Incomplete Property Checking for Asynchronous Reactive Systems

Wei Wei

submitted to the University of Konstanz
in partial fulfillment of the requirement for the degree of

Doctor of Engineering Science (Dr.-Ing.)

in Computer Science

April, 2008

Supervisor:
Prof. Dr. Stefan Leue
Universität Konstanz, Konstanz, Germany

Reviewers:
Prof. Dr. Stefan Leue
Universität Konstanz, Konstanz, Germany

Prof. Dr. George S. Avrunin
University of Massachusetts at Amherst, Amherst, U.S.A.

Wei Wei
Software Engineering Group
Fachbereich Informatik und Informationswissenschaft
Fach D 67
Universität Konstanz
D-78457 Konstanz
Germany
Wei.Wei@uni-konstanz.de

Abstract

Asynchronous reactive systems find applications in a wide range of software systems such as communication protocols, embedded software systems, etc. It is highly desirable to rigorously show that these systems are correctly designed, because a correct design is vital to providing services of high quality. However, formal approaches to the verification of these systems, such as model checking, are often difficult because these systems usually possess extremely large or even infinite state spaces. In fact, in case of infinite state systems, many interesting verification problems become undecidable and traditional finite state model checking techniques cannot be applied to those systems.

We propose an Integer Linear Program (ILP) solving based verification framework that concentrates on the local analysis of the cyclic behavior of each individual component of a system. This way we avoid the exploration of the huge or even infinite state space of the system. More precisely, we use automated abstraction techniques to transform an original system into a set of local control flow cycles and over-approximate the message passing effects of these cycles. Then, we derive a necessary condition for the violation of the considered property on the message passing effects of cycles. We further encode the necessary condition into an ILP problem whose solution space represents the property violating behavior. The infeasibility of the ILP problem then establishes the satisfaction of the property by the system. Moreover, the resulting ILP problem can be checked in polynomial time. We have applied our framework to the verification of the buffer boundedness and livelock freedom properties, both of which are undecidable for asynchronous reactive systems with an infinite state space.

On one hand, the verification framework that we propose is efficient since it needs not to consider an exponential number of all possible interleavings of the executions of the system components. Instead, it maintains the locality of the analysis of each component and reduces the original verification problem into a polynomial-time solvable problem. On the other hand, our framework is incomplete: it either proves the satisfaction of a property, or returns an inconclusive verdict "UNKNOWN". In the latter case, the property may or may not be satisfied by the system under scrutiny. This imprecision comes from the potential coarseness of the abstractions that our verification framework employs. After all, the incompleteness of the framework is inevitable since the properties that we check are undecidable.

While the precision of our framework remains an issue, we propose a counterexample guided abstraction refinement procedure based on the discovery of dependencies among control flow cycles. The discovered cycle dependencies can be efficiently encoded into linear inequalities that are used to augment the constraint set of the original property determination ILP problem. The newly added constraints may rule out certain spurious behavior that violates the property, and thus refine the abstraction. The cycle dependency discovery methods that we devise are also incomplete. This means that some spurious property violating behavior may never be eliminated by any cycle dependencies that we can discover.

We make the verification methods applicable to two widely used modeling languages, namely Promela and UML RT, by devising tailored code abstraction techniques. These techniques address abstraction issues concerning specific fea-

tures of the two languages, such as message definitions, process instantiations, buffer assignments, etc. In particular, we have developed an incomplete automated termination proving technique for program loops, which does not rely on the explicit construction of ranking functions. Finally, we implemented several prototype tools with which we obtained promising experimental results on real life system models.

Kurzfassung

Asynchrone Reaktive Systeme kommen in einem breiten Spektrum von Softwaresystemen, so z. B. in Kommunikationsprotokollen und eingebetteten Softwaresystemen, zur Anwendung. Es ist von großer Wichtigkeit zu zeigen, dass diese Systeme korrekt entworfen sind, da ein solcher korrekter Entwurf die Grundlage dafür ist, dass diese Systeme Dienste von hoher Qualität liefern. Formale Ansätze zur Verifikation solcher Systeme, wie z. B. Model Checking, sind aufgrund der extrem großen oder sogar unendlichen Zustandsräume, die diese Systeme besitzen, nur schwer anwendbar. Im Fall von Systemen mit unendlichen Zustandsraum werden viele interessante Verfikationsprobleme unentscheidbar, und traditionelle Model Checking Techniken für endliche Zustandsräume sind dann nicht mehr anwendbar.

Wir schlagen ein auf ganzzahliger Programmierung beruhendes Gerüst zur Verifikation vor, welches auf der lokalen Analyse des zyklischen Verhaltens einer jeden Systemkomponente beruht. Dadurch vermeiden wir die Exploration des sehr großen oder sogar unendlichen Zustandsraums des Systems. Um genau zu sein, verwenden wir automatisierte Abstraktionstechniken, um ein zu analysierendes System in eine Menge von Kontrollflusszyklen zu überführen und den Nachrichten-Austauscheffekt dieser Zyklen zu überapproximieren. Für eine zu analysierende Eigenschaft bestimmen wir notwendige Bedingungen an die Kontrollflusszyklen, welche die Verletzung dieser Eigenschaft anzeigen. Diese Bedingungen werden dann in ein ganzzahliges Programmierungsproblem überführt, so dass dessen Lösungsraum das Eigenschaften verletzende Verhalten des Systems repräsentiert. Die Unlösbarkeit des ganzzahligen Programmierungsproblems beweist damit die Erfüllung der Eigenschaft durch das System. Das ganzzahlige Programmierungsproblem kann dann in polynomieller Zeit auf seine Lösbarkeit hin überprüft werden. Wir haben unser Verifikationsgerüst für die Verifikation der konkreten Eigenschaften der Beschränktheit von Kommunikationspuffern (boundedness of buffers) und der Abwesenheit von divergierendem Systemverhalten (absence of livelock) konkretisiert. Diese beiden Eigenschaften sind für asynchrone reaktive Systeme mit einem unendlichen Zustandsraum unentscheidbar.

Dem gegenüber ist unser Verifikationsgerüst effizient, da es nicht alle exponentiell vielen linearisierten Ausführungsfolgen der Systemkomponenten betrachten muss. Es erhält die Lokalität der Analyse jeder einzelnen Systemkomponente und reduziert das urspüngliche Verifikationsproblem zu einem Problem, das in polynomieller Zeit gelöst werden kann. Auf der anderen Seite ist unser Verifikationsgerüst unvollständig: es zeigt entweder die Erfüllung einer Eigenschaft, oder es endet mit einem untentschiedenen Verdikt, das wir mit “UNKNOWN” bezeichnen. In diesem Fall ist die Eigenschaft für das untersuchte System entweder erfüllt, oder nicht erfüllt. Diese Unschärfe wird durch die Grobheit der von unserem Verifikationsgerüst verwendeten Abstraktion hervorgerufen. Die Unvollständigkeit unseres Verfikationsgerüstes ist in jedem Fall unvermeidbar, da die überprüften Eigenschaften unentscheidbar sind.

Da die Präzision unseres auf Abstraktion beruhenden Verifikationsgerüsts ein kritischer Punkt ist, schlagen wir automatische, auf Gegenbeispielen beruhende Abstraktionsverfeinerungsschritte vor. Diese beruhen auf der Entdeckung von Abhängigkeiten zwischen den Kontrollflusszyklen. Die entdeckten Abhängigkeiten können effizient in lineare Ungleichungen übersetzt werden, die dann dazu

verwendet werden, die Menge der zur Lösung des ursprünglichen ILP Problems ermittelten linearen Ungleichungen zu erweitern. Die zusätzlichen Ungleichungen schließen möglicherweise nicht zulässiges Verhalten aus, welches zu einer Eigenschaftsverletzung führte, und verfeinern somit die ursprüngliche Abstraktion. Die Methoden zur Entdeckung von Zyklenabhängigkeiten sind ebenfalls unvollständig. Dies bedeutet, dass ein Teil des unzulässigen Verhaltens, das Eigenschaftsverletzungen zur Folge hat, möglicherweise durch die von uns entdeckten Zyklenabhängigkeiten nie entdeckt werden kann.

Die resultierenden Verifikationsverfahren werden durch die Entwicklung von geeigneten Code-Abstraktionstechniken für zwei häufig verwendete Modellierungssprachen, nämlich Promela und UML RT, angepasst. Diese Techniken berücksichtigen spezifische Aspekte der beiden Sprachen, so besonders die Definition von Nachrichten, Prozessinstanziierungen, die Zuweisung von Nachrichtenpuffern, etc. Darüber hinaus haben wir eine automatische, unvollständige Methode zum Beweis der Termination von Programmschleifen entwickelt, welche ohne die explizite Konstruktion von Rangfolge-Funktionen auskommt. Schließlich haben wir mehrere Prototypen-Werkzeuge entwickelt, mit deren Hilfe wir viel versprechende experimentelle Resultate für reale Software-Modelle erhalten haben.

Acknowledgments

Firstly, I thank my supervisor Stefan Leue, not only for his wise guidance of my work, but also for teaching me how scientific research should be conducted rigorously and how research results can be written up precisely and concisely, which would invaluably benefit my future career. As more than an adviser to me, he has also been supportive in my personal life, generously offering as much help as he could.

I'm indebted to many people for their intelligent and thoughtful discussions with me that greatly helped to shape my research ideas and broaden my knowledge and skills in research. Among them are Husain Aljazzar, Ulrik Brandes, Thierry Jéron, Johannes Leitner, Richard Mayr, Alin Ştefănescu, and Evghenia Stegantova. I am also grateful to Daniel Butnaru and Quang Minh Bui for their assistance in the building of prototype tools.

I thank George Avrunin for his willingness to review the thesis.

I thank Bran Selic for providing us with the PBX model, which enabled us to evaluate our work on a real life system model. I also thank IBM for making Rational Rose tools available free of charge.

I want to show my gratitude to all my colleagues and friends in Freiburg and Konstanz for their warmest friendship, countless help and constant encouragement.

Finally, I must pay my greatest respect and appreciation to my parents who, as researchers themselves, are the very first reason for my initiative to achieve a doctoral degree. I also thank my wife whose love and understanding accompanied me through all the joyful as well as all the hard times in these five years.

Contents

Chapter 1

Introduction

A large number of distributed systems have their components physically distributed among different computers. The physical separation of components makes it impossible for components to share memory. As a consequence, the only possible way of inter-component communication is through message sending and receiving over computer networks. Examples of such systems include communications systems, embedded software systems, and Internet-based systems, among many other types of systems that we are using on a daily basis. These systems are mostly used to continuously serve requests from their environment rather than deliver some final computation results and terminate. We call such systems *asynchronous reactive systems*.

For asynchronous reactive systems, such as communication protocols, a correct design is vital to providing services of high quality. Such systems can also be safety-sensitive since their failure may cause tremendous inconvenience, high costs, and a potential indirect endangerment of human life. To illustrate this point, one may consider the situation in which one needs to reach emergency services by making phone calls in case of accidents or illnesses, but the local phone switch fails. Therefore, it is important to guarantee that these systems are correctly designed.

For an asynchronous reactive system to be correct, it is not enough to only assure that no runtime errors, such as stack overflows and null pointer dereferences, will ever occur. It is equally important to guarantee that the system delivers its intended functionalities. In other words, it needs to be assured that the implementation meets the specification. Hence, we must consider properties concerning the high level behavior of the whole system, which comprises the interaction between different system components as well as the interaction between the system and the environment. One example of such high-level properties is that the system will always respond to requests from its users. This thesis addresses the issue of checking specific high-level properties for asynchronous reactive systems.

It is extremely challenging to check properties for asynchronous reactive systems for the following two reasons. First, an asynchronous reactive system is usually very large and complex, which impedes the application of traditional verification techniques such as model checking. Model checking approaches suffer from the notorious *state explosion* problem. They are in practice often incapable of handling within reasonable space and time limits the enormous state spaces

that asynchronous reactive systems often possess. Therefore, they may need manual or automated abstractions to reduce a large state space to a manageable size. A manual abstraction procedure usually takes tremendous effort and is prone to errors. An automated abstraction technique employs a common abstraction strategy for all models and properties, and may fail to find sufficiently precise abstractions for specific models and properties. Second, asynchronous message-passing-based communication may allow the use of unbounded message buffers to store incoming messages that have yet to be processed, which may result in an infinite state space on which model checking methods become intractable. In fact, the checking problems of many interesting high-level properties are undecidable for asynchronous reactive systems [26]. Consequently, any method to check these properties is inevitably incomplete.

The goal of the work described in this thesis is to provide an efficient way of checking specific high-level properties for asynchronous reactive systems. Our approaches bear a certain degree of similarity to the verification framework proposed in [38] in that we encode the verification problem into an *Integer Linear Programming* (ILP) problem. This strategy avoids the exhaustive exploration of the vast or infinite state space of a system, and results in efficient property checking. However, the ILP encoding approach captures only certain aspects of an original system, and sometimes leaves us with a coarse abstraction from which no definite conclusion can be drawn. In order to address this imprecision problem, we also consider the refinement of abstractions while maintaining the efficiency of the whole verification procedure.

In the remainder of this chapter, we formally define asynchronous reactive systems and argue why it is difficult to verify these systems. At the end of this chapter, we list the main contributions and give the structure of this thesis.

1.1 Asynchronous Reactive Systems

Within the context of this thesis, a *system* is a collection of autonomous components (also called *processes*) collaborating with each other to accomplish jointly a set of tasks. Moreover, we consider only discrete state systems or the discrete state part of a system, i.e., the execution of such systems or parts can be described as a sequence of discrete states. In particular, many real-time systems are hybrid systems combining both a continuous timed part and a discrete control part. In our analysis we are only concerned with the behavior of the control part of a real-time system and abstract away the timed part.

An *asynchronous reactive system* is a system satisfying the following conditions:

- **Asynchronous inter-component communication**: The components of the system communicate with each other mainly by exchanging messages over message buffers in an asynchronous way. Two communicating components have at least one communication channel available to transmit messages. A sender component continues its local execution after sending a message over the channel. The message will be inserted into a message buffer at the receiver side. The message stays in the buffer until it is removed from the buffer and consumed by the receiver. Various policies can be used to decide the order in which the receiver consumes messages in a buffer.

- **Reactivity**: The goal of the system is not to eventually terminate and deliver some computational results. Instead, it maintains an ongoing activity to respond to the stimuli from the environment of the system.

There are many modeling and programming languages as well as CASE[1] tools that support the specification and construction of asynchronous reactive systems. Examples of modeling languages include Promela [68], UML RT [106], SDL [101], and Statecharts [65], among others. High-level programming languages are also extended with APIs and middle-ware supports to make it convenient to implement the paradigm of asynchronous message passing, such as Java Message Service (JMS) [4] and Visual C++ .NET [6].

The ever increasing complexity of asynchronous reactive systems makes it ultimately important to guarantee that they are correctly designed according to their specification. However, as we show in the following, it is a very difficult task to establish correctness for an asynchronous reactive system.

First, an asynchronous reactive system usually consists of a large number of components. Since a global state of the system must comprise the local state information of each system component, the number of global states is exponential in the number of components. Consequently, the system may possess a very large global state space that is hard to exhaustively explore for the verification of properties. Moreover, components can be dynamically created and destroyed in many systems, and the number of components in a system can grow unbounded. This may result in potentially infinite state systems for which many interesting verification problems become even undecidable.

Second, the use of communication buffers adds another level of complexity to asynchronous reactive systems. Because a global state of a system must also comprise the contents of all message buffers in the system, the number of global states is doubly exponential both in the number of message buffers and in the lengths of buffers if all are a priori bounded. Moreover, UML RT permits unbounded message buffers, also resulting in potentially infinite state systems.

Last, modeling languages like Promela and UML RT allow for the use of variables of data types with a large or even infinite domain, which constitutes another source of complexity. In particular, UML RT models built using the Rational Rose RealTime tool may have transition action code written in high-level programming languages such as Java and C++. In this case, a global state must also contain the runtime stack and heap information.

1.2 The Contributions of the Thesis

The main contributions of this thesis are listed as below:

- We show a great potential of developing highly scalable incomplete verification methods for large and infinite state systems, using problem-specific abstraction techniques. We propose a verification framework for asynchronous reactive systems based on Integer Linear Program solving, and apply the framework to check two important concrete properties, namely buffer boundedness and livelock freedom. The checking methods take advantage of the abstraction techniques focusing on the cyclic message

[1] CASE stands for Computer-Aided Software Engineering.

passing behavior of a system, which correspond to the two main characteristics of asynchronous reactive systems, i.e., asynchronous message passing and reactivity.

- We show how the precision of an incomplete verification method can be greatly improved by an automated abstraction refinement procedure. For our verification framework, we develop a counterexample driven abstraction refinement procedure based on the discovery of certain dependencies among control flow cycle executions.

- We show that the termination of program loops can be proved without explicitly constructing or synthesizing linear ranking functions. This is illustrated by a fully automated termination proving method that we propose for an important class of program loops.

A major portion of the work described in this thesis has been published in [83, 82, 84, 85, 81]. The buffer boundedness analysis as published in the research papers [83, 82] results from initial discussions and joint work with Richard Mayr who also provided the complexity results as described in Section 5.7.1. The paper [81] results from joint work with Alin Ştefănescu.

1.3 The Structure of the Thesis

- Chapter 2 discusses related work in the area of system verification.

- Chapter 3 introduces the notions, concepts, and formalisms used throughout the thesis. In particular, several languages for modeling asynchronous reactive systems are presented as well as ILP problems.

- Chapter 4 gives an overview of the verification framework that we propose.

- Chapter 5 and Chapter 6 explain with technical details how to check the buffer boundedness and livelock freedom properties respectively within the framework. Both checking methods are presented at the level of Communicating Finite State Machines (CFSMs), a common abstraction level for all high-level modeling languages.

- Chapter 7 and Chapter 8 discuss respectively the specific strategies and techniques to abstract Promela and UML RT, two high-level modeling languages, into CFSMs. Chapter 9 is dedicated to a special abstraction problem for UML RT models: how to abstract from program loops inside transition action code that can be written in a programming language.

- Chapters 10 – 12 are concerned with abstraction refinement.

- Chapter 13 presents some specific issues that we have encountered during the implementation of our verification methods.

- Chapter 14 reports experimental results on some representative case studies.

- Chapter 15 concludes the thesis and suggests future work.

Chapter 2

Related Work

In this section we mainly discuss related work in the area of formal system verification. In subsequent chapters, in which we explain the technical details of our work, we will also discuss existing work related to the respective techniques and methods that we propose.

2.1 Formal Verification

There are two main approaches to the formal verification of software systems, namely theorem proving [22] and model checking [34, 68, 91].

In a theorem proving approach, a logical framework is chosen to express the behavior of the software system as axioms, and the considered properties as formulas. In order to validate the satisfaction of the properties by the system, a proof that the axioms representing the system behavior entail the property formulas is derived with assistance of automated theorem provers [7, 1]. Such an approach often requires human intervention and comprehensive knowledge of both the software system under scrutiny and the theorem prover being used.

