a BIP model $\mathscr{M}_{\text{BIP}}$ and its transition system $\mathscr{T}_{BIP}$, we denote by $\mathscr{T}^R_{BIP}$ the reduced transition system. A transition in $\mathscr{T}^R_{BIP}$ is denoted by $c \xrightarrow{\Gamma_{sim}} c'$, where Γ_{sim} is a simultaneous set on c. Notice that the transition in $\mathscr{T}^R_{BIP}$ may no longer be transition in $\mathscr{T}_{BIP}$, but a representation of several transition sequences.

Formally, suppose $\Gamma_{sim} = \{\gamma_1 \dots \gamma_k\}$, a transition $c \xrightarrow{\Gamma_{sim}} c'$ in the reduced transition system $\mathscr{T}^R_{BIP}$ represents a set of transition sequences $c \xrightarrow{\gamma_{i_1} \cdots \gamma_{i_k}} c'$ in $\mathscr{T}_{BIP}$, where $i_1, \dots, i_k$ is a permutation of $1, \dots, k$. We say that each transition sequence $c \xrightarrow{\gamma_{i_1} \cdots \gamma_{i_k}} c'$ is a concretization of $c \xrightarrow{\Gamma_{sim}} c'$. It is not hard to see for each simultaneous set of size k, there are k factorial concretizations. The concretizations of a trace in $\mathscr{T}^R_{BIP}$ are extended in the standard way.

The correctness of simultaneous set reduction with respect to deadlock states reachability analysis is stated in the following theorem.

Theorem 5.1.3 *Every reachable deadlock state in $\mathscr{T}_{BIP}$ is also reachable in $\mathscr{T}^R_{BIP}$.*

Proof 5.1.4 *Assume that state c_e is a deadlock state in $\mathscr{T}_{BIP}$, which is reachable by the trace ρ, then we prove that c_e is also reachable in $\mathscr{T}^R_{BIP}$. The proof is by complete induction on the number of states in the trace ρ.*

For the base case of $|\rho| = 1$, the result trivially holds since the initial state is the deadlock. Assume the theorem holds for all the cases of $|\rho| <= n$, where $n >= 1$, then we prove it also holds for $|\rho| = n+1$. Assume that $\rho = c_0 \xrightarrow{\gamma_0} c_1 \xrightarrow{\gamma_1 \cdots \gamma_{n-1}} c_e$, we show how to construct a trace ρ_r in $\mathscr{T}^R_{BIP}$ that represents ρ and also results in the deadlock state c_e.

If the simultaneous set on state c_0 is $\Gamma^0_{sim} = \{\gamma_0\}$, then ρ_r is ρ. If $\Gamma^0_{sim} = \{\beta_i | i \in [1,k]\} \cup \{\gamma_0\}$, we have that all interactions β_i should be executed in ρ, i.e. for each $\beta_i, i \in [1,k]$, there must be an interaction $\gamma_j, j \in [1, n-1]$ such that $\beta_i = \gamma_j$. Otherwise, suppose there is an interaction $\beta_i, i \in [1,k]$, which is not present in ρ, then according to the definition of simultaneous set, β_i must be independent with all interactions $\gamma_j, j \in [1, n-1]$, then β_i should also be enabled on state c_e, contradicting the fact that c_e is a deadlock state.

Then by permuting adjcent independent interactions, we can obtain the following trace $\rho' = c_0 \xrightarrow{\gamma_0 \beta_1 \cdots \beta_k} c_{k+1} \xrightarrow{\gamma_{k+1} \cdots \gamma_{n-1}} c_e$, where the sequence of interactions $c_0 \xrightarrow{\gamma_0 \beta_1 \cdots \beta_k} c_{k+1}$ is a concretization of the transition labeled by the simultaneous set Γ^0_{sim}, i.e. $c_0 \xrightarrow{\Gamma^0_{sim}} c_{k+1}$. Based on the induction hypothesis, the sequence of interactions $c_{k+1} \xrightarrow{\gamma_{k+1} \cdots \gamma_{n-1}} c_e$ is also a concretization of some trace ρ'_r in $\mathscr{T}^R_{BIP}$. Thus, ρ_r is the concatenation of $(c_0 \xrightarrow{\Gamma^0_{sim}} c_{k+1})$ and ρ'_r, concluding the proof.

More generally, simultaneous set also preserves the reachability of local component states. The proof of the following theorem is straightforward. Since no interactions are postponed or

ignored, there is no need to solve the ignoring problem.

Theorem 5.1.5 *Given a BIP system model $\mathscr{M}_{\text{BIP}}$, and its labeled transition system $\mathscr{T}_{BIP}$, if $\langle l_i, \mathbf{V}_i \rangle$ is a local state of component i, then there is a state $c' = \langle \langle l_1, \mathbf{V}_1 \rangle', \ldots, \langle l_n, \mathbf{V}_n \rangle' \rangle \in C_{BIP}$, such that $\langle l_i, \mathbf{V}_i \rangle = \langle l_i, \mathbf{V}_i \rangle'$.*

In order to combine the simultaneous set reduction with lazy abstraction for BIP, we first lift the simultaneous set definition to abstract states as in Definition 4.2.5. Then we modify the Algorithm 2 to obtian the combination. The new algorithm is listed in Algorithm 6. It differs from Algorithm 2 in that when expanding a node, instead of creating successor nodes fror each enabled interaction in Γ_{enab}, it first computes the set of simultaneous sets Γ_{sim} by invoking the function *SimultaneousSet*, which will be elaborated in the next subsection, and then creates a successor node for each simultaneous set in Γ_{sim}. Notice that since a simultaneous set is a set of interactions, the node expansion procedure should also be accordingly adjusted. We also remark that we do not need cycle detection or full node expansion in the new algorithm, since no interactions are postponed to execute in the simultaneous set reduction.

Algorithm 6 Lazy abstraction with simultaneous set reduction for BIP

Input: a BIP model $\mathscr{M}_{\text{BIP}}$ and an error state
Output: either $\mathscr{M}_{\text{BIP}}$ is safe, or a counterexample *cex*

```
1:  create an ART 𝕋 with initial node η0
2:  create a worklist wl
3:  push η0 into wl
4:  while wl ≠ ∅ do
5:      η ← pop(wl)
6:      if IsError(η) then
7:          cex ← BuildCEX(η)
8:          if cex is real then
9:              return cex
10:         else
11:             Refine(𝕋, cex)
12:     else if Covering(η) then
13:         mark η as covered
14:     else
15:         Γ_S ← SimultaneousSet(η)
16:         for each Γ ∈ Γ_S do
17:             Expand(η, Γ)
18:             push the successor into wl
19: return M_BIP is safe
```

The following theorem states the correctness of Algorithm 6.

Theorem 5.1.6 *Given a BIP model $\mathscr{M}_{\text{BIP}}$ and an error state encoding the invariant property, for every terminating execution of Algorithm 6, the following two properties hold:*

1. *If a counterexample is returned, then it is a concrete counterexample in* $\mathscr{M}_{\mathrm{BIP}}$*;*
2. *If a safe ART is returned, then* $\mathscr{M}_{\mathrm{BIP}}$ *is safe.*

Proof 5.1.7 *If a counterexample is returned, we know that it is a feasible execution. If an ART is returned, and suppose there is a concrete execution to a deadlock state, then according to Theorem 5.1.3 and Theorem 4.1.6, we can conclude that this execution should also be in the abstraction, concluding the assumption that a safe ART is returned.*

5.1.3 Computing simultaneous set

In this section, we present an implmentation of function SimultaneousSet in Algorithm 6, which computes the set of simultaneous sets on an ART node. The independence relation is obtained in the same way as in Section 4.2.2. The implementation is listed in Algorithm 7. It uses two additional functions EnabledInteraction and DisabledInteraction. Function DisabledInteraction computes the set of disabled interactions on an ART node, which is the complement of the set of enabled interactions.

Algorithm 7 Simultaneous set computation

Input: an ART node $\eta = \langle\langle l_1, \phi_1\rangle, \ldots, \langle l_n, \phi_n\rangle, \phi\rangle$
Output: a set of simultaneous sets Γ_{sim}

```
1:  Γ_E ← EnabledInteraction(η)
2:  Γ_D ← DisabledInteraction(η)
3:  create a worklist of interaction sets wl
4:  push Γ_E into wl
5:  while wl ≠ ∅ do
6:      Γ ← pop(wl)
7:      if exists γ1, γ2 ∈ Γ, s.t. γ1, γ2 are dependent then
8:          copy1 ← Γ − {γ1}
9:          copy2 ← Γ − {γ2}
10:         push copy1, copy2 into wl
11:     else if exists γ1, γ2 ∈ Γ, γ3 ∈ Γ_D,
                s.t. γ3, γ1 are dependent, and γ3, γ2 are dependent then
12:         copy1 ← Γ − {γ1}
13:         copy2 ← Γ − {γ2}
14:         push copy1, copy2 into wl
15:     else
16:         if Γ_sim does not contain Γ then
17:             push Γ into Γ_sim
```

The basic idea is that starting from the set of enabled interactions, the algorithm progressively refines this set by splitting it into two sets, meaning that this set of interactions cannot be executed simultaneously. The criterion of splitting a set is the following: 1) either the two interactions from the set are dependent; 2) or they are independent with each other, but dependent with a disabled interaction. Then this set is split into two sets, each of which is

obtained by removing one of the interactions. Otherwise, if all interactions are independent of each other and with the disabled interactions, then the set is a simultaneous set and is added into the result set Γ_{sim}. The following theorem states the correctness of Algorithm 7.

Theorem 5.1.8 *Let Γ_{sim} be a set returned by Algorithm 7, then Γ_{sim} is a set of simultaneous sets on the given ART node.*

Proof 5.1.9 *Suppose there is a set $\Gamma \in \Gamma_{sim}$, and Γ is not a simultaneous set on the given ART node η. However, from the computation in Algorithm 7, we know that all interactions in Γ are independent. Then the reason preventing it from being a simultaneous set is that there is a finite execution $\eta \xrightarrow{\alpha} \eta_0 \xrightarrow{\beta_1} \eta_1 \ldots \eta_{n-1} \xrightarrow{\beta_n} \eta_n$, where $\alpha \in \Gamma$ and there is an interaction $\beta \in \Gamma$ and $\beta \notin \{\beta_1, \ldots, \beta_n\}$, β_n is dependent with some interaction $\gamma \in \Gamma$. If β_n is enabled on η, then γ and β_n should be in two different simultaneous sets. The above finite execution meets the simultaneous set definition. Thus, β_n is disabled on η, and based on the computation in Algorithm 7, α and γ should be splitted into two sets, which contradicts our assumption that $\alpha, \gamma \in \Gamma$. This concludes the proof.*

We remark that the above algorithm for computing a simultaneous set is only correct for BIP models we consider in this dissertation. Adopting the computation to other formalizations, e.g. Petri net, may not result in a correct simultaneous set that preserves deadlock states.

Consider the following model in Figure 5.2, consisting of two components B_1 and B_2. The initial states of the two components are S_1 and S_3 respectively. The only interaction synchronizes transition t_5 in component B_1 with transition t_4 in component B_2. Clearly, the two transitions t_1 and t_2 are independent on the initial state, however, the set $\{t_1, t_2\}$ is not a simultaneous set. To see why, consider the execution $\{t_2\}\{t_3\}\{t_4, t_5\}$ from the initial state, according to the definition of simultaneous set, $\{t_1\}$ should be independent of all the subsequent interactions after $\{t_2\}$, i.e. $\{t_3\}$ and $\{t_4, t_5\}$, which is not the case, since $\{t_1\}$ is dependent with $\{t_4, t_5\}$. Thus, the set $\{t_1, t_2\}$ does not form a simultaneous set. Our algorithm does not return the set $\{t_1, t_2\}$ as a simultaneous set on the initial state, because both interactions t_1 and t_2 are dependent with the disabled interaction $\{t_4, t_5\}$.

Considering the complexity of computing simultaneous set in Algorithm 7, we assume that, given two interactions γ_1 and γ_2, it takes $\mathcal{O}(1)$ time for the dependence check with precomputed dependence relation. The while loop executes at most $|\Gamma_E|$ times, where $|\Gamma_E|$ denotes the number of enabled interactions in Γ_E. Since in each loop execution at most two interactions will be split and one simultaneous set will be added into the worklist. In the worst case, $|\Gamma_E|^2 * |\Gamma_D|$ checks are needed to find the two interactions to be split in each loop execution. Thus, the worst case time complexity of Algorithm 7 is $\mathcal{O}(|\Gamma_E|^3 * |\Gamma_D|)$ in terms of the number of interactions in the model.

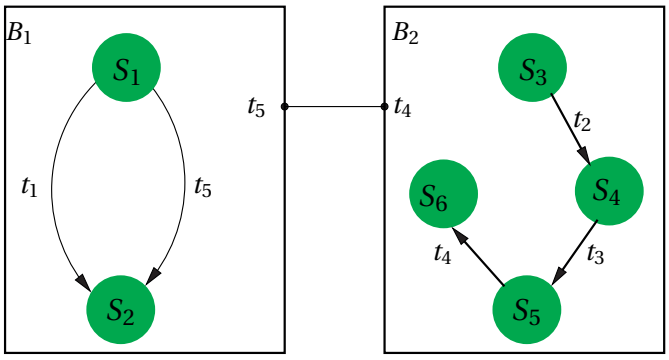

Figure 5.2 – The second example for illustrating simultaneous set

5.1.4 Discussions

In this subsection, we compare the simultaneous set reduction and persistent set reduction through an example borrowed from [143]. Arguably, it is not clear which of the two approaches, persistent set reduction and simultaneous set reduction has better performance. We try to give a preliminary theoretical comparison.

Consider first the example model in the left of Figure 5.3, which consists of n concurrent components. Each component defines three control locations and two transitions. No interactions between the components are enforced, thus, every component executes independently.

Clearly, there are $n!2^n$ different executions in this model, yielding a state space of size 3^n. On the initial state, there are 2^n possible simultaneous sets, each of which consists of one transition from each component. Simultaneous set reduction yields a reduced state space of size $2^n + 1$. In contrast, persistent set reduction executes one component at a time, which yields a state space with $2^{n+1} - 1$ states. Both reductions gain a significant saving over the full reachable state space, and moreover, simultaneous set yields approximately half additional saving over persistent set.

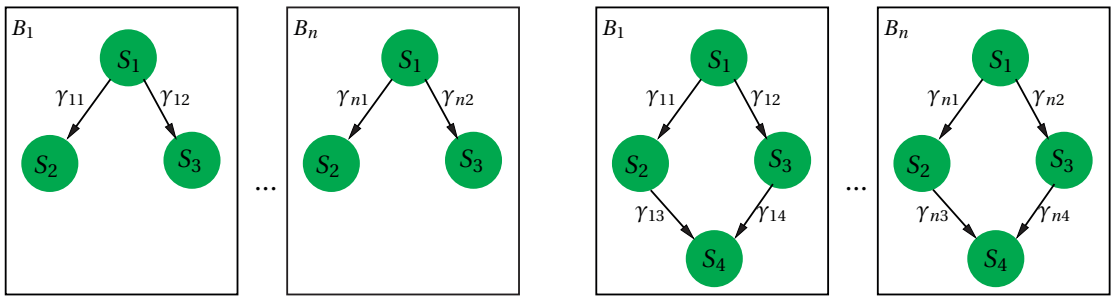

Figure 5.3 – Examples for comparing simultaneous and persistent sets

However, there is no guarantee that simultaneous set always outperforms persistent set. Consider now the example model in the right part of Figure 5.3, where each component has two additional transitions leading to a single terminal state. In this case, simultaneous set

results in $2^n + 2$ states, while properly implemented persistent set can construct a reduced state space of size $3 * n + 1$, which is tremendously better than simultaneous set reduction.

To conclude, in general there is no definite guarantee that one reduction outperforms the other one. Furthermore, in [143] the authors argue that in the best case, persistent set has the potential to offer good reductions that can never be achieved by simultaneous set reduction. Regrettably, the authors also noticed that there is no clear guarantee to obtain such a good reduction for persistent set reduction. Moreover, in persistent set reduction one has to resolve the ignoring problem in order to verify general safety properties. In our experiences, this task is computationally hard and affects the reduction performance of persistent set significantly. In the next section of experimental evaluation, we will compare the power of the two reductions for the practical point of view.

5.2 Experimental evaluation

We have implemented the proposed verification techniques for BIP. To evaluate the performance of the proposed techniques, we carried out a comprehensive experimental evaluation, where we took the set of benchmarks in the previous experiments. The details of all the benchmark models are provided in the Appendix. All the experiments have been run on a 64-bit Linux PC with a 2.8 GHz Intel i7-2640M CPU, with a memory limit of 4Gb and a time limit of 300 seconds per benchmark.

In the experiments, we denote by 'simset' in the plots our new technique and compare it to the following techniques: 1) plain lazy abstraction, denoted by 'plain' in the plots; 2) lazy abstraction with persistent set reduction, denoted by 'pset'; 3) the state-of-the-art IC3 algorithm for software model checking [37] implemented in nuXmv [33], denoted by 'IC3'; 4) implicit predicate abstraction model checking [138], denoted by 'IPA'. We do not compare the performance of our tool to DFinder [21] or the work [95], since they do not handle data transfer in interaction, or infinite-state models.

The detailed statistics data is attached in the Appendix A.7.

5.2.1 Comparing to lazy abstraction with reductions

In this subsection, we compare the lazy abstraction with simultaneous set reduction (simultaneous set reduction for simplicity) to plain lazy abstraction and to lazy abstraction with persistent set reduction (persistent set reduction for simplicity). In Figure 5.4 and Figure 5.5, we show the scatter plots of time for solving each benchmark. In the plots, symbol $\times$ represents a safe benchmark, and $\circ$ represents an unsafe benchmark. A point in the plots indicates the analysis time taken by the algorithms represented by x-axis and y-axis.

From the plots, we can conclude that 1) persistent set is slightly faster than simultaneous set on safe benchmarks, while simultaneous set is faster on unsafe benchmarks; 2) simultaneous

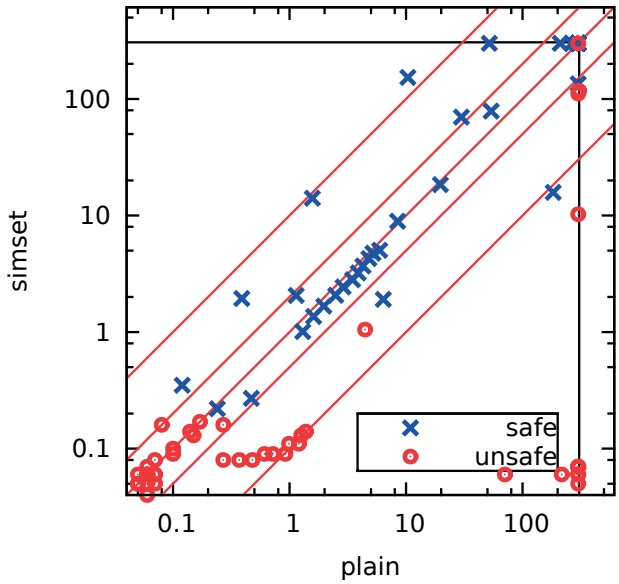

Figure 5.4 – Lazy abstraction vs. lazy abstraction with simultaneous set reduction

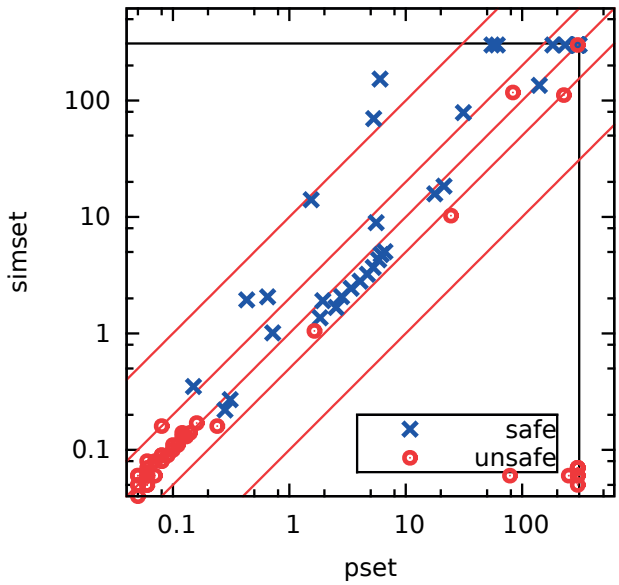

Figure 5.5 – Lazy abstraction with persistent set reduction vs. lazy abstraction with simultaneous set reduction

model	percentage	model	percentage
atm_safe_02	0.727273	atm_safe_03	0.873747
atm_unsafe_02	0.727273	leader_election_safe_02	0.400000
leader_election_safe_03	0.458333	leader_election_safe_04	0.525253
leader_election_safe_05	0.551122	leader_election_unsafe_02	0.666667
leader_election_unsafe_03	0.500000	leader_election_unsafe_04	0.534884
leader_election_unsafe_05	0.529703	quorum_safe_02	0.171429
quorum_safe_03	0.057971	quorum_safe_04	0.031311
quorum_unsafe_02	0.111111	quorum_unsafe_03	0.000000
railway_control_safe	0.000000	railway_control_unsafe	0.000000
temperature_safe	0.000000	temperature_unsafe	0.000000
ticket_safe	0.000000	ticket_unsafe	0.000000

Table 5.1 – Percentage of simultaneous set reduction

set is also faster than plain lazy abstraction on unsafe benchmarks, while on safe benchmarks, simultaneous set and plain lazy abstraction are comparable. One possible reason of the result that simultaneous set reduction works slightly better than persistent set on unsafe benchmarks is that in persistent set reduction, we blindly postpone the explorations of some interactions, which may have the effect of enlarge the length of counterexample, while, in simultaneous set we on the contrary shorten the length of executions by executing several interactions altogether, thus resulting in detecting a counterexample possibly faster.

As in the previous experiments, we also collect the time used by each subroutine of the algorithms and show them in the following bar plots. In simultaneous set reduction, it contains all the subroutines of lazy abstraction, and the time of simultaneous set computation, indicated as 'time_of_por' in the plots. The results are shown in Figure 5.6. We also only depict the results with total runtime greater than 1 second. The one in the left depicts the results with runtime greater than 10 seconds, and the one in the right depicts the results with runtime from 1 second to 10 seconds. From the plots we can see that the most expensive routines are the computation of transfer function and the coverage check, which is the same with lazy abstraction. Comparing to the persistent set reduction, simultaneous set reduction does not have the time of cycle detection, since it does not need to solve the ignoring problem. This may be the potential superiority over persistent set reduction for general safety property verification. The plots also show that our present simultaneous set computation is not efficient in some cases.

We also measure the percentage of successful reductions over the total attempts. The result is shown in Table 5.1. The benchamrks without any successful reductions are the railway control system, the ticket mutual exclusion protocol and the reactor temperature control system. It has less reduction achievements than persistent set reduction (shown in Table 4.1), which may explain the fact that on safe benchmarks persistent set reduction works better than simultaneous set reduction, as shown in Figure 5.5. A plausible result is that for unsafe

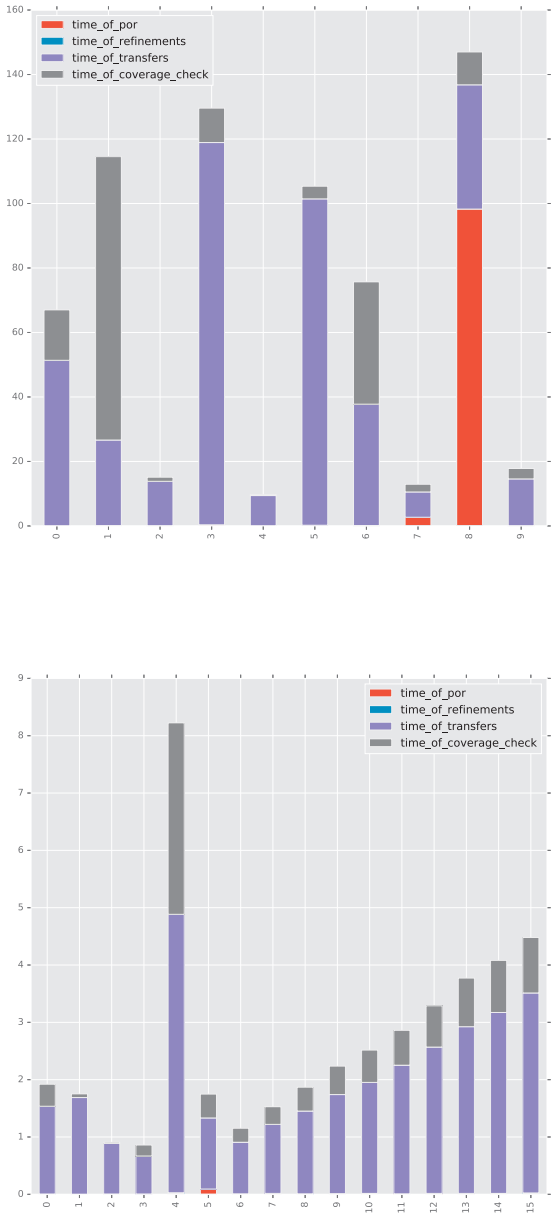

Figure 5.6 – Runtime of lazy abstraction with simultaneous set reduction subroutines

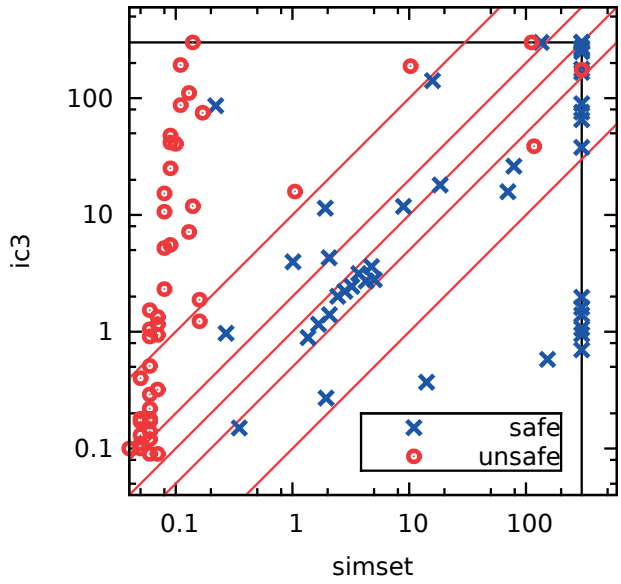

Figure 5.7 – Lazy abstraction with simultaneous set reduction vs. IC3

versions of railway control system, simultaneous set reduction obtains no reduction, while still outperforms the plain lazy abstraction. The reason is that in simultaneous set reduction we reorder the interactions to be explored, such that the interactions leading to the error state are always explored first. In plain lazy abstraction, the order of the interactions to be explored is chosen randomly.

5.2.2 Comparing to IC3 and IPA

We also compare each of our configurations to both IC3 and IPA in terms of the running time for solving each benchmark. The results are shown in the scatter plots in Figure 5.7 and Figure 5.8.

In Figure 5.7, we show the comparison with IC3. We can see that for unsafe benchmarks, simultaneous set reduction outperforms IC3. For safe benchmarks, our technique is comparable to IC3, while there are a number of models that can be solved by IC3, but the other technique runs out of time. In Figure 5.8, we show the comparison to IPA. For both safe and unsafe benchmarks, simultaneous set reduction performs better. IPA is unable to solve any unsafe benchmarks.

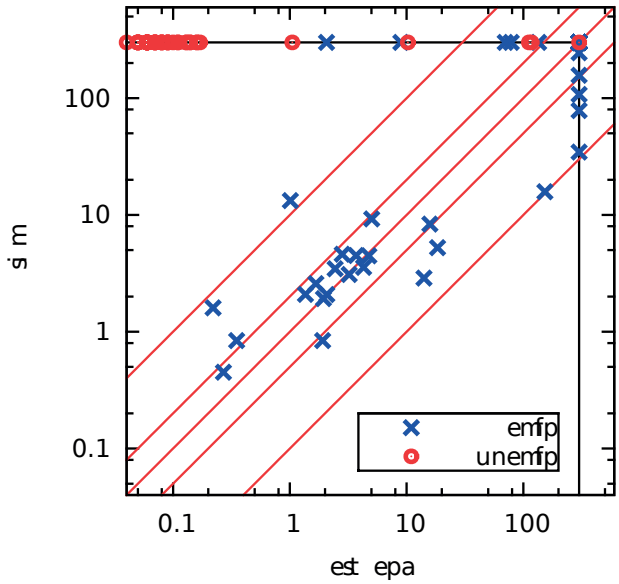

Figure 5.8 – Lazy abstraction with simultaneous set reduction vs. IPA

5.2.3 Cumulative plots

In this subsection, we present the cumulative plots that indicate the number of benchmark models that can be solved by each technique (y-axis) within the given time (x-axis). In Figure 5.9, we plot the cumulative time of solving the benchmarks for all techniques. The plot shows that overall IC3 can solve more benchmark models within the time limits. In total, IC3 has solved 84 benchmark models, IPA has solved 25 models and plain lazy abstraction, persistent set reduction and simultaneous set reduction have solved 63, 68, 70 models respectively. Among the three configurations of our prototype tool, persistent set reduction and simultaneous set reduction work better than plain lazy abstraction as expected. Simultaneous set reduction solves slightly more models than persistent set reduction.