Explicit state model checking techniques [34, 68] rely on the fully automated exploration of the whole reachable global state space of a system in search for property violating behavior. Therefore, no human intervention is needed, which means that model checking is easier to be integrated into a software development process than theorem proving. However, such an exhaustive verification approach poses the well-known state explosion problem. A model checker may run out of resources when applied to systems that possess too huge a state space, such as an asynchronous reactive system usually does. Moreover, model checking techniques become incomplete when applied to infinite state systems.

Symbolic model checking [91] was proposed as a solution to the state explosion problem. In symbolic model checking, the state space of a system is not explicitly represented as a state transition graph. Instead, states and transitions are encoded into logical formulas. The verification problems can be then reduced to the computation of greatest or least fixed points of logical formulas. It has been observed in practice that symbolic model checking techniques are more suitable for hardware verification tasks, while explicit state model checking outperforms symbolic model checking in software verification [47, 68]. While there has been no formal argument for an explanation of this observation, we

conjecture the reason to be the typical irregularities of the control structure of software systems, which cannot be effectively encoded and handled in the form of logical formulas used in symbolic model checking. On the other hand, partial order reduction [97] and other automated abstraction techniques used in explicit state model checking seem to be very efficient in reducing the size of irregularly structured software system models.

Verification techniques based on Integer Linear Program (ILP) solving have been proposed in [39, 49]. In these techniques, the control flow information of a system is over-approximated by a set of linear integer equations called state equations. One such equation can either (1) equate the incoming control flow of a control point with its outgoing control flow; or (2) represent the synchronized communication of two components of the system. The state equation based approach is mainly used for the verification of synchronous systems such as Ada programs and for the checking of reachability properties. It is claimed to also apply to asynchronous systems, which is however not elaborated [38]. Furthermore, this approach generates non-linear inequalities when checking liveness properties such as livelock freedom. One solution is to transform non-linear inequalities to linear ones by restricting the number of computation steps of a system. This however can only prove the satisfaction of the property within a certain finite number of computation steps [39].

2.2 Formal Verification of Infinite State Systems

An asynchronous reactive system may be an infinite state system. The verification problems of various infinite state system modeling formalisms have been studied, including Communicating Finite State Machines (CFSMs) [26, 98, 69], counter machines [71], and Petri nets[1] [50], among others [11, 20]. Many interesting problems, such as reachability problems, are undecidable for all above mentioned formalisms. The boundedness property that we will address in the thesis, however, is undecidable for CFSMs [26] and for counter machines [54] but decidable for Petri nets [50]. Many problems are decidable only for subclasses of these formalisms [53, 54]. As an example, the reachability problem is decidable for infinite state systems with unreliable message channels [12, 21]. This is because the inclusion of lossy behavior allows the set of all reachable system configurations to be represented as a regular language, for which reachability is decidable. Nevertheless, in order to arrive at more general verification approaches treating the whole classes of these formalisms, one has to resort to abstraction techniques as explained in the next section.

2.3 Abstraction and State Space Reduction

In Section 1.1 we gave three levels of difficulties in the verification of asynchronous reactive systems. These difficulties correspond to three sources of infiniteness: unbounded process instantiations, unbounded message buffers, and unbounded data domains. There are some existing techniques addressing these problems as discussed in the following.

[1] Petri nets can be regarded as a subclass of counter machines.

With regard to unbounded process instantiations, we have the following observation in realistic systems. Even though a system may contain a large number of components, there are not many different classes of components in the system. Instances of each component class exhibit identical or similar behavior. Therefore, there is a great potential to exploit symmetry-based state space reduction techniques [72].

Addressing unbounded message buffers, [44] proposes an abstraction method to eliminate Fist-In-First-Out (FIFO) message buffers in a system. Their observation is that the behavior of a state machine is only influenced by those messages that are consumed at runtime. The information of those never consumed messages is useless in the verification of the system. Based on this observation, they partially evaluate the effects of message passing events, and encode the effects using an evolving set of pending transitions that can be potentially triggered by the considered message receiving events. This partial evaluation results in an over-approximation of the original system, which may include behavior that the original system does not permit. Moreover, this abstraction technique applies only to FIFO message buffers.

Variables, pointers, and other high-level programming language features can be similarly treated based on the ideas of abstract interpretation, predicate abstraction, and shape analysis as used in program verification. Abstract interpretation [42] abstracts concrete data type domains into abstract domains of much smaller sizes, and computes the fixed point of the computation of the program on these abstract domains. Predicate abstraction [18] uses a set of boolean predicates to abstract away variables in a program. These predicates may represent numerical relations among variables. The boolean values of these predicates constitute abstract global states of the system which form a much smaller state space than the concrete state space. Shape analysis [118] can be used to abstract from pointers and aliases, and thereby check properties of heap operations and structures. These abstraction techniques apply mainly to sequential programs whose state spaces are far smaller than the state spaces of concurrent systems. Furthermore, these techniques are used mainly for the checking of runtime errors as debugging tools rather than verification tools.

2.4 Abstraction Refinement

The use of abstraction techniques results in over-approximations containing spurious behavior of original systems. Sometimes a coarse abstraction is even deliberately computed for efficiency reasons. These lead to imprecise verification and can be remedied only by refining the abstraction. The idea of automated counterexample guided abstraction refinement has been broadly adopted in system verification and especially in model checking methods [33, 17, 32]. Such a refinement approach first determines spuriousness for the counterexamples generated by the verification method. Then, the spurious behavior represented by these counterexamples is excluded by adding more constraints to the system behavior. For the ILP-based verification framework in [38], an abstraction refinement procedure is proposed in [108] to exclude unrealistic control flow by enforcing event orders and dependencies between acyclic paths and control flow cycles.

2.5 Summary

In this thesis we investigate the problem of checking properties for asynchronous reactive systems. In particular, we mainly consider the two concrete high-level properties that any correct asynchronous reactive system must satisfy: buffer boundedness and livelock freedom. Both properties describe conditions on the infinite executions of a system and are undecidable for infinite state systems, as we will see later. Based on the previous discussions in this chapter, we can show that the existing verification and abstraction techniques are insufficient to check these two properties.

- First, many asynchronous reactive systems possess huge, if finite at all, state spaces, which makes it hard to be explored completely by state enumeration based verification methods such as model checking. Even when using abstractions the verification of these systems using traditional model checking techniques is still a major challenge.
- Second, the verification methods other than model checking are also not able to check efficiently the properties that we consider for the class of asynchronous reactive systems. For instance, properties of infinite executions cannot be efficiently checked using the verification technique proposed in [39].
- Third, the existing common-purpose abstraction and state space reduction techniques may be very inefficient for the verification of the above mentioned two properties for asynchronous reactive systems. Especially, it remains an issue how to efficiently abstract from the contents of message buffers.
- Finally, while abstraction refinement has been well studied in the context of traditional model checking, we must devise an effective refinement procedure for the verification methods that we propose.

The objective of this thesis is to develop both scalable and precise verification methods for the checking of the two above properties, making use of tailored abstraction techniques and effective abstraction refinement procedures.

Chapter 3

Preliminaries

This chapter presents a brief introduction to some of the concepts, formalisms, modeling languages, and theories that we will use throughout the thesis.

3.1 Notions and Notations

Let D be an arbitrary domain. We denote a *sequence* over D by $\langle d_1, \ldots, d_n, \ldots \rangle$ where each $d_i \in D$, in which d_1 is the *head* of the sequence. A sequence can be finite or infinite. Let q_1 and q_2 be two sequences. We denote by $q_1.q_2$ the *concatenation* of the two sequences. For any subset $E \subseteq D$, we denote by E^* the set of all finite sequences whose elements are in E.

We use overlined version of variables $\bar{a}, \bar{x}, \bar{v}$ to denote rational-valued or integer vectors. By $\bar{x}^i$ we denote the i-th component of the vector $\bar{x}$. A vector $\bar{x}$ is *non-negative* if all its components are non-negative, i.e., $\bar{x}^i \geq 0$ for each i. A vector $\bar{x}$ is *positive* if it is non-negative and there exists at least one component $\bar{x}^j > 0$. As examples, $(0, 0, 0)$ is non-negative, $(2, 0, 0)$ is positive, and $(2, 0, -1)$ is neither non-negative nor positive.

3.2 Decision Procedures

For a certain property, it is a decision problem whether a system satisfies the property or not. An algorithm or a method to check this decision problem is a decision procedure.

A *decision problem* P is a yes-or-no problem on an infinite number of input instances [66]. A *decision procedure* for P is an algorithm to determine an answer to P on an arbitrary input. A decision procedure is *sound* if on any input it either terminates and delivers a correct answer or never terminates. A decision procedure is *complete* if and only if it always terminates on any input and delivers the correct answer. A decision procedure is *semi-complete* if it always terminates and delivers the correct answer on any input with a "yes" answer. A decision procedure is *incomplete* if it terminates and delivers correct answers on some inputs. We can easily see that a semi-complete decision procedure is incomplete but not vice versa. A decision problem is *decidable* if there exists a complete decision procedure for the problem. It is *undecidable* if no complete

decision procedure exists for the problem. It is *semi-decidable* if there exists a semi-complete but no complete decision procedure for the problem.

The checking problems of the properties that we address in this thesis are all undecidable. Consequently, any algorithm to check these properties is incomplete or, at best, semi-complete. According to the above definitions, if the algorithm is incomplete, then it may not terminate on some asynchronous reactive systems. However, any practically used algorithm is expected to terminate on any input instance. Therefore, we modify the above definitions of soundness and incompleteness by introducing a third kind of answers besides "yes" and "no": an inconclusive answer "unknown". We define that a correct answer must be a conclusive answer. A decision procedures is *sound* if it delivers either a correct answer or an inconclusive answer on any input. A decision procedure is *incomplete* if it delivers correct answers on some inputs and inconclusive answers on the other inputs.

3.3 Promela

In this thesis we will illustrate the application of our proposed verification methods to the two modeling languages Promela and UML RT. Both languages have rich and powerful features for the modeling of asynchronous reactive systems, and are widely used and studied both in industry and in academia. In particular, a great number of Promela models are available in public [9, 3, 2]. In this section we briefly introduce Promela. The UML RT language will be introduced in Section 3.4.

Promela is the input language of the *SPIN* explicit state model checker [68]. It has been successfully used for the modeling and analysis of many concurrent systems [74, 47, 112]. The operational semantics of Promela has been studied and formalized in [19, 46].

A Promela model consists of a set of `proctype` definitions. Each proctype represents a class of concurrent processes whose common behavior is specified by the sequential Promela code in the proctype definition. Processes, i.e., instances of proctypes can be created dynamically. Proctypes can be parameterized. There are three ways of inter-process communication: (1) communication through shared global variables; (2) synchronous rendezvous communication; and (3) asynchronous communication. The last two kinds of communication are achieved by message sending and receiving statements on communication buffers declared as `chan` variables. Each buffer can be used to exchange only a certain type of messages which is defined in its declaration. Moreover, each buffer has a predefined capacity n such that it can store no more than n messages at runtime. If the capacity of a buffer is 0, then the communication over this buffer is synchronous – the sending and receiving of a message must be synchronized. When the capacity is greater than 0, the communication over this buffer is asynchronous – the sender is blocked only if the buffer is full; the receiver is blocked only if the buffer is empty or it expects a message not available in the buffer. The above actually states the executability of message sending and receiving statements. Every statement in Promela has an executability. An assignment is always executable. A conditional statement is executable if the boolean condition in the statement evaluates to true.

Listing 3.1 shows an example Promela model. There are two types of pro-

```
1  mtype = {choice, result};
2
3  chan fromPlayer[2] = [1] of {mtype, bit};
4  chan toPlayer[2] = [1] of {mtype, int};
5
6  proctype player(int id){
7    int gain;
8    int sum = 5;
9
10   do
11   :: sum >= 0 ->
12      if
13      :: fromPlayer[id]!choice(1);
14      :: fromPlayer[id]!choice(0);
15      fi;
16
17      toPlayer[id]?result(gain) ->
18      sum = sum + gain;
19
20   :: sum < 0 ->
21      break;
22   od
23 }
24
25 active proctype judge(){
26   bit b1, b2;
27   int r1, r2;
28
29   do
30   :: fromPlayer[0]?choice(b1);
31      fromPlayer[1]?choice(b2);
32
33      if
34      :: b1 == 1 && b2 == 1 ->
35         r1 = -1; r2 = -1;
36      :: b1 == 0 && b2 == 1 ->
37         r1 = 1; r2 = -1;
38      :: b1 == 1 && b2 == 0 ->
39         r1 = -1; r2 = 1;
40      :: b1 == 0 && b2 == 0 ->
41         r1 = -2; r2 = -2;
42      fi;
43
44      toPlayer[0]!result(r1);
45      toPlayer[1]!result(r2);
46   od
47 }
48
49 init{
50   run player(0); run player(1);
51 }
```

Listing 3.1: A Promela model.

cesses `player` and `judge`. The proctype `judge` has a keyword `active` in its declaration. For such a proctype, one instance of the proctype is automatically created when the model starts to run. Instances of proctypes such as `player` must be created explicitly using `run` statements (Line 50). Such process instantiation tasks are usually carried out by a special process named `init`. Moreover, the proctype `player` has a formal parameter `id` as an integer. The two instances of `player` are created with different actual parameters 0 and 1 in the model. There are four message buffers in the model, declared as two buffer arrays `fromPlayer` and `toPlayer` that each contain two buffers. The instance `player(id)` uses the buffers `fromPlayer[id]` and `toPlayer[id]` to communicate with `judge`. The capacity of each buffer is defined within the square bracket after the "`=`" symbol in the corresponding declaration. Each buffer declaration also defines the format of messages exchanged in the buffer within the curly bracket. As an example, any message sent to a buffer of `fromPlayer` must contain two fields. The first field is of the type `mtype` and the second field is of the type `bit`. The special type `mtype` contains user defined constant symbols that can be used to indicate the type of messages.

The above Promela model describes a game in which two players independently choose between 0 or 1 in each round (Line 13 and Line 14), and send the chosen bit to the judge (Line 17). The judge decides how many points each player can gain depending on both of their choices (Line 33 – Line 42). The only chance for a player to get a positive number of points is to choose 0 in hope that another player would choose 1 (Line 36 – Line 39). This is risky because if both have chosen 0 then both will even lose more points (Line 40 and Line 41). Each player starts with 5 points, and the game ends when one player runs out of points. Repeated rounds correspond to `do` loops in the code (Line 10 and Line 29). Inside a `do` loop there might be several branches. A branch can be taken in one iteration of the loop if the first statement in the branch is executable. If multiple branches can be taken, then one of them will be taken and which one to take is completely nondeterministic. The same branch selection method applies also to the branching statement `if`. We can see that the two branches at Line 13 and Line 14 can be taken when the buffer `fromPlayer[id]` is not full. This corresponds to the fact that each player can freely choose a bit without the influence of another player.

Promela models are simulated and validated by the SPIN model checker that can only verify properties for finite state systems. Therefore, the Promela language has several syntax restrictions to guarantee the finiteness of a model. For instance, any data type has a finite domain so that a variable of the type can only have a finite number of runtime values. Furthermore, each message buffer has an a priori fixed capacity such that no buffer can contain an unbounded number of messages at runtime. However, as we are interested in infinite state system for which many interesting problems are undecidable, we disregard all these syntactic restrictions, e.g., to assume that all buffers have an unbounded capacity. In this way, a buffer may contain an unbounded number of messages, the model may therefore possess an infinite number of states.

3.4 UML RT

UML RT has its root in the graphical modeling language ROOM [106]. ROOM has later been reconciled with the UML standard to form a UML compatible language for the modeling of real-time systems [107]. Modeling in UML RT is supported by the CASE tool *Rational Rose RealTime* that has evolved into the *Rational Rose Technical Developer* tool and currently is a part of the IBM/Rational software development product line [8]. In this thesis all example UML RT models will take the graphic notations of modeling elements as used in Rational Rose RealTime.

UML RT permits the description of the communication structure and the dynamic behavior of the systems. A system is decomposed into a set of concurrent components called *capsules*. Capsules can be decomposed hierarchically into sub-capsules. The communication interfaces of the capsules are called *ports*. Each port is associated with a *protocol* that stipulates which types of messages can be sent through the port and which types of messages can be received from the port. Ports can be associated with each other using bidirectional *connectors* - the presence of a connector between ports indicates a communication channel. Connectors can be built either at compile time or dynamically through program control. Note that connectors are not places to store incoming messages but merely transmission media. UML RT models employ a set of FIFO message buffers at runtime to store exchanged messages that have not been delivered to the proper port. Any port is associated with a runtime buffer to receive messages from. However, it completely depends on the actual scheduling mechanism used in the runtime environment to determine which port is associated with which buffer and when a message can be delivered to a certain port. As a proper abstraction of all possible scheduling mechanisms, we assume in this thesis that every port is assigned a separate message buffer and the port can receive a message from the assigned buffer as long as the message is available – when it is on the top of the buffer. Unlike in the Promela language, inter-capsule communication in UML RT is exclusively by asynchronous message passing, i.e., no shared variable communication is defined in UML RT.

The behavior of each capsule is described using a communicating, extended, hierarchical finite state machine. These state machines are derived from ROOM-Charts [106] which, in turn, are a variant of Statecharts [65]. However, as opposed to Statecharts, the state machines in UML RT are strictly sequential, i.e., the orthogonality concept of Statecharts is absent in UML RT. The operational semantics of UML RT is characterized by two key features:

- First, transitions in state machines can only be triggered by message reception, i.e., there are no spontaneous transitions. The only exception is that every state machine has an initial state whose unique outgoing transition does not need to consume any message. It is fired automatically when the state machine starts to execute.

- Second, each transition is associated with a piece of action code. In the course of the transition, the action code is executed. It may include one or more message sending events. Transition action code is not constrained by the UML RT definition, and its specification depends on concrete implementations of UML RT. The Rational Rose RealTime tool allows action code to be written in C++ or Java.

More details of the hierarchical structures of UML RT state machines are mentioned in Section 8.2.

Consider the UML RT model of the Alternating Bit Protocol as shown in Figure 3.1. The model consists of three classes of capsules: `System`, `Sender`, and `Receiver`. The capsule `System` is the topmost level capsule as a container of all other capsule instances. Its structural graph is shown in Figure 3.2 in which there is an instance of the capsule class `Sender` and an instance of `Receiver`. Their communication structure is also defined: The instance `sender` has a port named `sp` whose protocol is the `AB_Protocol` shown in Figure 3.1. The protocol has a set of incoming message types as {`ACK(int)`} and a set of outgoing message types as {`BIT(int)`}. It stipulates that any message received by `sender` from the port `sp` must have an identifier `ACK` and carry an integer number. Any message that `sender` sends through the port `sp` must have an identifier `BIT` and carry an integer number. The identifier of each message is also called a *message signal*. Furthermore, the instance `receiver` has a port named `rp` whose protocol is the conjugated version of `AB_Protocol` – the set of incoming message types and the set of outgoing message types of `AB_Protocol` are reversed. The rectangle that represents the port `rp` is hollow, which indicates that the protocol associated with `rp` is conjugated. The two ports `sp` and `rp` are connected. Any message sent by `sender` through `sp` will be transmitted to the message buffer associated with the port `rp`, which will be then delivered to `rp` to trigger some transition in the state machine of `receiver`.