In Figure 5.10 and Figure 5.11, we plot the cumulative time of solving safe and unsafe benchmarks respectively. The plot in Figure 5.10 shows that for safe benchmarks, IC3 is able to solve more benchmark models. The other techniques are comparable, while persistent set reduction performs slightly better.

For unsafe benchmarks, our techniques perform much better than IC3. However, simultaneous set reduction solves almost the same amout of safe benchmarks as plain lazy abstraction. Only for unsafe benchmarks, it is able to solve more and faster than lazy abstraction. In other words, simultaneous set reduction is more efficient to find counterexamples. This result is reasonable because with simultaneous set reduction, some independent interactions are executed simultaneously, thus reducing both the length of counterexamples and the time

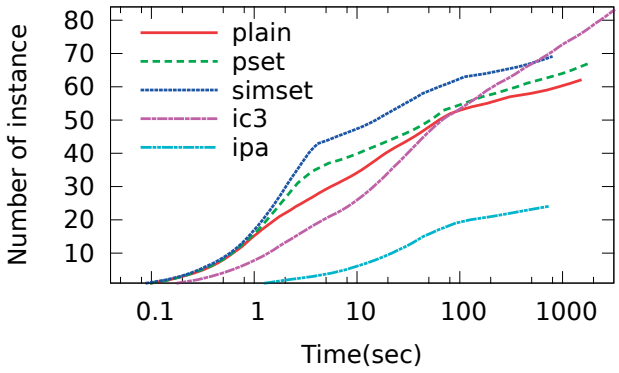

Figure 5.9 – Cumulative plot of time for all benchmarks

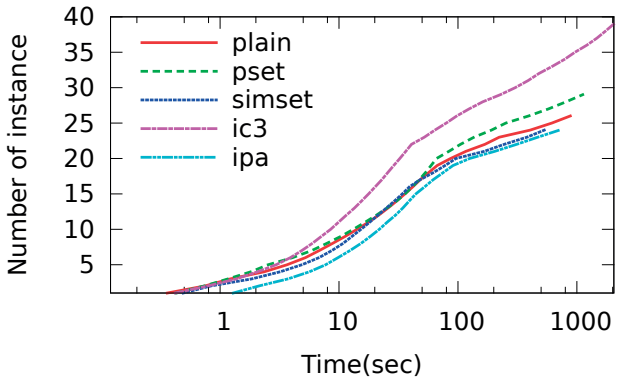

Figure 5.10 – Cumulative plot for safe benchmarks

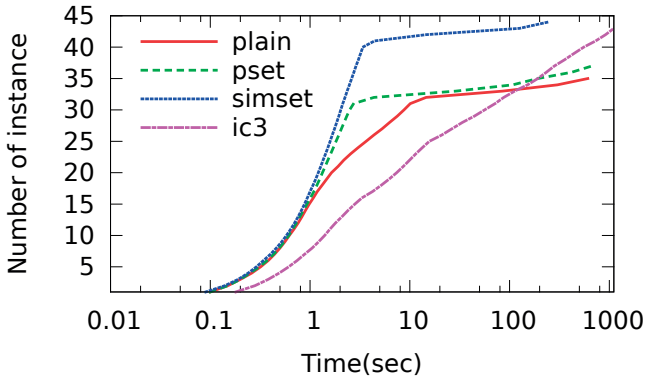

Figure 5.11 – Cumulative plot for unsafe benchmarks

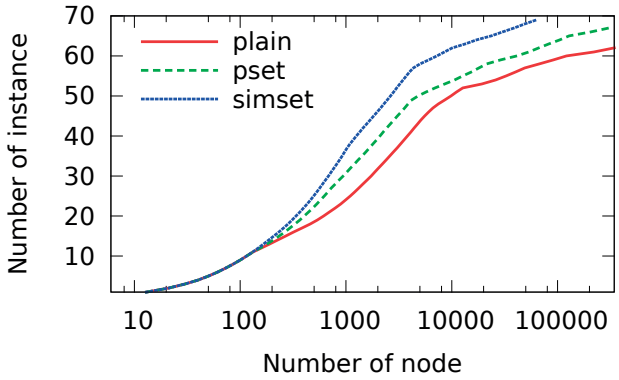

Figure 5.12 – Cumulative plot of ART size

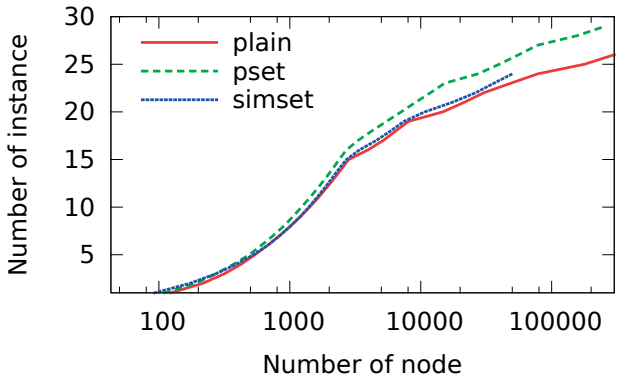

Figure 5.13 – Cumulative plot of ART size for safe benchmarks

to detect them. There is no data in Figure 5.11 for IPA, since it fails to solve all the unsafe benchmarks.

We also measure the memory usage of our techniques in terms of the size of ART. We collect the number of nodes that are needed to solve each benchmark model. The cumulative plots are shown in Figure 5.12, Figure 5.13 and Figure 5.14. They show that for solving the same amount benchmarks, plain lazy abstraction needs to create more ART nodes than the other two, thus consuming more memory usage. The other two are comparable, though simultaneous set reduction creates less ART nodes for solving unsafe benchmarks.

5.3 Partial order reduction under symmetry

In this section, we focuse on the class of BIP system models, which have certain symmetry features, and present how to exploit such symmetries to improve partial order reduction.

We build this work on top of the framework presented in Chapter 4. First of all, we extend the notion of interaction independence in Definition 4.2.1 by taking into account the system

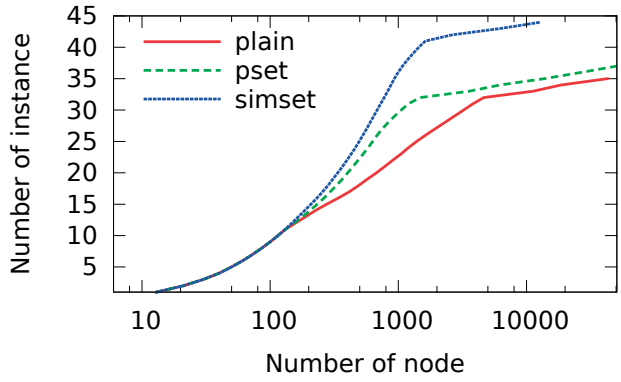

Figure 5.14 – Cumulative plot of ART size for unsafe benchmarks

symmetries, i.e. two interactions are independent if they commute under some symmetries. The original definition of independence is then a special case of this one with identical symmetry. Second, we adopt the persistent set based partial order reduction technique [78] by relying on this new notion of independence and show how to combine it with lazy abstraction of BIP presented in Chapter 4. We have also implemented the proposed verification algorithm and performed a set of experiments. The results show that for systems with certain symmetries, our new algorithm outperforms the others significantly.

5.3.1 Motivating example

We illustrate the basic idea of our verification approach, using the ticket mutual exclusion protocol.

Example 5.3.1 (Ticket mutual exclusion protocol [110]) *Consider again the ticket mutual exclusion protocol in Figure. 5.15. Upon entering the critical section $C_i, i = 1,2$, each process requests a fresh ticket from the controller, then the process waits until its ticket is with the number to be served next. When leaving the critical section, the process resets the ticket and the controller increases the number to be served by one.*

Assume we start the state space exploration from the initial state $\langle\langle I_1, ticket_1 = 0\rangle, \langle S, number = 1, next = 1\rangle, \langle I_2, ticket_2 = 0\rangle\rangle$, then the following two interactions γ_1, γ_2 will have to be explored, where $\gamma_1 = \langle true, \{request, request_1\}, ticket_1 = number\rangle$, and $\gamma_2 = \langle true, \{request, request_2\}, ticket_2 = number\rangle$. Apparently, this two interactions γ_1, γ_2 are not independent according to Definition 4.2.1, since they both modify the variable $number$ in the controller, and the interleavings $\gamma_1;\gamma_2$ and $\gamma_2;\gamma_1$ lead to two different states, that is, $\langle\langle W_1, ticket_1 = 1\rangle, \langle S, number = 3, next = 1\rangle, \langle W_2, ticket_2 = 2\rangle\rangle$ and $\langle\langle W_1, ticket_1 = 2\rangle, \langle S, number = 3, next = 1\rangle, \langle W_2, ticket_2 = 1\rangle\rangle$.

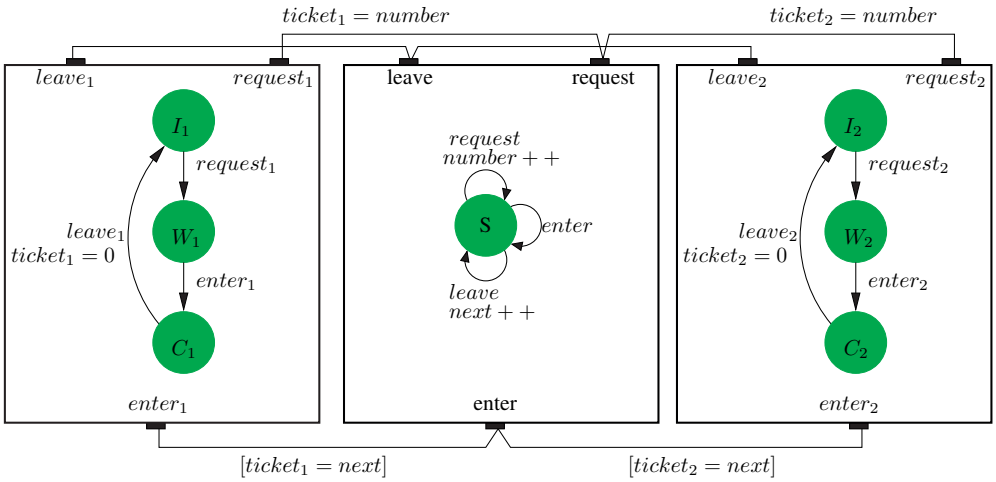

Figure 5.15 – Ticket mutual exclusion protocol

Our observation is that under the following permutation of the process components $\pi = \{1 \mapsto 2, 2 \mapsto 1\}$, the above two states become identical. Further, mutual exclusion is also invariant under this permutation. Thus, only one of the two states is needed to verify the mutual exclusion property. However, instead of participating the state space using symmetry reduction as in [61, 45, 98], we take a different view that under the above permutation, the two interactions γ_1, γ_2 commute with each other, and the partial order reductions presented in Chapter 4 can be refined by using this new commutativity property.

5.3.2 Symmetry reduction

Initially proposed in [61, 45, 98], symmetry reduction is a useful tool to reduce the search space during the verification of a transition system. It exploits the symmetry of a transition system. Intuitively, a transition system has symmetry if the transition relations remain invariant when states are rearranged by certain permutations.

Definition 5.3.2 *A symmetry of a labeled transition system $\mathcal{T} = \langle C, \Sigma, R, C_0 \rangle$ is a permutation π over $C \cup \Sigma$, that satisfies the following conditions:*

1. $\pi(C) = C$ *and* $\pi(\Sigma) = \Sigma$, *and*
2. $\langle c_1, \gamma, c_2 \rangle \in R$ *iff* $\langle \pi(c_1), \pi(\gamma), \pi(c_2) \rangle \in R$, *and*
3. $\pi(C_0) = C_0$.

Given a transition system $\mathcal{T}$, the set of all symmetries of $\mathcal{T}$ forms a group under the function composition, denoted by $Aut(\mathcal{T})$. However, obtaining $Aut(\mathcal{T})$ is computationally expensive, since one has to explore the whole state space. In practice, subgroups of $Aut(\mathcal{T})$, which can be obtained from the high-level system structure are used. Example subgroups include rotation group, full component symmetry group and also the Cartesian product of such subgroups.

A subgroup $\mathscr{G} \subseteq Aut(\mathscr{T})$ induces an equivalence relation $\equiv_{\mathscr{G}}$ on $\mathscr{T}$ as follows: $s \equiv_{\mathscr{G}} t \Leftrightarrow \exists \pi \in \mathscr{G}.s = \pi(t)$. The equivalence relation $\equiv_{\mathscr{G}}$ is also called the orbit relation, and it induces a quotient model $\mathscr{T}_{\mathscr{G}}$, which is bisimilar to $\mathscr{T}$ [61, 45]. Model checking of a symmetric property, i.e. a property remains invariant under permutations in $\mathscr{G}$, can be performed on the quotient model. We remark that deadlock states are trivially invariant under symmetry permutations.

As noticed in [44], under arbitrary symmetries, detecting state equivalence is as hard as the graph isomorphism problem. In order to bypass the orbit relation, for some specific symmetry subgroups, e.g. full component symmetry, rotation symmetry, one can select some representatives from the orbit relation and define a mapping funtion that computes these representatives [45, 62, 63]. Then during the state space exploration, states are dynamically mapped to their respective representatives.

5.3.3 Persistent set under symmetry

In this section, we extend the persistent set based partial order reduction [78] by taking into account the system symmetries. First of all, we generalize the definition of interaction independence.

Definition 5.3.3 *Given a symmetry group $\mathscr{G}$, two interactions γ_1 and γ_2 are independent under symmetry $\mathscr{G}$, if and only if for every state c in the global system, there is a symmetry permutation $\pi \in \mathscr{G}$, such that the following conditions hold:*

1. *if γ_1 is enabled in c, then γ_2 is enabled in c iff γ_2 is enabled in c', where $c \xrightarrow{\gamma_1} c'$.*
2. *if γ_1 is enabled in c, then γ_2 is enabled in c iff γ_2 is enabled in c', where $c \xrightarrow{\gamma_1} c'$.*
3. *if γ_1 and γ_2 are both enabled in c, then $c'_1 = \pi(c'_2)$, where $c \xrightarrow{\gamma_1\gamma_2} c'_1$, and $c \xrightarrow{\gamma_2\gamma_1} c'_2$.*

This new definition differs the one in Definition 4.2.1 in that two interactions are viewed as indepdenent if their executions commute under some symmetry permutations. As before, for a given interaction γ, we denote by $\mathscr{D}_\gamma$ the set of interactions that are not independent under symmetry.

In the previous chapter, we obtain an under-approximation of independence relation statically from system specification: two interactions are *independent* if they do not share a common component. Though being easy to obtain, this approximation is too coarse, and many independent transitions are ignored. For instance, the following two interactions in Figure 5.15, $\gamma_1 = \langle [ticket_1 = next], \{enter, enter_1\}, skip\rangle$, and $\gamma_2 = \langle [ticket_2 = next], \{enter, enter_2\}, skip\rangle$, are independent, but using the above static analysis, they are considered as dependent.

In this chapter, we apply a finer static analysis to check if two interactions are independent or not. Given two interactions $\gamma_1 = \langle g_1, \mathscr{P}_1, f_1\rangle$, $\gamma_2 = \langle g_2, \mathscr{P}_2, f_2\rangle$, we check if they are independent by checking the validity of the following three formulae:

1 $\forall c.\exists c'.c \models g_1 \wedge c \xrightarrow{\gamma_1} c' \implies (c \models g_2 \equiv c' \models g_2)$
2 $\forall c.\exists c'.c \models g_2 \wedge c \xrightarrow{\gamma_2} c' \implies (c \models g_1 \equiv c' \models g_1)$
3 there is a permutation $\pi \in \mathcal{G}$, such that the formula $\forall c.\exists c_1, c_2.c \models g_1 \wedge c \models g_2 \wedge c \xrightarrow{\gamma_1\gamma_2} c_1 \wedge c \xrightarrow{\gamma_2\gamma_1} c_2 \implies c_1 = \pi(c_2)$ is valid.

Considering the complexity, the number of interactions is linear to the size of the system model. Thus, the number of validity checks is also linear to the size of system model. In order to detect the state equivalence under symmetry, one intuitive approach is to traverse all permutations in the symmetry group $\mathcal{G}$. However, this would blow up the analysis, even for full component symmetry group, whose complexity is factorial in the number of components. As in [63], we use a sorting function that maps a state to a representative in the orbit relation, then two states are equivalent if they can be mapped to the same representative. The sorting function requires a total order on the symbolic states of each component. We say a symbolic state c_1 is greater than another c_2 if $c_1 > c_2$ is valid.

Since we focus on infinite-state systems, our new partial order reduction should apply to a symbolic abstraction structure, e.g. an abstract reachability tree. As in section 4.2, we say two interactions are independent under a symmetry $\mathcal{G}$ on an abstract state η, if they are independent on every concrete state of η.

As also noticed in section 4.2, the independent transitions do not commute under symmetry on symbolic abstraction structures. However, the following lemma shows that independent transitions still commute under symmetry on the concrete states represented by the abstraction structures. Thus, exploiting independence on the symbolic abstraction structure is still sound.

Lemma 5.3.4 *Let γ_1 and γ_2 be two independent transitions under a symmetry permutation π, and let η be an abstract state, then for all $c \in \beta(\widehat{post}(\widehat{post}(\eta,\gamma_1)),\gamma_2)$, if there is a state $c_1 \in \beta(\eta)$, such that $c = post(post(c_1,\gamma_1),\gamma_2)$, then $\pi(c) \in \beta(\widehat{post}(\widehat{post}(\eta,\gamma_2)),\gamma_1)$.*

Proof 5.3.5 *Assume we have $c = post(post(c_1,\gamma_1),\gamma_2)$, since γ_1 and γ_2 are independent under symmetry π, then we also have $\pi(c) = post(post(c_1,\gamma_2),\gamma_1)$. According to the semantics of $\widehat{post}$, it holds that $\pi(c) \in \beta(\widehat{post}(\widehat{post}(\eta,\gamma_2),\gamma_1))$.*

We then extend the persistent set in Definition 3.2.10 by relying on the notion of independence under symmetry and by generalising to the symbolic abstraction structure.

Definition 5.3.6 *Given a symmetry $\mathcal{G}$, a set of interactions Γ in an abstract state η is persistent under $\mathcal{G}$, if the following conditions hold:*

1. *$\Gamma \subseteq en(\eta)$ and $\Gamma = \emptyset$ if and only if $en(\eta) = \emptyset$;*
2. *for every trace $\eta \xrightarrow{\gamma_1 \ldots \gamma_n} \eta_n$, where $\gamma_i \notin \Gamma, i \in [1,n]$, γ_n is abstractly independent under symmetry $\mathcal{G}$ with all interactions in Γ;*

3. *for every execution* $\eta = \eta_0 \xrightarrow{\gamma_1 \dots \gamma_n} \eta_n$, *where* η_n *is implied by some* η_i, $i \in [0, n-1]$, *and for every* $\gamma \in en(\eta_j)$, $j \in [1, n]$, *there is* $k \in [1, n]$ *such that* $\gamma \in \Gamma(\eta_k)$.

We remark that a persistent set on an abstract state may not be a persistent set on some of its concrete states, because some interactions may be disabled on the concrete states. But the set of enabling interactions constitute a persistent set.

We also use Algorithm 5 in section 4.2 to compute the persistent set, and the intergration with lazy abstraction is straightforward as in Algorithm 4. We skip the elaboration here. The following theorem states the correctness of selective search over symbolic abstraction structure by using this new persistent set above.

Theorem 5.3.7 *Given a* BIP *model* $\mathscr{M}_{\text{BIP}}$, *and an error state encoding a safety property that is invariant under symmetry, for every terminating execution of Algorithm 4 with persistent set under symmetry in Definition 5.3.6, the following properties hold:*
1. *If a counterexample is returned, then there is concrete counterexample in* $\mathscr{M}_{\text{BIP}}$;
2. *If a safe ART is returned, then* $\mathscr{M}_{\text{BIP}}$ *satisfies the given safety property.*

Proof 5.3.8 *Item 1 holds for the same argument with Theorem 4.1.6. In order to prove the second item, in the following we prove that the returned ART safely over approximates the reachable states of* $\mathscr{M}_{\text{BIP}}$, *that is, for every reachable state* c *in* $\mathscr{M}_{\text{BIP}}$, *there is a symmetry permutation* π, *and an ART node* η *such that* $\pi(c) \models \eta$. *The proof is similar to the one in 4.2.6.*

According to Theorem 4.1.6, for every reachable state c *in* $\mathscr{M}_{\text{BIP}}$, *there is a node* η^w *in the ART returned by plain lazy abstraction in Algorithm 2, such that* $c \models \eta^w$. *Suppose the path to node* η^w *is* $\eta_0 \xrightarrow{\gamma_1 \dots \gamma_n} \eta_n$, *where* η_0 *is the root and* $\eta_n = \eta^w$.

Now we prove that there is another path in the ART returned by Algorithm 4 with persistent set under symmetry, $\eta_0 \xrightarrow{\gamma'_1 \dots \gamma'_n} \eta'_n$, *such that* γ'_1 *is in the persistent set* $\Gamma(\eta_0)$ *and* $\pi(c) \models \eta'_n$ *for some permutation* π.

Case 1 : if η^w *is a deadlock node, then we show that at least one interaction* γ_i, $i \in [1, n]$ *in the path* $\eta_0 \xrightarrow{\gamma_1 \dots \gamma_n} \eta_n$ *is in the persistent set* $\Gamma(\eta_0)$. *Otherwise, suppose that none of interactions* $\gamma_i, i \in [1, n]$ *is in the persistent set* $\Gamma(\eta_0)$, *then by the second condition of persistent set in Definition 5.3.6, the interactions in persistent set* $\Gamma(\eta_0)$ *will still be enabled in* η_n, *which controdicts the assumption that* η^w *is a deadlock node.*

Thus, there is at least one interaction $\gamma_i, i \in [1, n]$ *in the persistent set* $\Gamma(\eta_0)$. *Assume the first such interaction is* $\gamma_j, j \in [1, n]$, *then for all interactions* $\gamma_k, k < j$, *we have* γ_j *is independent with* γ_k. *Thus,* γ_j *can be moved to the beginning of the path. The new path would be the same one with* $\gamma_1 \dots \gamma_n$, *except* γ_j *has been moved to the first.*

Then applying 5.3.4, we can conclude that $\pi(c) \models \eta'_n$.

Case 2 : if η^w is not a deadlock node, but covered by some other node in the same path, assume the covering node is η_i, $i \in [0, n-1]$. Assume also no interactions in the persistent set $\Gamma(\eta_0)$ occur in γ_j, $j \in [1, i]$. Then we know that interactions in $\Gamma(\eta_0)$ are also enabled in η_i and η_n as well. Then according to the third condition of persistent set in Definition 5.3.6, we know that at least one interaction in $\Gamma(\eta_0)$ occurs in γ_j, $j \in [i+1, n]$. Let γ_k, $k \in [i+1, n]$ be the first such interaction, then for the same reason as the case 1, γ_k can be shifted to the beginning of the path. The new path would be the same one with $\gamma_1 \ldots \gamma_n$, except γ_k has been moved to the first.

Then applying 5.3.4, we can conclude that $\pi(c) \models \eta'_n$.

Case 3 : if η^w is covered by some node in another path. It is equivalent to prove the conclusion for another path. Since our models are finite branching, we are guaranteed that there is a path such that argument in case 2 applies.

Case 4 : if η^w is not covered, then it is possible to extend the path, where case 2 or case 3 applies.

5.4 Experimental evaluation

We have implemented the proposed verification technique in our prototype model checker for BIP. In the experimental evaluation, we took a subset of the benchmarks from the previous experiments, which have certain component symmetries. These include the ticket mutual exclusion protocol in star topology, a leader election protocol in ring topology, and a consensus protocol in star topology. All these benchmarks are scalable in terms of the number of components, and all are infinite-state, and they all use data transfer on interactions. We model them in BIP and for each benchmark, we create a safe and an unsafe version, and for each version, we have 10 instances. All the experiments are performed on a 64-bit Linux PC with a 2.8 GHz Intel i7-2640M CPU, with a memory limit of 4Gb and a time limit of 300 seconds per benchmark.

We run the following configurations of our prototype tool and compare the running time for solving the benchmarks: 1) plain lazy abstraction of BIP (represented as 'plain' in the plots); 2) lazy abstraction with persistent set reduction (represented as 'pset' in the plots); 3) lazy abstraction with simultaneous set reduction (represented as 'simset' in the plots); 4) our new algorithm (represented as 'sympor' in the plots). For simplicity, we call this new algorithm as reduction under symmetry in the sequel. We also compare to a variant of the the state-of-the-art invariant verification algorithm IC3 [37]. We do not compare with DFinder [21], since it does not handle data transfer.

The detailed statistics data is attached in the Appendix A.8.

5.4.1 Scatter plots

In the first experiment, we compare our new algorithm to the others in terms of the running time for solving each benchmark. The scatter plots are shown in the Figure 5.16, Figure

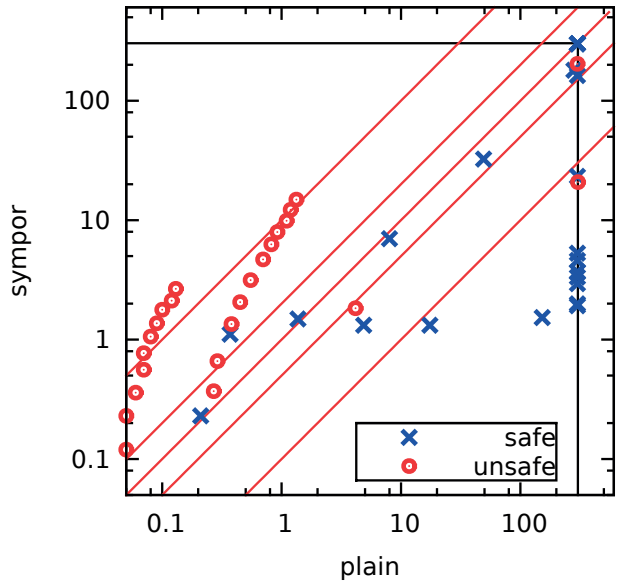

Figure 5.16 – Lazy abstraction vs. lazy abstraction with reduction under symmetry

5.17, Figure 5.18 and Figure 5.19. In all plots, symbol × represents a safe benchmark, and ○ represents an unsafe benchmark. A point in the plots indicates the analysis time of the algorithms represented by x-axis and y-axis.

In Figure 5.16, Figure 5.17 and Figure 5.18, we compare our new algorithm 'sympor' to plain lazy abstraction, lazy abstraction with persistent set reduction and lazy abstraction with simultaneous set reduction respectively. The results show that our new algorithm is always faster to prove the correctness, while for unsafe benchmark models, our new algorithm is less faster than the others. In Figure 5.19, we compare our new algorithm to IC3 and we find that for both safe benchmarks and unsafe ones, our new algorithm is always more efficient.