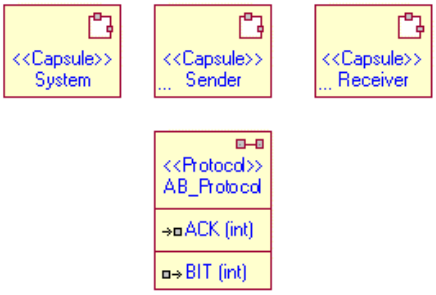

Figure 3.1: A UML RT model of the Alternating Bit Protocol.

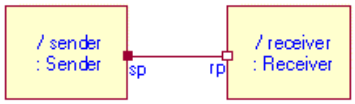

Figure 3.2: The structural diagram of the `System` capsule in the UML RT model in Figure 3.1.

The state machines of `Sender` and `Receiver` are shown on the left and right respectively in Figure 3.3. Initial states are represented by solid red circles. In the state machine of `Sender`, the transition `init` sends a message `BIT(1)` through the port `sp`, and the control then waits at the state `bit1` until a message `ACK(1)` is received from `sp`. The message `ACK(1)` is an acknowledgment from `Receiver` of the previously sent message `BIT(1)`. The arrival of a message

`ACK(1)` acts as the trigger of the transition `send0` along which `Sender` sends a message `BIT(0)` and reaches the state `bit0`. At the state `bit0`, an acknowledgment message `ACK(0)` must be received to enable the transition `send1` along which a message `BIT(1)` is sent. In this way, the sender alternatively sends bits 1 or 0, and it sends the next bit only after an acknowledgment is received for a previously sent bit. In the state machine of `Receiver`, the transition `donothing` has no action to execute. The receiver sends back an `ACK(1)` after receiving a `BIT(1)` (the transition `ack1`), and replies with an `ACK(0)` while getting a `BIT(0)` (the transition `ack0`).

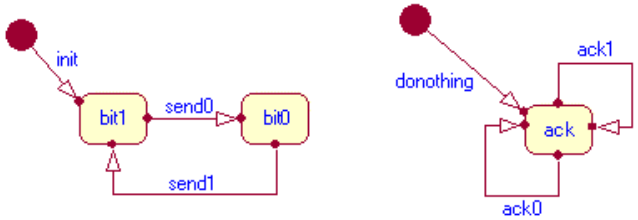

Figure 3.3: The state machines in the UML RT model in Figure 3.1: the state machine of `Sender` on the left and the state machine of `Receiver` on the right. The text attached to a transition is the name of the transition and indicates the message sending activities taken along the transition. A transition may also receive a message, as the trigger of the transition, which is left out in the name of the transition.

At the time of writing, there is no complete formal operational semantics for UML RT in the literature that we are aware of. Some guidance regarding the interpretation of UML RT models can be derived from the Rational Rose RealTime tool, in particular from its simulation capabilities, from the code of its runtime system and, in the worst case, from the code synthesized by the code generation functionality of Rational Rose RealTime. Many of the semantic features are implementation dependent. However, it turns out that the precise definition of a formal semantics is not a prerequisite for the work pursued here. We will present an approach that is taking advantage of a significant amount of abstraction in preparation of the analysis. These abstractions amount to an over-approximation of the actual system behavior so that subtle issues in the UML RT semantics, such as the ordering of message events and the treatment of message priorities, are not meaningful in the abstract system and hence for our analysis.

3.5 Communicating Finite State Machines

Communicating Finite State Machines (CFSMs) [26] are natural models for asynchronous reactive systems. Many modeling languages can be seen as being based on CFSMs, including Statecharts [65], Estelle [110], SDL [101], Promela [68], and UML-RT [106], among others. Therefore, we choose CFSMs as the initial abstract formalism for all the modeling languages to which our verification framework applies.

Intuitively, a CFSM system consists of a set of concurrently running processes. The behavior of each process is captured by a state machine. The

concurrency of processes is captured at the semantics level of CFSMs by the interleaving of processes executions. Processes exchange messages between each other asynchronously over a set of message buffers. Message buffers are interpreted at the semantic level as unbounded FIFO message queues. A sender process continues its local execution after sending a message to a buffer, and a receiver process is blocked when it tries to receive a message that is not available in the respective buffer.

We formally define the syntax and semantics of CFSMs for self-containedness. We change some parts of the definition in [26] for the convenient abstraction of the modeling languages that we consider in the thesis.

Definition 3.1 (Communicating Finite State Machines)**.** A system of *communicating finite state machines* (CFSM) is a quadruple

$$(P, M, B, succ)$$

where

- P is a finite set of *processes*. Each process p_i is a pair (S_i, s_0^i) where S_i is a finite set of *states* of p_i and $s_0^i \in S_i$ is the initial state. For any two different processes p_i and p_j, we put the restriction that $S_i \cap S_j = \emptyset$, i.e., their sets of states are disjoint.
- M is a finite set of *message symbols*.
- B is a finite set of *message buffers*. Each buffer is associated with a subset of message symbols $M' \subseteq M$ such that only the messages in M' can be exchanged in the buffer. Moreover, for each buffer $b \in B$ and each message symbol m in the subset of M associated with b, we call (b, m) a *message type*.
- *succ* is a finite set of *local transitions* (s, e, s') where s and s' are states of some same process p_i, and e is either empty or a message passing event in the form $b!m$ or $b?m$ such that (1) $b \in B$ and (2) $m \in M$ can be exchanged in the buffer b.

Example 3.1. One simple example of CFSMs is shown in Figure 3.4, which consists of two processes named `client` and `server`.

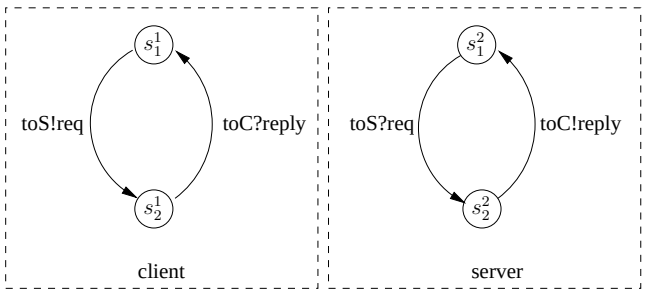

Figure 3.4: A simple CFSM system

Semantics. The semantics of CFSMs is defined using the concepts of configurations and reachability.

Definition 3.2 (Configuration)**.** Given a CFSM system $(P, M, B, succ)$, a *configuration* (or *global state*) of the system is a tuple $(s^1, \ldots, s^{|P|}, q^1, \ldots, q^{|B|})$ such that

- each s^i is a state of the process p_i, and
- each q^i is a queue of messages exchangeable in the buffer b_i.

A configuration of a CFSM system does not only contain the information of which local state each process is currently at, but also the information of the content of each message buffer denoted as a message queue. Since message buffers have unbounded capacities, there may be an infinite number of configurations of the system. The initial configuration c_0 is where each process is at its initial state and all message buffers contain an empty queue.

We define an auxiliary function *upd* to update the contents of message buffers according to the message passing event of a transition. We need the *upd* function in the definition of reachability. Intuitively, the function *upd* restricts message queues to be first-in-first-out: When a new message is sent to a buffer, the message is attached to the end of the queue. Moreover, only the top message in a buffer can be received. Given a configuration $(s^1, \ldots, s^{|P|}, q^1, \ldots, q^{|B|})$ and assume that the buffer b_i corresponds to the message queue q^i, we define

$$upd(q^i, e) = \begin{cases} q^i.m, & \text{if } e \text{ is } b_i!m; \\ q \text{ where } q^i = m.q, & \text{if } e \text{ is } b_i?m; \\ q^i, & \text{otherwise.} \end{cases}$$

and

$$upd(q^1, \ldots, q^{|B|}, e) = (upd(q^1, e), \ldots, upd(q^{|B|}, e)).$$

Consider two configurations c_1 and c_2. Let $c_1 = (s_1^1, \ldots, s_1^{|P|}, q_1^1, \ldots, q_1^{|B|})$ and $c_2 = (s_2^1, \ldots, s_2^{|P|}, q_2^1, \ldots, q_2^{|B|})$. We define that c_2 is a *successor* of c_1, denoted by $c_1 \Rightarrow c_2$, if the following is satisfied:

- There exists a process $p_i \in P$ such that (1) for all $j \neq i$ we have that $s_1^j = s_2^j$; and (2) $(s_1^i, e, s_2^i) \in succ$.
- $upd(q_1^1, \ldots, q_1^{|B|}, e) = (q_2^1, \ldots, q_2^{|B|})$.

Let $\Rightarrow^+$ be the reflexive and transitive closure of $\Rightarrow$. A configuration c is *reachable* from another configuration c' if $c' \Rightarrow^+ c$. In particular, c is defined to be a *reachable* configuration if it is reachable from the initial configuration c_0. An *execution* of the CFSM system is a sequence of configurations $\langle c_0, \ldots, c_i, \ldots \rangle$ such that $c_i \Rightarrow c_{i+1}$ for each c_i in the sequence. An execution can be finite or infinite. The relation $\Rightarrow^+$ essentially defines a reachability graph for a CFSM system, and every finite or infinite path from the initial configuration in the graph is a possible execution of the system. Figure 3.5 shows the reachability graph of the previous simple client-server example in Figure 3.4. While this example system has a finite reachability graph and contains only 4 configurations,

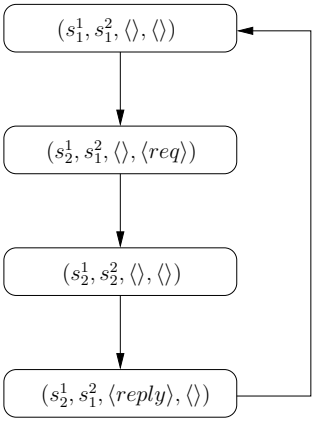

Figure 3.5: The reachability graph of the CFSM system in Figure 3.4.

a CFSM system in general may have an exponential or even infinite number of reachable configurations due to unbounded message buffers.

Our definition of CFSMs deviates from the original definition given in [26] mainly in the definition of message buffers. According to the definition in [26], for each pair of different processes (p_i, p_j) there exists exactly one message buffer such that p_i is the sender of the buffer and p_j is the receiver of the buffer. On the contrary, our definition disassociates message buffers from processes. A buffer may therefore have multiple senders and receivers. A process can also send messages to or receive messages from multiple buffers. Such deviation enables a more straightforward abstraction from Promela models into CFSM systems since Promela may put no restrictions on which processes may use a certain buffer. Although it is rarely observed in realistic models that a process uses more than one buffer to store messages from one other process and that a buffer allows more than one sender and receiver, such modification comes at no cost and all theoretical results in [26] that we use in this thesis are still valid for our modified definition.

We can further modify the definition of CFSM systems by allowing a sequence of message passing events for a transition, with the restriction that the sequence can contain at most one message receiving event and the receiving event must be the head of the sequence. This modification results in a more compact abstraction of UML RT models in which the action code of a transition may contain a number of message sending events. The necessary modification of semantics is straightforward: just re-define the *upd* function so that it updates message queues appropriately with a sequence of message passing events in order to maintain the FIFO condition of message buffers.

For CFSM systems, most non-trivial problems are undecidable, including buffer boundedness and reachability among others. In particular, the following problem is proved to be undecidable in [26].

Problem 1. Executability of a message reception in a CFSM system.

Instance:	A CFSM system S and a state s in S having an outgoing transition t labeled by the message receiving event $b?m$
Question:	Does there exist an execution of S in which the message events $b?m$ occurs at s?

3.6 Integer Linear Programming

In this section we give a brief introduction to linear programming and integer linear programming. A thorough account can be found in [105]. We first introduce some preliminary concepts.

Let $x_1, \ldots, x_n$ be rational-valued variables. A *linear expression* over these variables is an expression $a_1x_1 + \cdots + a_nx_n + a_{n+1}$ where $a_1, \ldots, a_{n+1}$ are rational numbers, e.g., $2.5x_1 + 6x_3 - 0.7$. A *linear combination* of variables is an expression $a_1x_1 + \cdots + a_nx_n$ such as $2.5x_1 + 6x_3$. A *linear inequality* is in the form $lc > b$, or $lc \geq b$, or $lc < b$, or $lc \leq b$ where lc is a linear combination of variables and b is a rational number, e.g., $2.5x_1 + 6x_3 \geq 10$. A *variable valuation* is a mapping from variables to rational numbers $\sigma : x_1 = v_1, \ldots, x_n = v_n$. Given a linear expression e, we denote by $\sigma(e)$ the expression $e[x_i \mapsto v_i]$ obtained from e by simultaneously substituting every variable x_i with its corresponding value v_i in the valuation σ. A variable valuation σ satisfies a linear inequality $lc \prec b$ where $\prec \in \{>, \geq, <, \leq\}$ if $\sigma(lc) \prec b$ is true. For instance, let $\sigma_1 : x_1 = 2, x_2 = 1$ and $\sigma_2 : x_1 = -1, x_2 = 4.6$ and $\sigma_3 : x_1 = 1, x_2 = -1$. Both σ_1 and σ_2 satisfy $2.5x_1 + 6x_3 \geq 10$ but σ_3 does not. A linear programming (LP) problem is to maximize or minimize a linear expression under the constraint of a set of linear inequalities.

Definition 3.3 (Linear Programming Problem)**.** A *linear programming* problem is in the form

$$max \text{ (or } min) : a_1^0x_1 + \cdots + a_n^0x_n + a_{n+1}^0 \tag{3.1}$$

$$a_1^1x_1 + \cdots + a_n^1x_n + a_{n+1}^1 \prec b^1 \tag{3.2}$$

$$\cdots$$

$$a_1^mx_1 + \cdots + a_n^mx_n + a_{n+1}^m \prec b^m \tag{3.3}$$

where $x_1, \ldots, x_n$ are rational-valued variables, each of a_j^i and b^i is a rational number, and $\prec \in \{>, \geq, <, \leq\}$.

In the above definition, the linear expression (3.1) is the *objective function* of the LP problem. The objective function is optional. Linear inequalities (3.2–3.3) form the *constraint* of the LP problem. The *solution space* of the LP problem is the set of all variable valuations that satisfy all linear inequalities in the constraint. When the solution space is non-empty, the LP problem is *feasible*. Otherwise, it is *infeasible*. While the objective function exists and is in the form $max : e$, an *optimal solution* to the LP problem is a valuation σ in the solution space such that for any other valuation σ' in the solution space $\sigma(e) \geq \sigma'(e)$. In this case $\sigma(e)$ is the optimal value for the objective function. Optimal solutions and optimal objective function values can be similarly defined while the objective function is in the form $min : e$. For an LP problem, an optimal objective function value does not exist if the solution space is empty, i.e., the problem is infeasible. Moreover, the objective function value can be

unbounded within the solution space and an optimal value does not exist either in this case.

Example 3.2. Consider the following LP problem.

$$max : x + 4y \tag{3.4}$$
$$x + y \leq 10 \tag{3.5}$$
$$5x + 2y \geq 20 \tag{3.6}$$
$$-x + 2y \geq 0 \tag{3.7}$$

This LP problem is feasible. An optimal solution is $x = 0, y = 10$, and the optimal value of the objective function is 40.

LP problems can be solved in polynomial time [77]. The mostly used algorithm for solving LP problem is the simplex method [45], which works efficiently in practice and has however an exponential time worst-case complexity. Polynomial time LP solving algorithms include the ellipsoid method [77] and Karmarkar's algorithm based on interior point methods [75], among others.

In Definition 3.3, if we require all variables to be integer variables, the LP problem becomes an *integer linear programming* (ILP) problem. The solving of ILP problems is no longer in polynomial time but NP-complete [37]. ILP solving methods include the Branch-and-Bound algorithm [80] and Gomory's cutting plane method [58], among others.

Chapter 4

Overview

As an alternative approach to model checking, we propose an efficient verification framework for asynchronous reactive systems, which avoids the enumeration of the whole global state space of a system. The proposed verification framework is based on ILP solving, similar to the previously mentioned ILP-based verification in [38] but applied to a different setting.

The basic idea of our verification framework is as follows. We encode the behavior of a system and the necessary condition of the *negation* of the checked property into an ILP problem. The solution space of the ILP problem therefore represents the property violating behavior of the system. When there is no solution to the ILP problem, the satisfaction of the property is assured. Our verification approach is different from the work in [38] in the following aspects.

First, the abstraction procedure that we use in encoding the system into an ILP problem is not based on state equations that focus on control flows. Instead, our abstraction techniques are closely related to the two key characteristics of the class of systems that we consider: reactivity and asynchronous communication. (1) The abstraction procedure is centered around control flow cycles because of the fact that the execution of a reactive system amounts to the repetitions of control flow cycles in the components of the system. More precisely, the property checking ILP problem of a system will contain the information how control flow cycles in the system can be combined together in the execution of the system. Note that, although there may be an infinite number of control flow cycles in a system, the number of *elementary* cycles is always finite. Our analysis therefore concentrates on the study of the behavior of elementary cycles. (2) We pay special attention to the effects of asynchronous communication on message buffers. In particular, we study the combined message passing effects of control flow cycles and check how the behavior of the system is influenced.

Second, ILP problems constructed in our verification framework share a common property with which these ILP problems can be solved very efficiently in polynomial time even though ILP solving is NP-hard. This will be explained later when we come to checking a concrete property.

Our verification framework is illustrated in Figure 4.1. It consists of three main procedures as *abstraction*, *property checking*, and *refinement*. The aim of the abstraction procedure is to encode the checking of the considered property into an ILP problem. The concrete abstraction techniques used in this procedure should be specifically and individually designed with respect to different prop-

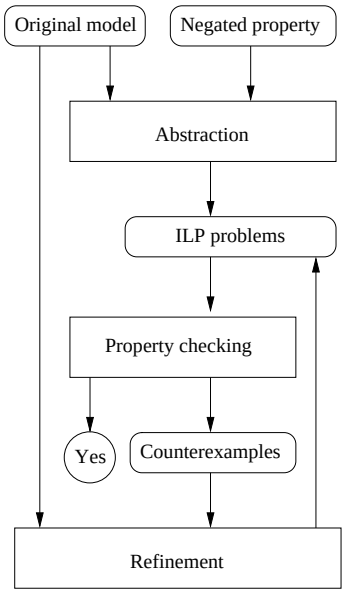

Figure 4.1: An ILP-based verification framework for asynchronous reactive systems.

erties to be checked. This is opposed to having a common abstraction approach for a general class of properties. We believe that there is a great potential for designing problem-specific abstraction strategies to make the future verification to be both more efficient and more precise. Moreover, the abstraction procedure is usually conservative, i.e., it keeps all possible behavior of the original system and may introduce some behavior that is not allowed in the original system. The imprecision is unavoidable for most properties as their checking problems are undecidable for infinite state systems. The imprecision is also necessary as a consequence of the trade-off between efficiency and accuracy of verification.

The property checking procedure mainly solves the ILP problems constructed by the abstraction procedure, and interprets the results. In case solutions to the ILP problems are found, it also constructs *counterexamples* from these ILP solutions to indicate certain property violating scenarios. Counterexamples constructed in our framework may be different from the counterexamples in the context of explicit state model checking. A counterexample returned by a model checker describes a trajectory in the global state transition graph of the system, starting from the initial state to a property offending state. However, in our setting, a counterexample usually does not give directly an erroneous execution of the system but only specifies some constraints that apply to a property violating execution. This is because the counterexample is constructed from the abstraction of the original system and therefore under-specifies the concrete executions of the original system. For instance, a counterexample can indicate which cycles are to be repeated infinitely often in a property violating execution. However, it does not specify how the executions of the infinitely repeated cycles are combined.

As the abstraction procedure is conservative and may result in an imprecise analysis, a refinement procedure is necessary to improve the accuracy by recov-

ering information lost during abstractions. Following the idea in [33], the abstraction refinement that we propose is guided by the abstract counterexamples that we obtain. The whole verification procedure is iterative, which corresponds to the loop in Figure 4.1 through the property checking, counterexamples, and refinement blocks: Given a system, for reasons of efficiency a relatively coarse abstraction is constructed during the first iteration of our verification approach. The model may be too coarse, and a counterexample may be constructed which does not correspond to any real execution of the original system. In this case certain information of the original system is recovered in order to exclude the spurious behavior that the counterexample corresponds to. The property is then checked on the refined abstraction in the next iteration. The model is gradually refined in this way until either it is precise enough to verify the property or the abstraction cannot be refined anymore.

In the subsequent chapters we detail the verification methods: the checking of buffer boundedness in Chapter 5; the checking of livelock freedom in Chapter 6; abstraction issues in Chapter 7, Chapter 8, and Chapter 9; refinement techniques in Chapter 11 and Chapter 12.

Part I

Integer Linear Programming Based Verification

Chapter 5

Checking Buffer Boundedness

In this chapter we present the core of a buffer boundedness test as one application of the verification framework that we described in the previous chapter. As mentioned previously, CFSM systems are used in our analysis as a common level of abstractions that a model of an asynchronous reactive systems will be abstracted into. We therefore present the boundedness test at the level of CFSM systems. In Chapter 7 and Chapter 8 we will address respectively the issues of abstracting models described in Promela and UML RT, as examples among other modeling languages. Moreover, the abstraction refinement procedure for the boundedness test will be discussed in Chapter 12.

The unboundedness of the message buffers in a CFSM model can have several negative effects. First, if the model represents a software design, the unboundedness of one or more of the message buffers hints at a possible design fault: The unbounded buffers may result in buffer overflows, or loss of messages, or other undesirable behavior in the system that implements the design. This is particularly important if the unboundedness cannot be ruled out by the relative speed of processes or by the process scheduling mechanism involved in implementing the system. Second, buffers with unbounded capacity impede automated finite state analyzability since they induce an infinite state space that renders state space exploration incomplete in finite time.

In spite of the potential unboundedness of the message buffers in CFSM systems, one commonly observes that, for many actual CFSM models, the buffer occupancy is bounded by some small constant k. This means that, although their length is not fixed a priori, they can never contain more than k messages, because of the particular dynamics of the system. If this is the case then one can safely replace the unbounded buffers by k-bounded buffers without changing the behavior of the system. Such a system with k-bounded buffers is a finite state system. Ideally, one wants to find individual bounds k_i for every buffer B_i, or at least to determine whether such bounds exist.

The boundedness test that we propose can automatically determine boundedness for CFSM systems and can also automatically estimate buffer bounds for systems whose boundedness has been proved. The test is however incomplete: it may not be able to determine boundedness for some CFSM systems

and delivers in such occasions an inconclusive verdict. In fact buffer boundedness is undecidable for CFSM systems and no complete algorithm therefore exists to determine boundedness for all CFSM systems. Moreover, the test is efficient and scales to realistic models of large size, as we will show both in the complexity (Section 5.7) and experimental results (Section 14.1).

Structure. We formally define and discuss the buffer boundedness property in Section 5.1. An overview of the boundedness test is given in Section 5.2 before the details are discussed in the subsequent sections. In particular, the soundness of the test is argued in Section 5.6, and we show the complexity results in Section 5.7. In the end of the chapter, a comparison to the boundedness test proposed by Brand and Zafiropulo is given.

5.1 Buffer Boundedness

We formally define buffer boundedness for CFSM systems as follows.

Definition 5.1 (Buffer Boundedness of CFSM Systems)**.** Given a CFSM system, a message buffer b in the system is *bounded* if and only if there exists a natural number k such that, in any reachable configuration of the system, the buffer b contains no more than k messages. If no such k exists, then b is *unbounded*. The system is *bounded* if and only if all the buffers are bounded. The system is *unbounded* if and only if at least one buffer is unbounded.

The above definition of buffer boundedness can be straightforwardly adapted for other modeling formalisms and languages for describing asynchronous reactive systems, such as Queue Automata [27, 28], UML RT, and Promela, among others.

Proposition 5.1. *If a CFSM system is bounded, then it has finitely many reachable configurations.*

The above proposition is obvious, and it states a desired property of bounded CFSM systems: Their reachability graphs are finite so that many interesting problems become decidable, e.g., the determination of deadlock and livelock freedom that can be checked by finite state verification techniques such as model checking. However, many modeling languages permit other sources of infiniteness besides unbounded buffers. As an example UML RT allows variables of types with an infinite domain such as integers. Moreover, it also supports unbounded capsule instantiations. These factors in a UML RT model may still contribute to an infinite global state space even if the buffers are all bounded.

In the following we show several simple CFSM systems and manually check their boundedness.

Example 5.1. Consider the four CFSM systems in Figure 5.1. All systems consist of two state machines. We assume that there is one unique buffer to store each kind of messages such as a, b or c. In this way we can omit the names of buffers in the transition labels for succinctness reasons. Moreover, since we are only interested in infinite executions, all finite prefixes, e.g., transitions initializing the system, have been disregarded.

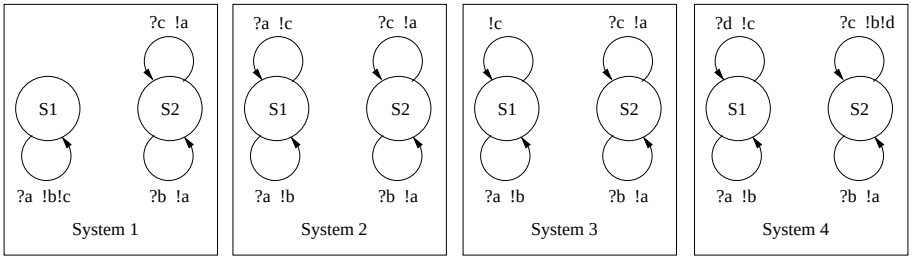

Figure 5.1: Various examples of CFSM systems

It is easy to see that System 1 is unbounded if at least one message of any type is initially available. Any execution of the cycle through the state $S1$ in the left process will consume a message a and produce two messages b and c. Each one of these messages would then produce one message a when the right process cycles through $S2$. To the contrary, System 2 is bounded since a message a generates only one single b or c message, while the consumption of a message b or c triggers the generation of a single a message. System 3 contains a spontaneous self-transition that may obviously flood the buffer with an unbounded number of messages c. Deciding the boundedness of System 4 is less obvious. Whenever the system generates a message c, the buffer may be flooded. However, if the system only ever executes the cycles in which messages a and b are exchanged, the buffer filling remains bounded. Whether a message c ever gets exchanged obviously depends on how the system is initialized, which is information we will abstract away in the boundedness test.

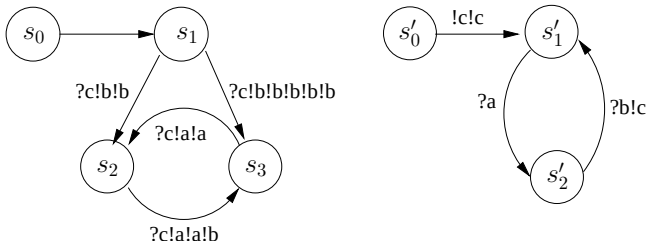

Figure 5.2: A simple CFSM system whose boundedness is not easily seen

While in the above case the boundedness is easy to see by manual inspection, this is generally not the case. Consider the example given in Figure 5.2. For succinctness reasons, we again omit the names of buffers, and assume that each process uses one unique buffer to send messages to the other process. The left process can send messages a and b to the right process, and the right process can send messages c to the left process. The actual boundedness of this system is far from obvious. This calls for automated methods in support of testing a system's boundedness. However, as stated in Theorem 5.2, buffer boundedness is undecidable for CFSM systems, which implies that there is no complete algorithm to determine boundedness for all CFSM systems. Consequently, the boundedness test that we propose is inevitably incomplete.

Theorem 5.2. *It is in general undecidable whether a CFSM system is bounded.*

A detailed proof of Theorem 5.2 is given in Appendix A in [25]. The proof is by a reduction from the undecidable halting problem of Turing machines.

Before we start to describe our boundedness test in the next section, we give an in-depth discussion of the buffer boundedness property and review existing boundedness analysis techniques in the remainder of this section.

5.1.1 Buffer Boundedness and the Safety-Liveness Classification

We discuss the buffer boundedness property in relation to the widely accepted interpretation of properties as sets of executions, and in particular, to the safety-liveness classification of properties [78]. By this discussion we may gain more insights on both the boundedness property and the challenges in its verification.

The safety-liveness classification was suggested and informally characterized in [78] while formal definitions of safety and liveness properties were later given in [79, 13, 109, 14, 30]. The classification is broadly adopted in practice because the verification principles for these two classes of properties have been well studied [95, 88]. In the following we give a brief introduction to the classification.

The Safety-Liveness Classification

A *property* is usually interpreted as a set of executions. If the permitted executions of a system S is a subset of a property P, then S is said to *satisfy* P. Informally a safety property states that some "bad" thing never happens. A more formal definition of safety properties is as follows.

Definition 5.2 (Safety Property)**.** A property P is a *safety property* if and only if, for any execution $r \notin P$, there exists a prefix r_p of r such that $r_p.r_s \notin P$ for any suffix r_s.

Intuitively, if an execution r does not satisfy P, then there is a finite prefix r_p of r that can witness the violation of P, i.e., the "bad" thing must have happened at some point within r_p. Any extension of r_p therefore cannot satisfy P.

On the contrary, a liveness property states that some "good" thing will eventually happen. A more formal definition of liveness properties is as follows.

Definition 5.3 (Liveness Property)**.** A property P is a *liveness property* if and only if, for any finite execution r_p, there exists a suffix r_s such that $r_p.r_s \in P$.

Intuitively, any finite execution can be extended to one execution satisfying P if the extending suffix contains the "good" thing.

There is one limitation of interpreting properties as sets of executions: it may only describe properties of individual executions and is incapable to capture relations among executions. We would argue that the buffer boundedness property for Queue Automata and UML RT models cannot be represented as a set of executions. Consequently, their buffer boundedness falls outside the safety-liveness classification.

Non-Set-Representable Buffer Boundedness of Queue Automata

We define one-queue automata and its buffer boundedness property as follows.

Definition 5.4 (One-Queue Automaton)**.** A *one-queue automaton*

$$(\Sigma, Q, q_0, F, \Gamma, z_0, \delta)$$

is an ω-automaton augmented with one FIFO queue where

- Σ is a finite input alphabet;
- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of accepting states;
- Γ is a finite queue alphabet;
- $z_0 \in \Gamma^*$ is the initial content of the queue;
- $\delta \subseteq (\Sigma \times Q \times \Gamma) \times (Q \times \Gamma^*)$ is a transition relation.

A *configuration* is a pair (q, z) where $q \in Q$ and $z \in \Gamma^*$, and the initial configuration is $c_0 = (q_0, z_0)$. We define the *successor* relation ($\Rightarrow$) for configurations as follows: For any two configurations (q_1, z_1) and (q_2, z_2), $(q_1, z_1) \Rightarrow (q_2, z_2)$ if and only if there exists a transition $(a, q_1, \gamma, q_2, z) \in \delta$ such that (1) $z_1 = \gamma.z_1'$ for some sequence $z_1' \in \Gamma^*$ and (2) $z_2 = z_1'.z$. An *execution* of the automaton is a sequence of configurations $\langle c_0, c_1, \dots \rangle$ such that $c_i \Rightarrow c_{i+1}$ for any $i \geq 0$. Let $\Rightarrow^+$ be the reflexive and transitive closure of $\Rightarrow$. A configuration c is *reachable* if $c_0 \Rightarrow^+ c$. An execution is *accepted* if it is infinite and at least one accepting state occurs infinitely often in the execution. The language of a one-queue automaton is the set of the input sequences of all accepted executions.

Definition 5.5 (Queue Boundedness of One-Queue Automata[1])**.** A one-queue automaton is *bounded* if and only if there exists a natural number k such that, in any reachable configuration of the automaton, the queue used in the automaton contains no more than k elements.

Proposition 5.3. *The queue boundedness property of one-queue automata cannot be represented as a set of executions.*

Proof. By contradiction we assume that the queue boundedness property of one-queue automata can be represented by a set of executions P. Let us consider an infinite collection of one-queue automata in which each automaton A_i ($i \geq 1$) is defined as in Figure 5.3. Each A_i has one input symbol a, one queue symbol γ, and $i+1$ states $q_0 \dots q_i$ where q_0 is the initial state and q_i is the only accepting state. The queue contains initially one element γ. Each transition from the state q_k to the state q_{k+1} ($0 \leq k < i$) takes an input symbol a, removes an element γ from the queue, and appends two γ's to the queue, thereby increasing the length of the queue by 1. The behavior of the self-transition at q_i is the same as the behavior of the other transitions, except that it only appends one γ to the queue and therefore does not change the length of the queue.

[1]We refer to the buffer boundedness of one-queue automata as queue boundedness because the term "queue" is used in the definition of one-queue automata.

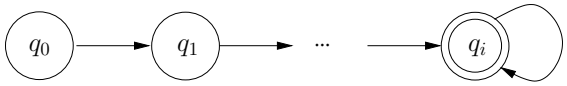

Figure 5.3: A bounded one-queue automaton

Every automaton A_i is obviously bounded because in any accepted execution the length of the queue can be increased only for i times and is unchanged after entering the accepting state. Let E_i denote the set of accepted executions of A_i. According to the definition of properties, a system satisfies a property if and only if the set of its accepted executions is included in the set of executions representing the property. As a result, we have

$$E_i \subseteq P \qquad \text{for all } i \geq 1. \tag{5.1}$$

Next, we construct another one-queue automaton A as in Figure 5.4. The automaton A has one input symbol a, one queue symbol γ, and two states: q_0 as the initial state and q_1 as the accepting state. The behavior of the self-transition at q_0 is the same as the behavior of those non-self-transitions in A_i, namely that it increases the length of the queue by 1. The transition from q_0 to q_1 and the self-transition at q_1 have the same behavior as the self-transition in A_i, which keeps the length of the queue unchanged.

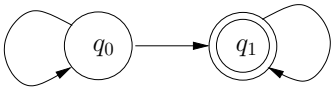

Figure 5.4: An unbounded one-queue automaton

A is unbounded: For any given number k, there exists an accepted execution of A that reaches q_1 after cycling at q_0 for k times, which increases the length of the queue up to $k + 1 > k$. However, we can easily see that every accepted execution of A is in P. The reason is that any accepted execution of A is an accepted execution of some A_i, i.e., using E to denote the set of accepted executions of A, we have

$$E = \bigcup_{i=1}^{\infty} E_i. \tag{5.2}$$

From the statements (5.1) and (5.2) above, we know that $E \subseteq P$, which means that A satisfies the queue boundedness property. This contradicts the fact that A is unbounded. □

Non-Set-Representable Buffer Boundedness of UML RT Models

We show informally that the buffer boundedness property of UML RT models is also not representable by a set of executions, based on a similar argument as for one-queue automata. Consider a simple UML RT model that consists of only one capsule whose state machine is depicted in Figure 5.5. The capsule may cycle through the state $S1$ to increase the value of an integer variable y for an

arbitrary or even infinite number of times[2]. Once the capsule stops to increase y and transits to the state $S2$, it sends to the message buffer b as many messages m as the value of the variable y. Apparently the buffer b is unbounded in all the reachable configurations of the model because the value of y is unbounded upon reaching $S2$. However, b is bounded in every individual execution: The capsule may either increase the value of y forever and therefore never increase the buffer length, or it reaches the state $S2$ with a finite y value by which the buffer length is bounded afterward in this particular execution. Therefore, any set of executions P to represent the buffer boundedness property of UML RT models cannot exclude any of the executions of the model. This implies that the behavior of the model is a subset of P. This contradicts the fact that the model is actually unbounded.

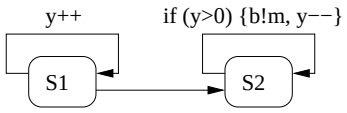

Figure 5.5: A UML RT model

Bounded Executions and Buffer Boundedness of CFSM systems

We define the property of *bounded executions* as follows. An execution of a system is *bounded* if and only if all buffers in the system are bounded in the execution. It is easy to see that the property of bounded executions is a liveness property: Any finite partial execution can be extended to a bounded execution by appending an infinite suffix in which all buffers in the system are bounded.

As seen from the above arguments, the fact that in general a system permits only bounded executions is not sufficient to guarantee the boundedness of the system. This is because a system may have an infinite number of executions and the bounds of buffers in individual executions may be unbounded. Examples have been shown as the systems in the figures 5.4 and 5.5.

However, we have the following conjecture. For a CFSM system, if all the executions of the system are bounded, then the system is bounded. If this conjecture is true then the buffer boundedness property of CFSM systems can be represented as a set of executions and it is also a liveness property. The intuition behind our conjecture is that, unlike one-queue automata and UML RT models, the behavior of a CFSM system is neither constrained by acceptance conditions nor by boolean conditions on variable values. Its behavior is only constrained by the sending and receiving of messages. Therefore, if the buffer bounds in individual bounded executions of the system are unbounded, then there must exist a common cyclic segment in some bounded executions which does not decrease the lengths of all buffers and increases the length of at least one buffer. This segment is repeated for different finite numbers of times in these executions so that these executions can be bounded. But infinitely repeating this segment can certainly result in an unbounded execution. So, if the buffer

[2]In UML RT every transition must be triggered by an incoming message. To respect this restriction we may regard all the transitions in the example model to be triggered by a special type of messages m' that are always abundant. We disregard the message type m' and the supply of m' because they are not relevant to our argument.

bounds in bounded executions are unbounded, there must exist an unbounded execution. In future work we will work out a formal proof of our conjecture.