The result that our new algorithm is more efficient to prove the correctness is as expected, since in our new algorithm more reduction power is gained by exploiting symmetry. The percentage of the successful reduction for each solvable benchmark mdoel is listed in Table 5.2. This percentage does not measure the reduction of the search space, but the ratio of successful reductions over all attempts. That is, a positive percentage means that in some node expansion, a successful reduction is achieved and explorations of some interactions are ignored, but we do not count how many interactions are ignored. Percentage '1' in the table means that in every node expansion, a successful reduction is achieved. Comparing to the percentage of persistent set reduction in Table 4.1, we can see that our new algorithm achieves more reductions. For instance, for the ticket mutual exclusion protocol, persistent set without considering symmetry is unable to obtain any reduction, which is, however overcame by our

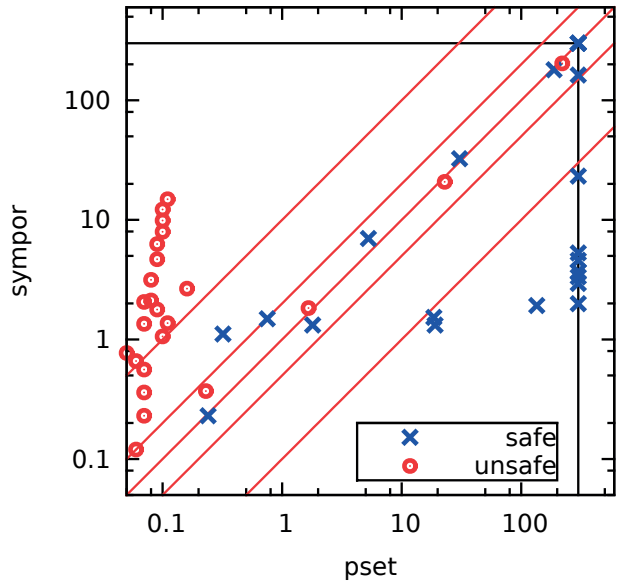

Figure 5.17 – Lazy abstraction with persistent set reduction vs. lazy abstraction with reduction under symmetry

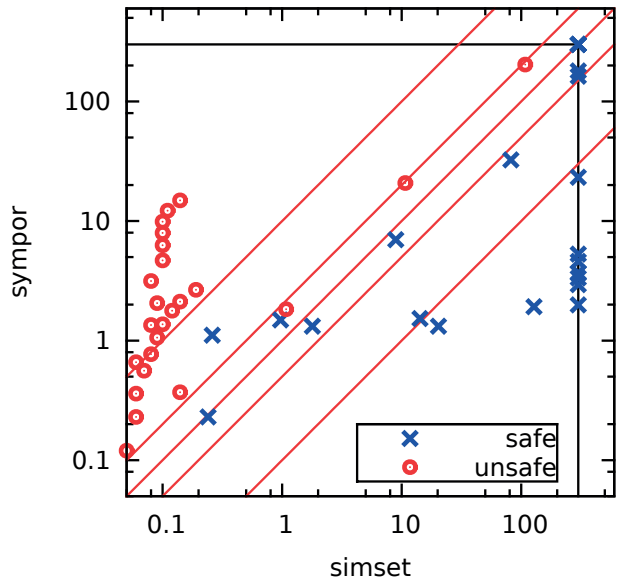

Figure 5.18 – Lazy abstraction with simultaneous set reduction vs. lazy abstraction with reduction under symmetry

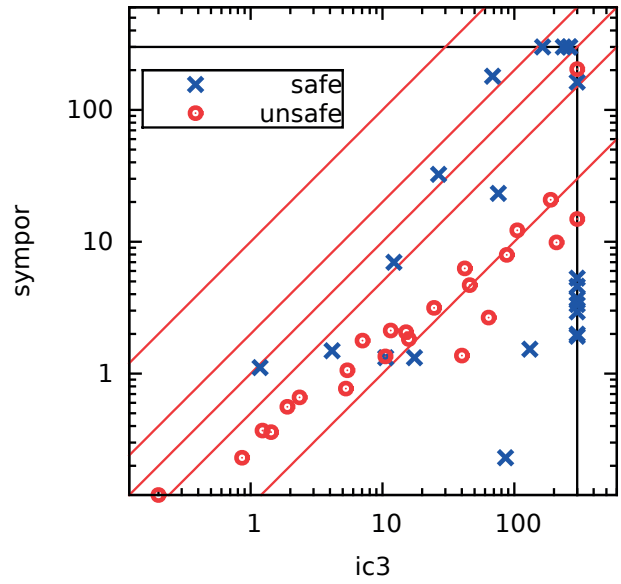

Figure 5.19 – IC3 vs. lazy abstraction with reduction under symmetry

new algorithm.

model	percentage	model	percentage
leader_election_safe_02	1.000000	leader_election_safe_03	1.000000
leader_election_safe_04	1.000000	leader_election_safe_05	1.000000
leader_election_safe_06	1.000000	leader_election_safe_07	1.000000
leader_election_safe_08	1.000000	leader_election_safe_09	1.000000
leader_election_safe_10	1.000000	leader_election_safe_11	1.000000
leader_election_unsafe_02	0.466667	leader_election_unsafe_03	0.555556
leader_election_unsafe_04	0.465347	leader_election_unsafe_05	0.395445
quorum_safe_02	0.297872	quorum_safe_03	0.327189
quorum_safe_04	0.320191	quorum_safe_05	0.285627
quorum_unsafe_02	0.400000	quorum_unsafe_03	0.400000
quorum_unsafe_04	0.400000	quorum_unsafe_05	0.400000
quorum_unsafe_06	0.400000	quorum_unsafe_07	0.400000
quorum_unsafe_08	0.400000	quorum_unsafe_09	0.400000
quorum_unsafe_10	0.400000	quorum_unsafe_11	0.400000
ticket_safe_02	0.230769	ticket_safe_03	0.142857
ticket_safe_04	0.049645	ticket_safe_05	0.037185
ticket_unsafe_02	0.500000	ticket_unsafe_03	0.600000
ticket_unsafe_04	0.666667	ticket_unsafe_05	0.714286
ticket_unsafe_06	0.750000	ticket_unsafe_07	0.777778

ticket_unsafe_08	0.800000	ticket_unsafe_09	0.818182
ticket_unsafe_10	0.833333	ticket_unsafe_11	0.846154

Table 5.2 – Percentage of partial order reduction under symmetry

In order to understand the result that for unsafe benchmarks, our new algorithm takes more time to detect the counterexamples, we draw the plots that show the running time of each subroutine, as in the previous sections. For our new algorithm, the subroutines constitute the computation of transfer function, i.e. the ART node expansion, the computation of persistent set, the computation of node coverage, the cycle detection, the counterexample analysis and abstraction refinement, and also the computation of independence relation. The result is depicted in Figure 5.20. We split the plot into two parts, the first one depicts the results with time greater than 5 seconds, and the second one depicts the rest.

The plots show that for the models that can be solved quickly, e.g. within 5 seconds, the computation of independence relation contributes a major part to the total running time. Most of these models are the unsafe ones. We believe that this is the main reason of taking longer to detect counterexamples for our new algorithm. The other algorithms do not have this cost of independence relation computation. They use static analysis of the system model to approximate the independence relation, whose cost is negligible. The plot also shows that for models that take more verification time, the costs of coverage check and cycle detection are significant.

5.4.2 Cumulative plots

In Figure 5.21, we plot the cumulative time (x-axis) of solving a number of benchmarks (y-axis). A point (x, y) in the plot tells us the total number y of benchmarks, each of which can be verified in the given time bound x by the corresponding method. We remark that time x is not the accumulation of the analysis time of all y benchmarks. We see that our new algorithm can always solve more instances in a given time bound than IC3, while comparing to other algorithms, it is not always faster, but can still solve more instances in a larger time bound. This tells that the analysis time of our new algorithm grows slower than the other algorithms.

In Figure 5.22 and Figure 5.23, we plot the cumulative time of solving safe and unsafe benchmarks respectively. For safe benchmarks, our new algorithm is always more efficient than all the others. While for unsafe benchmarks, our new algorithm is less faster, due to the reasons we have discussed above.

5.5 Related work

Relevant partial order reduction and abstraction techniques have already been discussed in the previous chapter. Exceptionally, we remark that in [144], the authors explore an idea, which

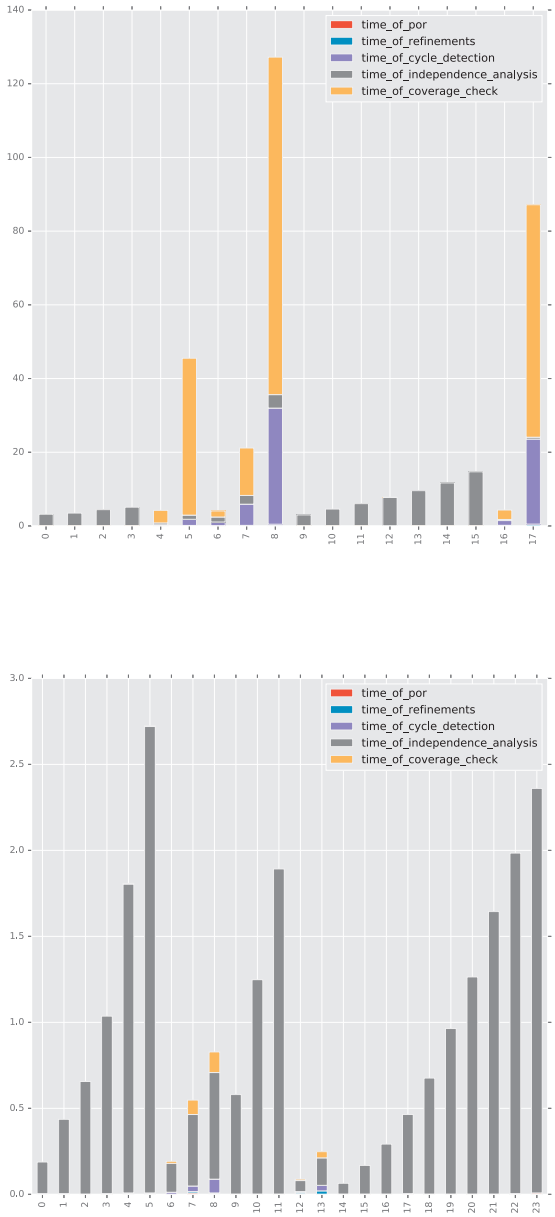

Figure 5.20 – Runtime of lazy abstraction with reduction under symmetry subroutines

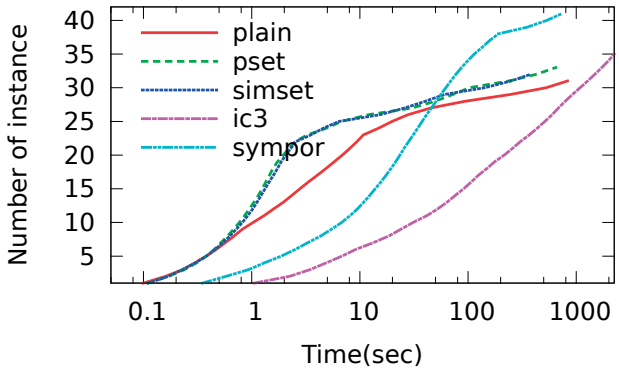

Figure 5.21 – Cumulative plot of time for all benchmarks

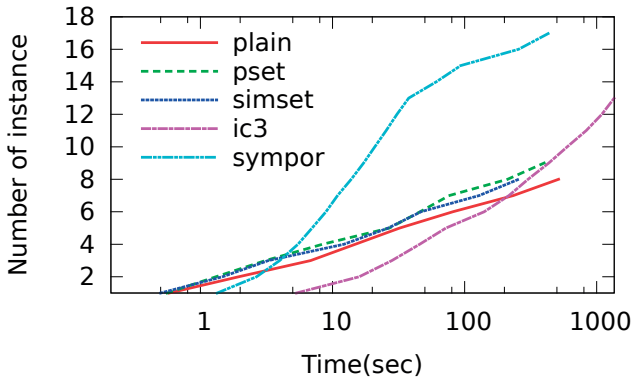

Figure 5.22 – Cumulative plot of time for safe benchmarks

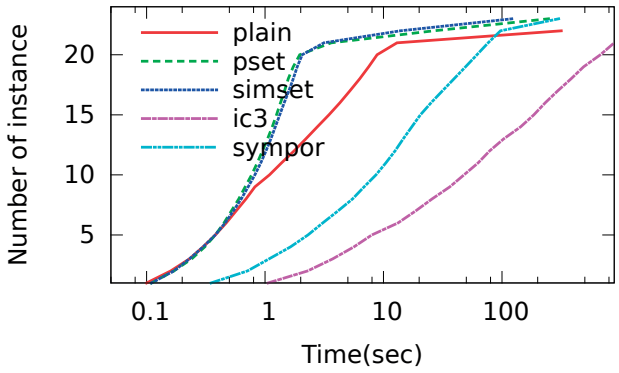

Figure 5.23 – Cumulative plot of time for unsafe benchmarks

is similar to our simultaneous set approach, to compute the reachable states of a Petri net as a covering step graph. Independent transitions of a Petri net are also fired simultaneously under certain conditions. However, in their work no abstraction is used.

State space symmetries have been extensively investigated in model checking community over the decades, leading to a variety of symmetry reduction techniques [61, 45, 98, 63]. However, most work focuses on finite state systems. We refer to [121, 146] for the detailed review.

In [53], the authors propose a symmetry aware counterexample guarded abstraction refinement technique for replicated non-recursive C programs. Their abstraction technique is eager in the sense that an abstraction model is constructed first, which differs from our lazy abstraction technique. In [57, 97], the authors investigate how to combine symmetry reduction with ample-set-based partial order reduction. However, both work focus on finite state models, and no abstraction techniques are used.

6 Design and verification of parameterized systems in BIP

In this chapter, we focuse on the modeling and verification of parameterized systems, where the number of components in the system is not fixed a priori. The verification problem asks whether the property holdes for all system instances. Many efforts have been made in the past decades to identify decidable fragments and draw the boundaries between decidability and undecidability. The decidability depends on several factors, with the most important being the underlying communication graph (e.g. rings, stars, cliques), and the means of synchronization (e.g. token passing with/without information-carrying tokens, broadcast). However, there is no uniform framework that can capture various computational models that occur in the literature, or enables automatic verification of parameterized systems. As discussed in Chapter 1, there is also a gap between the mathematical formalisms from the parameterized verification research and the verification practice. That is, in order to verify a parameterized system, the engineers have to understand the underlying mathematical model of the system and then identify the suitable verification techniques if any. This might be a difficult task since it requires a deep understanding of the subtle differences between various mathematical models. Thus, a uniform modeling and automatic verification framework would be useful.

We first present a uniform modeling framework for parameterized systems, by extending the current BIP component framework introduced in Chapter 2. The core of this framework is a formal language for system architecture and communciation primitives, called first order interaction logic. We show that many interesting parameterized systems can be uniformly specified in this logic. Then we present how to perform automated parameterized verification within our new framework. We also present some decidability results for the verification of parameterized BIP models.

This chapter is based on the following publication:

– *Parameterized systems in BIP: design and model checking*, Konnov, Igor and Kotek, Tomer and **Wang, Qiang** and Veith, Helmut and Bliudze, Simon and Sifakis, Joseph, Proceedings of the 27th International Conference on Concurrency Theory (CONCUR 2016), pages 30–1, 2016, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

The idea of modeling parameterized systems using first order interaction logic was initiated by Prof.Joseph Sifakis. The author formalized it and applied it to the verification of parameterized systems with the help of other collaborators. The author also did the prototype implementation, and proved the decidability results.

6.1 Parameterized BIP without priorities

We rely on the notions of BIP component type and interaction introduced in Chapter 2. Recall that a component type is a transition system $\mathbb{B} = \langle \mathbb{V}, \mathbb{L}, \mathbb{P}, \mathbb{E}, \ell \rangle$ over the finite sets $\mathbb{L}$ and $\mathbb{P}$. We will put the following restrictions on the parameterized BIP framework: 1) states of the components do not have specific internal structure, or integer variables; 2) we do not consider interaction priorities.

Since in parameterized systems, we have an unbounded number of components communicating with each other, thus, the number of interactions is unbounded, and an interaction may also involve unbounded number of actions, the explicit representation of interactions as sets, which is the way how we represent interactions in Chapter 2, becomes infeasible.

In this dissertation, we propose the first order interaction logic as a uniform and formal language for system topologies and coordination mechanisms in parameterized systems.

6.1.1 FOIL: First order interaction logic

In this section, we fix a tuple of component types $\langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1} \rangle$.

FOIL vocabulary. For each port $p \in \mathbb{P}_i$ of an i^{th} component type, we introduce a unary *port predicate* with the same name p. Further, we introduce a tuple of constants $\bar{n} = \langle n_0, \ldots, n_{k-1} \rangle$, which represent the number of components of each type. We also assume the standard vocabulary of Presburger arithmetics, that is, $\langle 0, 1, \leq, + \rangle$.

FOIL syntax. Assume an infinite set of index variables $\mathscr{I}$. We say that ψ is a first order interaction logic formula, if it is constructed according to the following grammar:

$$\psi ::= p(i) \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \exists i :: type_j : \phi.\, \psi \mid \forall i :: type_j : \phi.\, \psi\,,$$

where $p \in \mathbb{P}_0 \cup \cdots \cup \mathbb{P}_{k-1}$, $i \in \mathscr{I}$, and ϕ is a formula in Presburger arithmetic over index variables and the vocabulary $\langle 0, 1, \leq, +, \bar{n} \rangle$.

Informally, $\mathbf{Q}\ i :: type_j : \phi.\, \psi$, where $\mathbf{Q} \in \{\exists, \forall\}$, restricts the index variable i to be associated with the component type $\mathbb{B}_j$. Notice, however, that the syntax of FOIL does not enforce type correctness of ports. For instance, one can write a formula $\exists i :: type_j : p(i)$ with some $p \notin \mathbb{P}_j$. While this formula is syntactically correct, since it is not in line with Definition 2.3.2 of

interaction given in Section 2.3, where it requires that an interaction can only involve a port defined in some component. To this end, we say that a FOIL formula is *natural*, if for each of its subformulae $\mathbf{Q}\ i :: type_j : \phi.\psi(i)$, for $\mathbf{Q} \in \{\exists, \forall\}$, and every atomic formula $p(i)$ of ψ, it holds that $p \in \mathbb{P}_j$. From here on, we assume that all FOIL formulae we consider in this disseration are natural.

FOIL semantics. We give semantics of a FOIL formula by the means of structures. A *first-order interaction logic structure* (FOIL structure) is a pair $\xi = (\mathbb{N}, \alpha_\xi)$, which consists of the set of natural numbers, i.e. the domain of ξ, the interpretation α_ξ of all unary predicates and of the constants $\bar{n}$. The symbols 0, 1, $\leq$, and + have the natural interpretations over $\mathbb{N}$.

By $\sigma : \mathscr{I} \to \mathbb{N}$ we denote an assignment that gives values to free variables in ψ, and by $\sigma[x \mapsto j]$ we denote the assignment that differs from σ in that the index variable x is mapped to the value j. For a FOIL structure ξ and an assignment σ, the semantics of FOIL is formally given as follows (the semantics of Boolean operators and universal quantifiers is defined in the standard way):

$$
\begin{array}{lll}
\xi, \sigma \models_{\text{FOIL}} p(i) & \textit{iff} & \alpha_\xi(p) \text{ is true on } \sigma(i) \\
\xi, \sigma \models_{\text{FOIL}} \exists i :: type_j : \phi.\ \psi & \textit{iff} & \text{there is } l \in [0, \alpha_\xi(n_j)) \text{ such that} \\
 & & \xi, \sigma[i \mapsto l] \models_{\text{FOIL}} \psi \text{ and } \xi, \sigma[i \mapsto l] \models_{\text{FO}} \phi
\end{array}
$$

where $\models_{\text{FO}}$ to denotes the standard 'models' relation of first-order logic.

Finally, for a FOIL formula ψ without free variables and a structure ξ, we write $\xi \models_{\text{FOIL}} \psi$, if $\xi, \sigma_0 \models_{\text{FOIL}} \psi$ for the valuation σ_0 that assigns 0 to every index $i \in \mathscr{I}$. Since ψ has no free variables, our choice of σ_0 is arbitrary: for all σ we have $\xi, \sigma \models_{\text{FOIL}} \psi$ if and only if $\xi, \sigma_0 \models_{\text{FOIL}} \psi$.

Decidability. It is easy to show that although checking validity of a FOIL formula is undecidable, FOIL contains an important fragment, which is known to be decidable:

Theorem 6.1.1 (Decidability of FOIL) *The following results about FOIL hold:*

(i) Validity of FOIL sentences is undecidable.

(ii) Validity of FOIL sentences in which all additions are of the form $i + 1$ is decidable.

Proof 6.1.2 *(i) FOIL contains Presburger arithmetic with unary predicates, in which satisfiability is undecidable [85]. (ii) The formula $j = i + 1$ is definable in FOIL by $i \leq j \wedge j \neq i \wedge \psi_{consecutive}(i, j)$, where $\psi_{consecutive}(i, j) = \forall \ell :: type_t.\ (j \leq \ell \wedge \ell \leq i) \to (\ell = i \vee \ell = j)$, where t is the type of i and j. Hence, we can rewrite any FOIL sentence ψ in which all additions are of the form $i + 1$ as an equi-satisfiable first-order logic sentence ψ' without using addition (+). The sentence ψ' belongs to WS1S, the weak monadic second order theory of $(\mathbb{N}, 0, 1, \leq)$, which is decidable, see [137].*

In the following, we only refer to addition with $i+1$.

6.1.2 Interactions as FOIL structures

In contrast to Definition 2.3.2 of a standard interaction, which is represented explicitly as a finite set of ports, we use first order interaction logic formulae to define all the possible interactions in parameterized systems. Our key insight is that each structure of a formula uniquely defines at most one interaction, and the set of all possible interactions is the union of the interactions derived from the structures satisfying the formula.

Intuitively, if $p(i)$ evaluates to true in a structure, then the i^{th} instance of the respective component type — uniquely identified by a port — takes part in the interaction identified with the structure. Thus, we can reconstruct a standard BIP interaction from a FOIL structure by taking the set of ports, whose indices are evaluated to true by the unary predicates.

Formally, given a FOIL structure $\xi = (\mathbb{N}, \alpha_\xi)$, we define the $\gamma_\xi = \{(p,m) \mid j \in [0,k),\ p \in \mathbb{P}_j,\ m \in [0, \alpha_\xi(n_j)),\ \alpha_\xi(p)(m) = \mathit{true}\}$, where the notation (p,m) denotes the port p of the m^{th} component.

Notice that not every γ_ξ is an interaction in the sense of Definition 2.3.2. Indeed, γ_ξ may include several ports of the same component. We say that ξ *induces an interaction*, if γ_ξ is an interaction in the sense of Definition 2.3.2.

Definition 6.1.3 (Parameterized BIP Model) *A parameterized* BIP *model is a tuple* $\mathcal{M}_{\text{PBIP}} = \langle \mathbb{B}, \bar{n}, \psi, \epsilon \rangle$, *where* $\mathbb{B} = \langle \mathbb{B}_0, \ldots, \mathbb{B}_{k-1} \rangle$ *is a tuple of component types,* ψ *is a sentence in FOIL over the port predicates and a size tuple* $\bar{n} = \langle n_0, \ldots, n_{k-1} \rangle$, *and* ϵ *is a linear constraint over* $\bar{n}$.

The tuple $\bar{n}$ consists of the size parameters for all component types, and the constraint ϵ restricts these parameters, e.g. the formula $n_0 = 1 \wedge n_1 \geq 10$ requires every instance of the parameterized BIP model to contain only one component of the first type and at least ten components of the second type. The sentence ψ in FOIL restricts both the system topology and the communication mechanisms.

Definition 6.1.4 (PBIP Instance) *Given a parameterized* BIP *model* $\mathcal{M}_{\text{PBIP}} = \langle \mathbb{B}, \bar{n}, \psi, \epsilon \rangle$ *and a tuple of natural numbers* $\bar{N}$, *a* PBIP instance *is a BIP model* $\mathcal{M}_{\text{BIP}} = \langle \mathscr{B}, \Gamma \rangle$, *where* $\mathscr{B}$ *and* Γ *are defined as follows:*

1. *the numbers* $\bar{N}$ *satisfy the size constraint* ϵ,
2. *the set of components* $\mathscr{B}$ *is* $\{\mathbb{B}_i[j] \mid i \in [0,k), j \in [0,N_j)\}$, *and*
3. *the set of interactions* Γ *is the set of all interactions* γ_ξ *satisfying* ψ *and referring to the ports of components with indices up to the numbers in* $\bar{N}$, *that is,* $\xi \models_{\text{FOIL}} \psi$ *and* $\alpha_\xi(\bar{n}) = \bar{N}$.

For a given PBIP instance model, its semantics is defined as in Definition 2.3.7. The labeled transition system semantics for a parameterized BIP model is then the union of all the transition systems, one for each PBIP instance.

Example 6.1.5 (Broadcast in a star) *Let* $\langle\langle \mathbb{B}_0, \mathbb{B}_1\rangle, \langle n_0, n_1\rangle, \psi, \epsilon\rangle$ *be a parameterized* BIP *model with two component types and the size constraint* $\epsilon \equiv (n_0 = 1)$. *We also assume component type* $\mathbb{B}_0$ *has only one port called send and component type* $\mathbb{B}_1$ *defines only one port called receive, i.e.* $\mathbb{P}_0 = \{send\}$ *and* $\mathbb{P}_1 = \{receive\}$. *The FOIL formula* $\psi = \exists i :: type_1.\ send(i)$ *specifies broadcast from the component* $\mathbb{B}_0[0]$, *the center of the star, to the leaves of type* $\mathbb{B}_1$. *The set of interactions defined by* ψ *consists of all sets of ports of the form* $\{(send, 0)\} \cup \{(receive, d) \mid d \in D)\}$ *for all* $D \subseteq [0, n_1)$ *(including* $D = \emptyset$*).*

Example 6.1.6 (Milner's scheduler [60]) *The components of a token ring schedule tasks in succession along the ring. We follow the formulation by Emerson & Namjoshi [60]. The component type* $\mathbb{B}_0$ *is:*

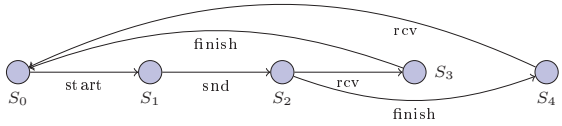

Figure 6.1 – Component type of Milner's scheduler

A component has the token if it is in locations S_0, S_1, *or* S_4. *A component must have the token when it initiates a task (by interacting on port start). The token is then sent to the component's neighbor by interaction on port snd. The component then waits until (a) its initiated task has finished, and (b) the component has received the token again. When both (a) and (b) have occurred, the component may initiate a new task. Note that (a) and (b) may occur in either order.*

The parameterized BIP *model of Milner's scheduler is* $\langle\langle \mathbb{B}_0\rangle, \langle n_0\rangle, \psi, true\rangle$, *where*

$$\begin{aligned} \psi &= \exists i, j :: type_0 : (j = (i+1) \bmod n_0).\ snd(i) \wedge rcv(j) \wedge \psi_{only}(i, j) \\ \psi_{only}(i, j) &= \forall \ell :: type_0 : \ell \neq i \wedge \ell \neq j.\ \neg(snd(\ell) \vee rcv(\ell)) \end{aligned}$$

ψ *is a formula without free variables which holds for a structure* ξ *if its induced interaction* γ_ξ *is a send-receive interaction along some edge* $i \to j$ *of the ring, where* j *is* $i+1$ *modulo* n_0. $\psi_{only}(i, j)$ *excludes any component other than* i *and* j *from participating in the interaction. The modulo notation abbreviates the expression* $(i = n_0 - 1 \to j = 0) \wedge (i < n_0 - 1 \to j = i + 1)$. *We discuss how to ensure that exactly one component starts with the token in Section 6.5.4.*

Example 6.1.7 (Barrier [28]) *Here we consider a barrier synchronization protocol, cf. [28, Example 6.6]. The component type* $\mathbb{B}_0$ *is: (the self-loops are labeled by the ports* $loop_{ms}$, $loop_{nt}$, *and* $loop_{sl}$*)*

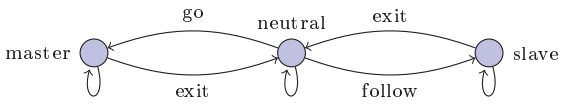

Figure 6.2 – Component type of a barrier synchronization protocol

The initial location is neutral. *A synchronization episode consists of three stages: (i) First, a single component enters the barrier by moving to* master. *(ii) Then, each of the others components moves to* slave. *(iii) Finally, the master triggers a broadcast and all components leave the barrier into* neutral.