If our conjecture is true and buffer boundedness for CFSM systems is a liveness property, then one may consider to use the existing liveness property checking methods to check buffer boundedness. Unfortunately, liveness properties are undecidable for CFSM systems [26, 53]. The existing verification methods must rely on abstractions, e.g., by reducing the verification of a liveness property into the problem of checking coverability of Petri nets [52, 59], which is very inefficient.

5.1.2 Existing Boundedness Analysis Methods

Practitioners usually notice the boundedness of a system either by manual inspection of the system, or by running random simulations. However, manual inspection is neither reliable nor applicable to systems of considerable complexity. Random simulations are more useful in detecting unbounded executions than in achieving full confidence in the boundedness of a system, since random simulations cannot cover all the possible executions in general. This calls for either formal techniques to prove rigorously a system's boundedness, or effective and automated methods to detect unboundedness, or realistic scheduling mechanisms to rule out any unbounded executions at run time. In the following we review several existing techniques of these kinds.

Formal Methods for Boundedness Determination

We have known that buffer boundedness is undecidable for CFSM systems. However, it is decidable for some subclasses of CFSM systems[3] [25, 61, 104, 55, 73, 76, 60, 52, 51]. However, these subclasses are too restrictive and thus not useful in practice. It is therefore of greater practical interest to develop boundedness determination methods for the whole class of CFSM systems even though any such methods are inevitably incomplete.

Buffer boundedness for CFSM systems is semi-decidable. One example of a semi-complete algorithm for determining boundedness is to construct the reachability graph of a CFSM system [100]. When a system is bounded, it possesses a finite reachability graph and the algorithm certainly terminates on it. However, if the system is unbounded, the algorithm will never terminate because the reachability graph is infinite. This algorithm has little practical use because, in order to prove boundedness for a system, it has to construct the whole reachability graph that is triply exponential in the number of processes, in the number of message buffers, and in the maximal runtime length of each message buffer.

There are some incomplete boundedness tests with the help of abstractions that work more efficiently than the above mentioned semi-complete algorithm. [90] proposes a method that first abstracts a CFSM system into a Petri net by disregarding message orders in buffers[4], and then determines the boundedness of the resulting Petri net. Since the abstraction into Petri nets is over-approximating, not all the bounded CFSM systems can be proved to

[3] The decidability results for subclasses of CFSM systems have been summarized in [73].

[4] We also abstract from message orders in our boundedness analysis, which will be explained in more detail later.

be bounded. Moreover, the boundedness problem of Petri nets is EXPSPACE-complete, which limits the ability of the method to handle large systems. A more efficient method was earlier proposed by Brand and Zafiropulo in [25], which we abbreviate as the BZ test. The method is based on a combinatorial analysis of the message passing behavior of control flow cycles in CFSM systems, which is a similar approach to our test. The complexity of the BZ test is exponential in the number of control flow cycles because it uses NP-complete integer linear program solving. It is in contrast to the polynomial complexity of our test. In Section 5.9 more comparisons between the BZ test and our test will be given.

Unboundedness Tests

While boundedness determination remains a difficult task, the detection of potential unboundedness is valuable in revealing design errors in a system. [73] proposes an unboundedness test based on a sufficient condition for unbounded executions. This unboundedness condition is satisfied by a finite execution prefix if there are two global states c_1 and c_2 in the prefix such that the segment between c_1 and c_2 can be repeated at c_2 and produce more messages in the buffers in the system without losing any messages. The test explores the reachability graph of a CFSM system in an on-the-fly fashion to search for finite execution prefixes that satisfy the sufficient condition. Any such finite execution prefix being found then assures the existence of unbounded executions. The test is incomplete and has an exponential complexity.

Bounded Scheduling

Bounded scheduling has been studied for Kahn process networks [96, 57] and recently for Petri nets [87]. It aims at finding a scheduling policy to ensure that only bounded executions occur at runtime even though the system is unbounded. To our knowledge, there is currently no existing work addressing the bounded scheduling of CFSM systems. However, since CFSM systems can be abstracted into Petri nets, any bounded scheduling for an abstract Petri net model is also a bounded scheduling for the original CFSM system.

5.2 Overview of the Boundedness Test

We give an overview of our buffer boundedness test. The test makes use of a series of abstract steps that leaves us with an over-approximating abstraction as an independent cycle system. Boundedness for independent cycle systems can be determined efficiently using integer linear program solving techniques. The test can also estimate an upper bound for each individual message buffer for CFSM systems whose boundedness has been proved. By the very nature of over-approximations, not every bounded CFSM can be detected as such by this method and the obtained bounds are not necessarily optimal. However, the computed bounds are certainly upper bounds, which is sufficient to make the system treatable by finite-state verification methods.

The underlying idea of our boundedness test is to determine whether at all it is possible to combine the cyclic executions of all the processes in a CFSM system in such a way that the filling of at least one of the message buffers can be

"blown up" in an unbounded way. Note that any infinite execution of a CFSM system can be understood as a certain way of repeating infinitely some or all elementary control flow cycles in the processes of the system.

5.3 Abstraction

In this section we describe a sequence of conceptual abstractions used in the boundedness test. Each abstraction level corresponds to a computational model for which complexity results for the boundedness problem are either known or provided by our work. The abstraction is conceptual since the tool that we develop does not perform the transformations described in this section to generate the intermediate abstraction at each level, but uses a more direct approach that will be described in Section 13.1. The purpose of the conceptual abstraction is to show how the complexity of the boundedness problem can be reduced by each abstraction step. As mentioned above, we want to check the boundedness of the CFSM system in terms of summary message passing effects of control flow cycles. The goal of our conceptual abstraction is to arrive at a data structure that allows us to reason about these summary effects using linear combination analysis.

The abstract computational model that we obtain is an over-approximation of the original CFSM system in the following senses:

- All behavior of the original system is also possible in the abstract model. However, there can exist some behavior that is possible in the abstract model, but not in the original system.
- The abstraction preserves the number of messages in every message buffer of the CFSM system. In particular, if some buffer is unbounded in the CFSM system, then it is also unbounded in the over-approximation. Furthermore, if a buffer is bounded by a constant k' in the over-approximation, then it is bounded by some constant $k \leq k'$ in the original system.

In the following we show each conceptual abstraction step, the resulting computational model of the step, the definition of boundedness for this model, and the complexity of the corresponding boundedness problem.

Level 0: UML RT or Promela. We start in the beginning with models described in UML RT, or Promela, or other modeling languages for asynchronous reactive systems. For these models, boundedness is undecidable since these modeling languages are usually more expressive than CFSM systems and can therefore simulate Turing-machines. For instance, UML RT models may contain arbitrary program code written in high-level programming languages such as C++ or Java.

Level 1: CFSM Systems. First, we abstract the model described in UML RT or Promela to an over-approximating CFSM system. This step involves the abstraction from the general program code, e.g., the code on the transitions of a UML RT model. The code abstraction retains only the finite control structure of the capsules in the UML RT model and their message passing via message buffers. This abstraction step would provide different treatment for different

modeling languages. As we focus in this chapter on the core idea of the boundedness test at the level of CFSM systems, we will discuss the abstraction of Promela and UML RT models respectively in Chapter 7 and in Chapter 8. For CFSM systems, it has been shown that boundedness is undecidable [26].

Level 2: Parallel-Composition-VASS. In the next step we abstract from the order of messages in the buffers and consider only the number of messages of any given type. For example, the buffer with contents *abbacb* would be represented by the integer vector $(2, 3, 1)$, representing 2 messages of type a, 3 messages of type b and 1 message of type c. Consequently, no distinction can be made at this level between *abbacb* and *bbbaac* that are both represented by the same integer vector. In this way we also abstract from the order of message sending and receiving events in a transition and use an integer vector to denote how many massages of each type are sent or received along the transition. We call such an integer vector an *effect vector*. A positive component in an effect vector denotes the number of messages of the corresponding type being sent; a negative component denotes the number of messages of the corresponding type being received. Consider the CFSM system in Figure 5.2. The transition from the state s_2 to the state s_3 has the effect vector $(2, 1, -1)$, and the transition from s'_1 to s'_2 has the effect vector $(-1, 0, 0)$.

For the purpose of complexity analysis it is helpful to relate the obtained abstraction to the theory of Petri nets [93]. The numbers of messages in any buffer can be represented by the number of tokens in Petri net places. We then obtain a *vector addition system with states* (VASS) [21]. The control states correspond to the states of the processes of the CFSM systems and the Petri net places represent the integer vectors approximating buffer contents. More exactly, we obtain a *parallel-composition-VASS*. This is a VASS whose finite-control is the parallel composition of several finite state machines. We give the definition of parallel-composition-VASS as follows.

Definition 5.6 (Parallel-Composition-VASS)**.** A *parallel-composition-VASS* is a triple

$$(P, X, \delta)$$

where

- P is a finite number of *parts*. Each part p_i is a pair (Q_i, q_0^i) where Q_i is a finite set of control states and $q_0^i \in Q_i$ is the initial state in the part. For any two different parts p_i and p_j, we put the restriction that their sets of control states are disjoint, i.e., $Q_i \cap Q_j = \emptyset$ for any $i \neq j$. Let $Q = \bigcup_i Q_i$;
- X is a finite set of integer variables called *places* and the values of these variables are called *tokens*;
- $\delta \subseteq (Q \times \mathbb{Z}^{|X|} \times Q)$ is a transition relation. We require that, for any transition $(q_1, \bar{v}, q_2)$, q_1 and q_2 are control states of the same part.

Let $|P| = m$ and $|X| = n$ in the above definition. A *configuration* is a pair $(\bar{q}, \bar{v})$ where $\bar{q} \in Q_1 \times \cdots \times Q_m$ and $\bar{v} \in \mathbb{N}^n$. Let $\bar{q}_0 = (q_0^1, \ldots, q_0^m)$. A configuration $(\bar{q}, \bar{v})$ can be an initial configuration, usually denoted by c_0, if $\bar{q} = \bar{q}_0$. The execution of a VASS system can therefore be started with any

finite number of tokens. We define the *successor* relation ($\Rightarrow$) for configurations as follows: For any two configurations $(\bar{q}_1, \bar{v}_1)$ and $(\bar{q}_2, \bar{v}_2)$, $(\bar{q}_1, \bar{v}_1) \Rightarrow (\bar{q}_2, \bar{v}_2)$ if and only if there exists a transition $(q_1, \bar{v}, q_2)$ in a part p_i such that (1) $q_1 = \bar{q}_1^i$ and $q_2 = \bar{q}_2^i$; (2) for any part p_j where $i \neq j$, $\bar{q}_1^j = \bar{q}_2^j$; and (3) $\bar{v}_2 = \bar{v}_1 + \bar{v}$. An *execution* of the system is a sequence of configurations $\langle c_0, c_1 \ldots \rangle$ such that $c_i \Rightarrow c_{i+1}$ for all $i \geq 0$. Let $\Rightarrow^+$ be the reflexive and transitive closure of $\Rightarrow$. Given an initial configuration c_0, a configuration c is *reachable from* c_0 if $c_0 \Rightarrow^+ c$. Note that a parallel-composition-VASS is not exactly the same as the parallel composition of several VASS, because the places are shared by all parallel parts of the finite control.

By the abstraction from message orders, we obtain a parallel-composition-VASS system from the CFSM system of Level 1. It is easy to see that the obtained VASS system has the following properties. First, the control flow structures of the CFSM system and the VASS system are isomorphic: (1) The state machine of each process in the CFSM system corresponds to a unique part in the VASS system; (2) There is a bijective mapping f from the local states in the CFSM system to the control states in the VASS system such that f preserves initial states: If s is an initial state, then $f(s)$ is also an initial state. Given a state s in the CFSM system and a state s' in the VASS system, if $s' = f(s)$ then we say that s and s' are *congruent*; (3) There is a bijective mapping g from the transitions in the CFSM system to the transitions in the VASS system such that, for any transition $t = (s_1, l, s_2)$ in the CFSM system, there is a transition $g(t) = (f(s_1), \bar{v}, f(s_2))$ in the VASS system, in which $\bar{v}$ is the effect vector that over-approximates the message passing effect of the event sequence in l. Second, each place of the VASS system corresponds to a unique type of messages in the CFSM system. Figure 5.6 shows the abstract parallel-composition-VASS system for the CFSM system in Figure 5.2. We can see that the control flow structure of the CFSM system is completely preserved.

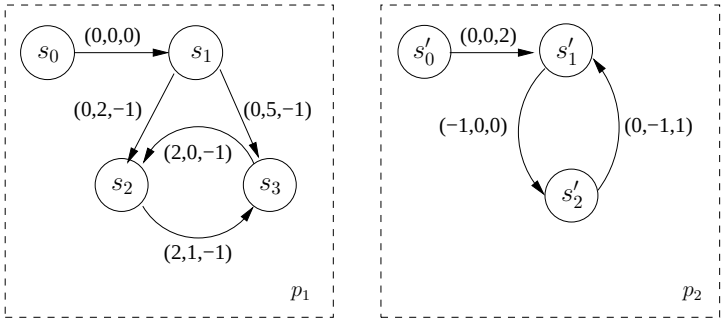

Figure 5.6: The parallel-composition-VASS system corresponding to the CFSM system in Figure 5.2

We define boundedness for parallel-composition-VASS as follows.

Definition 5.7 (Boundedness of Parallel-Composition-VASS)**.** A parallel-com -position-VASS system is bounded with respect to an initial configuration c_0 if and only if there exists a natural number k such that, in any reachable configuration from c_0, the values of all the integer variables (i.e., the numbers of tokens in all places) are no larger than k.

The boundedness problem for parallel-composition-VASS is to determine, given an initial configuration c_0, whether the system is bounded with respect to c_0. This problem is polynomially equivalent to the boundedness problem for Petri nets, which is EXPSPACE-complete [119].

Level 3: Parallel-Composition-VASS with Arbitrary Initial Tokens. We now abstract from activation conditions of cycles in the control-graph of the parallel-composition-VASS. Any combination of control flow cycles in the system has a minimal activation condition, i.e., a minimal number of tokens needed to get it started. In principle, it is decidable if there is a reachable configuration that satisfies these minimal requirements, but this involves solving the coverability problem for Petri nets: given a Petri net, whether there exists a reachable marking which is bigger than a given marking. This problem is decidable, but at least EXPSPACE-hard [86, 50], and thus not practical. So, we assume instead that there are always enough tokens present to start the cycle. More precisely, we assume that any control flow cycle that has an overall non-negative effect on all places can be repeated arbitrarily often. By this abstraction, we replace the boundedness problem "Is the system bounded with respect to a given initial configuration?" by the problem of the so-called structural boundedness:

Definition 5.8 (Structural Boundedness of Parallel-Composition-VASS)**.** A parallel-composition-VASS system is *structurally bounded* if and only if the system is bounded with respect to any initial configuration.

It will be shown in Section 5.7 that the structural boundedness problem for parallel-composition-VASS is co-NP-complete, unlike for standard Petri nets where it is polynomial [92, 50]. The reason for this difference is that an encoding of control states by Petri net places does not preserve structural boundedness, because it is not assured that only one of these places in each part is marked at any time. Furthermore, the co-NP-lower bound even holds if the number of elementary cycles in the control-graph is only linear in the size of the system.

Level 4: Independent Cycle System. Finally, we abstract from the fact that certain control flow cycles within one part or among several parts in a VASS system may depend on each other. For example, cycles might be mutually exclusive so that executing one cycle makes another cycle unreachable. Cycles may also depend on each other, i.e., one cannot repeat some cycle infinitely often without repeating some other cycle infinitely often. The concept of cycle dependencies will be discussed and formalized in Chapter 11.

We abstract from cycle dependencies in order to obtain a level of abstractions for which boundedness can be more efficiently determined. However, the presence of some cycle dependencies is critical to the boundedness of a system and the loss of this information may cause our analysis to become imprecise. In order to overcome this disadvantage we have designed several static analysis techniques to discover cycle dependencies. These techniques constitute a major part of the abstraction refinement in our verification framework and will be explained in Chapter 11. In fact the discovery of cycle dependencies is costly but the discovery procedure will be called only when needed, i.e., if the boundedness test fails.

Here, by abstracting from cycle dependencies, we assume that all cycles are independent and any combination of them is executable infinitely often,

provided that the combined effect of this combination on all places is non-negative. In this way we may abstract the VASS system of Level 3 to a set of independent elementary control flow cycles with their summary effect vectors. We call this system an independent cycle system.

Definition 5.9 (Independent Cycle Systems)**.** An *independent cycle system* is a triple

$$(n, C, \mathit{eff})$$

where

- $n \geq 1$ is a fixed dimension of integer vectors;
- C is a finite set of cycles;
- $\mathit{eff} : C \to \mathbb{Z}^n$ is a function that associates with every cycle in C an integer vector of dimension n as its *effect vector*;

In the above definition, let $v_c \in \mathbb{N}$ for each $c \in C$, and we call

$$\sum_{c \in C} v_c \cdot \mathit{eff}(c)$$

a *linear combination of cycles*. Figure 5.7 shows the independent cycle system that we obtain from the parallel-composition-VASS system in Figure 5.6 after abstracting from cycle dependencies.

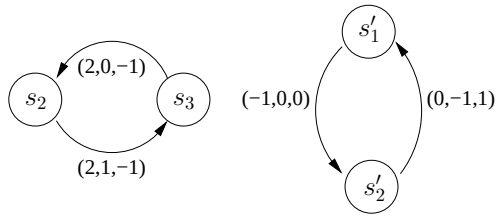

Figure 5.7: The independent cycle system corresponding to the VASS system in Figure 5.6, and corresponding to the CFSM system in Figure 5.2

Now we define boundedness for independent cycle systems as follows.

Definition 5.10 (Boundedness of Independent Cycle Systems)**.** An independent cycle system $\mathcal{C} = (n, C, \mathit{eff})$ is *bounded* if, for any integer vector $\bar{v}$ of the dimension n, there exists an upper bound $\bar{b}$ such that the following is satisfied: For any linear combination of cycles $\sum_{c \in C} v_c \cdot \mathit{eff}(c)$ such that $\bar{v} + \sum_{c \in C} v_c \cdot \mathit{eff}(c) \geq 0$, we have that $\bar{v} + \sum_{c \in C} v_c \cdot \mathit{eff}(c) \leq \bar{b}$. Otherwise, $\mathcal{C}$ is *unbounded*.

We will show in Section 5.6 (soundness results) that the boundedness of independent cycle systems is a *sufficient* condition for the boundedness of any previous levels of abstractions. Equivalently, the unboundedness of independent cycle

systems is a *necessary* condition for the unboundedness of any previous levels of abstractions. Hence, if boundedness can be proved at this level, then boundedness for other levels is immediately implied, which means that the original model is also bounded. However, if the independent cycle system is unbounded, then the models of other levels may not necessarily also be unbounded.

The following proposition gives a sufficient and necessary condition for the boundedness of an independent cycle system.

Proposition 5.4. *An independent cycle system is bounded if and only if there exists no positive linear combination of cycles.*

In order to prove the above proposition, we need the following auxiliary lemma.

Lemma 5.5. *For any infinite sequence of integer vectors of same dimension $s = \langle \bar{v}_1, \bar{v}_2, \dots \rangle$, if there exists an integer vector $\bar{b}$ such that $\bar{v}_i \geq \bar{b}$ holds for each i, then there is an infinite subsequence $\langle \bar{v}_{u_1}, \bar{v}_{u_2}, \dots \rangle$ of s such that $\bar{v}_{u_i} \leq \bar{v}_{u_j}$ holds for all $u_i < u_j$.*

Proof. We prove the lemma using the induction on the dimension k of the vectors in the sequence.

Induction base: Let $k = 1$. Then, s is actually a sequence of integers $\langle v_1, v_2, \dots \rangle$ bounded below by b. By contradiction, we assume that there is no infinite non-decreasing subsequence of s. Then, there must exists an upper bound b' such that $v_i \leq b'$ for every i. Because there are only finitely many integers v such that $b \leq v \leq b'$, there must exist infinitely many elements in s that are equivalent. These elements form a non-decreasing subsequence of s, which contradicts the assumption that there is no such subsequence.

Induction step: Assume that the lemma holds for all sequences of integer vectors of dimension k, and that s consists of vectors of dimension $k+1$. Let $trunc(\bar{v}_i)$ be an integer vector of dimension k constructed from $\bar{v}_i$ in s such that $trunc(\bar{v}_i)^j = \bar{v}_i^j$ for each $1 \leq j \leq k$. According to the induction assumption, we can select an infinite subsequence s' of s: $\langle \bar{v}_{w_1}, \bar{v}_{w_2}, \dots \rangle$ such that $trunc(\bar{v}_{w_i}) \leq trunc(\bar{v}_{w_j})$ holds for all $w_i < w_j$. The $(k+1)$-th components of all elements in s' form an infinite sequence of integers, from which we can select an infinite non-decreasing subsequence of integers $\langle \bar{v}_{u_1}^{k+1}, \bar{v}_{u_2}^{k+1}, \dots \rangle$, based on the argument for the induction case. Apparently, $\langle \bar{v}_{u_1}, \bar{v}_{u_2}, \dots \rangle$ satisfies the property that $\bar{v}_{u_i} \leq \bar{v}_{u_j}$ holds for all $u_i < u_j$. □

Now we prove Proposition 5.4.

Proof. It is equivalent to prove the following: Given a finite set of integer vectors $\bar{v}_1, \dots, \bar{v}_n$ of the same dimension, say m, the two conditions below are equivalent.

- Condition **C1**: There exists no $x_i \in \mathbb{N}$ $(1 \leq i \leq n)$ such that

$$\sum_{i=1}^{n} x_i \bar{v}_i > \bar{0}. \tag{5.3}$$

- Condition **C2**: For any vector $\bar{v}$, there exists a vector $\bar{b}$ such that, for all $x_i \in \mathbb{N}$ $(1 \leq i \leq n)$,

$$\bar{v} + \sum_{i=1}^{n} x_i \bar{v}_i \geq \bar{0} \rightarrow \bar{v} + \sum_{i=1}^{n} x_i \bar{v}_i \leq \bar{b}. \tag{5.4}$$

Let $f(x_1, \ldots, x_n) := \bar{v} + x_1\bar{v}_1 + \cdots + x_n\bar{v}_n$. We first prove `C1` $\rightarrow$ `C2` by contradiction and assume that there exists some k such that no upper bound exists for $f(\bar{x})^k$ under the condition $f(\bar{x}) \geq \bar{0}$. This implies that there exists an infinite sequence of non-negative vectors $s_c = \langle \bar{c}_1, \bar{c}_2, \ldots \rangle$ of dimension n such that (1) $f(\bar{c}_i) \geq \bar{0}$ (2) $\lim_{i\rightarrow\infty} f(\bar{c}_i)^k = +\infty$.

Following Lemma 5.5, we may select from s_c an infinite subsequence $s_d = \langle \bar{d}_1, \bar{d}_2, \ldots \rangle$ such that $f(\bar{d}_i)^k$ is strictly increasing. Again, from s_d we can select an infinite subsequence $s_e = \langle \bar{e}_1, \bar{e}_2, \ldots \rangle$ such that $f(\bar{e}_i)$ is strictly increasing. Moreover, from s_e we can select an infinite strictly increasing subsequence $s_g = \langle \bar{g}_1, \bar{g}_2, \ldots \rangle$.

From s_g we select randomly two elements $\bar{g}_i$ and $\bar{g}_j$ such that $\bar{g}_i < \bar{g}_j$. Based on the construction of s_g, we know that $f(\bar{g}_i) < f(\bar{g}_j)$. It is easy to see that $\bar{h} = \bar{g}_j - \bar{g}_i > 0$ and $\sum_{p=1}^{n} \bar{h}^p \cdot \bar{v}_p = f(\bar{g}_j) - f(\bar{g}_i) > 0$. Therefore, $\sum_{p=1}^{n} \bar{h}^p \cdot \bar{v}_p$ is a non-negative solution for Inequality 5.3, which results in a contradiction.

Second, we prove `C2` $\rightarrow$ `C1` by contradiction and assume that there exists a non-negative vector $\bar{c}_0$ such that $f(\bar{c}_0) > \bar{0}$. Then we can construct an infinite sequence $\langle \bar{c}_1, \bar{c}_2, \ldots \rangle$ where $\bar{c}_i = i \cdot \bar{c}_0$. We can easily see that $f(\bar{c}_i)$ is strictly increasing and no upper bound therefore exists for $f(\bar{c}_i)$, which leads to a contradiction. □

We will show in the next section that boundedness for independent cycle systems can be automatically determined by integer linear program solving, and that the time required is only polynomial in the number of cycles in the cycle system. It is unlike the complexity at Level 3 where the problem is co-NP-hard even for a linear number of elementary cycles, which justifies for further abstraction to Level 4. This abstraction step is very significant, since for instances derived from typical CFSM systems, the number of elementary cycles is usually small (see Section 5.7). However, in the worst case the number of elementary cycles can be exponential in the size of a control flow graph. Consequently we have an exponential-time upper bound on the *worst-case* complexity of checking boundedness at this abstraction level 4.

5.4 Boundedness Test

In the previous section we have reduced the buffer boundedness problem of a CFSM system to the following problem of an independent cycle system:

`P1`: *Is there any positive linear combination of cycles?*

Such a linear combination may correspond to a combined effect of control flow cycles that sends at least one message without consuming any messages. Some message buffers will then be flooded by repeating the execution of this combination of cycles.

We now describe how to check the above problem `P1` by encoding the problem into an ILP problem as follows. Given an independent cycle system (n, C, eff) where $C = \{C_1, \ldots, C_m\}$, the sufficient and necessary condition as given in Proposition 5.4 for its *unboundedness* can be formalized as below:

$$\exists x_1, \ldots, x_m \in \mathbb{N}. \sum_{i=1}^{m} x_i \cdot \mathit{eff}(C_i) > \bar{0} \tag{5.5}$$

This can straightforwardly be transformed into the following ILP problem and solved by standard linear programming tools.

$$x_j \geq 0 \quad \text{for each} \quad 1 \leq j \leq m \tag{5.6}$$

$$\sum_{j=1}^{m} x_j \cdot \mathit{eff}(C_j)^k \geq 0 \quad \text{for each} \quad 1 \leq k \leq n \tag{5.7}$$

$$\sum_{j=1}^{m} \sum_{k=1}^{n} x_j \cdot \mathit{eff}(C_j)^k > 0 \tag{5.8}$$

In the above ILP problem, Inequality (5.6) requires all the coefficients x_j to be non-negative since any linear combination of cycles can contain only natural number coefficients. Inequality (5.7) requires only all the linear combination of cycles to be non-negative. The positivity of combinations is then enforced by Inequality (5.8).

If the above ILP problem has no solution for x_j values, then the independent cycle system is bounded. Consequently, the original CFSM system is bounded. If the ILP problem has solutions, then the independent cycle system is unbounded. However, the CFSM system is not necessarily unbounded in this case. The unboundedness could simply be due to the coarseness of the over-approximation. Thus, this test yields an answer of the form "BOUNDED" in case no positive linear combination of cycles exists, and "UNKNOWN" if such a linear combination exists.

Example 5.2. Consider the CFSM system in Figure 5.2. We have abstracted the system into the independent cycle system in Figure 5.7. We encode the unboundedness condition for the independent cycle system into the following ILP problem:

$$x_1 \geq 0 \tag{5.9}$$

$$x_2 \geq 0 \tag{5.10}$$

$$4x_1 - x_2 \geq 0 \tag{5.11}$$

$$x_1 - x_2 \geq 0 \tag{5.12}$$

$$-2x_1 + x_2 \geq 0 \tag{5.13}$$

$$3x_1 - x_2 > 0 \tag{5.14}$$

This ILP problem has no solution. As a result, both the independent cycle system and the original CFSM system are bounded. We conclude "BOUNDED".

Example 5.3. Consider System 1 from Figure 5.1. Its corresponding independent cycle system consists of three cycles, one through the state $S1$ and two through $S2$. Their effect vectors are respectively $\langle -1, 1, 1\rangle, \langle 1, 0, -1\rangle, \langle 1, -1, 0\rangle$.

From these vectors we obtain the following ILP problem to determine boundedness for the independent cycle system:

$$x_1 \geq 0 \tag{5.15}$$
$$x_2 \geq 0 \tag{5.16}$$
$$x_3 \geq 0 \tag{5.17}$$
$$-x_1 + x_2 + x_3 \geq 0 \tag{5.18}$$
$$x_1 - x_3 \geq 0 \tag{5.19}$$
$$x_1 - x_2 \geq 0 \tag{5.20}$$
$$x_1 > 0 \tag{5.21}$$

Obviously $x_1 = x_2 = 1, x_3 = 0$ is a solution to the above ILP problem. So, the independent cycle system is unbounded and we can only conclude "UNKNOWN". In the next section we will show that we may derive information of potential sources of unboundedness from any such particular ILP solutions.

Complexity. Given an independent cycle system (n, C, eff) where C contains m cycles, there are no more than $m+n+1$ inequalities in the ILP problem that represents the unboundedness condition of the system. Furthermore, the size of each inequality is bounded by m. So, the size of the ILP problem is bounded by $(m+n+1) \times m$, which is polynomial in the size of the cycle system. In Section 5.7.2 we will show that the solving of such an ILP problem is in polynomial time in spite of the NP-completeness of solving general ILP problems. This is due to a special property of ILP problems representing the unboundedness condition for independent cycle systems.

5.5 Counterexamples

When the boundedness determination ILP problem for a CFSM system has solutions, these solutions represent particular linear combinations of cycles that contribute to the unboundedness of the abstract independent cycle system. Any such linear combination is actually a counterexample to boundedness at the level of cycle systems. However, a counterexample in the cycle system does not necessarily correspond to any real execution of the original CFSM system because of the over-approximating abstraction steps taken in the boundedness test. In the following we introduce the concept of *abstract* counterexamples and their spuriousness at the level of CFSM systems, which can be straightforwardly adapted for UML RT and Promela. We will use these concepts later to guide the refinement procedure to improve the precision of the boundedness test. Note that any control flow cycle in a CFSM system corresponds to one unique cycle in the corresponding independent cycle system.

Definition 5.11 (Abstract Counterexamples of CFSM systems)**.** Given a CFSM system S, an *abstract counterexample to the buffer boundedness property* is a set $\mathcal{C}$ of control flow cycles such that the following conditions are satisfied:

- In the corresponding independent cycle system of S, there exists a positive linear combination L of cycles.

- All the cycles in the cycle system that correspond to those cycles in $\mathcal{C}$ have non-zero coefficients in the combination L.

If an abstract counterexample is *real*, then there exists an unbounded execution in which only the cycles in the counterexample are repeated infinitely often. All the other cycles are either repeated only a finite number of times, or not executed at all. Otherwise, if no such an unbounded execution exists, then the counterexample is said to be *spurious*. In the remainder of this thesis, we will simply refer to an abstract counterexample as a counterexample for convenience reasons.

Example 5.4. Consider Example 5.3 in the previous section, in which the boundedness determination ILP problem has a solution $x_1 = x_2 = 1, x_3 = 0$. From this solution, we may construct a counterexample $\{C_1, C_2\}$ where C_1 denotes the cycle through the state $S1$ and C_2 denotes the upper cycle through $S2$. By manual inspection we can easily see that this counterexample is real when there is an initialization transition leading to $S2$, along which a message `a` is sent. In such case the counterexample corresponds to the following unbounded execution of System 1: After the initialization transition is taken and a message `a` is available, C_1 is executed to consume the message `a` while sending one message `b` and one message `c`. Then, one execution of C_2 immediately follows to only consume the message `c` and send another `a`, leaving the message `b` in the message buffer. If we repeat the executions of C_1 and C_2 alternatively forever without ever executing the lower cycle through $S2$ to consume messages `b`, the message buffer containing `b` messages will be apparently flooded.

While in the above case we can easily check that the counterexample is real, it is not the case in general. There are many sources of spuriousness introduced in all the abstraction steps. We will discuss these sources in Chapter 10, and propose automated methods to determine and eliminate spurious counterexamples in Chapter 11 and in Chapter 12.

5.6 Soundness

In this section we show the soundness of the boundedness test that we described in the previous sections, by proving that the boundedness of each abstraction level implies the boundedness of the previous level.

Level 4 → Level 3. We first prove that the boundedness of an independent cycle system is a sufficient condition for the structural boundedness of the corresponding parallel-composition-VASS system.

Proposition 5.6. *Given a parallel-composition-VASS system S, if its abstract independent cycle system C is bounded, then S is structurally bounded.*

Proof. Any finite prefix of an execution of S can be decomposed into an acyclic part and a cyclic part. Since there are finitely many possible acyclic parts for execution prefixes, there is an upper bound on the effect vectors generated by acyclic parts for any initial configuration. Because C is bounded, the buffer lengths in any finite execution prefix are bounded by some vector $\bar{b}$, following Proposition 5.4. This shows that S is structurally bounded. □

Level 3 → Level 2. By Definition 5.8, it is obvious that if a VASS system is structurally bounded then the system is bounded with respect to any initial configuration.

Level 2 → Level 1. We now prove that the boundedness of VASS systems implies the buffer boundedness of CFSM systems, as stated in Proposition 5.8. We need the following auxiliary proposition in the proof.

Proposition 5.7. *Given a CFSM system S, for every finite execution of S of length k: $r = \langle c_0, \ldots, c_{k-1} \rangle$, there exists a finite execution $r' = \langle c'_0, \ldots, c'_{k-1} \rangle$ of length k, starting with no tokens in all the places, of its abstract parallel-composition-VASS system S' such that the following is satisfied:*

- *For any type of messages m, the number of messages m in c_{k-1} is equivalent to the number of tokens in the corresponding place in c'_{k-1};*
- *For any process p_i in S, its state in c_{k-1} is congruent to the state of the corresponding part of S' in c'_{k-1}.*

Proof. We prove this proposition by an induction on the length of r.

Induction base: Let $r = \langle c_0 \rangle$ in S, which is of length 1. Initially there are no messages in buffers. So, the number of messages of any type is 0 in c_0. The initial configuration c'_0 of S' has also no tokens in all the places. Moreover, for every process p_i, it is in its initial state in c_0 and its corresponding part of S' is also in the initial state in c'_0. From the isomorphism of the control flow structures of the two systems, these two initial states are congruent. The proposition trivially holds for this case.

Induction step: Let us assume that the proposition holds for any execution r of length k in S. Now consider an execution $r = \langle c_0, \ldots, c_{k-1}, c_k \rangle$ of length $k+1$ in S. By the induction assumption, for the prefix $r_k = \langle c_0, \ldots, c_{k-1} \rangle$ of r of length k, there exists a finite execution of S' of length k: $r' = \langle c'_0, \ldots, c'_{k-1} \rangle$ such that (1) the number of messages of each type in c_{k-1} is equivalent to the number of tokens in the corresponding place in c'_{k-1}; and (2) for each process p_i in S, its state in c_{k-1} is congruent to the state of the corresponding part of S' in c'_{k-1}.

According to the definition of executions of CFSM systems, we have that $c_{k-1} \Rightarrow c_k$. By the definition of the successor relation of CFSM configurations, there exists a local transition $t = (s_1, l, s_2)$ in the state machine of some process p_i such that (1) s_1 is the p_i state in c_{k-1} and s_2 is the p_i state in c_k; (2) For all processes p_j, where $i \neq j$, the p_j states are the same in c_{k-1} and c_k, i.e., no process other than p_i is executed. Because of the control flow structure isomorphism, there exists also a transition $t' = (s'_1, \bar{v}, s'_2)$ in a part p of the VASS system such that (1) p corresponds to p_i; (2) s_1 and s'_1 are congruent and s_2 and s'_2 are congruent; (3) $\bar{v}$ is the effect vector to describe the message passing behavior of t. By the definition of the successor relation for VASS systems, there exists a configuration c'_k such that $c'_{k-1} \Rightarrow c'_k$ such that c'_{k-1} and c'_k only differ in the states in the part p and the number of tokens. Moreover, s_2 is in c_k, which is congruent to s'_2 in c'_k. Consequently, for any process p_j in S, its state in c_k is congruent to the state of the corresponding part of S' in c'_k.

For any type of messages being neither sent nor received by t, its corresponding component in $\bar{v}$ is 0, thereby not changing the number of tokens in

the corresponding place in c'_k. If t receives a message m, then the component corresponding to the message type m in $\bar{v}$ is -1, thereby reducing the number of tokens in the corresponding place by 1 in c'_k. If t sends n number of messages m, then the corresponding component in $\bar{v}$ is n, thereby increasing the tokens in the corresponding place by n in c'_k. Consequently, the number of messages of each type in c_k is still equivalent to the number of tokens in the corresponding place in c'_k. □

Proposition 5.8. *Given a CFSM system S, if its abstract parallel-composition-VASS system S' is bounded with respect to the initial configuration in which there are no tokens in all the places, then S is bounded.*

Proof. By contradiction we assume that S' is bounded with respect to the initial configuration in which there are no tokens in all the places, and let b be an upper bound for the number of tokens in each place in any reachable configurations. Furthermore, we assume that S is unbounded. Then, there exists an execution r of S in which some buffer is unbounded. It is obvious that the number of at least one type of messages stored in this buffer, say m, must be unbounded in r. Furthermore, there must exist a configuration c in r in which the number of messages of type m is larger than b. Let us take the prefix of r leading to c. By Proposition 5.7, there exists a finite execution r' of S' in which the number of tokens in the place corresponding to m is equivalent to the number of messages m in c, which is larger than b. This contradicts the assumption that b is an upper bound for the number of tokens in all the VASS configurations. □

Theorem 5.9 (Soundness)**.** *Given a CFSM system, if the boundedness test returns "BOUNDED", then the system is actually bounded.*

The above theorem follows immediately the propositions 5.4, 5.6, and 5.8.