The parameterized BIP *model of the barrier synchronization protocol is* $\langle\langle \mathbb{B}_0\rangle, \langle n_0\rangle, \psi, true\rangle$, *where* $\psi = \psi_{go} \vee \psi_{follow} \vee \psi_{exit}$, *and*

$$\begin{array}{lcl} \psi_{go} & = & \exists i :: type_0.\ go(i) \quad \wedge \forall j :: type_0 : i \neq j.\ loop_{nt}(j) \\ \psi_{follow} & = & \exists i :: type_0.\ loop_{ms}(i) \wedge \forall j :: type_0 : i \neq j.\ \psi_{fllw\text{-}loop}(j) \\ \psi_{exit} & = & \forall i :: type_0.\ exit(i) \\ \psi_{fllw\text{-}loop}(j) & = & follow(j) \vee loop_{nt}(j) \vee loop_{sl}(j) \end{array}$$

ψ_{go}, ψ_{follow}, *and* ψ_{exit} *describe the interactions of stages (i), (ii), and (iii) respectively.*

Example 6.1.8 (Semaphore) *This example has two component types, the semaphore type* $\mathbb{B}_0$ *on the right and the process type* $\mathbb{B}_1$ *on the left:*

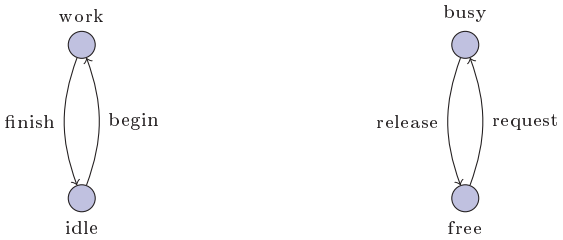

Figure 6.3 – Component type of a semaphore example

The system has exactly one semaphore and may have an unbounded number of processes. The components communicate by pairwise rendezvous on a star whose center is the semaphore. The processes start in the initial location idle *and the semaphore starts in the initial location* free. *Any process c may rendezvous with the semaphore by an interaction between the* begin *port of the process and the* request *port of the semaphore. Once such an interaction occurs, the only possible next interaction is between the same process c and the semaphore; this interaction consists of the process c's* finish *port and the semaphore's* release *port. The semaphore is now free to interact with any process* c'.

We model this semaphore example by a parameterized BIP *model* $\langle\langle \mathbb{B}_0, \mathbb{B}_1\rangle, \langle n_0, n_1\rangle, \psi, \epsilon\rangle$, *where*

$\epsilon \equiv (n_0 = 1)$, $\psi = \psi_{request} \vee \psi_{release}$, *and*

$$\begin{array}{lcl} \psi_{request} & = & \exists i :: type_0 : i = 0.\ \exists j :: type_1.\ request(i) \wedge begin(j) \wedge \psi_{only}(j) \\ \psi_{release} & = & \exists i :: type_0 : i = 0.\ \exists j :: type_1.\ release(i) \wedge finish(j) \wedge \psi_{only}(j) \\ \psi_{only}(j) & = & \forall \ell :: type_0 : \ell \neq j.\ \neg(begin(\ell) \vee finish(\ell)) \end{array}$$

$\psi_{request}$ *describes the* request-begin *interactions.* $\psi_{release}$ *describes the* release-finish *interactions.* $\psi_{only}(j)$ *excludes any component of type* $\mathbb{B}_1$*, other than* j*, from participating in the interaction.*

Example 6.1.9 (Guarded protocol [59]) *This example considers a class of parameterized systems, called guarded protocols [59].*

We assume a single component type and specialize the atomic propositions to be the finite set of control locations. Assume $\phi(j)$ *is a boolean formula over atomic propositions of a component, indexed with a free index variable* j*. A disjunctive guard is of the form* $\exists j :: type_0 : j \neq i.\phi(j)$*, where* i *is a free index variable of the same type with* j *and* $\phi(j)$ *is a disjunction over atomic propositions of component* j*. A disjunctive guarded protocol is a parameterized system, whose transitions are associated with disjunctive guards.*

A disjunctive guard protocol can be specified in one-type parameterized BIP *model* $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$ *as follows. On each control location* q *of the component type, there is a self-loop transition* $loop_q$*. For a given guarded transition* $(q, \exists i, j :: type_0 : j \neq i.\phi(j), p, q')$*, where* p *is the port,* $\phi(j) = q_1(j) \vee q_2(j) \ldots \vee q_k(j)$*, and* $q_1, q_2, \ldots, q_k$ *are control locations, it can be simulated by a pairwise rendezvous defined by the following FOIL formula:* $\exists i, j :: type_0 : j \neq i.p(i) \wedge (loop_{q_1}(j) \vee loop_{q_2}(j) \ldots \vee loop_{q_k}(j))$*, where* $loop_{q_1}, \ldots, loop_{q_k}$ *are self-loop transitions on locations* $q_1, \ldots, q_k$ *respectively.*

Similarly, a conjunctive guard is of the form $\forall j :: type_0.j \neq i.\phi(j)$*, and a conjunctive guarded protocol is a parameterized system, whose transitions are associated with conjunctive guards. Given a guarded transition* $(q, \forall j :: type_0 : j \neq i.\phi(j), p, q')$*, where* $\phi(j)$ *is of the same form as above, it can be simulated by the following FOIL formula:* $\exists i :: type_0.\forall j :: type_0 : j \neq i.\ \ p(i) \wedge (loop_{q_1}(j) \vee loop_{q_2}(j) \ldots \vee loop_{q_k}(j))$*, where* $loop_{q_1}, \ldots, loop_{q_k}$ *are self-loop transitions on locations* $q_1, \ldots, q_k$ *respectively.*

6.2 Parameterized model checking

In this section, we review the syntax and semantics of the indexed version of CTL^*, called $ICTL^\star$, which is often used to specify the properties of parameterized systems [28]. Though we use indexed temporal logics to define the standard parameterized model checking problem, these logics are not the focus of this paper. Further, we introduce the parameterized model checking problem for parameterized BIP design, and show its undecidability.

Syntax. For a set of index variables $\mathscr{I}$, the *ICTL** *state formulae* are written according to the grammar:

$$\theta ::= true \mid at(q,i) \mid \neg\theta \mid \theta_1 \wedge \theta_2 \mid \exists i :: type_j : \phi.\ \theta \mid \forall i :: type_j : \phi.\ \theta \mid \mathbf{E}\varphi \mid \mathbf{A}\varphi,$$

where q is a location from $\bigcup_{0\le j<k} \mathbb{L}_j$, and i is an index from the set $\mathscr{I}$, and φ is a path formula (to be defined below), and ϕ is a formula in Presburger arithmetic over size variables $\bar{n}$ and index variables from the set $\mathscr{I}$.

The *path formulae* are written according to the following grammar:

$$\varphi ::= \theta \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi_1 \mathbf{U} \varphi_2, \quad \text{where } \theta \text{ is a state formula.}$$

Example 6.2.1 *In ICTL***, the response property in Example 6.1.6 can be written as* $\forall i :: type_0 : 0 \le i < n_0.\ \mathbf{AG}(at(S_0,i) \rightarrow \mathbf{AF}\ at(S_1,i))$*, and the mutual exclusion property in Example 6.1.8 can be written as* $\neg(\exists i,j :: type_1 : 0 \le i < n_1 \wedge 0 \le j < n_1 \wedge j \ne i : (at(busy,i) \wedge at(busy,j)))$.

Semantics. Given a BIP model $\mathscr{M}_{\text{BIP}}$ with the transition system $\mathscr{T}_{BIP}$ (i.e. $\mathscr{T}_{BIP} = \langle C, \Sigma, R, C_0\rangle$ by definition 2.3.7), we inductively define the semantics of *ICTL** formulae. We briefly discuss semantics to highlight the role of quantifiers in indexed temporal logics. For further discussions and additional definitions, we refer the reader to a textbook, e.g. see [47].

State formulae are interpreted over a configuration s and a valuation of index variables $\sigma : \mathscr{I} \rightarrow \mathbb{N}$ (the semantics of Boolean operators and universal quantifiers is defined in the standard way):

$\mathscr{T}_{BIP}, s, \sigma \models_{\text{ICTL}^\star} at(q,i)$	*iff*	$q = s(j, \sigma(i))$, where $q \in \mathbb{L}_j$
$\mathscr{T}_{BIP}, s, \sigma \models_{\text{ICTL}^\star} \exists i :: type_j : \phi.\ \theta$	*iff*	for some $l \in [0, N_j)$, both $\mathscr{T}_{BIP}, s, \sigma[i \mapsto l] \models_{\text{ICTL}^\star} \theta$ and $\langle \mathbb{N}, 0, 1, \le, +, \bar{N}\rangle, \sigma[i \mapsto l] \models_{\text{FO}} \phi$
$\mathscr{T}_{BIP}, s, \sigma \models_{\text{ICTL}^\star} \mathbf{E}\varphi$	*iff*	$\mathscr{T}_{BIP}, \rho, \sigma \models_{\text{ICTL}^\star} \varphi$ for some infinite path ρ starting from s

Path formulae are interpreted over an infinite path ρ, and the valuation function σ as follows (the semantics for Boolean operators and temporal operators **F** and **G** is defined in the standard way):

$\mathscr{T}_{BIP}, \rho, \sigma \models_{\text{ICTL}^\star} \theta$	*iff*	$\mathscr{T}_{BIP}, s, \sigma \models_{\text{ICTL}^\star} \theta$, where s is the first configuration of the path ρ
$\mathscr{T}_{BIP}, \rho, \sigma \models_{\text{ICTL}^\star} \mathbf{X}\varphi$	*iff*	$\mathscr{T}_{BIP}, \rho^1, \sigma \models_{\text{ICTL}^\star} \varphi$
$\mathscr{T}_{BIP}, \rho, \sigma \models_{\text{ICTL}^\star} \varphi_1 \mathbf{U} \varphi_2$	*iff*	$\exists j \ge 0. \forall i < j.\ \mathscr{T}_{BIP}, \rho^j, \sigma \models_{\text{ICTL}^\star} \varphi_2$ and $\mathscr{T}_{BIP}, \rho^i, \sigma \models_{\text{ICTL}^\star} \varphi_1$,

where ρ^i is the suffix of the path ρ starting with the i^{th} configuration.

Finally, given a formula φ without free variables, we say that $\mathscr{T}_{BIP}$ satisfies φ, written as $\mathscr{T}_{BIP} \models_{\text{ICTL}^\star} \varphi$, if $\mathscr{T}_{BIP}, s_0, \sigma_0 \models_{\text{ICTL}^\star} \varphi$ for the valuation σ_0 that assigns zero to each index from the set $\mathscr{I}$.

The choice of σ_0 is arbitrary, as for all σ, it holds that $\mathscr{T}_{BIP}, s_0, \sigma \models_{\text{ICTL}^\star} \varphi$ if and only if $\mathscr{T}_{BIP}, s_0, \sigma_0 \models_{\text{ICTL}^\star} \varphi$.

Now we are at a position to formulate the parameterized model checking problem for BIP:

Problem 6.2.2 (Parameterized model checking) *The verification problem for a parameterized* BIP *model* $\langle \mathbb{B}, \bar{n}, \psi, \epsilon \rangle$ *and an ICTL* state formula* θ *without free variables, is whether every instance* $\mathscr{M}_{\text{BIP}}$ *satisfies* θ.

Not surprisingly, Problem 6.2.2 is undecidable in general.

Theorem 6.2.3 (Undecidability) *Given a two-counter machine* M_2*, one can construct an ICTL*-formula* $\mathbf{G}\neg halt$ *and a parameterized BIP model* $\mathscr{M}_{\text{PBIP}} = \langle \mathbb{B}, \bar{n}, \psi, \epsilon \rangle$ *that simulates* M_2 *and has the property:* M_2 *does not halt if and only if* $\mathscr{M}_{\text{BIP}} \models \mathbf{G}\neg halt$ *for all instances of* $\mathscr{M}_{\text{PBIP}}$.

Proof 6.2.4 *The idea is to simulate a multi-valued token passing ring system within parameterized BIP, and in [60] the authors have shown how to simulate a two-counter machine with a multi-valued token ring, thus, combining them gives us the full proof of the theorem.*

Fix a finite set T *of token values with* $|T| \geq 2$ *and the component type* $\mathbb{B}_0 = \langle \mathbb{V}, \mathbb{L}, \mathbb{P}, \mathbb{E}, \ell \rangle$*, where 1) control locations are partitioned into three sets:* $\mathbb{L} = \mathbb{L}^T \cup \mathbb{L}^N \cup \{start\}$*. Locations in* $\mathbb{L}^T$ *represent holding the token, while the ones in* $\mathbb{L}^T$ *are without the token; 2)* $\ell = start$*; 3) the set of ports is* $\mathbb{P} = \{send_t, receive_t \mid t \in T\} \cup \{init\_token, init\}$*; 4) every transition* $(q, send_t, q') \in \mathbb{E}$ *for token* $t \in T$ *satisfies* $q \in \mathbb{L}^T$ *and* $q' \in \mathbb{L}^N$*, and every transition* $(q, receive_t, q') \in \mathbb{E}$ *for token* $t \in T$ *satisfies* $q' \in \mathbb{L}^T$ *and* $q \in \mathbb{L}^N$*. Transition* $(start, init\_token, q') \in \mathbb{E}$*,* $q' \in \mathbb{L}^T$ *initializes the component with a token, and transition* $(start, init, q') \in \mathbb{E}$*,* $q' \in \mathbb{L}^N$ *initializes the component without a token.*

Then the parameterized BIP *model is* $\mathscr{M}_{\text{PBIP}} = \langle \langle \mathbb{B}_0 \rangle, \langle n_0 \rangle, n_0 \geq 2, \psi \rangle$*, where* $\psi = \psi_1 \vee \psi_2$ *and* $\psi_1 = \bigvee_{t \in T} (\exists x, y :: type_0 : 0 \leq x, y < n_0 \wedge (y = x+1 \bmod n_0).send_t(x) \wedge receive_t(y) \wedge \neg(init(x) \vee init(y) \vee init\_token(x) \vee init\_token(y)) \wedge \bigwedge_{t' \neq t} \neg(send_{t'}(x) \vee receive_{t'}(x) \vee send_{t'}(y) \vee receive_{t'}(y)) \wedge \forall z :: type_0 : 0 \leq z < n_0 \wedge z \neq x \wedge z \neq y.\neg(\bigwedge_{t'' \in T} send_{t''}(z) \vee receive_{t''}(z) \vee init(z) \vee init\_token(z)))$*, and* $\psi_2 = \exists x :: type_0.\forall y :: type_0 : 0 \leq x < n_0 \wedge 0 \leq y < n_0 \wedge y \neq x.init\_token(x) \wedge init(y) \wedge \neg init(x) \wedge \neg init\_token(y) \wedge \bigwedge_{t \in T} \neg(send_t(x) \vee receive_t(x) \vee send_t(y) \vee receive_t(y))$*.* ψ_1 *specifies the pairwise rendezvous between a component* x *and its neighbour* $x+1$ *for token passing, while* ψ_2 *distributes the token initially in the ring.*

6.3 Decidability results for parameterized BIP

In this section, we present a fragment of parameterized BIP models, called well-structured parameterized BIP, and prove that certain safety properties are decidable for this class of BIP. First of all, we review the theory of the well-structured transition system [1, 66].

6.3.1 Well-structured transition system

The theory of well-structured transition system is a powerful tool for the verification of infinite-state systems. In brief, well-structured transition systems are transition systems, whose sets of states are well-quasi ordered and whose transition relations exhibit the monotonicity property with respect to a well-quasi ordering. Well-known computation models that are well-structured include, e.g. communication finite state automaton, Petri nets.

A preorder on a set D, denoted by $\preceq$, is a reflexive and transitive binary relation on D. A set $U \subseteq D$ is said to be upward closed with respect to $\preceq$ if $d \in U$ and $d \preceq d'$ implies $d' \in U$. Given $d \in D$, we define by $\mathscr{U}(d) = \{d' \mid d \preceq d'\}$ the upward closure of d with respect to preorder $\preceq$. Given a set $B \subseteq D$, we define similarly $\mathscr{U}(B) = \bigcup_{b \in B} \mathscr{U}(b)$. The preorder $\preceq$ is said to be a well quasi-order if for all infinite sequences $d_0, d_1, d_2, \ldots$ in D, there are i, j, $i < j$, such that $d_i \preceq d_j$. Equivalently, if a preorder is a well quasi-order, then there is no infinite sequence of upward closed sets.

For an upward closed set U, we define a minor set of U to be the set $Min(U)$, such that $\mathscr{U}(Min(U)) = U$ and for all $c, c' \in Min(U)$, if $c \preceq c'$, then $c = c'$. Elements in $Min(U)$ are also called the generators of U. Assume that $\preceq$ is a well quasi-order, then the minor set of an upward closed set is finite. Otherwise, we would have an infinite set of incomparable elements, contradicting the assumption of a well quasi-order. Every upward closed set can be represented by its minor set.

Definition 6.3.1 (Monotonicity) *Given a labeled transition system $\mathscr{T} = \langle C, \Sigma, R, C_0 \rangle$ and a preorder $\preceq$ on the state space C, the transition relation R is monotonic with respect to $\preceq$, if for each $c_1, c_2, c_3 \in C$, $c_1 \preceq c_2$ and $\langle c_1, t, c_3 \rangle \in R$, there is $c_4 \in C$, such that $c_3 \preceq c_4$ and $\langle c_2, t', c_4 \rangle \in R$.*

Monotonicity means that greater states can always simulate smaller ones. Thus, any finite executions can be simulated from above, starting from a greater state.

Definition 6.3.2 (Well-structured transition system) *Given a labeled transition system $\mathscr{T} = \langle C, \Sigma, R, C_0 \rangle$ and a preorder $\preceq$ on the state space C, $\mathscr{T}$ is well-structured, if the following conditions hold:*

1. *$\preceq$ is a well quasi-order;*
2. *R is monotonic with respect to $\preceq$;*
3. *for each state $c \in C$, the minor set $Min(pre(\mathscr{U}(c)))$ is computable.*

The coverability problem for well-structured transition system is defined as follows: given a well-structured transition system with the preorder $\preceq$, and an upward closed set U of states, the coverability problem asks if some states in U are reachable from some initial states. It is known that this problem is decidable for well-structured transition system, and can be solved

by a general symbolic algorithm [1, 66, 131]. The algorithm performs a backwards reachability analysis from the set of bad states and checke if some initial states can be reached. Starting from an upward closed set U of bad states, the algorithm repeatedly applies the predecessor computation, and generates a sequence $U_0, U_1, U_2, \ldots$ of upwards closed sets, where $U_0 = U$, and $U_{i+1} = U_i \cup pre(U_i)$ for $i \geq 0$. Intuitively, each U_i represents the set of states from which U is reachable within i steps. The iteration terminates when we reach a point $i > 0$ such that $U_i = U_{i-1}$. In such a case, U_i consists of the set of states from which U is reachable. The termination is guaranteed when $\preceq$ is a well quasi-order.

6.3.2 Well-structured parameterized BIP

We consider the fragment of parameterized BIP models with clique topology and a single component type. Extensions to multi-typed models are straightforward.

We identify a fragment of the first order interaction logic, called upward closed FOIL. For this purpose, we define an preorder $\prec$ over the FOIL structures. Given two structures ξ and ξ', we denote by $\xi \prec \xi'$ if $\alpha_\xi(n_1) < \alpha_{\xi'}(n_1)$ and there is a monotonic injection $h : [0, \alpha_\xi(n_1)) \mapsto [0, \alpha_{\xi'}(n_1))$ such that for all $p \in \mathbb{P}_1$, if $\alpha_\xi(p)(i)$ evaluates to true, for some $i \in [0, \alpha_\xi(n_1))$, then $\alpha_{\xi'}(p)(j)$ evaluates to true, for some $j \in [0, \alpha_{\xi'}(n_1))$, $j = h(i)$.

Definition 6.3.3 (Upward closed FOIL) *A FOIL formula ψ is upward closed if for all structures ξ, ξ', such that $\xi \models_{FOIL} \psi$ and $\xi \prec \xi'$, then it holds $\xi' \models_{FOIL} \psi$.*

In terms of BIP interactions, an upward closed FOIL formula ψ has the following property. Given two structures ξ and ξ' of ψ, suppose the two induced interactions are $\gamma_\xi = \{(p, i) \mid p \in \mathbb{P}_1,\ i \in [0, \alpha_\xi(n_1)),\ \alpha_\xi(p)(i) = true\}$ and $\gamma_{\xi'} = \{(p, j) \mid p \in \mathbb{P}_1,\ j \in [0, \alpha_{\xi'}(n_1)),\ \alpha_{\xi'}(p)(j) = true\}$, respectively. If $\xi \prec \xi'$, then there is a monotonic injection $h : [0, \alpha_\xi(n_1)) \mapsto [0, \alpha_{\xi'}(n_1))$, such that for each $(p, i) \in \gamma_\xi$, there is $j = h(i)$ and $(p, j) \in \gamma_{\xi'}$. Intuitively, if $\xi \prec \xi'$, then the interaction γ_ξ is a subset of the interaction $\gamma_{\xi'}$ under an injective mapping on the port indices. Upward closedness means that adding more ports still preserve the validity of the interaction.

Example upward closed FOIL formulae includes the ones with only existential quantifiers. FOIL formulae with positive predicates in the scope of universal quantifiers are not upward closed in general.

Example 6.3.4 *Consider the FOIL formula $\psi = \exists i :: type_1.\ send(i)$ in Example 6.1.5. FOIL formula ψ is in the upward closed fragment, since for instance, given a structure ξ, where $\alpha_\xi(n_1) = 2$, and $\alpha_\xi(send)(0) = true$, $\alpha_\xi(send)(1) = false$, then for any structure ξ', if $\xi \prec \xi'$, we have $\alpha_{\xi'}(send)(0) = true$. It holds that $\xi' \models \psi$.*

Example 6.3.5 *Consider the FOIL formula $\psi = \psi_{go} \vee \psi_{follow} \vee \psi_{exit}$ in Example 6.1.7. It is not in the upward closed fragment, because of $\psi_{exit} = \forall i :: type_0.exit(i)$. One can check that, for*

instance, given a structure ξ, where $\alpha_\xi(n_0) = 2$, and $\alpha_\xi(exit)(0) = true$, $\alpha_\xi(exit)(1) = true$, there is another structure ξ', where $\alpha_{\xi'}(n_0) = 3$, and $\alpha_{\xi'}(exit)(0) = true$, $\alpha_{\xi'}(exit)(1) = true$, $\alpha_{\xi'}(exit)(2) = false$, such that $\xi \prec \xi'$, but it does not hold that $\xi' \models \psi_{exit}$.

Given a parameterized BIP model $\mathscr{M}_{\text{PBIP}} = \langle \mathbb{B}, \bar{n}, \psi, \epsilon \rangle$, we say that $\mathscr{M}_{\text{PBIP}}$ is well-structured if its LTS $\mathscr{T}_{PBIP} = \langle C_{PBIP}, \Sigma_{PBIP}, R_{PBIP}, C_{0_{PBIP}} \rangle$ is well-structured, i.e. the union of LTSs of all instance models is well-structured.

The preorder $\preceq_{PBIP}$ on the state space C_{PBIP} is defined as follows. Given two states $c = q_0 \dots q_{m-1} \in C_{PBIP}$ and $c' = q'_0 \dots q'_{n-1} \in C_{PBIP}$, we denote by $c \preceq_{PBIP} c'$ if there is a monotonic injection $h : [0, m) \mapsto [0, n)$, such that $q_i = q'_{h(i)}$ for each $i \in [0, m)$, $m \leq n$. It has been shown in [5] that this preorder is a well quasi-order. We remark that this preorder $\preceq_{PBIP}$ is different from the order $\prec$ over FOIL structures.

Proposition 6.3.6 *Given a parameterized* BIP *model $\mathscr{M}_{\text{PBIP}} = \langle \mathbb{B}, \bar{n}, \epsilon, \psi \rangle$, suppose its LTS $\mathscr{T}_{PBIP} = \langle C_{PBIP}, \Sigma_{PBIP}, R_{PBIP}, C_{0_{PBIP}} \rangle$. If ψ is in upward closed FOIL, then R_{PBIP} is monotonic with respect to the preorder $\preceq_{PBIP}$.*

Proof 6.3.7 *We have to prove that $\forall c_1, c'_1, c_2$, if $c_1 \xrightarrow{\gamma} c'_1$ and $c_1 \preceq_{PBIP} c_2$, then $\exists c'_2$, $c_2 \xrightarrow{\gamma'} c'_2$ and $c'_1 \preceq_{PBIP} c'_2$. If the interaction γ in state c_1 is induced by the FOIL structure ξ_1, then due to the fact that ψ is upward closed we know that in state c_2, there is another structure ξ_2, $\xi_1 \prec \xi_2$, which defines an interaction γ'. Then it is sufficient to prove that γ' is enabled in c_2 and labels the transition $c_2 \xrightarrow{\gamma'} c'_2$.*

Assume $c_1 = q_0 \dots q_{m_1-1}$ and $c_2 = q'_0 \dots q'_{m_2-1}$, since $c_1 \preceq_{PBIP} c_2$, then there is a monotonic injection $h : [0, m_1) \mapsto [0, m_2)$, for all $i \in [0, m_1)$, $q_i = q'_{h(i)}$. Using the injection h, we can derive a new structure ξ_2 from ξ_1, where $\alpha_{\xi_2}(n) = m_2$, and for each $p \in \mathbb{P}_1$, $\alpha_{\xi_2}(p)(j) = true$, for some $j \in [0, m_2)$, if $\alpha_{\xi_2}(p)(i) = true$, for some $i \in [0, m_1)$ and $j = h(i)$. Thus, $\xi_1 \prec \xi_2$. Since ψ is upward closed, we have $\xi_2 \models \psi$. Then for each port $(p, i) \in \gamma$, we have $(p, j) \in \gamma'$, where $j = h(i)$.

Since (p, i) is enabled on q_i, and $q_i = q'_{h(i)}$, then (p, j) is also enabled on $q'_{h(i)}$. Thus, γ' is enabled on state c_2, which completes the whole proof.

The following theorem states that if we restrict FOIL formulae to the upward closed fragment, the parameterized BIP model is well-structured.

Theorem 6.3.8 *Given a parameterized* BIP *model $\mathscr{M}_{\text{PBIP}} = \langle \mathbb{B}, \bar{n}, \epsilon, \psi \rangle$, if the FOIL formula ψ is upward closed, then $\mathscr{M}_{\text{PBIP}}$ is well-structured with respect to the preorder $\preceq_{PBIP}$.*

Proof 6.3.9 *Assume the LTS of $\mathscr{M}_{\text{PBIP}}$ is $\langle C_{PBIP}, \Sigma_{PBIP}, R_{PBIP}, C_{0_{PBIP}} \rangle$, according to Proposition 6.3.6, the transition relation R_{PBIP} is monotonic with respect to $\preceq_{PBIP}$. Moreover, since the*

preorder $\preceq_{PBIP}$ is a well quasi-order, it remains to prove that for each state $c \in C_{PBIP}$, the minor set $Min(pre(\mathscr{U}(c)))$ is computable. In fact, $Min(pre(\mathscr{U}(c)))$ equals to $Min(pre(c))$, if ψ is upward closed. In other words, in order to compute the predecessors of an upward closed set, we only need to compute the predecessors of the generators.

It suffices to prove that $\forall c' \in \mathscr{U}(c)$ and $\forall c_1 \in C_{PBIP}, c_1 \xrightarrow{\gamma'} c'$, then $\exists c_2 \in C_{PBIP}, c_2 \xrightarrow{\gamma} c$ and $c_1 \in \mathscr{U}(c_2)$.

Suppose $c = q_0^c \dots q_{m-1}^c$, $c_1 = q_0^{c_1} \dots q_{m-1}^{c_1}$, and $c' = q_0^{c'} \dots q_{m'-1}^{c'}$, $c_2 = q_0^{c_2} \dots q_{m'-1}^{c_2}$. Since $c' \in \mathscr{U}(c)$, i.e. $c \preceq_{PBIP} c'$, then there is a monotonic injection $h : [0,m) \rightarrowtail [0,m')$, $m \leq m'$, $q_i^c = q_j^{c'}$, for each $i \in [0,m)$, $j \in [0,m')$ and $j = h(i)$. Using h, we construct c_1 from c_2 as follows: $q_i^{c_1} = q_j^{c_2}$, for each $i \in [0,m)$, $j \in [0,m')$ and $j = h(i)$. By construction, we have $c_2 \in \mathscr{U}(c_1)$.

Assume the FOIL structure for interaction γ' is $\xi_{\gamma'}$, we can construct another structure ξ_γ from $\xi_{\gamma'}$ using the above h, such that $\alpha_\xi(p)(i) = \alpha_{\xi'}(p)(j)$, for each $i \in [0,m)$, $j \in [0,m')$ and $j = h(i)$. Since ψ is upward closed, $\xi_\gamma \models \psi$, otherwise, it violates the assumption $\xi_{\gamma'} \models \psi$.