So far we only argued the soundness of the boundedness test at the level of CFSM systems. If we consider also the UML RT or Promela model at Level 0, then the soundness of our test becomes the following: Given a UML RT (or Promela) model, if the boundedness test returns "BOUNDED", then the UML RT (or Promela) model is actually bounded. Obviously the soundness at this level relies on the soundness results of all the abstraction techniques that are used to abstract UML RT (or Promela) models into CFSM systems, which will be discussed in Chapter 7 and Chapter 8 where these abstraction techniques are presented.

5.7 Complexity

In this section we analyze the complexity of the problem of checking buffer boundedness in general. It has already been mentioned in Section 5.3 that the boundedness problem is undecidable for UML RT or Promela models (Level 0), undecidable for CFSM systems (Level 1), EXPSPACE-complete for parallel-composition-VASS (Level 2), co-NP-complete for parallel-composition-VASS with arbitrary initial tokens (Level 3), and polynomial in the number of elementary cycles for independent cycle systems (Level 4). While the complexity results for Level 0 – 2 are either trivial or known from existing work, we need to prove the complexity results for Level 3 and Level 4.

5.7.1 Co-NP-completeness of the Structural Boundedness Problem of Parallel-Composition-VASS

We first prove that the structural boundedness problem is co-NP-complete, by first showing that it is co-NP-hard (Proposition 5.10), and then that it is in co-NP (Proposition 5.11).

Proposition 5.10. *The structural boundedness problem of parallel-composition-VASS is co-NP-hard.*

Proof. We will show that the structural boundedness problem is co-NP-hard even if the control flow graph of each part of the system is strongly connected and contains only polynomially many elementary cycles. The proof is by a reduction from the NP-complete boolean satisfiability problem (SAT) to the unboundedness of parallel-composition-VASS with respect to some initial configuration.

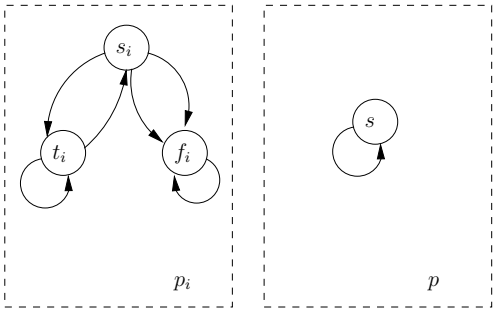

Figure 5.8: The parallel-composition-VASS system constructed from a boolean formula $\Phi := F_1 \wedge \cdots \wedge F_k$

Let $\Phi := F_1 \wedge \cdots \wedge F_k$ be a boolean formula over boolean variables $x_1, \ldots, x_n$. Each clause F_j is a disjunction of literals and each literal is either a variable or the negation of a variable.

We now construct a parallel-composition-VASS system, as shown in Figure 5.8, from Φ in polynomial time as follows: The system contains $k+2$ places, $y_1, \ldots, y_k, l, g$, where each of the first k places corresponds to one clause. For every boolean variable x_i we construct a part p_i in the VASS system with three control states s_i, t_i, u_i. The transitions from s_i to t_i, from t_i to s_i, from s_i to f_i, and from f_i to s_i have the following same effect: They reduce the number of tokens in the place l by 1 and leave all other places unaffected. The self-transition at t_i removes one token from the place g, and adds one token to the place y_j for all j such that Clause F_j contains x_i. The self-transition at f_i removes one token from g, and adds one token to p_j for all j such that Clause F_j contains $\neg x_i$. Finally, we add another part p to the VASS system with one state s. The self-transition at s has an effect $(-1, \ldots, -1, 0, n+1)$, i.e., removing one token from each of the places y_k and adding $n+1$ tokens to g. Furthermore, the initial states are $s_1, \ldots, s_n, s$. Note that each p_i and p are strongly connected and that the total number of elementary cycles in the system is $4n+1$. We now show that the constructed VASS is structurally bounded if and only if Φ is *not* satisfiable.

For the "only if" part, we assume by contradiction that the VASS system is structurally bounded and Φ is satisfiable. Then, there exists a variable assignment that makes all clauses F_j true. From such an assignment, we may construct an execution of the system as follows: First, for each part p_i, if x_i is true in the assignment, then the transition to t_i is taken; otherwise, the transition to f_i is taken. The combined effect of this is $(0, \ldots, 0, -n, 0)$. Next, each p_i executes the cycle at t_i or f_i accordingly exactly once, producing a combined effect denoted by $(e_1, \ldots, e_k, 0, -n)$. We can show that (*) $e_j \geq 1$ for all j. Next, the cycle at s in the part p is executed exactly once, producing the effect $(-1, \ldots, -1, 0, n+1)$. Note that, by executing the previously mentioned cycle combination in all processes p_i and the cycle at s together once, the combined effect is larger than $(0, \ldots, 0, 1)$, which can therefore be repeated infinitely often in our constructed execution. Obviously the execution is unbounded starting at the configuration $(0, \ldots, 0, n, n)$. This contradicts the assumption that the system is structurally bounded.

Now, we prove that the above statement (*) is true. Because each clause F_j is true, at least one literal in F_j must be true. For each F_j, we consider two cases. (1) If a literal x_i in F_j is true, then the part p_i would move to t_i and generate one token to the place y_j by executing the cycle at t_i each time. (2) If a literal $\neg x_i$ in F_j is true, then the part p_i would move to f_i and generate one token to y_j by executing the cycle at f_i each time. In both cases, we have that $e_j \geq 1$.

For the "if" part, we assume by contradiction that Φ is not satisfiable and the VASS system is unbounded with respect to an initial configuration. Note that the transitions between s_i and t_i/f_i in any part p_i can be taken only a finite number of times because the number of tokens in the place l is decreased along these transitions and never increased anywhere in the VASS system. Consequently, the execution of each p_i will eventually either terminate, or cycle only through t_i, or cycle only through f_i. Furthermore, since all the cycles at each state t_i or f_i remove a token from g, they can be repeated infinitely often only if the cycle at s in p is also executed infinitely often to add tokens to g. Moreover, because Φ is not satisfied, we can show that no combination consisting of at most one cycle at t_i or f_i from each p_i can have a positive effect on all the places $y_1, \ldots, y_k$: From any such combination, we may construct a variable assignment as follows. For each part p_i, if none of its cycles is in the combination, then x_i can take any truth value; if its cycle at t_i is in the combination, then x_i takes the value `true`; if its cycle at f_i is in the combination, then x_i takes the value `false`. Now we assume that one combination exists to have a positive effect on each place y_j. Then, for each place y_j, there exists at least one part p_i of the VASS system in which the cycle at t_i or f_i is in the combination to put one token to y_j. This means that one of x_i or $\neg x_i$ is a true literal and included in the clause F_j that corresponds to the place y_j. Consequently, every F_j is true, which makes Φ true. This contradicts that Φ is unsatisfiable. As a result, any combination of cycles at t_i of f_i has a non-positive effect, i.e., an effect 0 on at least one place, and can therefore be repeated jointly with the cycle at s only a finite number of times. Since no cycle in the VASS system can be executed infinitely often, all executions starting at any initial configuration have a finite length and the system is certainly structurally bounded. This contradicts the assumption that the system is unbounded with respect to an initial configuration. □

Proposition 5.11. *Structural boundedness of parallel-composition-VASS is in co-NP.*

Proof. We reduce the negation of the structural boundedness problem, i.e., unboundedness with respect to some initial configuration, to the NP-complete language-non-emptiness problem for reversal-bounded counter machines. These are Minsky-multi-counter machines where the counters can only switch from increasing-mode to decreasing-mode (and vice versa) a bounded number of times [70].

Given a parallel-composition-VASS system consisting of parts $p_1 \dots p_m$, we first partition the control flow graph of each p_i into strongly connected components. This can be done in polynomial time. We choose nondeterministically a strongly connected component A_i in every p_i. In each A_i we choose nondeterministically some state s_0^i as the initial state of A_i.

From the above VASS system, we obtain a reversal-bounded counter machine C as follows. C contains the same states as those states in all the strongly connected components A_i, and s_0^1 is the initial state of C. For all i we add an unlabeled transition from s_0^i to s_0^{i+1}. If the VASS system has d places, then we have $2d$ reversal-bounded counters in C such that there are two counters c_j^+ and c_j^- for every place x_j. These counters are initially 0. For each transition labeled with a vector $\bar{v}$ in the VASS system, the counters are changed in the corresponding transition in C as follows. For all components $\bar{v}^j$ of the vector $\bar{v}$, if $\bar{v}^j \geq 0$ then c_j^+ is increased by $\bar{v}^j$. If $\bar{v}^j < 0$ then c_j^- is increased by $-\bar{v}^j$. It follows that $c_j^+ - c_j^-$ represents the complete effect of the transition on the place x_j. Finally, we add an extra control state s and a transition from s_0^m to s. At state s the counter machine C checks if $c_j^+ \geq c_j^-$ for all j, and if there exists some j such that $c_j^+ > c_j^-$. It accepts an execution if and only if these conditions are satisfied. C has a polynomial size, contains a polynomial number of counters that are all reversal-bounded by 1. The one reversal is needed for the final check at the end, because one has to subtract c_j^- from c_j^+. Checking the language of C for non-emptiness is in NP, as shown in [63].

It remains to show that the parallel-composition-VASS is not structurally bounded if and only if the language of C is non-empty.

If the parallel-composition-VASS is not structurally bounded then there exists some initial configuration from which there is an unbounded execution. Any such execution is infinite and can be decomposed into an acyclic part and a cyclic part, i.e., a combination of executions of control flow cycles. The effect of this combination must have an overall non-negative effect on all places and a positive effect on at least one place. Furthermore, each cycle must be contained in some strongly connected component A_j of a part p_j. By the construction of C an unbounded execution exists if and only if C can reach an accepting state. □

Theorem 5.12. *Structural boundedness of parallel-composition-VASS is co-$\mathcal{NP}$-complete.*

The above theorem follows immediately Proposition 5.10 and Proposition 5.11.

5.7.2 Polynomial Time Complexity of the Boundedness Problem of Independent Cycle Systems

Given an independent cycle system (n, C, eff), an ILP problem (Inequalities 5.6–5.8) that represents the sufficient and necessary condition for its unboundedness can be built in time polynomial in the size of the cycle system. In general, solving ILP problems is NP-complete. However, any ILP problem generated in our test to determine boundedness has a special property: For each inequality in the ILP problem, the left-hand side does not contain any constant items, and the right-hand side is 0. Such an ILP problem is called a *homogeneous* ILP problem. A homogeneous linear programming problem has the following property: Given any rational solution $(v_1, \ldots, v_n)$ to the problem and any positive rational number t, $(t \cdot v_1, \ldots, t \cdot v_n)$ is still a solution to the problem. This is because every inequality in a homogeneous linear programming problem can be transformed into the form $\sum_i x_i \cdot e_i \geq 0$ (or $\sum_i x_i \cdot e_i > 0$). Given any value v_i of x_i, if $\sum_i v_i \cdot e_i \geq 0$ (or $\sum_i v_i \cdot e_i > 0$, respectively), then $t \sum_i v_i \cdot e_i \geq t \cdot 0 = 0$ (or $t \sum_i v_i \cdot e_i > t \cdot 0 = 0$, respectively) for any positive rational number t. By this property, we may turn the ILP problem into a linear programming problem by allowing all the variables x_i to have rational values. Then, we can solve the linear programming problem to obtain rational solutions for x_i, which is known to be in polynomial time [105]. Next, we compute the least common denominator d of all the x_i values in the rational solution, which is also in polynomial time [40]. We can get an integer value for each x_i by multiplying the rational solution of x_i with d. All the obtained integer values for x_i give an integer solution to the problem.

From the above argument, we know that the complexity of the boundedness problem of an independent cycle system is polynomial in the size of the cycle system, i.e., the number of elementary control flow cycles in all processes of the original CFSM systems. In the worst case, the number of elementary cycles in a CFSM process can be exponential in the size of the process. Consequently, the worst-case complexity of the boundedness problem of independent cycle systems is exponential in the size of each process. This is also the worst-case complexity of our boundedness test. It should be noted that this is still normally much smaller than the complexity of the semi-complete boundedness test by constructing reachability graphs, which is triply exponential as already shown in Section 5.1.2. Moreover, the control flow graphs derived from realistic models of asynchronous reactive systems are normally very sparse, and the number of elementary control flow cycles in them is normally polynomial, rather than exponential. Therefore, the boundedness test requires only polynomial time in practice.

5.8 Estimating Buffer Bounds

A more refined problem is to compute upper bounds on the lengths of individual buffers in the system. In particular, some buffers might be bounded even if the whole system is unbounded. Since normally not all buffers can reach maximal length simultaneously, the analysis is done individually for each buffer B.

For a bounded CFSM system, the least upper bound of a buffer B, i.e., the maximal number of messages stored in B, can be obtained by simply explor-

ing the reachability graph of the system in order to find those configurations in which the length of B reaches the maximum. This approach is sound and complete with respect to bounded systems because the reachability graph is finite. However, since the reachability graph is exponentially large, this approach has an exponential complexity and is thus not practical. Furthermore, the approach by reachability graph exploration becomes incomplete when considering unbounded systems. Consequently, it does not work when we try to determine the upper bound for a bounded buffer in an unbounded system. In the following we propose another approach also based on integer linear program solving, which delivers coarser upper bounds for buffers but is more efficient and works for both bounded and unbounded CFSM systems.

Consider an arbitrary CFSM system and a type of messages m exchanged in the system. Given a finite execution of the system r_f, we use $n_m(r_f)$ to denote the number of messages m in the end of r_f. It is easy to see that $lub_m = max\{n_m(r_f) \mid r_f \text{ is a finite execution}\}$ is the least upper bound on the number of messages m that may ever occur in the system at any point of runtime. Let $m_1, \ldots, m_k$ be the types of messages stored in a buffer B. The number of messages stored in a buffer B is then bounded by $ub_B = \sum_{i=1}^{k} lub_{m_i}$. Note that ub_B is not necessarily the least upper bound because the numbers of messages of all types normally do not reach the maximums at same time.

Note that any finite execution r_f can be decomposed into a cyclic part r_f^c and an acyclic part r_f^a starting from the initial configuration of the system. The cyclic part r_f^c can be further decomposed into a linear combination of executions of control flow cycles in each process. The acyclic part r_f^a can be projected into a set of acyclic paths p_f^a in each individual process p from the respective initial local state. Let $\mathit{eff}(r)$ denote the overall message passing effect, denoted by an effect vector, generated by a finite execution r. Let x_i to denote the number of times a cycle c_i is repeated in r_f^c. Then, we have that

$$\mathit{eff}(r_f^c) = \sum_i x_i \cdot \mathit{eff}(c_i) \tag{5.22}$$

and

$$\mathit{eff}(r_f^a) = \sum_p \mathit{eff}(p_f^a) \tag{5.23}$$

Assume that the message type m corresponds to the j-th component in effect vectors. Then, we have that $n_m(r_f) = \mathit{eff}(r_f^c)^j + \mathit{eff}(r_f^a)^j$. Furthermore, $lub_m = max\{\mathit{eff}(r_f^c)^j + \mathit{eff}(r_f^a)^j \mid r_f \text{ is a finite execution}\}$. However, it is certainly infeasible to compute lub_m by checking all possible finite executions, since they are exponentially many for boundedness systems and infinitely many for unbounded systems. In the following we compute an over-approximating bound $ub_m \geq lub_m$. The over-approximation relies on the following two properties. In particular, the first property eliminates the need of considering an exponential number of combinations of acyclic paths and control flow cycles, at the cost of obtaining coarser results, i.e., larger buffer bounds.

Proposition 5.13. *The following properties trivially hold:*

1. $lub_m = max\{\mathit{eff}(r_f^c)^j + \mathit{eff}(r_f^a)^j\} < max\{\mathit{eff}(r_f^c)^j\} + max\{\mathit{eff}(r_f^a)^j\}$;

2. For any finite execution r_f, $n_m(r_f) \geq 0$.

In the above proposition, the first property gives a larger upper bound for lub_m as a result of abstracting from the dependences between the acyclic part and the cyclic part of finite executions. The second property actually gives a trivial fact about any possible finite executions: in the end of any finite execution there cannot be a negative number of messages of any type. In the ILP problem that we construct to estimate buffer bounds, this fact will be used as the constraints of the optimization. Now, the problem becomes to compute $max\{\mathit{eff}(r_f^c)^j\} + max\{\mathit{eff}(r_f^a)^j\}$. Let the vector $lub^a = (max\{\mathit{eff}(r_f^a)^1\}, \ldots, max\{\mathit{eff}(r_f^a)^q\})$ if there are q message types. Obviously lub^a denotes the least upper bound on the maximal contribution of all the acyclic paths, and it can be precisely computed in finite time by checking all the acyclic paths in all the processes (See Equation 5.23). This is even true for unbounded systems because there are always finitely many acyclic paths in any finite control flow graph. Since the number of acyclic paths in a process is exponential in the size of the process, the complexity to compute lub^a is also exponential. However, we will show in Chapter 13 that the computation of the maximal contribution of acyclic paths can be made a byproduct of the algorithm for cycle detection that we use in our prototype implementation. Therefore, it hardly costs extra overhead in our boundedness analysis. Furthermore, we again abstract from message orders, activation conditions of cycles, and cycle dependencies to allow any combination of cycles. In this way we can optimize the maximal contribution of cyclic parts of executions $max\{\mathit{eff}(r_f^c)^j\}$ by optimizing the overall effects of linear combinations of cycles. Based on the above discussion, we can now encode the computation of an upper bound for the number of messages m into the following ILP problem:

$$max: \quad max\{\mathit{eff}(r_f^a)^j\} + \textstyle\sum_i x_i \cdot \mathit{eff}(c_i)^j \tag{5.24}$$

$$\mathit{constraints}: \quad lub^a + \textstyle\sum_i x_i \cdot \mathit{eff}(c_i) \geq 0 \tag{5.25}$$

$$x_i \geq 0 \quad \text{for all } i \tag{5.26}$$

The above ILP problem is to maximize the objective function (5.24), which represents the upper bound $ub_m = max\{\mathit{eff}(r_f^c)^j\} + max\{\mathit{eff}(r_f^a)^j\}$, under the constraints stating that there cannot be a negative number of messages at any time. The constraints (5.25) are actually a weaker condition than the second property in Proposition 5.13, which is again an over-approximation making the computed bound coarser. The constraints (5.26) simply restrict all x_i variables to be non-negative since there cannot be a negative number of executions of any cycle.