From the construction of c_1 and ξ_γ, it's straightforward to see that the interaction γ is also enabled on c_1 and labels the transition $c_1 \xrightarrow{\gamma} c$. This completes the proof.

Example 6.3.10 *In this example, we show that the disjunctive guarded protocol in Example 6.1.9 is well-structured.*

First of all, we define the preorder on the state space. Given two states $c = (q_1, q_2, ..., q_m)$, $c' = (q'_1, q'_2, ..., q'_n)$, and $m \leq n$, we define $c \preceq c'$ if and only if $\exists h : [1,m] \rightarrowtail [1,n]$, such that for each $i \in [1,m]$, $q_i = q'_{h(i)}$. This preorder is useful for characterizing certain safety properties, e.g. mutual exclusion, as upward closed sets. Since if a state violates the mutual exclusion property, then any larger state would also violate the mutual exclusion property. Then we show that the disjunctive guard protocol exhibits a monotonic transition with respect to this preorder.

Suppose a guarded transition $(q, \exists j :: type_0.j \neq i.\phi(j), p, q')$ is enabled in state $c = (q_1, q_2, ..., q_m)$ for process i, i.e. $q_i = q$ and $\exists j :: type_0.j \neq i$, such that $q_j \models \phi(j)$, then this guarded transition is also enabled in any state $c' = (q'_1, q'_2, ..., q'_n)$, $c \preceq c'$. This is because that if $c \preceq c'$, there is $j' \in [1,n]$, such that $j' = h(j)$ for $j \in [1,m]$ and $q'_{j'} = q_j$, thus, $q'_{j'} \models \phi(j')$. Suppose furthur the successor state of c is c_t, it would not be hard to see that there is a successor state c'_t of c', such that $c_t \preceq c'_t$.

Similarly, we can prove that conjunctive guarded protocol is not well-structured with respect to the above preorer. This is because the transition relation defined by a conjunctive guard does not guarantee the monotonicity property, i.e. adding more states may turn the conjunctive guard to be unsatisfiable.

We consider the following verification problem for parameterized BIP.

Problem 6.3.11 *Given a parameterized* BIP *model $\mathscr{M}_{\text{PBIP}} = \langle \mathbb{B}, \bar{n}, \psi, \epsilon \rangle$ and an ICTL* state*

formula θ with only temporal operator **G** *and universal path quantifier* **A** *and existential quantifier* $\exists$*, and without free variables, this verification problem asks whether it holds that every instance model $\mathscr{M}_{\text{BIP}}$ satisfies θ.*

Corollary 6.3.12 *The above verification problem for a parameterized* BIP *model $\mathscr{M}_{\text{PBIP}} = \langle \mathbb{B}, \tilde{n}, \epsilon, \psi \rangle$ is decidable, if ψ is upward closed.*

Proof 6.3.13 *The set of states satisfying ICTL* formulae with only temporal operator* **G** *and universal path quantifier* **A** *and existential quantifier* $\exists$ *and without free variables are upward closed sets. Then the decidability proof follows directly from Theorem 6.3.8 and the result that coverability problem is decidable for well-structured transition system [1, 131].*

6.4 A framework of automated parameterized verification in BIP

In this section, we present a general framework for automated parameterized verification in BIP. It is shown in Figure 6.4. It takes as input a parameterized system design specified in parameterized BIP, and then identifies the architecture model of the given system. According to the identification, a suitable parameterized verification technique is chosen. The development of parameterized verification technique is orthogonal to the architecture identification. In the rest of this dissertation, we focuse on the latter task.

Architecture identification plays an important step in our verification framework. Parameterized BIP can capture various specific architectures: token rings, broadcast in cliques, rendezvous in stars, etc. In the non-parameterized case, knowing the architecture is not crucial, as there are model checking algorithms that apply in general to arbitrary transition systems. However, the architecture dramatically affects both the decidability and the techniques of *parameterized* model checking. It is crucial to understand the architecture model in parameterized case in order to achieve automation.

6.5 Identifying the architecture of a parameterized BIP model

In this section, we present how to identify system architectures automatically, and show the applications to parameterized verification. For the sake of exposition, we assume that the parameterized BIP models have only one component type. Our identification framework extends easily to the general case.

Given an architecture $\mathscr{A}$, e.g. the token ring architecture, an expert in parameterized model checking creates formula templates in FOIL (*FOIL-templates*) and in temporal logic (*TL-templates*). *FOIL-templates* describe the system topology and communication mechanism for architecture $\mathscr{A}$. *TL-templates* describe the behavior of the component type required by architecture $\mathscr{A}$, e.g. in a token ring, a component which does not have the token cannot send.

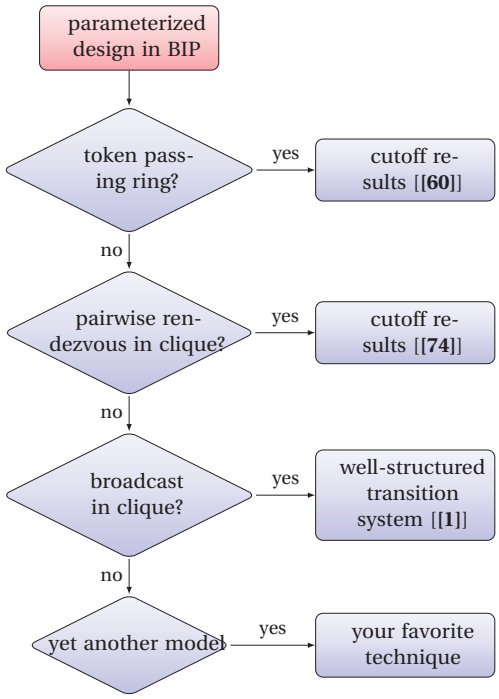

Figure 6.4 – Framework of automated parameterized verification in BIP

These templates are designed once for all parameterized BIP models compliant with $\mathscr{A}$. In the sequel, *TL-templates* are only used for token rings, thus we omit them from the discussion of other architectures.

Given a parameterized BIP model $\langle\langle\mathbb{B}\rangle,\langle n\rangle,\psi,\epsilon\rangle$ — not necessarily compliant with the architecture $\mathscr{A}$ — the templates for the architecture $\mathscr{A}$ are instantiated to first-order formulae $\varphi_1^{FOIL},\ldots,\varphi_m^{FOIL}$, and temporal logic formulae $\varphi_1^{TL},\ldots,\varphi_\ell^{TL}$. The first-order logic formulae restrict the set of interactions expressed by the FOIL formula ψ. The temporal logic formulae restrict the behavior of the component type $\mathbb{B}$. The *identification criterion* is as follows: if $\varphi_1^{FOIL}\wedge\cdots\wedge\varphi_m^{FOIL}$ valid [1] and $\mathbb{B}\models_{\text{TL}}\varphi_1^{TL}\wedge\cdots\wedge\varphi_\ell^{TL}$ holds, then the parameterized model $\langle\langle\mathbb{B}\rangle,\langle n\rangle,\psi,\epsilon\rangle$ is compliant with the architecture $\mathscr{A}$. In practice, we use an SMT solver to check validity of the FOIL formulae and a model checker to check that the component type $\mathbb{B}$ satisfies the temporal formulae.

In the rest of this section we construct FOIL-templates and TL-templates for well-known architectures: cliques of processes communicating via broadcast, cliques of processes communicating via rendezvous, token rings, processes organized in a star and communicating via rendezvous. We show that the provided templates identify the architectures in a sound way.

1. A FOIL formula without free variables is valid if it is satisfied by all FOIL structures ξ.

6.5.1 The common templates for BIP semantics

As we discussed in Section 6.1.2, not every FOIL structure induces a BIP interaction. We show that one can write an FOIL-template that restricts FOIL structures to be BIP interactions. The following template $\eta^{FOIL}_{interaction}(\mathbb{P}_1)$ expresses that there is no interaction with more than one active port belonging to the same component: $\forall j :: type_1 . \bigwedge_{p,q \in \mathbb{P}_1,\ q \neq p} \neg p(j) \vee \neg q(j)$

As expected, the template $\eta^{FOIL}_{interaction}(\mathbb{P}_1)$ restricts FOIL structures to BIP interactions:

Proposition 6.5.1 *Let $\mathbb{P}_1$ be a set of ports, and η be the instantiation of $\eta^{FOIL}_{interaction}$ with $\mathbb{P}_1$. A FOIL structure ξ satisfies η if and only if ξ induces an interaction.*

In the following, we often need to express that a component has at least one active port. template $active(j) \equiv \bigvee_{p \in \mathbb{P}_1} p(j)$. We omit the parameterization of $active(j)$ by $\mathbb{P}_1$ for to simplify notation.

6.5.2 Pairwise rendezvous in a clique

In BIP, two components communicate with pairwise rendezvous, if each of them has an active port — forming an interaction — and the other components do not have active ports. In this case, both components make their transitions simultaneously, and the other components stutter on their states. Pairwise rendezvous has been widely used as a basic primitive in the parameterized model checking literature, e.g. in [74, 10].

FOIL-templates. We construct a template using two formulae $\eta^{FOIL}_{\leq 2}(\mathbb{P}_1)$ and $\eta^{FOIL}_{\geq 2}(\mathbb{P}_1)$:

- The formula $\eta^{FOIL}_{\leq 2}(\mathbb{P}_1)$ expresses that every interaction has at most two ports: $\forall i, j, \ell :: type_1 .\ active(i) \wedge active(j) \wedge active(\ell) \rightarrow i = j \vee j = \ell \vee i = \ell$.
- The formula $\eta^{FOIL}_{\geq 2}(\mathbb{P}_1)$ expresses that every interaction has at least two ports: $\exists i, j :: type_1 : i \neq j .\ active(i) \wedge active(j)$.

We show that the combination of $\eta^{FOIL}_{interaction}$, $\eta^{FOIL}_{\geq 2}$, and $\eta^{FOIL}_{\leq 2}$ defines pairwise rendezvous communication in cliques of all sizes:

Theorem 6.5.2 *Given a one-type parameterized* BIP *model $\langle\langle \mathbb{B} \rangle, \langle n \rangle, \psi, \epsilon\rangle$, if $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2})$ is valid, then for every instance $\mathbb{B}^{N,\Gamma}$, the following holds:*

1. *every interaction is of size 2, that is, $|\gamma| = 2$ for $\gamma \in \Gamma^{\tilde{N}}$, and*
2. *for every pair of indices i and j such that $0 \leq i, j < N$ and $i \neq j$ and every pair of ports $p, q \in \mathbb{P}_1$, there is a FOIL structure ξ such that $\xi \models_{\text{FOIL}} \psi \wedge p(i) \wedge q(j)$.*

Proof 6.5.3 *Fix an instance $\mathbb{B}^{N,\Gamma}$ of $\langle\langle \mathbb{B} \rangle, \langle n \rangle, \psi, \epsilon\rangle$.*

To show Point 1, fix an interaction γ of $\mathbb{B}^{N,\Gamma}$. By Definition 6.1.4, there is a FOIL structure ξ such that $\xi \models_{\text{FOIL}} \psi$ and $\gamma = \gamma_\xi$. As ξ induces an interaction, by Proposition 6.5.1, we immediately have that γ_ξ satisfies an instantiation of $\eta^{FOIL}_{interaction}$. Hence, since $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2})$ is valid we conclude that ξ also satisfies $\eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2}$. This immediately gives us the required equality $|\gamma_\xi| = 2$.

To show Point 2, fix a pair of indices i and j such that $0 \leq i, j < N$ and $i \neq j$ and a pair of ports $p, q \in \mathbb{P}_1$. The set $\gamma = \{(p,i),(q,j)\}$ is an interaction. Obviously, one can construct a FOIL structure ξ that induces γ. Since $i \neq j$ and $|\gamma_\xi| = 2$, it holds that $\xi \models_{\text{FOIL}} \eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2}$. Thus, since $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2})$ is valid, it follows that $\xi \models_{\text{FOIL}} \psi$. From this and that ξ induces the interaction γ, we conclude that $\xi \models_{\text{FOIL}} \psi \wedge p(i) \wedge q(j)$.

In Theorem 6.5.2, the right-hand side of the equivalence does not restrict pairs of ports that are included into interactions, e.g., it does not require the ports to be the same. Thus, if the formula ψ is more restrictive than the right-hand side of the equivalence, validity will not hold. Obviously, one can further restrict the equivalence to reflect additional constraints on the allowed pairs of ports.

Applications. Theorem 6.5.2 gives us a criterion for identifying parameterized BIP models, where all processes may interact with each other using rendezvous communication. To verify such parameterized BIP models, we can immediately invoke the seminal result by German & Sistla [74, Sec. 4]. Their result applies to specifications written in indexed linear temporal logic without the operator **X**.

More formally, we say that an *ICTL** path formula $\chi(i)$ is a *LTL*$\backslash X$ formula, if χ has only one index variable i and χ does not contain quantifiers $\exists$, $\forall$, A, E, nor temporal operator **X**. Given a parameterized BIP model $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$ and a *LTL*$\backslash X$ formula χ, one can check in polynomial time, whether every instance $\mathbb{B}^{N,\Gamma}$ satisfies the formula $E\,\exists i :: type_1 :\ true.\ \chi(i)$.

6.5.3 Broadcast in a clique

In BIP, components communicate via broadcast, if there is a "trigger" component whose sending port is active, and the other components either have their receiving port active, or have no active ports. In this section, we denote the sending port with *send* and the receiving port with *receive*. Our results can be easily extended to treat multiple sending and receiving ports. In a broadcast step, all the components with the active ports make their transitions simultaneously. Broadcasts were extensively studied in the parameterized model checking literature [64, 131].

One way to enforce all the processes to receive a broadcast, if they are ready to do so, is to use priorities in BIP: an interaction has priority over any of its subsets. In BIP without priorities — considered in this paper — one can express broadcast by imposing the following restriction

on the structure of the component type $\mathbb{B}$: *every location has a transition labeled with the port receive*. This restriction enforces all interactions to involve all the components, though some of the components may not change their location by firing a self-loop transition. This requirement can be statically checked on the transition system of $\mathbb{B}$, and if the component type does not fulfill the requirement, it is easy to modify the component type's transition system by adding required self-loops.

FOIL-templates. First, we define the formula $\eta^{FOIL}_{bcast}(\mathbb{P}_1)$, which guarantees that every interaction includes one sending port by one component and the receiving ports of the other components:

$$\exists i :: type_1.\ send(i) \wedge \forall j :: type_1 : j \neq i.\ receive(j)$$

We show that the combination of $\eta^{FOIL}_{interaction}$ and η^{FOIL}_{bcast} defines broadcast in cliques of all sizes:

Theorem 6.5.4 *Given a one-type parameterized* BIP *model* $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$, *if* $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{bcast} \wedge \eta^{FOIL}_{interaction})$ *is valid, then for every instance* $\mathbb{B}^{N,\Gamma}$, *the following holds:*

1. *every interaction is of size N consisting of one send port and receive ports.*
2. *for every index c, such that* $0 \leq c < N$, *there is a FOIL structure* ξ *satisfying the following:* $\xi \models_{\text{FOIL}} \psi \wedge send(c) \wedge \forall j :: type_1 : j \neq c.\ receive(j)$.

Proof 6.5.5 *The proof follows the same principle as the proof of Theorem 6.5.2.*

Applications. Theorem 6.5.4 gives a criterion for identifying parameterized BIP models in which all components may send and receive broadcast. Its implications are two-fold. First, it is well-known that parameterized model checking of safety properties is decidable [1] (cf. the discussion in [64]), and there are tools for well-structured transition systems applicable to model checking of parameterized BIP. Second, parameterized model checking of liveness properties is undecidable [64]. From the user's perspective, this indicates the need to construct abstractions, or to use semi-decision procedures.

Identifying sending and receiving ports. Now we illustrate how to automatically detect the sending and receiving ports in a parameterized BIP model. We say that a port $p \in \mathbb{P}_1$ in the component type may be a sending port, if in every interaction exactly one component uses this port. Similarly, we say that a port $q \in \mathbb{P}_1$ in the component type may be a receiving port, if in every interaction all but one component use this port. Intuitively, we have to enumerate all port types and check, whether they are acting as sending ports or receiving ports. Formally, to find, if p is a potential sending port and q is a potential receiving port, we check, whether the

following is valid:

$$\psi \wedge \eta^{FOIL}_{interaction} \wedge \exists i :: type_1 . p(i) \vee q(i) \rightarrow \Big(\exists i :: type_1 .\ p(i) \wedge \forall j :: type_1 : j \neq i .\ q(j)\Big)$$

6.5.4 Token rings

Token ring is a classical architecture: (i) all processes are arranged in a ring, and (ii) one component owns the token and can pass it to its neighbors. It is easy to express token-passing with rendezvous, so we re-use the formulae from Section 6.5.2. We assume that there is a pair of ports: the port *send* giving away the token and the port *receive* accepting the token. We do not allow the token to change its type, as the parameterized model checking problem in this case is undecidable [136, 60]. Nevertheless, it is easy to extend our results to multiple token types. Here the token is passed in one direction, i.e. every component can only receive the token from one neighbor and send it to the other neighbor.

TL-templates. Following the standard assumption [60], we require that every process sends and receives the token infinitely often. We encode this requirement as a local constraint in a form of an LTL formula that is checked against the component type (not a BIP instance):

$$\mathbf{G}\Big(receive \rightarrow \mathbf{X}\ (\neg receive\ \mathbf{U}\ send)\Big) \wedge \mathbf{G}\Big(send \rightarrow \mathbf{X}\big(\neg send\ \mathbf{U}\ receive\big)\Big)$$

The left conjunct forces a component to eventually send the token, if the component has received the token. The right conjunct does not allow a component to send the token twice without receiving the token before the second send.

FOIL-templates. We extend the pairwise rendezvous templates with an additional formula $\eta^{FOIL}_{bcast}(\mathbb{P}_1)$ that restricts the interactions to be performed only among the neighbors in one direction:

$$\exists i, j :: type_1 .\ (j = (i+1)\ mod\ n_1).\ active(i) \wedge active(j) \wedge send(i) \wedge receive(j)$$

Note that the modulo notation "$j = (i+1)\ mod\ n_1$" can be seen as syntactic sugar, as it expands into $(i = n_1 - 1 \rightarrow j = 0) \wedge (i < n_1 - 1 \rightarrow j = i + 1)$.

Theorem 6.5.6 *Given a one-type parameterized* BIP *model* $\langle\langle\mathbb{B}\rangle, \langle n\rangle, \psi, \epsilon\rangle$, *if* $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2} \wedge \eta^{FOIL}_{uniring})$ *is valid, then every instance* $\mathbb{B}^{N,\Gamma}$ *satisfies:*

1. *every interaction* $\gamma \in \Gamma^{\bar{N}}$ *is of the form* $\{send(c), receive(d)\}$ *for some indices* c *and* d *such that* $0 \leq c, d < N$ *and* $d = (c+1)\ mod\ N$, *and*

2. *for every index c such that $0 \leq c < N$ and the index $d = (c+1) \bmod N$, there is a FOIL structure ξ such that $\xi \models_{\text{FOIL}} \psi \wedge send(c) \wedge receive(d)$.*

Proof 6.5.7 *The proof follows the same principle as the proof of Theorem 6.5.2.*

Distributing the token. The token ring architecture assumes that initially only one component has the token. Emerson & Namjoshi [60] assumed that the token was distributed using a "daemon", but this primitive is obviously outside of the token ring architecture. Our framework encompasses token distribution. To this end, we restrict the transition system of the component as follows:

– We assume that the location set $\mathbb{L}_1$ of the component type $\mathbb{B}_1$ is partitioned into two sets: $\mathbb{L}_1^{tok}$ is the set of locations possessing the token, and $\mathbb{L}_1^{ntok}$ is the set of locations without the token. The initial location does not possess the token: $\ell^0 \in \mathbb{L}_1^{ntok}$.
– We assume that there are two auxiliary ports called *master* and *slave* that are only used in a transition from the initial location ℓ^0. There are only two transitions involving ℓ^0: the transition from ℓ^0 to a location in $\mathbb{L}_1^{tok}$ that broadcasts via the port *master*, and the transition from ℓ^0 to a location in $\mathbb{L}_1^{ntok}$ that receives the broadcast via the port *slave*. The broadcast interaction can be checked with the constraints similar to those in Section 6.5.3.

Applications. Theorem 6.5.6 gives us a criterion of identifying parameterized BIP models that express a unidirectional token ring. This criterion has a great impact: one can apply non-parameterized BIP tools to verify parameterized BIP designs expressing token rings. As Emerson & Namjoshi showed in their celebrated paper [60], to verify parameterized token rings, it is sufficient to run model checking on rings of small size. The bound on the ring size — called a *cut-off* — depends on the specification and typically requires two or three components.

6.5.5 Pairwise rendezvous in a star

In a star architecture, one component acts as the center, and the other components communicate only with the center. The components communicate via rendezvous considered in Section 6.5.2. This architecture is used in client-server applications. Parameterized model checking for the star architecture was investigated by German & Sistla [74]. We assume that a parameterized BIP model contains two component types: $\mathbb{B}_1$ with only one instance, and $\mathbb{B}_2$ that may have many instances.

FOIL-templates. The essential requirements of rendezvous communication are defined in Section 6.5.2. We add the following restriction that the center is involved in every interac-

Benchmark	Architecture model	Outcome	Time (sec.)	Memory (MB)
Milner's scheduler	uni-directional token ring	**positive**	0.068	≤ 10
Milner's scheduler	broadcast in clique	negative	0.016	≤ 10
Semaphore	pairwise rendezvous in star	**positive**	0.096	≤ 10
Semaphore	pairwise rendezvous in clique	negative	0.084	≤ 10
Barrier	broadcast in clique	**positive**	0.028	≤ 10
Barrier	pairwise rendezvous in star	negative	0.008	≤ 10

Table 6.1 – Experimental results of identifying architecture models.

tion η^{FOIL}_{center}:

$$\exists i :: type_1.\ active_1(i)$$

By restricting ϵ to have only one instance of type $\mathbb{B}_1$, we arrive at the following simple theorem, which to a large extent is a consequence of Theorem 6.5.2:

Theorem 6.5.8 *Given a two-component parameterized* BIP *model* $\langle\langle \mathbb{B}_1, \mathbb{B}_2\rangle, \langle n_1, n_2\rangle, \psi, \epsilon\rangle$*, if* $(\psi \wedge \eta^{FOIL}_{interaction}) \leftrightarrow (\eta^{FOIL}_{interaction} \wedge \eta^{FOIL}_{\geq 2} \wedge \eta^{FOIL}_{\leq 2} \wedge \eta^{FOIL}_{center})$ *and* $\epsilon \leftrightarrow (n_1 = 1)$ *are both valid, then every instance* $\langle \mathbb{B}_1, \mathbb{B}_2\rangle^{\langle N1, N2\rangle, \Gamma}$ *admits only the rendezvous interactions with the center, i.e. the only component of type* $\mathbb{B}_1$.

Applications. Theorem 6.5.8 gives us a criterion for identifying parameterized BIP models, where the user processes communicate with the coordinator via rendezvous. To verify such parameterized BIP models, we can immediately invoke several results by German & Sistla [74, Sec. 3]. First, one can analyze such parameterized BIP models for deadlocks, which is of extreme importance to the practical applications of BIP. Second, the results [74] reduce parameterized model checking to reachability in Petri nets, which allows one to use the existing tools for Petri nets.

6.6 Prototype implementation and experiments

We have implemented a prototype of the framework introduced in Section 6.5. This prototype uses the templates for pairwise rendezvous and broadcast in cliques, tokens rings, and rendezvous in stars. The implementation uses nuXmv [33] for model-checking and Z3 [51] for SMT-solving. To deal with quantifiers, we run a customized solver with tactic 'qe' (i.e. quantifier elimination). The implementation and benchmarks are available at http://risd.epfl.ch/parambip.

Table 6.1 summarizes our experiments with three benchmarks. The column "Outcome" indicates, whether the benchmark was recognized to have the given architecture (positive), or not (negative). The experiments were performed on a 64-bit Linux machine with 2.8GHz ×

4 CPU and 7.8GiB memory. We conducted the experiments with two kinds of templates: the original architecture of the benchmark, and an architecture different from the original one. In all cases, the architectures were identified as expected. Our preliminary experiment results demonstrate both the correctness and the efficiency of our technique. In the future, we will implement a full-featured tool and perform thorough experimental evaluations.

6.7 Related work

In the research line of parameterized verification, one of the widely used techniques is based on the framework of well-structured transition system [1, 66, 131]. A well-structured transition system naturally generalises several infinite-state models such as Petri nets. In [1, 66], the authors show that for certain safety properties, such as coverability, are decidable on this class of systems. They also present a practical backward reachability analyses algorithm, and the termination is guaranteed by the fact that such systems are monotonic with respect to a well-quasi ordering. Given a parameterized system, we look at its transition system, which defines its operational semantics. If the transition system is well-structured, then certain safety properties are decidable and the algorithmic verification can be achieved via a backward reachability analysis from the error states. However, in most cases well-structureness is rarely satisfied. A solution to this problem is the monotonic abstraction [5, 7, 6]. In this abstraction technique, parameterized systems containing global conditions within guards are abstracted into a well-structured one in order to apply algorithmic verification. Later, in [4], the authors extend monotonic abstraction to CEGAR style reasoning.

Regular model checking [30, 3] is another widely used technique being developed for algorithmic verification of several classes of infinite-state systems whose configurations can be modeled as words over a finite alphabet. Examples include parameterized systems consisting of an arbitrary number of homogeneous finite-state processes connected in a linear or ring-formed topology, and systems that operate on queues, stacks, integers, and other linear data structures. The main idea is to use regular languages as the representation of sets of configurations, and finite-state transducers to describe transition relations. In general, the verification problems considered are all undecidable, so the work has consisted in developing semi-algorithms, and decidability results for restricted cases.

Besides the backward analysis of well-structured systems, the first notable forward algorithm to solve the coverability problem was proposed in [103] for Petri net. In [56], the authors attempt to generalise the forward algorithm for broadcast protocols, a class of well-structured systems that are made up of an unbounded number of finite state processes communicating via rendezvous and broadcast. They present a forward reachability analysis algorithm for such systems based on the construction of a covering graph. However, in [64], the authors show that the algorithm in [56] may not terminate for broadcasting protocols and the termination is retained by applying the backward reachability analysis based on well-structured transition system. A forward reachability analysis technique, called Expand, Enlarge and Check (EEC), is

proposed in [73, 72], It is a general algorithmic schema that allows to define forward analysis techniques to solvle the coverability problem of well-structured transition system.

In [20], the authors propose to model parameterized systems using a single WS1S transition system, where WS1S refers to the weak monadic second order theory of one successor. In a WS1S transition system, variables are set (second order) variables and transitions can be expressed as WS1S formulae. The idea is that set variables encode the set of processes that reside in certain control locations. They also present techniques to abstract a WS1S transition system into a finite state system, which can be automatically verified.

In [12], the authors present a compositional verification technique for parameterized component-based timed systems. Their technique relies on a cutoff result to reduce the parameterized verification to the verification of finite state systems. The cutoff result is obtained by restricting the formulae used to describe the parameterized systems to a certain fragment, which has a small model theorem [100].

In line of modeling system architectures, the authors proposed Dynamic BIP to model fixed size, but dynamic architectures, where interactions of components may evolve during the execution [31]. In a recent work [114], configuration logic is proposed as a formal specification of architecture families. Our first order interaction logic differs from the configuration logic in that a formula in interaction logic describes a certain architecutre, while in configuration logic, formulae describe a set of architectures.

7 Conclusions and perspectives

In this chapter, we first conclude the dissertation by describing the main objectives of this work and the goals we have achieved. Then we also give some directions for the future work.

7.1 Summary of the dissertation

While algorithmic verification has made impressive advances recently thanks to the novel symbolic model checking techniques, such as lazy abstraction [90, 88], interpolation [119], IC3/PDR for hardware [32, 55] and for software [35, 37, 25], concurrent systems that consist of either bounded or unbounded number of components still pose a formidable challenge of efficient verification.

The effectiveness of model checking in the presence of bounded concurrency is severely limited by the state explosion caused by interleavings of interactions, which are not handled by the above mentioned symbolic model checking techniques. Consequently, the first insight of this disseration is that combining techniques that can reduce the redundant interaction interleavings, such as partial order reduction [139, 124, 78] would be feasible way to improve the scalability of the symbolic model checking techniques.

We have presented an efficient safety property verification technique for infinite-state BIP models with a fixed number of components in this dissertation. Our technique is based on the idea of combining abstraction techniques with partial order reductions. Particularly, our technique applies sophisticated counterexample guided abstraction refinement techniques to reason about the sequential computations in the atomic components, and also incorporates the persistent set based partial order reduction technique to deal with concurrent interactions between the components. We have implemented the proposed technique and the experimental evaluations justify our arguments about the competitiveness and efficiency of the proposed technique. Moreover, we have also presented two advanced reductions for BIP. The first one reduces the redundant interleavings by exploring independent interactions simultaneously as many as possible, and the second one exploits the system symmetries to improve the

persistent set reduction for the class of models that exhibit such symmetries.

Another source of state explosion is due to the unboundedness of the number of participating components in the system. Verification of the system with an unbounded number of components is also known as parameterized verification, in which the task is to prove the correctness for all instrances of the system. Being undecidable in its general form [11], there are therefore roughly two approaches to circumvent this problem: one is to identify decidable fragments and devise verification techniques for them, such as the cutoff techniques [60, 42] that decomposes the parameterized verification problem into several finite-state verification problems, and another one gives rise to incomplete methods, that apply abstractions or approximations to achieve efficiency, such as the counter abstraction [127].

Whether the parameterized verification problem for a concurrent system is decidable depends on several factors, the most important being the underlying communication graph (e.g. rings, stars, cliques), and the means of synchronization (e.g. token passing with/without information-carrying tokens, broadcast). We refer to [28] for more details. Unfortunately, there is currently no uniform concurrent system model in the literature. Hence, if one is faced with a parameterized verification problem for a given system, it is difficult to tell whether there is a published computational model that naturally captures the system's semantics. One of the key insight of this dissertation is that it is useful to provide a uniform framework that incorporates many of the foundational computational models that have appeared in the parameterized model checking literature on undecidability and decidability.

To this end, we have extended the current BIP framework to provide a general model for uniform concurrent systems that captures a large class of systems from the literature. Our model includes different forms of communication, like token-passing, rendezvous, or broadcast, as well as different communication graphs, like cliques, rings, stars. We also have showed that our framework encompasses several prominent parameterized model checking techniques. To our understanding, other seminal results that can be integrated into our framework are as follows: the cut-off results for disjunctive and conjunctive guards [59], network decomposition techniques [42, 10], and techniques based on well-structured transition systems [1] and monotonic abstraction [6].

As the core of our framework, first-order interaction logic extends propositional interaction logic [27]. Other extensions of propositional interaction logic are Dynamic BIP [31] and configuration logic [114]. Dynamic BIP extends propositional interaction logic with quantification, but is not expressive enough to write Presburger arithmetic formulas. Configuration logic uses second-order formulas to represent sets of topologies. The benefit of using FOIL is in using SMT solvers, which is essential for the design of a practical framework.

7.2 Perspectives of the future work

For the algorithmic verification of component-based systems with bounded concurrency, we believe that offering partial order semantics to the abstraction techniques is an aspiring way to tackle the state explosion problem. Following this idea, one direction we would like to explore in the future is how to combine partial order reduction techniques with IC3/PDR style reasoning, in particular the Tree-based IC3 [35] for concurrent software verification. In fact, similar idea has already been investigated recently in [82], where a dynamic reduction technique that extends Lipton's original work [108] has been proposed and incorporated into IC3 for the model checking of concurrent software. Their reduction technique differs from the partial order reduction techniques in that they use specialized encodings to instrument the multi-threaded programs such that interleavings of independent actions will not be explored in the model checking. Though sharing the same goal of avoiding redundant interleavings, no persistent set is used in their reduction.

Orthogonally, it is noticed in [2] that persistent set based partial order reduction is not optimal in the sense that multiple representatives of a Mazurkiewicz trace might be explored even with the precise persistent set. Thus, optimal reduction techniques, e.g. the one in [2], might achieve more enhancement when combining with abstraction techniques. It is also noticed in [82, 143] that in partial order reduction techniques, usually an up-front static analysis of the system model is conducted in order to obtain an over-approximation of the dependence relation and the set of transitions to be explored. The accuracy of the static analysis turns into a severe bottleneck for good reductions. Techniques that can improve the accuracy of static analysis might result in better reductions. Besides, we also plan to apply our prototype to some real-life systems that are constructed in BIP, e.g. the software running in the control and data management subsystem (CDMS) of CubETH satellite [99].

On the other hand, making formal verification beneficial in practice requires not only efficient algorithmic verification algorithm, but also some useful diagnostic information that can help a human understand why the system under verification might actually be correct or incorrect. For instance, sometimes when the verifier reports a real counterexample, it might not be easy for the programmers to figure out the core sources of the violation, in particular when the counterexample is tedious. In this case, we believe that techniques that can either automatically localize the faults [148], or produce diagnosis for explaining the bugs [105] would be useful in order to assist the programmers to understand what is neccessary in the counterexample to cause the violation.

As for the verification of parameterized systems, it is a less developed domain, compared to the non-parameterized case. For the future work, first of all we plan to fully implement the proposed parameterized verification framework in a prototype tool that integrates multiple parameterized model checking techniques to verify parameterized BIP designs.

An interesting topic we would like to talk about in the next step is the verification of parameterized systems with mixed architectures. The current verification techniques can only handle

systems with a fixed architecture, however, in reality systems may have a mixed architecture, such as the mixture of star and ring topology. The cutoff results for the star or ring topology may not apply in this case, due to the interference of each other. It is unclear to us now how to reuse the results of a single architecture for a mixed architecture.

We will also investigate novel parameterized verification techniques for component-based systems. For the safety properties, parameterized verification essentially boils down to the computation of quantified inductive invariants that are strong enough to imply the properties. We would consider how to compute such an invariant in a compositional way as in [22], that is, we first compute an invariant for each component type and an invariant for the interactions of all components, and then the invariant of the global system is obtained as the conjunction of both. The difficulty in the parameterized case is how to produce a quantified invariant that talks about all the instances of the parameterized system. One possible way to achieve this goal, as reported in [54], might be the following: we first compute an invariant for a small number of instances, and then generalize it for all instances. Further, we can use the failed generalization to guide the strengthening of the invariant, as in IC3 [32].

Finally, we will also study the second-order extensions of FOIL to express more complex architectures such as server-client whose coordinator is chosen non-deterministically. Nevertheless, this is a long-term effort.

A Appendix

A.1 An ATM transaction protocol in BIP

```
package ATM_model
    port type Sync()
    port type MsgPassing(int x)
    atom type User()
        data int request
        data int returned
        export port Sync insert()
        export port Sync enter()
        export port Sync invalid()
        export port Sync valid()
        export port MsgPassing amount(request)
        export port MsgPassing withdraw(returned)
        export port Sync restart()
        export port Sync goError()
        place l1, l2, l3, l4, l5, l6, ERROR
        initial to l1 do {request = 10; returned = 0; }
            on insert from l1 to l2
            on enter from l2 to l3
            on valid from l3 to l4
            on invalid from l3 to l5 do {request = 0;returned = 0;}
            on amount from l4 to l6
            on restart from l5 to l1
            on withdraw from l6 to l5
            on goError from l5 to ERROR provided(request != returned)
    end
    atom type ATM()
        data int requestedMoney
        data int returnedMoney
        data int time
        export port Sync insert()
        export port Sync enter()
        export port Sync invalid()
        export port Sync valid()
        export port MsgPassing amount(requestedMoney)
        export port MsgPassing withdraw(returnedMoney)
        export port Sync restart()
        export port Sync getMoney()
        export port Sync tick()
        place l1, l2, l3, l4, l5, l6, l7, l8, l9, l10
        initial to l1 do{requestedMoney = 0; returnedMoney = 0; time = 0; }
            on insert from l1 to l2
            on enter from l2 to l3
            on valid from l3 to l7
            on invalid from l3 to l8
            on amount from l7 to l9
                       on tick from l9 to l9 do {time = time + 1;}
            on getMoney from l9 to l10 provided(time >= 10) do {returnedMoney =
              requestedMoney;}
            on withdraw from l10 to l8
            on restart from l8 to l1
    end
    connector type rendezvous2(Sync p1, Sync p2)
        define [ p1 p2 ]
    end
    connector type msgpassing2(MsgPassing sender, MsgPassing receiver)
        define [ sender receiver ]
        on sender receiver down {receiver.x = sender.x ;}
    end
```

```
58      connector type single(Sync p1)
59          define [ p1 ]
60      end
61      compound type AtmSystemType()
62          component User user1()
63          component User user2()
64          component ATM atm1()
65          component ATM atm2()
66          connector single user1_goError(user1.goError)
67          connector single atm1_getMoney(atm1.getMoney)
68          connector single atm1_tick(atm1.tick)
69          connector single user2_goError(user2.goError)
70          connector single atm2_getMoney(atm2.getMoney)
71          connector single atm2_tick(atm2.tick)
72          connector rendezvous2 u1_atm1_restart(user1.restart, atm1.restart)
73          connector rendezvous2 u2_atm2_restart(user2.restart, atm2.restart)
74          connector rendezvous2 u1_atm1_insert (user1.insert, atm1.insert)
75          connector rendezvous2 u2_atm2_insert (user2.insert, atm2.insert)
76          connector rendezvous2 u1_atm1_enter (user1.enter, atm1.enter)
77          connector rendezvous2 u2_atm2_enter (user2.enter, atm2.enter)
78          connector rendezvous2 u1_atm1_invalid (user1.invalid, atm1.invalid)
79          connector rendezvous2 u2_atm2_invalid (user2.invalid, atm2.invalid)
80          connector rendezvous2 u1_atm1_valid (user1.valid, atm1.valid)
81          connector rendezvous2 u2_atm2_valid (user2.valid, atm2.valid)
82          connector msgpassing2 u1_atm1_amount (user1.amount, atm1.amount)
83          connector msgpassing2 u2_atm2_amount (user2.amount, atm2.amount)
84          connector msgpassing2 u1_atm1_withdraw (atm1.withdraw, user1.withdraw)
85          connector msgpassing2 u2_atm2_withdraw (atm2.withdraw, user2.withdraw)
86      end
87  end
88  BIPTARGET (user1.ERROR, user2.ERROR )
```

A.2 A leader election protocol in BIP

```
package leader_election
    port type DataTransfer(int tdata)
    port type Local()
    atom type process(int id)
        data int receipt
        data int tdata
        export port DataTransfer receive(receipt)
        export port DataTransfer send(tdata)
        export port Local decide()
        export port Local resend()
        export port Local back()
        place S1,S2,S3,S4
        initial to S1 do {receipt = 0; tdata = id;}
        on send from S1 to S2
        on receive from S2 to S3
        on decide from S3 to S4 provided( receipt == id)
        on back from S3 to S2 provided( receipt < id)
        on resend from S3 to S1 provided( receipt > id) do {tdata = receipt;}
    end
    atom type channel(int id)
        data int tdata
        export port DataTransfer receive(tdata)
        export port DataTransfer send(tdata)
        place S1, S2
        initial to S1 do {tdata = 0;}
        on receive from S1 to S2
        on send from S2 to S1
    end
    connector type SendReceive(DataTransfer S, DataTransfer R)
        define S R
        on S R down {R.tdata = S.tdata;}
    end
    connector type LocalTransition(Local a)
        define a
    end
    compound type System()

    component process P1(1)
    component process P2(2)
    component channel C1(1)
    component channel C2(2)
    connector SendReceive P1C1(P1.send,C1.receive)
    connector SendReceive P2C2(P2.send,C2.receive)
    connector SendReceive C1P2(C1.send, P2.receive)
    connector SendReceive C2P1(C2.send, P1.receive)
    connector LocalTransition D1(P1.decide)
    connector LocalTransition D2(P2.decide)
    connector LocalTransition B1(P1.back)
    connector LocalTransition B2(P2.back)
    connector LocalTransition R1(P1.resend)
    connector LocalTransition R2(P2.resend)
    end
end
BIPTARGET (P1.S4, P2.S4 )
```

A.3 A quorum consensus protocol in BIP

```
package Quorum
    port type FirstType()
    port type SecondType(int x)
    port type ThirdType(int x, int y)
    atom type Customer(int id)
        data int proposedValue
        data int decidedValue
        data int time
        data int clientId
        export port ThirdType send(clientId,proposedValue)
        export port ThirdType receive(clientId,decidedValue)
        export port SecondType decide(decidedValue)
        export port FirstType tick()
        export port FirstType restart()
        place S1,S2,S3,S4
        initial to S1 do {proposedValue=id;decidedValue=0;time=0;clientId =id;}
        on send from S1 to S2
        on tick from S2 to S2 do {time=time+1;}
        on receive from S2 to S3
        on decide from S3 to S4
        on restart from S4 to S1
    end
    atom type Judger()
        data int decision1
        data int decision2
        data int decision
        export port SecondType decide1(decision)
        export port SecondType decide2(decision)
        export port FirstType goError()
        export port FirstType restart()
        place S1,S2,S3, ERROR
        initial to S1 do {decision = 0; decision1 = 0; decision2 = 0;}
        on decide1 from S1 to S2 do {decision1 = decision;}
        on decide2 from S2 to S3 do {decision2 = decision;}
        on goError from S3 to ERROR provided(decision1 != decision2)
        on restart from S3 to S1 provided(decision1 == decision2)
    end
    atom type Server()
        data int decidedValue
        data int proposedValue
        data int ClientId
        export port ThirdType receive1(ClientId,proposedValue)
        export port ThirdType receive2(ClientId,proposedValue)
        export port ThirdType send(ClientId,decidedValue)
        place RECEIVE, SEND, START
        initial to START do {decidedValue=0; ClientId=0; proposedValue=0;}
        on receive1 from START to SEND do {decidedValue=proposedValue;}
        on receive2 from RECEIVE to SEND
        on send from SEND to RECEIVE
    end
    connector type OneMessagePassing(SecondType sender, SecondType receiver)
        define sender receiver
        on sender receiver down { receiver.x = sender.x; }
    end
    connector type TwoMessagePassing(ThirdType sender, ThirdType receiver)
        define sender receiver
        on sender receiver down { receiver.x=sender.x; receiver.y=sender.y;}
    end
```

```
59      connector type TowMessageWGuard(ThirdType sender, ThirdType receiver )
60          define sender receiver
61          on sender receiver provided (sender.x == receiver.x) down { receiver.y=
              sender.y;}
62      end
63      connector type Singleton (FirstType p)
64          define p
65      end
66      compound type System()
67
68          component Customer C1(1)
69          component Customer C2(2)
70          component Judger judger()
71          component Server server()
72          connector Singleton Restart1(C1.restart)
73          connector Singleton Restart2(C2.restart)
74          connector Singleton Tick1(C1.tick)
75          connector Singleton Tick2(C2.tick)
76          connector TwoMessagePassing Client1ToServer1 (C1.send,server.receive1)
77          connector TwoMessagePassing Client2ToServer1 (C2.send,server.receive1)
78          connector TwoMessagePassing Client1ToServer2 (C1.send,server.receive2)
79          connector TwoMessagePassing Client2ToServer2 (C2.send,server.receive2)
80          connector TowMessageWGuard ServerToClient1 (server.send,C1.receive)
81          connector TowMessageWGuard ServerToClient2 (server.send,C2.receive)
82          connector OneMessagePassing C1Decide1(C1.decide, judger.decide1)
83          connector OneMessagePassing C2Decide1(C2.decide, judger.decide1)
84          connector OneMessagePassing C1Decide2(C1.decide, judger.decide2)
85          connector OneMessagePassing C2Decide2(C2.decide, judger.decide2)
86          connector Singleton judgerGoError(judger.goError)
87          connector Singleton judgerRestart(judger.restart)
88      end
89  end
90  BIPTARGET ( judger.ERROR )
```

A.4 A railway control protocol in BIP

```
package RailwayControl
    port type NullPort()
    connector type Single(NullPort p)
        define p
    end
    connector type Rendezvous2(NullPort p1, NullPort p2)
        define p1 p2
    end
    atom type Train()
        data int time
        export port NullPort request()
        export port NullPort reset()
        export port NullPort authorised()
        export port NullPort unauthorised()
        export port NullPort tick()
        export port NullPort timeout()
        place Idle, Waiting, Critical
        initial to Idle do {time=0;}
        on request  from Idle   to Waiting
        on authorised   from Waiting    to Critical
        on reset    from Critical   to Idle
        on unauthorised from Waiting    to Idle
        on tick     from Waiting    to Waiting do {time=time+1;}
        on timeout  from Waiting    to Idle provided(time>20) do {time = 0;}
    end
    atom type Monitor()
        data int occupied
        data int time
        export port NullPort request()
        export port NullPort authorised()
        export port NullPort unauthorised()
        export port NullPort setVar()
        export port NullPort unsetVar1()
        export port NullPort unsetVar2()
        export port NullPort reset()
        export port NullPort tick()
        export port NullPort timeout()
        place S1, S2, S3, S4, S5
        initial to S1 do {occupied = 0; time = 0;}
        on request from S1 to S2
        on tick from S2 to S2 do {time = time+1;}
        on timeout from S2 to S1 provided(time>30) do {time=0;}
        on authorised from S2 to S3 provided(occupied == 0)
        on unauthorised from S2 to S4 provided(occupied != 0)
        on setVar from S3 to S1 do {occupied = 1;}
        on unsetVar1 from S4 to S1
        on reset from S1 to S5
        on unsetVar2 from S5 to S1 do {occupied = 0;}
    end
    compound type System()
        component Monitor mtor()
        component Train p1()
        component Train p2()
        connector Single monitor_setVar(mtor.setVar)
        connector Single monitor_unsetVar1(mtor.unsetVar1)
        connector Single monitor_unsetVar2(mtor.unsetVar2)
        connector Single monitor_tick(mtor.tick)
        connector Single monitor_timeout(mtor.timeout)
```

```
59          connector Single p1_tick(p1.tick)
60          connector Single p2_tick(p2.tick)
61          connector Single p1_timeout(p1.timeout)
62          connector Single p2_timeout(p2.timeout)
63          connector Rendezvous2 p1_request(p1.request, mtor.request)
64          connector Rendezvous2 p2_request(p2.request, mtor.request)
65          connector Rendezvous2 p1_authorised(p1.authorised, mtor.authorised)
66          connector Rendezvous2 p2_authorised(p2.authorised, mtor.authorised)
67          connector Rendezvous2 p1_unauthorised(p1.unauthorised, mtor.unauthorised)
68          connector Rendezvous2 p2_unauthorised(p2.unauthorised, mtor.unauthorised)
69          connector Rendezvous2 p1_reset(p1.reset, mtor.reset)
70          connector Rendezvous2 p2_reset(p2.reset, mtor.reset)
71      end
72  end
73  BIPTARGET (  p1.Critical,  p2.Critical )
```

A.5 Statistics for lazy abstraction

0	atm_safe_02	unsatisfiable	1.14	0s	323.0	0.0	304.0	0.008s	6.0	0s	0.0	790.0	0.544035s	0.476028s	6.0	0.0	278.0	0.0
1	atm_safe_03	unsatisfiable	29.87	0s	7002.0	0.0	6699.0	0.008002s	6.0	0s	0.0	7222.0	19.2892s	8.86855s	6.0	0.0	6673.0	0.0
2	atm_unsafe_02	satisfiable	305.51	0s	7177.0	0.0	6553.0	0.264016s	24.0	0s	20.0	31046.0	28.7778s	267.769s	8.0	0.0	2490.0	0.0
3	leader_election_safe_02	unsatisfiable	0.47	0s	127.0	0.0	108.0	0.012s	2.0	0s	0.0	78.0	0.312018s	0.044004s	6.0	0.0	41.0	0.0
4	leader_election_safe_03	unsatisfiable	6.38	0s	1253.0	0.0	1126.0	0.02s	3.0	0s	0.0	991.0	4.7763s	0.78805s	12.0	0.0	411.0	0.0
5	leader_election_safe_04	unsatisfiable	183.11	0s	19591.0	0.0	18299.0	0.064005s	4.0	0s	0.0	17896.0	147.129s	27.6177s	21.0	0.0	4796.0	0.0
6	leader_election_unsafe_02	satisfiable	0.27	0s	82.0	0.0	76.0	0.004s	1.0	0s	0.0	45.0	0.164012s	0.020002s	3.0	0.0	58.0	0.0
7	leader_election_unsafe_03	satisfiable	4.45	0s	870.0	0.0	826.0	0.008001s	2.0	0s	0.0	690.0	3.40421s	0.656039s	9.0	0.0	458.0	0.0
8	quorum_safe_02	unsatisfiable	1.30	0s	313.0	0.0	279.0	0.012001s	2.0	0s	1.0	255.0	0.792043s	0.308024s	4.0	0.0	217.0	0.0
9	quorum_safe_03	unsatisfiable	8.46	0s	1792.0	0.0	1624.0	0.028004s	3.0	0s	1.0	1546.0	4.78031s	3.01218s	7.0	0.0	1462.0	0.0
10	quorum_safe_04	unsatisfiable	53.78	0s	9540.0	0.0	8753.0	0.084006s	4.0	0s	1.0	8542.0	26.4257s	22.8694s	10.0	0.0	8356.0	0.0
11	quorum_unsafe_02	satisfiable	0.27	0s	79.0	0.0	75.0	0.004s	1.0	0s	1.0	44.0	0.176011s	0.008001s	5.0	0.0	51.0	0.0
12	quorum_unsafe_03	satisfiable	0.37	0s	112.0	0.0	108.0	0.004001s	1.0	0s	1.0	50.0	0.264015s	0.004s	5.0	0.0	71.0	0.0
13	quorum_unsafe_04	satisfiable	0.48	0s	151.0	0.0	147.0	0.004s	1.0	0s	1.0	56.0	0.336022s	0.008s	5.0	0.0	94.0	0.0
14	quorum_unsafe_05	satisfiable	0.61	0s	196.0	0.0	192.0	0.004s	1.0	0s	1.0	62.0	0.424028s	0.012s	5.0	0.0	120.0	0.0
15	quorum_unsafe_06	satisfiable	0.72	0s	247.0	0.0	243.0	0.004s	1.0	0s	1.0	68.0	0.524032s	0.012s	5.0	0.0	149.0	0.0
16	quorum_unsafe_07	satisfiable	0.92	0s	304.0	0.0	300.0	0.008s	1.0	0s	1.0	74.0	0.688044s	0.008s	5.0	0.0	181.0	0.0
17	quorum_unsafe_08	satisfiable	0.99	0s	367.0	0.0	363.0	0.012001s	1.0	0s	1.0	80.0	0.760048s	0s	5.0	0.0	216.0	0.0
18	quorum_unsafe_09	satisfiable	1.20	0s	436.0	0.0	432.0	0.008s	1.0	0s	1.0	86.0	0.848054s	0.012001s	5.0	0.0	254.0	0.0
19	quorum_unsafe_10	satisfiable	1.26	0s	511.0	0.0	507.0	0.016001s	1.0	0s	1.0	92.0	0.952061s	0.016001s	5.0	0.0	295.0	0.0
20	quorum_unsafe_11	satisfiable	1.38	0s	592.0	0.0	588.0	0.012001s	1.0	0s	1.0	98.0	1.05207s	0.020001s	5.0	0.0	339.0	0.0
21	railway_control_safe_02	unsatisfiable	0.12	0s	96.0	0.0	94.0	0s	1.0	0s	1.0	105.0	0.064003s	0.012002s	0.0	0.0	86.0	0.0
22	railway_control_safe_03	unsatisfiable	0.39	0s	347.0	0.0	335.0	0.004s	1.0	0s	1.0	356.0	0.236015s	0.084005s	0.0	0.0	317.0	0.0
23	railway_control_safe_04	unsatisfiable	1.57	0s	1330.0	0.0	1276.0	0.012s	2.0	0s	1.0	1375.0	0.840059s	0.416019s	0.0	0.0	1056.0	0.0
24	railway_control_safe_05	unsatisfiable	10.36	0s	7104.0	0.0	6681.0	0.064004s	2.0	0s	1.0	6473.0	5.24433s	3.64423s	0.0	0.0	3460.0	0.0
25	railway_control_safe_06	unsatisfiable	51.82	0s	32512.0	0.0	30033.0	0.700043s	3.0	0s	1.0	27471.0	26.1016s	18.4492s	0.0	0.0	9276.0	0.0
26	railway_control_safe_07	unsatisfiable	210.68	0s	108591.0	0.0	99510.0	13.6289s	4.0	0s	1.0	81051.0	94.2619s	67.6442s	0.0	0.0	23888.0	0.0
27	railway_control_safe_08	unsatisfiable	264.92	0s	133429.0	0.0	122494.0	28.3698s	5.0	0s	1.0	88683.0	112.443s	71.9485s	0.0	0.0	59693.0	0.0
28	railway_control_unsafe_02	satisfiable	0.05	0s	30.0	0.0	31.0	0s	0.0	0s	0.0	24.0	0.004s	0s	0.0	0.0	31.0	0.0
29	railway_control_unsafe_03	satisfiable	70.43	0s	8545.0	0.0	7492.0	0.480027s	31.0	0s	31.0	5057.0	52.0472s	15.249s	0.0	0.0	515.0	0.0
30	railway_control_unsafe_04	satisfiable	216.75	0s	27073.0	0.0	24005.0	1.32408s	62.0	0s	31.0	13623.0	163.014s	42.9787s	0.0	0.0	815.0	0.0
31	temperature_safe_02	unsatisfiable	1.61	0s	184.0	0.0	80.0	0.016s	7.0	0s	22.0	59.0	1.02006s	0.332024s	0.0	0.0	37.0	0.0
32	temperature_safe_03	unsatisfiable	1.97	0s	239.0	0.0	99.0	0.012001s	7.0	0s	22.0	71.0	1.30408s	0.468029s	0.0	0.0	49.0	0.0
33	temperature_safe_04	unsatisfiable	2.49	0s	294.0	0.0	118.0	0.004001s	7.0	0s	22.0	83.0	1.6041s	0.540035s	0.0	0.0	61.0	0.0
34	temperature_safe_05	unsatisfiable	2.88	0s	349.0	0.0	137.0	0.016001s	7.0	0s	22.0	95.0	1.91611s	0.616047s	0.0	0.0	73.0	0.0
35	temperature_safe_06	unsatisfiable	3.48	0s	404.0	0.0	156.0	0.016s	7.0	0s	22.0	107.0	2.26414s	0.776054s	0.0	0.0	85.0	0.0
36	temperature_safe_07	unsatisfiable	3.89	0s	459.0	0.0	175.0	0.016004s	7.0	0s	22.0	119.0	2.54816s	0.936058s	0.0	0.0	97.0	0.0
37	temperature_safe_08	unsatisfiable	4.27	0s	514.0	0.0	194.0	0.012001s	7.0	0s	22.0	131.0	2.84018s	0.940058s	0.0	0.0	109.0	0.0
38	temperature_safe_09	unsatisfiable	4.80	0s	569.0	0.0	213.0	0.008001s	7.0	0s	22.0	143.0	3.2202s	1.04807s	0.0	0.0	121.0	0.0
39	temperature_safe_10	unsatisfiable	5.20	0s	624.0	0.0	232.0	0.016002s	7.0	0s	22.0	155.0	3.50822s	1.15207s	0.0	0.0	133.0	0.0
40	temperature_safe_11	unsatisfiable	5.94	0s	679.0	0.0	251.0	0.012s	7.0	0s	22.0	167.0	3.95225s	1.29208s	0.0	0.0	145.0	0.0
41	temperature_unsafe_02	satisfiable	0.05	0s	5.0	0.0	6.0	0s	0.0	0s	0.0	2.0	0.004001s	0s	0.0	0.0	6.0	0.0
42	temperature_unsafe_03	satisfiable	0.06	0s	6.0	0.0	7.0	0s	0.0	0s	0.0	2.0	0.004001s	0s	0.0	0.0	7.0	0.0
43	temperature_unsafe_04	satisfiable	0.05	0s	7.0	0.0	8.0	0s	0.0	0s	0.0	2.0	0.004s	0s	0.0	0.0	8.0	0.0
44	temperature_unsafe_05	satisfiable	0.06	0s	8.0	0.0	9.0	0s	0.0	0s	0.0	2.0	0.004s	0s	0.0	0.0	9.0	0.0
45	temperature_unsafe_06	satisfiable	0.06	0s	9.0	0.0	10.0	0s	0.0	0s	0.0	2.0	0.008s	0s	0.0	0.0	10.0	0.0
46	temperature_unsafe_07	satisfiable	0.07	0s	10.0	0.0	11.0	0s	0.0	0s	0.0	2.0	0.012001s	0s	0.0	0.0	11.0	0.0
47	temperature_unsafe_08	satisfiable	0.07	0s	11.0	0.0	12.0	0s	0.0	0s	0.0	2.0	0.004001s	0s	0.0	0.0	12.0	0.0
48	temperature_unsafe_09	satisfiable	0.06	0s	12.0	0.0	13.0	0s	0.0	0s	0.0	2.0	0.004s	0s	0.0	0.0	13.0	0.0
49	temperature_unsafe_10	satisfiable	0.07	0s	13.0	0.0	14.0	0s	0.0	0s	0.0	2.0	0.012s	0s	0.0	0.0	14.0	0.0
50	temperature_unsafe_11	satisfiable	0.06	0s	14.0	0.0	15.0	0s	0.0	0s	0.0	2.0	0.008s	0s	0.0	0.0	15.0	0.0
51	ticket_safe_02	unsatisfiable	0.24	0s	51.0	0.0	43.0	0s	2.0	0s	0.0	36.0	0.140008s	0.024002s	5.0	0.0	27.0	0.0

52	ticket_safe_03	unsatisfiable	19.74	0s	1725.0	0.0	1227.0	0.072005s	14.0	0s	0.0	1282.0	15.621s	3.07618s	36.0	0.0	232.0	0.0
53	ticket_unsafe_02	satisfiable	0.05	0s	9.0	0.0	10.0	0s	0.0	0s	0.0	5.0	0.004s	0s	0.0	0.0	10.0	0.0
54	ticket_unsafe_03	satisfiable	0.06	0s	16.0	0.0	17.0	0s	0.0	0s	0.0	6.0	0.008s	0s	0.0	0.0	17.0	0.0
55	ticket_unsafe_04	satisfiable	0.06	0s	25.0	0.0	26.0	0s	0.0	0s	0.0	7.0	0.008001s	0s	0.0	0.0	26.0	0.0
56	ticket_unsafe_05	satisfiable	0.08	0s	36.0	0.0	37.0	0s	0.0	0s	0.0	8.0	0.012002s	0s	0.0	0.0	37.0	0.0
57	ticket_unsafe_06	satisfiable	0.07	0s	49.0	0.0	50.0	0s	0.0	0s	0.0	9.0	0.016001s	0s	0.0	0.0	50.0	0.0
58	ticket_unsafe_07	satisfiable	0.10	0s	64.0	0.0	65.0	0s	0.0	0s	0.0	10.0	0.024002s	0s	0.0	0.0	65.0	0.0
59	ticket_unsafe_08	satisfiable	0.10	0s	81.0	0.0	82.0	0s	0.0	0s	0.0	11.0	0.028002s	0s	0.0	0.0	82.0	0.0
60	ticket_unsafe_09	satisfiable	0.15	0s	100.0	0.0	101.0	0s	0.0	0s	0.0	12.0	0.028002s	0s	0.0	0.0	101.0	0.0
61	ticket_unsafe_10	satisfiable	0.14	0s	121.0	0.0	122.0	0s	0.0	0s	0.0	13.0	0.044002s	0s	0.0	0.0	122.0	0.0
62	ticket_unsafe_11	satisfiable	0.17	0s	144.0	0.0	145.0	0s	0.0	0s	0.0	14.0	0.048003s	0s	0.0	0.0	145.0	0.0

Table A.1 – Verification statistics for lazy abstraction. From the second column, each represents model, result, total running time, time of partial order reduction, number of transfers, number of reduction attempts, number of nodes created, time of refinements, number of refinements, time of cycle detection, number of local predicates, number of coverage check, time of transfers, time of coverage check, number of global predicates, number of successful reductions, art size and number of cycle detections respectively.

A.6 Statistics for lazy abstraction with persistent set reduction

0	atm_safe_02	unsatisfiable	0.65	0.004s	164.0	66.0	153.0	0s	2.0	0.068006s	0.0	124.0	0.396018s	0.076009s	3.0	33.0	146.0	151.0
1	atm_safe_03	unsatisfiable	5.30	0.016001s	1683.0	546.0	1634.0	0.004001s	2.0	0.45203s	0.0	1361.0	3.42421s	1.03207s	3.0	269.0	1627.0	1632.0
2	atm_safe_04	unsatisfiable	61.17	0.34402s	16633.0	4237.0	16273.0	0.004s	2.0	9.96864s	0.0	13913.0	35.9942s	11.8927s	3.0	1873.0	16266.0	16271.0
3	atm_unsafe_02	satisfiable	83.28	0s	2884.0	929.0	2648.0	0.236015s	24.0	13.4248s	20.0	9716.0	14.2929s	51.7192s	8.0	374.0	843.0	10148.0
4	leader_election_safe_02	unsatisfiable	0.31	0s	78.0	20.0	68.0	0.004001s	2.0	0.008001s	0.0	60.0	0.204013s	0.016s	6.0	8.0	24.0	60.0
5	leader_election_safe_03	unsatisfiable	1.94	0.004s	340.0	92.0	320.0	0.020002s	3.0	0.048002s	0.0	302.0	1.5001s	0.092007s	12.0	50.0	94.0	302.0
6	leader_election_safe_04	unsatisfiable	17.75	0.012001s	2140.0	515.0	2034.0	0.048003s	4.0	0.18801s	0.0	2002.0	14.4609s	2.06812s	21.0	248.0	529.0	2002.0
7	leader_election_safe_05	unsatisfiable	140.41	0.044003s	12831.0	2892.0	12300.0	0.16001s	5.0	1.21607s	0.0	12250.0	108.531s	25.9416s	26.0	1188.0	2908.0	12250.0
8	leader_election_unsafe_02	satisfiable	0.24	0s	56.0	15.0	53.0	0s	1.0	0.004s	0.0	44.0	0.140008s	0.012002s	3.0	7.0	43.0	44.0
9	leader_election_unsafe_03	satisfiable	1.64	0s	295.0	81.0	284.0	0.008s	2.0	0.068006s	0.0	265.0	1.25208s	0.112006s	9.0	45.0	156.0	265.0
10	leader_election_unsafe_04	satisfiable	24.45	0.012002s	2158.0	505.0	2086.0	0.032003s	3.0	0.248011s	0.0	2053.0	18.3851s	4.13226s	18.0	235.0	1179.0	2053.0
11	leader_election_unsafe_05	satisfiable	228.11	0.02s	13325.0	2898.0	12967.0	0.116007s	4.0	1.80412s	0.0	12916.0	166.798s	49.7071s	21.0	1146.0	7264.0	12916.0
12	quorum_safe_02	unsatisfiable	0.72	0.004s	125.0	47.0	118.0	0.012002s	3.0	0.076006s	1.0	140.0	0.384024s	0.144009s	6.0	14.0	74.0	158.0
13	quorum_safe_03	unsatisfiable	5.57	0.020002s	663.0	217.0	618.0	0.048002s	8.0	0.952057s	1.0	1036.0	2.41616s	1.6401s	7.0	71.0	445.0	1151.0
14	quorum_safe_04	unsatisfiable	31.24	0.072004s	2878.0	837.0	2702.0	0.124008s	14.0	6.25238s	1.0	6379.0	10.1927s	13.4608s	10.0	268.0	2386.0	6886.0
15	quorum_safe_05	unsatisfiable	183.56	0.096009s	12986.0	3256.0	12256.0	0.296021s	19.0	33.5421s	1.0	34713.0	47.227s	97.2101s	14.0	930.0	10608.0	36901.0
16	quorum_unsafe_02	satisfiable	0.06	0s	21.0	10.0	22.0	0s	0.0	0s	0.0	13.0	0.004s	0s	0.0	4.0	22.0	16.0
17	quorum_unsafe_03	satisfiable	0.07	0s	24.0	10.0	25.0	0s	0.0	0s	0.0	13.0	0.012001s	0s	0.0	4.0	25.0	16.0
18	quorum_unsafe_04	satisfiable	0.08	0s	27.0	10.0	28.0	0s	0.0	0.004s	0.0	13.0	0.008001s	0s	0.0	4.0	28.0	16.0
19	quorum_unsafe_05	satisfiable	0.09	0s	30.0	10.0	31.0	0s	0.0	0s	0.0	13.0	0.004001s	0s	0.0	4.0	31.0	16.0
20	quorum_unsafe_06	satisfiable	0.08	0s	33.0	10.0	34.0	0s	0.0	0s	0.0	13.0	0.016001s	0s	0.0	4.0	34.0	16.0
21	quorum_unsafe_07	satisfiable	0.09	0.008s	36.0	10.0	37.0	0s	0.0	0s	0.0	13.0	0.012003s	0s	0.0	4.0	37.0	16.0
22	quorum_unsafe_08	satisfiable	0.11	0.004001s	39.0	10.0	40.0	0s	0.0	0s	0.0	13.0	0.020002s	0s	0.0	4.0	40.0	16.0
23	quorum_unsafe_09	satisfiable	0.10	0.012s	42.0	10.0	43.0	0s	0.0	0s	0.0	13.0	0.008001s	0s	0.0	4.0	43.0	16.0
24	quorum_unsafe_10	satisfiable	0.12	0.008001s	45.0	10.0	46.0	0s	0.0	0s	0.0	13.0	0.008s	0s	0.0	4.0	46.0	16.0
25	quorum_unsafe_11	satisfiable	0.12	0.016s	48.0	10.0	49.0	0s	0.0	0s	0.0	13.0	0.016001s	0s	0.0	4.0	49.0	16.0
26	railway_control_safe_02	unsatisfiable	0.15	0s	84.0	24.0	82.0	0s	1.0	0.012001s	1.0	93.0	0.060004s	0.012001s	0.0	6.0	74.0	93.0
27	railway_control_safe_03	unsatisfiable	0.43	0.020001s	276.0	71.0	264.0	0s	1.0	0.044004s	1.0	285.0	0.224012s	0.060006s	0.0	27.0	246.0	285.0
28	railway_control_safe_04	unsatisfiable	1.54	0.084003s	971.0	210.0	917.0	0.012s	2.0	0.216016s	1.0	998.0	0.756047s	0.240016s	0.0	85.0	741.0	998.0
29	railway_control_safe_05	unsatisfiable	6.01	0.192009s	3592.0	603.0	3337.0	0.072005s	2.0	0.82405s	1.0	3157.0	2.96818s	1.34008s	0.0	222.0	2393.0	3157.0
30	railway_control_safe_06	unsatisfiable	55.01	0.976056s	24303.0	2798.0	21824.0	0.728045s	3.0	11.2247s	1.0	19350.0	24.2815s	12.3808s	0.0	583.0	6505.0	19350.0
31	railway_control_safe_07	unsatisfiable	234.05	3.24019s	87629.0	8732.0	78389.0	14.4609s	4.0	58.4636s	1.0	59748.0	86.6734s	45.9029s	0.0	1455.0	16665.0	59748.0
32	railway_control_safe_08	unsatisfiable	308.91	10.4126s	112235.0	11821.0	101300.0	29.6419s	5.0	64.052s	1.0	67975.0	104.074s	49.1111s	0.0	2992.0	41225.0	67975.0
33	railway_control_unsafe_02	satisfiable	0.05	0s	30.0	5.0	31.0	0s	0.0	0s	0.0	24.0	0.008s	0s	0.0	0.0	31.0	24.0
34	railway_control_unsafe_03	satisfiable	78.56	0s	8545.0	1217.0	7492.0	0.46003s	31.0	10.9927s	31.0	5057.0	50.4551s	14.6049s	0.0	0.0	515.0	5057.0
35	railway_control_unsafe_04	satisfiable	251.30	0.008s	27073.0	3482.0	24005.0	1.38809s	62.0	32.298s	31.0	13623.0	163.558s	42.6427s	0.0	0.0	815.0	13623.0
36	temperature_safe_02	unsatisfiable	1.84	0s	184.0	55.0	80.0	0.012002s	7.0	0.320017s	22.0	59.0	1.01206s	0.328027s	0.0	0.0	37.0	59.0
37	temperature_safe_03	unsatisfiable	2.52	0s	239.0	66.0	99.0	0.008001s	7.0	0.440025s	22.0	71.0	1.36409s	0.416025s	0.0	0.0	49.0	71.0
38	temperature_safe_04	unsatisfiable	2.81	0s	294.0	77.0	118.0	0.020001s	7.0	0.496027s	22.0	83.0	1.59611s	0.504027s	0.0	0.0	61.0	83.0
39	temperature_safe_05	unsatisfiable	3.39	0s	349.0	88.0	137.0	0.012s	7.0	0.584035s	22.0	95.0	1.94412s	0.636041s	0.0	0.0	73.0	95.0
40	temperature_safe_06	unsatisfiable	4.05	0s	404.0	99.0	156.0	0.016001s	7.0	0.740048s	22.0	107.0	2.26014s	0.736044s	0.0	0.0	85.0	107.0
41	temperature_safe_07	unsatisfiable	4.68	0.004001s	459.0	110.0	175.0	0.020001s	7.0	0.848054s	22.0	119.0	2.52016s	0.820051s	0.0	0.0	97.0	119.0
42	temperature_safe_08	unsatisfiable	5.25	0.004s	514.0	121.0	194.0	0.012s	7.0	0.936055s	22.0	131.0	2.89618s	0.93206s	0.0	0.0	109.0	131.0
43	temperature_safe_09	unsatisfiable	5.85	0.004s	569.0	132.0	213.0	0.020001s	7.0	1.00806s	22.0	143.0	3.22019s	1.02407s	0.0	0.0	121.0	143.0
44	temperature_safe_10	unsatisfiable	6.15	0.016s	624.0	143.0	232.0	0.012s	7.0	1.08406s	22.0	155.0	3.56023s	1.13607s	0.0	0.0	133.0	155.0
45	temperature_safe_11	unsatisfiable	6.63	0.008001s	679.0	154.0	251.0	0.012s	7.0	1.17206s	22.0	167.0	3.86824s	1.24409s	0.0	0.0	145.0	167.0
46	temperature_unsafe_02	satisfiable	0.05	0s	5.0	1.0	6.0	0s	0.0	0s	0.0	2.0	0.004s	0s	0.0	0.0	6.0	2.0
47	temperature_unsafe_03	satisfiable	0.05	0s	6.0	1.0	7.0	0s	0.0	0s	0.0	2.0	0.004s	0s	0.0	0.0	7.0	2.0
48	temperature_unsafe_04	satisfiable	0.06	0s	7.0	1.0	8.0	0s	0.0	0s	0.0	2.0	0s	0s	0.0	0.0	8.0	2.0
49	temperature_unsafe_05	satisfiable	0.06	0s	8.0	1.0	9.0	0s	0.0	0s	0.0	2.0	0.008001s	0s	0.0	0.0	9.0	2.0
50	temperature_unsafe_06	satisfiable	0.05	0.004s	9.0	1.0	10.0	0s	0.0	0s	0.0	2.0	0.004001s	0s	0.0	0.0	10.0	2.0
51	temperature_unsafe_07	satisfiable	0.05	0s	10.0	1.0	11.0	0s	0.0	0s	0.0	2.0	0.004001s	0s	0.0	0.0	11.0	2.0

52	temperature_unsafe_08	satisfiable	0.06	0s	11.0	1.0	12.0	0s	0.0	0s	0.0	2.0	0.008001s	0s	0.0	0.0	12.0	2.0
53	temperature_unsafe_09	satisfiable	0.07	0s	12.0	1.0	13.0	0s	0.0	0s	0.0	2.0	0.004s	0s	0.0	0.0	13.0	2.0
54	temperature_unsafe_10	satisfiable	0.06	0s	13.0	1.0	14.0	0s	0.0	0s	0.0	2.0	0.012s	0s	0.0	0.0	14.0	2.0
55	temperature_unsafe_11	satisfiable	0.06	0s	14.0	1.0	15.0	0s	0.0	0s	0.0	2.0	0.008s	0s	0.0	0.0	15.0	2.0
56	ticket_safe_02	unsatisfiable	0.28	0s	51.0	24.0	43.0	0.004001s	2.0	0.012s	0.0	36.0	0.176011s	0.020001s	5.0	0.0	27.0	36.0
57	ticket_safe_03	unsatisfiable	21.30	0s	1725.0	570.0	1227.0	0.068004s	14.0	1.47208s	0.0	1282.0	15.149s	3.2042s	37.0	0.0	232.0	1282.0
58	ticket_unsafe_02	satisfiable	0.06	0s	9.0	4.0	10.0	0s	0.0	0s	0.0	5.0	0s	0s	0.0	0.0	10.0	5.0
59	ticket_unsafe_03	satisfiable	0.06	0s	16.0	5.0	17.0	0s	0.0	0s	0.0	6.0	0.008s	0s	0.0	0.0	17.0	6.0
60	ticket_unsafe_04	satisfiable	0.06	0s	25.0	6.0	26.0	0s	0.0	0s	0.0	7.0	0.012001s	0s	0.0	0.0	26.0	7.0
61	ticket_unsafe_05	satisfiable	0.08	0s	36.0	7.0	37.0	0s	0.0	0s	0.0	8.0	0.016001s	0s	0.0	0.0	37.0	8.0
62	ticket_unsafe_06	satisfiable	0.08	0.004s	49.0	8.0	50.0	0s	0.0	0s	0.0	9.0	0.016001s	0s	0.0	0.0	50.0	9.0
63	ticket_unsafe_07	satisfiable	0.08	0.004001s	64.0	9.0	65.0	0s	0.0	0s	0.0	10.0	0.020001s	0s	0.0	0.0	65.0	10.0
64	ticket_unsafe_08	satisfiable	0.10	0.004001s	81.0	10.0	82.0	0s	0.0	0s	0.0	11.0	0.028001s	0s	0.0	0.0	82.0	11.0
65	ticket_unsafe_09	satisfiable	0.13	0s	100.0	11.0	101.0	0s	0.0	0s	0.0	12.0	0.044003s	0s	0.0	0.0	101.0	12.0
66	ticket_unsafe_10	satisfiable	0.14	0.008001s	121.0	12.0	122.0	0s	0.0	0s	0.0	13.0	0.044002s	0s	0.0	0.0	122.0	13.0
67	ticket_unsafe_11	satisfiable	0.16	0.016s	144.0	13.0	145.0	0s	0.0	0s	0.0	14.0	0.056005s	0s	0.0	0.0	145.0	14.0

Table A.2 – Verification statistics for lazy abstraction with persistent set reduction. From the second column, each represents model, result, total running time, time of partial order reduction, number of transfers, number of reduction attempts, number of nodes created, time of refinements, number of refinements, time of cycle detection, number of local predicates, number of coverage check, time of transfers, time of coverage check, number of global predicates, number of successful reductions, art size and number of cycle detections respectively.

A.7 Statistics for lazy abstraction with simultaneous set reduction

0	atm_safe_02	unsatisfiable	2.06	0s	394.0	55.0	340.0	0.004s	3.0	0.0	338.0	1.5321s	0.384019s	6.0	40.0	326.0
1	atm_safe_03	unsatisfiable	69.87	0.012001s	8543.0	499.0	6618.0	0.008001s	5.0	0.0	6880.0	51.3472s	15.641s	9.0	436.0	6593.0
2	atm_unsafe_02	satisfiable	117.32	0s	4373.0	55.0	3642.0	0.092003s	24.0	20.0	13993.0	26.5497s	87.8895s	8.0	40.0	1379.0
3	leader_election_safe_02	unsatisfiable	0.27	0s	60.0	5.0	50.0	0s	2.0	0.0	31.0	0.192011s	0.008s	6.0	2.0	19.0
4	leader_election_safe_03	unsatisfiable	1.91	0s	338.0	24.0	276.0	0s	3.0	0.0	165.0	1.6881s	0.064004s	12.0	11.0	102.0
5	leader_election_safe_04	unsatisfiable	15.81	0.036002s	1833.0	99.0	1564.0	0.008s	3.0	0.0	1243.0	13.7929s	1.27208s	18.0	52.0	895.0
6	leader_election_safe_05	unsatisfiable	134.94	0.356023s	11710.0	401.0	9962.0	0.008001s	5.0	0.0	7839.0	118.539s	10.6687s	24.0	221.0	3913.0
7	leader_election_unsafe_02	satisfiable	0.16	0s	36.0	3.0	33.0	0s	1.0	0.0	13.0	0.084005s	0s	3.0	2.0	22.0
8	leader_election_unsafe_03	satisfiable	1.05	0.004s	211.0	10.0	177.0	0s	2.0	0.0	65.0	0.884053s	0.020001s	9.0	5.0	82.0
9	leader_election_unsafe_04	satisfiable	10.26	0.020002s	1282.0	43.0	1032.0	0s	3.0	0.0	540.0	9.40459s	0.288019s	18.0	23.0	347.0
10	leader_election_unsafe_05	satisfiable	111.41	0.208012s	7930.0	202.0	6553.0	0.012001s	4.0	0.0	4429.0	101.21s	3.92025s	21.0	107.0	2184.0
11	quorum_safe_02	unsatisfiable	1.01	0s	211.0	35.0	190.0	0.004s	3.0	1.0	260.0	0.664039s	0.192012s	6.0	6.0	139.0
12	quorum_safe_03	unsatisfiable	8.93	0.012001s	1373.0	138.0	1185.0	0.016002s	5.0	1.0	2879.0	4.85631s	3.3402s	8.0	8.0	1076.0
13	quorum_safe_04	unsatisfiable	78.72	0.052001s	10558.0	511.0	8778.0	0.048004s	9.0	1.0	25751.0	37.6623s	37.9384s	12.0	16.0	6782.0
14	quorum_unsafe_02	satisfiable	0.08	0s	24.0	9.0	25.0	0s	0.0	0.0	14.0	0.008001s	0s	0.0	1.0	25.0
15	quorum_unsafe_03	satisfiable	0.08	0s	29.0	9.0	30.0	0s	0.0	0.0	14.0	0.012002s	0s	0.0	0.0	30.0
16	quorum_unsafe_04	satisfiable	0.08	0s	32.0	9.0	33.0	0s	0.0	0.0	14.0	0.008001s	0s	0.0	0.0	33.0
17	quorum_unsafe_05	satisfiable	0.09	0s	35.0	9.0	36.0	0s	0.0	0.0	14.0	0.016s	0s	0.0	0.0	36.0
18	quorum_unsafe_06	satisfiable	0.09	0s	38.0	9.0	39.0	0s	0.0	0.0	14.0	0.020002s	0s	0.0	0.0	39.0
19	quorum_unsafe_07	satisfiable	0.09	0.004s	41.0	9.0	42.0	0s	0.0	0.0	14.0	0.016001s	0s	0.0	0.0	42.0
20	quorum_unsafe_08	satisfiable	0.11	0.004s	44.0	9.0	45.0	0s	0.0	0.0	14.0	0.024001s	0s	0.0	0.0	45.0
21	quorum_unsafe_09	satisfiable	0.11	0.008s	47.0	9.0	48.0	0s	0.0	0.0	14.0	0.020002s	0s	0.0	0.0	48.0
22	quorum_unsafe_10	satisfiable	0.13	0.024002s	50.0	9.0	51.0	0s	0.0	0.0	14.0	0.012s	0.004s	0.0	0.0	51.0
23	quorum_unsafe_11	satisfiable	0.14	0.036001s	53.0	9.0	54.0	0s	0.0	0.0	14.0	0.016003s	0s	0.0	0.0	54.0
24	railway_control_safe_02	unsatisfiable	0.35	0.004s	162.0	22.0	153.0	0s	1.0	1.0	151.0	0.228017s	0.048s	0.0	0.0	143.0
25	railway_control_safe_03	unsatisfiable	1.94	0.092005s	761.0	67.0	690.0	0s	1.0	1.0	687.0	1.24007s	0.416029s	0.0	0.0	679.0
26	railway_control_safe_04	unsatisfiable	14.07	2.69617s	3965.0	176.0	3317.0	0s	1.0	1.0	3313.0	7.84448s	2.38015s	0.0	0.0	3305.0
27	railway_control_safe_05	unsatisfiable	152.96	98.2661s	17945.0	429.0	14161.0	0s	1.0	1.0	14156.0	38.5384s	10.1926s	0.0	0.0	14148.0
28	railway_control_unsafe_02	satisfiable	0.05	0s	15.0	4.0	16.0	0s	0.0	0.0	9.0	0.004s	0s	0.0	0.0	16.0
29	railway_control_unsafe_03	satisfiable	0.06	0s	17.0	4.0	18.0	0s	0.0	0.0	9.0	0.008001s	0s	0.0	0.0	18.0
30	railway_control_unsafe_04	satisfiable	0.06	0.004s	19.0	4.0	20.0	0s	0.0	0.0	9.0	0s	0s	0.0	0.0	20.0
31	railway_control_unsafe_05	satisfiable	0.07	0s	21.0	4.0	22.0	0s	0.0	0.0	9.0	0.004s	0s	0.0	0.0	22.0
32	railway_control_unsafe_06	satisfiable	0.05	0s	23.0	4.0	24.0	0s	0.0	0.0	9.0	0.008s	0s	0.0	0.0	24.0
33	railway_control_unsafe_07	satisfiable	0.06	0s	25.0	4.0	26.0	0s	0.0	0.0	9.0	0s	0s	0.0	0.0	26.0
34	railway_control_unsafe_08	satisfiable	0.07	0s	27.0	4.0	28.0	0s	0.0	0.0	9.0	0.008001s	0s	0.0	0.0	28.0
35	railway_control_unsafe_09	satisfiable	0.07	0s	29.0	4.0	30.0	0s	0.0	0.0	9.0	0.004s	0s	0.0	0.0	30.0
36	railway_control_unsafe_10	satisfiable	0.06	0s	31.0	4.0	32.0	0s	0.0	0.0	9.0	0.012001s	0s	0.0	0.0	32.0
37	railway_control_unsafe_11	satisfiable	0.07	0.004s	33.0	4.0	34.0	0s	0.0	0.0	9.0	0.008001s	0s	0.0	0.0	34.0
38	temperature_safe_02	unsatisfiable	1.36	0s	184.0	3.0	80.0	0.008s	7.0	22.0	59.0	0.900053s	0.244018s	0.0	0.0	37.0
39	temperature_safe_03	unsatisfiable	1.67	0s	239.0	4.0	99.0	0.016002s	7.0	22.0	71.0	1.20407s	0.308019s	0.0	0.0	49.0
40	temperature_safe_04	unsatisfiable	2.07	0s	294.0	5.0	118.0	0.008s	7.0	22.0	83.0	1.4441s	0.416018s	0.0	0.0	61.0
41	temperature_safe_05	unsatisfiable	2.44	0s	349.0	6.0	137.0	0.012001s	7.0	22.0	95.0	1.72811s	0.496031s	0.0	0.0	73.0
42	temperature_safe_06	unsatisfiable	2.81	0.004s	404.0	7.0	156.0	0.012001s	7.0	22.0	107.0	1.93611s	0.564041s	0.0	0.0	85.0
43	temperature_safe_07	unsatisfiable	3.22	0.004s	459.0	8.0	175.0	0.004s	7.0	22.0	119.0	2.24413s	0.608047s	0.0	0.0	97.0
44	temperature_safe_08	unsatisfiable	3.69	0.004s	514.0	9.0	194.0	0.012s	7.0	22.0	131.0	2.55216s	0.720046s	0.0	0.0	109.0
45	temperature_safe_09	unsatisfiable	4.28	0.004s	569.0	10.0	213.0	0.012001s	7.0	22.0	143.0	2.90819s	0.848047s	0.0	0.0	121.0
46	temperature_safe_10	unsatisfiable	4.75	0.004s	624.0	11.0	232.0	0.012001s	7.0	22.0	155.0	3.1562s	0.908056s	0.0	0.0	133.0
47	temperature_safe_11	unsatisfiable	5.03	0.012001s	679.0	12.0	251.0	0.012001s	7.0	22.0	167.0	3.48821s	0.968062s	0.0	0.0	145.0
48	temperature_unsafe_02	satisfiable	0.05	0s	5.0	1.0	6.0	0s	0.0	0.0	2.0	0s	0s	0.0	0.0	6.0
49	temperature_unsafe_03	satisfiable	0.04	0s	6.0	1.0	7.0	0s	0.0	0.0	2.0	0.004s	0s	0.0	0.0	7.0
50	temperature_unsafe_04	satisfiable	0.05	0s	7.0	1.0	8.0	0s	0.0	0.0	2.0	0.004s	0s	0.0	0.0	8.0
51	temperature_unsafe_05	satisfiable	0.05	0s	8.0	1.0	9.0	0s	0.0	0.0	2.0	0.004s	0s	0.0	0.0	9.0

52	temperature_unsafe_06	satisfiable	0.06	0s	9.0	1.0	10.0	0s	0.0	0.0	2.0	0.008001s	0s	0.0	0.0	10.0
53	temperature_unsafe_07	satisfiable	0.06	0.004s	10.0	1.0	11.0	0s	0.0	0.0	2.0	0.004001s	0s	0.0	0.0	11.0
54	temperature_unsafe_08	satisfiable	0.05	0s	11.0	1.0	12.0	0s	0.0	0.0	2.0	0.008001s	0s	0.0	0.0	12.0
55	temperature_unsafe_09	satisfiable	0.06	0s	12.0	1.0	13.0	0s	0.0	0.0	2.0	0.012001s	0s	0.0	0.0	13.0
56	temperature_unsafe_10	satisfiable	0.06	0.004s	13.0	1.0	14.0	0s	0.0	0.0	2.0	0.012001s	0s	0.0	0.0	14.0
57	temperature_unsafe_11	satisfiable	0.07	0.012001s	14.0	1.0	15.0	0s	0.0	0.0	2.0	0.012001s	0s	0.0	0.0	15.0
58	ticket_safe_02	unsatisfiable	0.22	0s	51.0	8.0	43.0	0s	2.0	0.0	36.0	0.13601s	0.016s	5.0	0.0	27.0
59	ticket_safe_03	unsatisfiable	18.40	0s	1924.0	22.0	1301.0	0.040001s	12.0	0.0	1443.0	14.5449s	3.2322s	31.0	0.0	228.0
60	ticket_unsafe_02	satisfiable	0.06	0s	9.0	4.0	10.0	0s	0.0	0.0	5.0	0.008s	0s	0.0	0.0	10.0
61	ticket_unsafe_03	satisfiable	0.06	0s	16.0	5.0	17.0	0s	0.0	0.0	6.0	0.004001s	0s	0.0	0.0	17.0
62	ticket_unsafe_04	satisfiable	0.06	0s	25.0	6.0	26.0	0s	0.0	0.0	7.0	0.004s	0s	0.0	0.0	26.0
63	ticket_unsafe_05	satisfiable	0.16	0s	36.0	7.0	37.0	0s	0.0	0.0	8.0	0.016001s	0s	0.0	0.0	37.0
64	ticket_unsafe_06	satisfiable	0.08	0s	49.0	8.0	50.0	0s	0.0	0.0	9.0	0.008s	0s	0.0	0.0	50.0
65	ticket_unsafe_07	satisfiable	0.09	0.004s	64.0	9.0	65.0	0s	0.0	0.0	10.0	0.020001s	0s	0.0	0.0	65.0
66	ticket_unsafe_08	satisfiable	0.10	0.008001s	81.0	10.0	82.0	0s	0.0	0.0	11.0	0.024001s	0s	0.0	0.0	82.0
67	ticket_unsafe_09	satisfiable	0.13	0.004s	100.0	11.0	101.0	0s	0.0	0.0	12.0	0.032002s	0s	0.0	0.0	101.0
68	ticket_unsafe_10	satisfiable	0.14	0.024001s	121.0	12.0	122.0	0s	0.0	0.0	13.0	0.036003s	0s	0.0	0.0	122.0
69	ticket_unsafe_11	satisfiable	0.17	0.044003s	144.0	13.0	145.0	0s	0.0	0.0	14.0	0.044002s	0s	0.0	0.0	145.0

Table A.3 – Verification statistics for lazy abstraction with simultaneous set reduction. From the second column of the table, each represents model, result, total running time, time of partial order reduction, number of transfers, number of reduction attempts, number of nodes created, time of refinements, number of refinements, number of local predicates, number of coverage check, time of transfers, time of coverage check, number of global predicates, number of successful reductions, art size respectively.

A.8 Statistics for lazy abstraction with persistent set reduction under symmetry

leader_election_safe_02	unsatisfiable	1.11	0s	14.0	5.0	0.004s	1.0	0s	0.184011	7.0	0.0	0s	0.028001s	3.0	5.0	13.0	13.0	13.0
leader_election_safe_03	unsatisfiable	1.32	0s	19.0	9.0	0.004s	1.0	0s	0.432027	9.0	0.0	0s	0.036002s	3.0	9.0	18.0	18.0	18.0
leader_election_safe_04	unsatisfiable	1.53	0s	24.0	12.0	0s	1.0	0s	0.656041	11.0	0.0	0s	0.032002s	3.0	12.0	23.0	23.0	23.0
leader_election_safe_05	unsatisfiable	1.92	0s	29.0	15.0	0.004s	1.0	0s	1.032060	13.0	0.0	0s	0.036002s	3.0	15.0	28.0	28.0	28.0
leader_election_safe_06	unsatisfiable	1.99	0.004s	34.0	18.0	0.004s	1.0	0s	1.796110	15.0	0.0	0.004001s	0.048003s	3.0	18.0	33.0	33.0	33.0
leader_election_safe_07	unsatisfiable	2.94	0.004s	39.0	21.0	0.004s	1.0	0s	2.712170	17.0	0.0	0.004s	0.072005s	3.0	21.0	38.0	38.0	38.0
leader_election_safe_08	unsatisfiable	3.34	0s	44.0	24.0	0.004001s	1.0	0s	3.176200	19.0	0.0	0s	0.060004s	3.0	24.0	43.0	43.0	43.0
leader_election_safe_09	unsatisfiable	3.69	0.004s	49.0	27.0	0.004s	1.0	0s	3.488220	21.0	0.0	0s	0.068005s	3.0	27.0	48.0	48.0	48.0
leader_election_safe_10	unsatisfiable	4.60	0.004s	54.0	30.0	0.008s	1.0	0s	4.412280	23.0	0.0	0s	0.076004s	3.0	30.0	53.0	53.0	53.0
leader_election_safe_11	unsatisfiable	5.29	0s	59.0	33.0	0.012001s	1.0	0s	5.080320	25.0	0.0	0s	0.076005s	3.0	33.0	58.0	58.0	58.0
leader_election_unsafe_02	satisfiable	0.37	0s	56.0	15.0	0s	1.0	0.012s	0.168010	43.0	0.0	0.012s	0.104007s	3.0	7.0	53.0	44.0	44.0
leader_election_unsafe_03	satisfiable	1.83	0.008s	295.0	81.0	0.008s	2.0	0.032003s	0.416026	156.0	0.0	0.084005s	1.13607s	9.0	45.0	284.0	265.0	265.0
leader_election_unsafe_04	satisfiable	20.81	0.024002s	2158.0	505.0	0.024001s	3.0	0.276017s	0.496031	1179.0	0.0	3.40821s	15.925s	18.0	235.0	2086.0	2053.0	2053.0
leader_election_unsafe_05	satisfiable	203.58	0.028002s	13325.0	2898.0	0.112007s	4.0	1.66411s	1.128070	7264.0	0.0	42.5787s	152.309s	21.0	1146.0	12967.0	12916.0	12916.0
quorum_safe_02	unsatisfiable	1.49	0.004001s	125.0	47.0	0.004s	3.0	0.080006s	0.620039	74.0	1.0	0.120007s	0.400023s	6.0	14.0	118.0	140.0	158.0
quorum_safe_03	unsatisfiable	6.99	0.008001s	663.0	217.0	0.048001s	8.0	1.00806s	1.360080	445.0	1.0	1.6481s	2.32015s	7.0	71.0	618.0	1036.0	1151.0
quorum_safe_04	unsatisfiable	32.46	0.032002s	2878.0	837.0	0.124007s	14.0	5.74438s	2.380150	2386.0	1.0	12.8768s	9.88462s	10.0	268.0	2702.0	6379.0	6886.0
quorum_safe_05	unsatisfiable	180.00	0.112007s	12986.0	3256.0	0.360023s	19.0	31.502s	3.664230	10608.0	1.0	91.5817s	45.8309s	14.0	930.0	12256.0	34713.0	36901.0
quorum_unsafe_02	satisfiable	0.66	0s	21.0	10.0	0s	0.0	0s	0.580037	22.0	0.0	0s	0.008s	0.0	4.0	22.0	13.0	16.0
quorum_unsafe_03	satisfiable	1.35	0s	24.0	10.0	0s	0.0	0s	1.248080	25.0	0.0	0s	0.008001s	0.0	4.0	25.0	13.0	16.0
quorum_unsafe_04	satisfiable	2.06	0s	27.0	10.0	0s	0.0	0s	1.892120	28.0	0.0	0s	0.008s	0.0	4.0	28.0	13.0	16.0
quorum_unsafe_05	satisfiable	3.15	0s	30.0	10.0	0s	0.0	0s	2.992190	31.0	0.0	0s	0.012s	0.0	4.0	31.0	13.0	16.0
quorum_unsafe_06	satisfiable	4.69	0.004s	33.0	10.0	0s	0.0	0s	4.580290	34.0	0.0	0s	0.016001s	0.0	4.0	34.0	13.0	16.0
quorum_unsafe_07	satisfiable	6.28	0.004s	36.0	10.0	0s	0.0	0s	6.048380	37.0	0.0	0s	0.008001s	0.0	4.0	37.0	13.0	16.0
quorum_unsafe_08	satisfiable	7.95	0.004001s	39.0	10.0	0s	0.0	0s	7.756490	40.0	0.0	0.004s	0.012s	0.0	4.0	40.0	13.0	16.0
quorum_unsafe_09	satisfiable	9.88	0s	42.0	10.0	0s	0.0	0.004s	9.572600	43.0	0.0	0s	0.016001s	0.0	4.0	43.0	13.0	16.0
quorum_unsafe_10	satisfiable	12.22	0.004s	45.0	10.0	0s	0.0	0s	11.672700	46.0	0.0	0s	0.020002s	0.0	4.0	46.0	13.0	16.0
quorum_unsafe_11	satisfiable	14.88	0.008s	48.0	10.0	0s	0.0	0s	14.684900	49.0	0.0	0s	0.016002s	0.0	4.0	49.0	13.0	16.0
ticket_safe_02	unsatisfiable	0.23	0.004001s	29.0	13.0	0.008s	2.0	0.004s	0.064004	9.0	0.0	0.008001s	0.068003s	5.0	3.0	22.0	18.0	18.0
ticket_safe_03	unsatisfiable	1.32	0s	175.0	56.0	0.020001s	6.0	0.032001s	0.160010	23.0	0.0	0.036003s	0.868055s	20.0	8.0	122.0	79.0	79.0
ticket_safe_04	unsatisfiable	23.24	0.004s	1905.0	423.0	0.17601s	15.0	1.28408s	0.296018	17.0	0.0	2.56817s	17.2171s	40.0	21.0	1178.0	968.0	968.0
ticket_safe_05	unsatisfiable	162.55	0.008001s	8822.0	1506.0	0.488028s	18.0	23.0774s	0.504032	91.0	0.0	63.0839s	68.9883s	44.0	56.0	6082.0	10714.0	10714.0
ticket_unsafe_02	satisfiable	0.12	0s	7.0	4.0	0s	0.0	0s	0.064003	8.0	0.0	0s	0s	0.0	2.0	8.0	5.0	5.0
ticket_unsafe_03	satisfiable	0.23	0s	10.0	5.0	0s	0.0	0s	0.168010	11.0	0.0	0s	0.004001s	0.0	3.0	11.0	6.0	6.0
ticket_unsafe_04	satisfiable	0.36	0s	13.0	6.0	0s	0.0	0s	0.292018	14.0	0.0	0s	0.004s	0.0	4.0	14.0	7.0	7.0
ticket_unsafe_05	satisfiable	0.56	0s	16.0	7.0	0s	0.0	0.004s	0.460029	17.0	0.0	0s	0.004001s	0.0	5.0	17.0	8.0	8.0
ticket_unsafe_06	satisfiable	0.77	0s	19.0	8.0	0s	0.0	0s	0.676042	20.0	0.0	0s	0.008s	0.0	6.0	20.0	9.0	9.0
ticket_unsafe_07	satisfiable	1.06	0s	22.0	9.0	0s	0.0	0s	0.964060	23.0	0.0	0s	0.008s	0.0	7.0	23.0	10.0	10.0
ticket_unsafe_08	satisfiable	1.37	0s	25.0	10.0	0s	0.0	0s	1.264080	26.0	0.0	0s	0.008s	0.0	8.0	26.0	11.0	11.0
ticket_unsafe_09	satisfiable	1.78	0s	28.0	11.0	0s	0.0	0s	1.644100	29.0	0.0	0s	0.008001s	0.0	9.0	29.0	12.0	12.0
ticket_unsafe_10	satisfiable	2.12	0.004s	31.0	12.0	0s	0.0	0s	1.980120	32.0	0.0	0s	0.008001s	0.0	10.0	32.0	13.0	13.0
ticket_unsafe_11	satisfiable	2.66	0.008s	34.0	13.0	0s	0.0	0s	2.352150	35.0	0.0	0s	0.008001s	0.0	11.0	35.0	14.0	14.0

Table A.4 – Verification statistics for lazy abstraction with persistent set reduction under symmetry. From the first column of the table, each represents model, result, total running time, time of partial order reduction, number of transfers, number of reduction attempts, time of refinements, number of refinements, time of cycle detection, time of independence analysis, art size, number of local predicate, time of coverage check, time of transfers, number of global predicate, number of successful reductions, number of nodes created, number of coverage check, number of cycle detections respectively.

Bibliography

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 313–, Washington, DC, USA, 1996. IEEE Computer Society.

[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *ACM SIGPLAN Notices*, volume 49, pages 373–384. ACM, 2014.

[3] Parosh Aziz Abdulla. Regular model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(2):109–118, 2012.

[4] Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. Constrained monotonic abstraction: A cegar for parameterized verification. In *International Conference on Concurrency Theory*, pages 86–101. Springer, 2010.

[5] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS*, 2007.

[6] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Monotonic abstraction: on efficient verification of parameterized systems. *International Journal of Foundations of Computer Science*, 20(05):779–801, 2009.

[7] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *International Conference on Computer Aided Verification*, pages 145–157. Springer, 2007.

[8] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design*, 18(2):97–116, 2001.

[9] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.

[10] B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR*. 2014.

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

[12] Lăcrămioara Aştefănoaei, Souha Ben Rayana, Saddek Bensalem, Marius Bozga, and Jacques Combaz. Compositional verification of parameterised timed systems. In *NASA Formal Methods Symposium*, pages 66–81. Springer, 2015.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[14] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, July 2011.

[15] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[16] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 268–283, London, UK, UK, 2001. Springer-Verlag.

[17] Thomas Ball and Sriram K Rajamani. The slam toolkit. In *International Conference on Computer Aided Verification*, pages 260–264. Springer, 2001.

[18] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[19] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, May 2011.

[20] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, TACAS '00, pages 188–203, London, UK, UK, 2000. Springer-Verlag.

[21] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *International Conference on Computer Aided Verification*, pages 614–619. Springer, 2009.

[22] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *International Symposium on Automated Technology for Verification and Analysis*, pages 64–79. Springer, 2008.

[23] Dirk Beyer and M Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[24] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[25] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (ctigar). In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 831–848, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[26] Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab, and Qiang Wang. Formal verification of infinite-state bip models. In *International Symposium on Automated Technology for Verification and Analysis*, pages 326–343. Springer, 2015.

[27] Simon Bliudze and Joseph Sifakis. The algebra of connectors: Structuring interaction in bip. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 11–20, New York, NY, USA, 2007. ACM.

[28] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. Decidability of parameterized verification. *Synthesis Lectures on Distributed Computing Theory*, 2015.

[29] Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *CAV*, pages 135–148, 2008.

[30] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *International Conference on Computer Aided Verification*, pages 403–418. Springer, 2000.

[31] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using dy-bip. In *Proceedings of the 11th International Conference on Software Composition*, SC'12, pages 1–16, Berlin, Heidelberg, 2012. Springer-Verlag.

[32] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[33] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[34] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 385–395. IEEE Computer Society, 2003.

[35] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 277–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Kratos: A software model checker for systemc. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 310–316, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.

[38] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[39] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[40] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[41] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ANSI-C. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 570–574. Springer, 2005.

[42] Edmund Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. Verification by network decomposition. In *International Conference on Concurrency Theory*, pages 276–291. Springer, 2004.

[43] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[44] Edmund M Clarke, E Allen Emerson, Somesh Jha, and A Prasad Sistla. Symmetry reductions in model checking. In *CAV*, 1998.

[45] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 1996.

[46] Edmund M Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *STTT*, 1999.

[47] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[48] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[49] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.

[50] Satyaki Das, David L Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171. Springer, 1999.

[51] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[53] Alastair F Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 2012.

[54] Michael Dooley and Fabio Somenzi. Proving parameterized systems safe by generalizing clausal proofs of small instances. In *International Conference on Computer Aided Verification*, pages 292–309. Springer, 2016.

[55] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134. IEEE, 2011.

[56] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, LICS '98, pages 70–, Washington, DC, USA, 1998. IEEE Computer Society.

[57] E Allen Emerson, Somesh Jha, and Doron Peled. Combining partial order and symmetry reductions. In *TACAS*. 1997.

[58] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Proceedings of the 17th International Conference on Automated Deduction*, CADE-17, pages 236–254, London, UK, UK, 2000. Springer-Verlag.

[59] E Allen Emerson and Vineet Kahlon. Model checking guarded protocols. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 361–370. IEEE, 2003.

[60] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 85–94, New York, NY, USA, 1995. ACM.

[61] E Allen Emerson and A Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 1996.

[62] E Allen Emerson and Richard J Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *CHDVM*. 1999.

[63] E Allen Emerson and Thomas Wahl. Dynamic symmetry reduction. In *TACAS*. 2005.

[64] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. *LICS*, 1999.

[65] Sami Evangelista and Christophe Pajault. Solving the ignoring problem for partial order reduction. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):155–170, 2010.

[66] Alain Finkel and Ph Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001.

[67] Kathleen Fisher. Using formal methods to enable more secure vehicles: Darpa's hacms program. *SIGPLAN Not.*, 49(9):1–1, August 2014.

[68] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121. ACM, 2005.

[69] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 191–202. ACM, 2002.

[70] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *International SPIN Workshop on Model Checking of Software*, pages 213–224. Springer, 2003.

[71] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[72] G. Geeraerts, J. F. Raskin, and L. Van Begin. Expand, enlarge and check: New algorithms for the coverability problem of wsts. *J. Comput. Syst. Sci.*, 72(1):180–203, February 2006.

[73] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, enlarge, and check: New algorithms for the coverability problem of wsts. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 287–298. Springer, 2004.

[74] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, July 1992.

[75] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards smt model checking of array-based systems. In *International Joint Conference on Automated Reasoning*, pages 67–82. Springer, 2008.

[76] Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In *International Joint Conference on Automated Reasoning*, pages 22–29. Springer, 2010.

[77] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90, pages 176–185, London, UK, UK, 1991. Springer-Verlag.

[78] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.

[79] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer aided verification*, pages 72–83. Springer, 1997.

[80] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 405–416, New York, NY, USA, 2012. ACM.

[81] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

[82] Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 246–265. Springer, 2017.

[83] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 331–344. ACM, 2011.

[84] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, volume 6806 of *LNCS*, pages 412–417. Springer, 2011.

[85] J. Y Halpern. Presburger arithmetic with unary predicates is π_1^1 complete. *The Journal of Symbolic Logic*, 1991.

[86] Henri Hansen. *Abstractions for transition systems with applications to stubborn sets*, pages 104–123. Lecture Notes in Computer Science. Springer International Publishing, 1 2017.

[87] Henri Hansen, Shang-Wei Lin, Yang Liu, Truong Khanh Nguyen, and Jun Sun. Diamonds are a girl's best friend: Partial order reduction for timed automata with abstractions. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 391–406, 2014.

[88] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 232–244, New York, NY, USA, 2004. ACM.

[89] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *International Conference on Computer Aided Verification*, pages 262–274. Springer, 2003.

[90] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 2002.

[91] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 1–15, Berlin, Heidelberg, 2006. Springer-Verlag.

[92] Thomas A. Henzinger and Joseph Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, October 2007.

[93] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[94] Hossein Hojjat, Filip Konečnỳ, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems. In *International Symposium on Formal Methods*, pages 247–251. Springer, 2012.

[95] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. Horn clauses for communicating timed systems. *arXiv preprint arXiv:1412.1153*, 2014.

[96] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[97] Radu Iosif. Symmetry reductions for model checking of concurrent dynamic software. *STTT*, 2004.

[98] C Norris Ip and David L Dill. Better verification through symmetry. *Formal methods in system design*, 1996.

[99] Anton Ivanov, Louis Masson, Stefano Rossi, Federico Belloni, Reto Wiesendanger, Volker Gass, Markus Rothacher, Christine Hollenstein, Benjamin Männel, Patrick Fleischmann, Heinz Mathis, Martin Klaper, Marcel Joss, and Erich Styger. CubETH: low cost GNSS space experiment for precise orbit determination. Technical report, 2014.

[100] Taylor T. Johnson and Sayan Mitra. A small model theorem for rectangular hybrid automata networks. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32Nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 18–34, Berlin, Heidelberg, 2012. Springer-Verlag.

[101] Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

[102] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 398–413, Berlin, Heidelberg, 2009. Springer-Verlag.

[103] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969.

[104] Yonit Kesten, Amir Pnueli, Elad Shahar, and Lenore D. Zuck. Network invariants in action. In *Proceedings of the 13th International Conference on Concurrency Theory*, CONCUR '02, pages 101–115, London, UK, UK, 2002. Springer-Verlag.

[105] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 165–176, New York, NY, USA, 2015. ACM.

[106] Shuvendu K Lahiri, Randal E Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *Computer Aided Verification*, pages 141–153. Springer, 2003.

[107] Shuvendu K Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT techniques for fast predicate abstraction. In *Computer Aided Verification*, pages 424–437. Springer, 2006.

[108] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 1975.

[109] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Form. Methods Syst. Des.*, 6(1):11–44, January 1995.

[110] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[111] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *International Colloquium on Theoretical Aspects of Computing*, pages 183–197. Springer, 2006.

[112] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[113] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[114] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration logics: Modelling architecture styles. In *Revised Selected Papers of the 12th International Conference on Formal Aspects of Component Software - Volume 9539*, FACS 2015, pages 256–274, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[115] A Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[116] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[117] Kenneth L McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.

[118] Kenneth L McMillan. Applications of craig interpolants in model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Springer, 2005.

[119] Kenneth L McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136. Springer, 2006.

[120] Kenneth L McMillan. Interpolants and symbolic model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 89–90. Springer, 2007.

[121] Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 2006.

[122] Steven P Miller, Michael W Whalen, and Darren D Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, 2010.

[123] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.

[124] Doron Peled. All from one, one for all: On model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 409–423, London, UK, UK, 1993. Springer-Verlag.

[125] Doron Peled. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 17–28, London, UK, UK, 1998. Springer-Verlag.

[126] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 82–97, London, UK, UK, 2001. Springer-Verlag.

[127] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 107–122, London, UK, UK, 2002. Springer-Verlag.

[128] Corneliu Popeea, Andrey Rybalchenko, and Andreas Wilhelm. Reduction for compositional verification of multi-threaded programs. In *FMCAD*, pages 187–194. IEEE, 2014.

[129] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[130] Marco Roveri, Iman Narasamdya, and Alessandro Cimatti. Software model checking with explicit scheduler and symbolic threads. *Logical Methods in Computer Science*, 8, 2012.

[131] S. Schmitz and P. Schnoebelen. The power of well-structured systems. In *CONCUR*, 2013.

[132] Joseph Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18(3):227–258, 1982.

[133] Joseph Sifakis. Rigorous system design. *Foundations and Trends® in Electronic Design Automation*, 6(4):293–362, April 2013.

[134] Joseph Sifakis. System design automation: Challenges and limitations. *Proceedings of the IEEE*, 103(11):2093–2103, 2015.

[135] Joseph Sifakis, Saddek Bensalem, Simon Bliudze, and Marius Bozga. A theory agenda for component-based design. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, volume 8950 of *Lecture Notes in Computer Science*, pages 409–439. Springer, 2015.

[136] I. Suzuki. Proving properties of a ring of finite-state machines. *Inf. Process. Lett.*, 1988.

[137] W. Thomas. *Languages, automata, and logic*. Springer, 1997.

[138] Stefano Tonetta. Abstract model checking without computing the abstraction. In *FM 2009: Formal Methods*, pages 89–105. Springer, 2009.

[139] Antti Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification*, pages 156–165. Springer, 1990.

[140] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.

[141] Antti Valmari. A state space tool for concurrent system models expressed in C++. In Jyrki Nummenmaa, Outi Sievi-Korte, and Erkki Mäkinen, editors, *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), Tampere, Finland, October 9-10, 2015.*, volume 1525 of *CEUR Workshop Proceedings*, pages 91–105. CEUR-WS.org, 2015.

[142] Antti Valmari. Stop it, and be stubborn!. *ACM Trans. Embed. Comput. Syst.*, 16(2):46:1–46:26, January 2017.

[143] Antti Valmari and Henri Hansen. Stubborn set intuition explained. In Lawrence Cabac, Lars Michael Kristensen, and Heiko Rölke, editors, *Petri Nets and Software Engineering. International Workshop, PNSE'16, Toruń, Poland, June 20-21, 2016. Proceedings*, volume 1591 of *CEUR Workshop Proceedings*, pages 213–232. CEUR-WS.org, 2016.

[144] François Vernadat, Pierre Azéma, and François Michel. Covering step graph. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 516–535, London, UK, UK, 1996. Springer-Verlag.

[145] Bjorn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 210–217. IEEE, 2013.

[146] Thomas Wahl and Alastair Donaldson. Replication and abstraction: Symmetry in automated formal verification. *Symmetry*, 2010.

[147] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS 2008*, volume 4963 of *LNCS*, pages 382–396. Springer, 2008.

[148] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

Wang Qiang

qiang.wang@epfl.ch

EDUCATION

PhD candidate in computer and communication science 2013-

Ecole Polytechnique Federale de Lausanne

I am currently a PhD candidate under the supervision of Prof.Joseph Sifakis, focusing on algorithmic verification of component-based systems as well as parameterized verification.

MSc. in information and communication science 2010-2013

National University of Defense Technology, China

Master thesis: formal analysis of security protocols.

Graduate with third-class merit.

BSc. in information and communication science 2006-2010

National University of Defense Technology, China

Bachelor thesis: provable security of Elliptic Curve cryptography algorithms.

Outstanding undergraduate ranking 2nd out of 155.

AWARDS

1. Third-class merit awarded by president of National University of Defense Technology in 2012.
2. Outstanding undergraduate student award in 2010.
3. Second place in Chinese National Mathematics Competition in 2009.
4. Third place in Chinese National Information Security Competition in 2008.

PUBLICATIONS

1. Exploiting Symmetry for Efficient Verification of Infinite-State Component-Based Systems, International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA 2016), pages 246–263, 2016, Springer.
2. Parameterized systems in BIP: design and model checking. With Igor Konnov, Tomer Kotek, Helmut Veith, Simon Bliudze and Joseph Sifakis. 27th International Conference on Concurrency Theory (CONCUR 2016). (Corresponding author)
3. Formal verification of infinite state BIP models. With Simon Bliudze, Alessandro Cimatti, Mohamad Jaber, Sergio Mover, Marco Roveri, Wajeb Saab. 13th International Symposium on Automated Technology for Verification and Analysis (ATVA 2015). (Corresponding author)

4. Verification of component-based systems via predicate abstraction and simultaneous set reduction. With Simon Bliudze. 10th International Symposium on Trustworthy Global Computing (TGC 2015). (Corresponding author)

5. SeBip: a symbolic symbolic executor for BIP. With Simon Bliudze. 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015). (Corresponding author)

6. Automatic fault localization for BIP. With Lei Yan, Simon Bliudze, Xiaoguang Mao. 1st Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA 2015). (Corresponding author)

7. TOA: A tag-owner-assisting RFID authentication protocol toward access control and ownership transfer. With Xie, W. and Xie, L. and Zhang, C. and Wang, C. and Tang, C. Security and Communication Networks. Volume 7, May 2014, Pages 934-944.

8. RFID seeking: Finding a lost tag rather than only detecting its missing. With Xie, W. and Xie, L. and Zhang, C. and Xu, J. and Zhang, Q. and Tang, C. Journal of Network and Computer Applications, Volume 42, June 2014, Pages 135-142.

SOFTWARE

1. Kratos4BIP: a model checker for infinite-state component-based systems in BIP.

2. BIPChecker: a model checker for parameterized component-based systems in BIP