Given a buffer B in which the types of messages are $m_1, \ldots, m_k$, we already know that an upper bound for the length of B is $ub_B = \sum_{i=1}^{k} lub_{m_i}$. By over-approximating the bound of the number of messages of each type allowed in B, we obtain also a larger upper bound for B as $ub_B^+ = \sum_{i=1}^{k} ub_{m_i} > ub_B$ by summing all the bounds for message types allowed in B. However, there is no need to compute bounds for these message types separately so that n ILP problems have to be solved. Instead we can easily modify the objective function (5.24) to encode the sum of the upper bounds of all message types as below and solve only one ILP problem for B's bound. Let us assume that the message

types allowed in B correspond to the effect vector components whose indices are in a set I.

$$max: \quad \sum_{k\in I} max\{\mathit{eff}(r_f^a)^k\} + \sum_{k\in I}\sum_i x_i \cdot \mathit{eff}(c_i)^k \tag{5.27}$$

Example 5.5. Having established boundedness for the CFSM system in Figure 5.2, we now compute the estimated upper bound for each buffer. First we compute the effect vectors for all acyclic paths of each process. They are listed in table 5.1. The first three rows contain all the acyclic paths in the left process, and the other rows contain all the acyclic paths in the right process.

Acyclic path	Effect vector	Acyclic path	Effect vector
$\langle s_0\rangle$	$(0,0,0)$	$\langle s_0, s_1\rangle$	$(0,0,0)$
$\langle s_0, s_1, s_2\rangle$	$(0,2,-1)$	$\langle s_0, s_1, s_2, s_3\rangle$	$(2,3,-2)$
$\langle s_0, s_1, s_3\rangle$	$(0,5,-1)$	$\langle s_0, s_1, s_3, s_2\rangle$	$(2,5,-2)$
$\langle s_0'\rangle$	$(0,0,0)$	$\langle s_0', s_1'\rangle$	(0,0,2)
$\langle s_0', s_1', s_2'\rangle$	(-1,0,2)		

Table 5.1: The effect vectors for all acyclic paths in the CFSM system in Figure 5.2

It is easy to see that we can first compute the least upper bound for the effect vectors of acyclic paths in each individual process, and then sum all these least upper bounds to obtain the least upper bound lub^a for the whole system. In our example, the least upper bound for the left process is $(2,5,0)$, and the one for the right process is $(0,0,2)$. So, $lub^a = (2,5,0) + (0,0,2) = (2,5,2)$. Then, we obtain the following ILP problem to compute the upper bound for the buffer B_1 that stores messages of types a and b.

$$max: 7 + 5x_1 - x_2 \tag{5.28}$$
$$2 + 4x_1 - x_2 \geq 0 \tag{5.29}$$
$$5 + x_1 - x_2 \geq 0 \tag{5.30}$$
$$2 - 2x_1 + x_2 \geq 0 \tag{5.31}$$
$$x_1 \geq 0 \tag{5.32}$$
$$x_2 \geq 0 \tag{5.33}$$

And, the following ILP problem is to compute the upper bound for the buffer B_2 that stores messages of type c.

$$max: 2 - 2x_1 + x_2 \tag{5.34}$$
$$2 + 4x_1 - x_2 \geq 0 \tag{5.35}$$
$$5 + x_1 - x_2 \geq 0 \tag{5.36}$$
$$2 - 2x_1 + x_2 \geq 0 \tag{5.37}$$
$$x_1 \geq 0 \tag{5.38}$$
$$x_2 \geq 0 \tag{5.39}$$

Linear Programming returns a value of 18 for the objective function (5.28) and a value of 6 for the objective function (5.34). So, B_1 can never contain more than 18 messages at runtime, and B_2 can never contain more than 6 messages at

runtime. By manual inspection that took effort, we determined that the actual bounds for B_1 and B_2 are respectively 7 and 2. The buffer bounds estimated by our methods are much larger than the actual bounds. We may see that the coarseness in this example is largely due to the abstraction from message orders. If we allow the message buffers used in the system to be pseudo-first-in-first-out, i.e., any message can be received as long as it is in the buffer, then the actual bounds become respectively 13 and 4 to which our estimation is much closer.

Complexity. For each message type m, the ILP problem constructed to compute an upper bound for the number of m messages has a size polynomial in the size of the abstract independent cycle system corresponding to the original CFSM system, which is the same as for boundedness determination ILP problems. More precisely, if there are n cycles and k message types, the size of any upper bound estimation ILP problem is bounded by $(n+1)\cdot(k+n+1)$. However, these upper bound estimation ILP problems may not be homogeneous since the left hand side of each inequality has a constant item $max\{\mathit{eff}(r_f^a)^j\}$ and this item may not be 0. Consequently, the solving of these problems is NP-complete, which is in contrast to the polynomial complexity for boundedness determination ILP problems. However, we observed in practice that buffer bounds can still be computed efficiently for realistic models as will be shown in the experimental results.

5.9 A Comparison with the Brand and Zafiropulo's Boundedness Test

In Appendix C of [25] Brand and Zafiropulo proposed a buffer boundedness test that we will refer to as the BZ test. It bears similarities to our test. In this section we compare the two tests with regard to their complexities, the coarseness of the computed buffer bounds, and the counterexample construction.

The BZ Test. The BZ test determines boundedness for a CFSM system by computing an upper bound for the number of messages of each individual type that may occur at any point of an execution. If an upper bound can be obtained for all message types, then the system is determined to be bounded. Otherwise, an "UNKNOWN" result is returned since the test is certainly incomplete. The computation of buffer bounds is similar to our estimation method described in the previous section. It first makes use of the same series of abstractions as in our test, i.e., abstracting from message orders, the activation condition of cycles, and cycle dependences. However, these abstractions are not explicitly presented in [25], and the complexity results for the abstract model of each intermediate level are not given either. In the next step in the BZ test, an ILP problem as follows is given to compute an upper bound for each message type m that corresponds to the j-th component in the effect vectors that the BZ test produces. The notations appearing in this ILP problem are interpreted in the same way as for the ILP problem given in (5.24– 5.26) except the newly introduced symbol ub^a as the upper bound on the acyclic path effects that the

BZ test obtains.

$$max: \quad ub_j^a + \sum_i x_i \cdot \mathit{eff}(c_i)^j \tag{5.40}$$

$$\mathit{constraints}: \quad ub^a + \sum_i x_i \cdot \mathit{eff}(c_i) \geq 0 \tag{5.41}$$

$$x_i \geq 0 \quad \text{for all } i \tag{5.42}$$

We can see that the only difference between the above ILP problem and the ILP problem constructed in our estimation is in the computation of acyclic path effects. For each message type m, the BZ test generates a larger upper bound ub_j^a for the number of m messages contributed by acyclic paths as follows: it collects all the local transitions in the system which send messages m. The collection only needs to scan the syntactic definition of the system and requires therefore only a runtime linear in the size of the system. It will then count the total number of messages being sent by these transitions and this total number is used as ub_j^a in the above ILP problem. It is easy to see that $ub_j^a \geq max\{\mathit{eff}(r_f^a)^j\}$. Consequently, the upper bound generated in the BZ test for the effect vectors of all acyclic paths is $ub^a = (ub_1^a, \ldots, ub_q^a) \geq lub^a$. As a result, the upper bound b_1 generated by the above ILP problem is no smaller than the upper bound b_2 computed in our test. In fact b_1 is usually much larger than b_2 as the following example illustrates.

Example 5.6. Consider the CFSM system in Figure 5.2, for which we have computed upper bounds for all the buffers as 18 and 6 respectively using our boundedness test. Now we use the BZ test to compute buffer bounds and compare the results to our estimation.

We first compute ub_j^a for each message type such as a, b, and c. For the type a that corresponds to the 1-st component in effect vectors, there are two transitions that send a messages: from s_2 to s_3 and from s_3 to s_2. Each transition sends 2 messages of type a and therefore $ub_1^a = 4$. Similarly, we get that $ub_2^a = 8$ and $ub_2^a = 3$. Recall that in out analysis the upper bounds for the number of messages of each type as generated by acyclic paths are respectively 2, 5, and 2. The ILP problems to determine buffer bounds are now

$$max: 12 + 5x_1 - x_2 \tag{5.43}$$

$$4 + 4x_1 - x_2 \geq 0 \tag{5.44}$$

$$8 + x_1 - x_2 \geq 0 \tag{5.45}$$

$$3 - 2x_1 + x_2 \geq 0 \tag{5.46}$$

$$x_1 \geq 0 \tag{5.47}$$

$$x_2 \geq 0 \tag{5.48}$$

and

$$max: 3 - 2x_1 + x_2 \tag{5.49}$$

$$4 + 4x_1 - x_2 \geq 0 \tag{5.50}$$

$$8 + x_1 - x_2 \geq 0 \tag{5.51}$$

$$3 - 2x_1 + x_2 \geq 0 \tag{5.52}$$

$$x_1 \geq 0 \tag{5.53}$$

$$x_2 \geq 0 \tag{5.54}$$

The buffer bounds produced by the above ILP problems are respectively 38 and 9, which are much larger than our estimation as 18 and 6.

Complexity. We have shown that the complexity of boundedness determination of our test is polynomial in the number of elementary cycles. On the contrary the BZ test has an exponential complexity because it determines boundedness by computing buffer bounds, unlike our test that determines boundedness without the need of obtaining buffer bounds for each buffer. However, the complexity of computing buffer bounds is NP-complete for the both tests.

Buffer Bounds. We have shown both theoretically and using an example that the buffer bounds generated by the BZ test are no smaller and usually much larger than the bounds generated by our test. A smaller upper bound for a message buffer is certainly desired in practice. For example, in the system that implements the CFSM model, the data structure used to realize a specific message buffer may have a size that is equal to the estimated buffer bound for the buffer, without any danger of losing messages or a buffer overflow occurring. A smaller buffer bound then reduces the runtime memory needed by the actual system.

Counterexample Generation. We have shown that our test may directly construct a counterexample to boundedness from any specific solution to the boundedness determination ILP problem, by collecting all the cycles whose coefficients are not zero in the solution. However, it is not the case for the BZ test: Any generated ILP problem is to compute buffer bounds, and if there exists any solution then the solution corresponds to bounded executions. When the number of messages of the respective type is unbounded, the ILP problem has no solutions. In either case, no evidence of unboundedness can therefore be straightforwardly recovered.

Chapter 6

Checking Livelock Freedom

We show in this chapter another application of our verification framework for asynchronous reactive systems – the checking of the livelock freedom property.

The main characteristic of a reactive system is that of maintaining an ongoing activity of exchanging and processing information. One salient property that any correct reactive system must satisfy is deadlock freedom, i.e., the execution of the system is non-blocking. However, a system may be free of deadlock and yet it does no progress in executing its tasks. For example, two components of a system may keep exchanging internal messages with each other and never respond to the outside world. Such a situation is referred to as livelock. The freedom from livelock is highly desirable as it is important to ensure that the execution of a system is not only continuous but also meaningful.

The livelock freedom property of finite state systems are mostly checked by explicit state model checking techniques that rely on the complete enumeration of global states of a system [47, 68, 64]. However, since an asynchronous reactive system usually possesses a large or even infinite global state space, these techniques suffer from the state explosion problem and are incapable to determine livelock freedom if the system has an infinite state space.

Based on ILP solving, we propose a verification method for the livelock freedom property of asynchronous reactive systems whose state spaces can be either finite or infinite. The observation behind the method is that control flow cycles play a central role in the "forever execution" behavior of reactive systems. In the following ways, our method is similar to the buffer boundedness test that we described in the previous chapter:

- Both methods use the same sequence of conceptual abstraction steps which results in an independent cycle system as the final abstraction of the original system (see Section 6.2.1). Therefore, both methods may use the same techniques for the abstraction of Promela and UML RT models and for the abstraction refinement procedure. Similarly as for the buffer boundedness test, we explain the livelock freedom verification method at the level of CFSM systems.

- The livelock freedom verification method also reduces the livelock freedom problem to a homogeneous ILP problem that can be solved in polynomial time (see Section 6.4).

- The livelock freedom verification method is incomplete since livelock freedom is undecidable for CFSM systems (see Theorem 6.1 and its proof in Section 6.1).

Structure. Section 6.1 formally defines the livelock freedom problem, while Section 6.2 and Section 6.3 present the core idea of checking livelock freedom. Sections 6.4 and 6.5 discuss respectively the soundness and the complexity results of the proposed testing method.

6.1 Livelock Freedom

Livelock has been defined variously in different contexts [67]. For concurrent systems, such as asynchronous reactive systems, livelock often means "individual starvation": a process is prevented from performing some particular actions [89]. These actions are normally intended to make progress, deliver outputs, or respond to the environment. We call such an action a *progress action*. For a reactive system, if only non-progress actions are executed after a certain point of the execution, i.e., all the progress actions are repeated only a finite number of times, then we say that the system is in a livelock situation.

We formally define livelock and livelock freedom for CFSM systems as follows.

Definition 6.1 (Livelock Freedom of CFSM Systems)**.** Given a CFSM system, we identify a set of local transitions as *progress transitions*. A *livelocked execution* is an infinite execution in which all the progress transitions are taken only a finite number of times. The system is *free of livelock* if no livelocked executions exist.

The definition above makes a clear distinction between deadlock and livelock, which are two different concepts even though they both result in a lack of progress in the system and are sometimes not distinguished from a practical point of view. However, these two concepts correspond to two different causes of non-progress of CFSM systems: A deadlock situation is where all the processes wait for inputs that will never come, or wait for certain conditions to come true which will however never happen. On the contrary, a livelock situation is where all processes, if still executing, decide to ignore forever those possibilities to make a progress. So, a deadlocked execution is always a finite execution while a livelocked execution is an infinite one. By enforcing in our definition that a livelocked execution must be infinite, we exclude any deadlocked execution to be a livelocked execution. In particular, if our method proves a system to be free of livelock, then it may still have deadlocks. Our distinction between deadlock and livelock is further attested by the fact that deadlock freedom and livelock freedom belong to two different classes of properties. Deadlock freedom is a safety property and livelock freedom is a liveness property. More precisely, livelock freedom is a fairness property. As a consequence, the techniques used to check these two properties are radically different. For instance, the SPIN model checker distinguishes in a similar fashion between deadlock and livelock [68]. In Promela models, "progress" labels are attached to progress actions. SPIN then checks livelock freedom by checking the absence of non-progress global cycles by a nested depth first search in the global state space. However, such a state

enumeration approach suffers from the state explosion problem and can only deal with finite state systems.

Example 6.1. Consider the CFSM system in Figure 6.1. The system describes a simple client-server model and consists of three processes: two client processes $client_1$ and $client_2$, and one server process $server$. Each client $client_i$ repeatedly requests a resource (req), waits for an acknowledgment (ack) from the server, and releases the resource (rel). The server can accept only one request at a time, and it chooses nondeterministically a request to handle. Each process has two exclusive message buffers to communicate with the server.

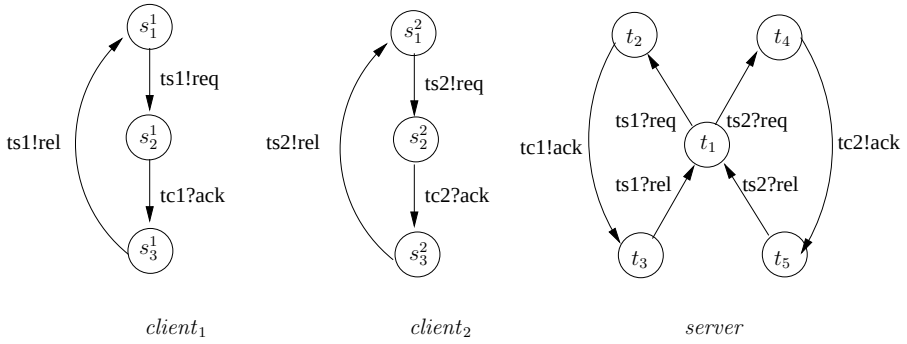

Figure 6.1: A client-server CFSM system

If we want to guarantee that the client $client_1$ can always receive acknowledgments from the server, then we identify the set of progress transitions to be $P = \{s_2^1 \rightarrow s_3^1\}$, i.e., the only progress transition is the one where $client_1$ receives an acknowledgment. Consider one execution in which the server accepts requests from $client_1$ only a finite number of times. If the execution that we consider is infinite, then the server will eventually accepts only requests from the other client $client_2$. Such an execution leads certainly to a livelock with regard to the set of progress transitions that we choose. If no such execution exists, then the server has to accept requests from both clients infinitely often and the system is free of livelocks. The goal of our test is to determine automatically whether the system is livelock free.

We can prove that livelock freedom is undecidable for CFSM systems. Consequently, no complete algorithm exists to determine livelock freedom for all CFSM systems. This implies that our test is also incomplete.

Theorem 6.1. *It is in general undecidable whether a CFSM system is livelock free.*

Proof. The proof uses a simple reduction from the undecidable message reception executability problem (Problem 1 in Section 3.5) of CFSM systems. Consider a CFSM system S and a local state s of S. There is an outgoing transition t from s, which is labeled by a message receiving event $b?m$. We construct another CFSM system S' from S such that an execution of S' becomes a livelocked execution after the transition t is taken, i.e., after the reception $b?m$ at s occurs. More precisely, S' is obtained from S by changing the target

state of t to a newly introduced local state s' that has a self-transition with an empty label. If s' is reached, then S' can do nothing else but repeatedly execute the always enabled self-transition at s'. In S' we set all transitions as progress actions except the self transition at s'. We show in the following that the $b?m$ can be executed at State s of S if and only if S' has a livelocked execution.

For the "if" part, if S' permits a livelocked execution r, then the state s' must be reached in r. Let us take the finite prefix of r leading to the first occurrence of s'. We call it r'. From the construction of S' from S, it is obvious that r' is also a finite execution of S. Moreover, $b?m$ at s is the last event on r'. So, $b?a$ is executable in S at s.

For the "only if" part, since $b!m$ is executable at s, let r' be a finite execution of S leading to the execution of $b?a$ at s. Then, we can simulate the same finite execution r' in S' and reach the state s'. At this point we obtain a livelocked execution r by repeating the self-transition at s' forever, which is the only non-progress transition of S'. □

6.1.1 Existing Verification Techniques for Livelock Freedom

We review some representatives of existing verification techniques for livelock freedom. As mentioned previously, there are various definitions of livelock. Here, we concentrate only on those methods that deal with a similar notion of livelock freedom to our definition.

Model Checking

In practice, the verification of livelock freedom for finite state systems are mainly tackled by explicit state model checking techniques that generally outperform symbolic model checking methods [47] for this problem.

In explicit state model checking, the basic approach to livelock freedom verification is to detect those cycles in the global state space that consist of non-progress transitions. It requires first the construction of the whole global state space of a system. The detection of cycles can then be done by a nested depth first search whose complexity is linear in the size of the state space, as implemented in [68]. However, the size of the state space is usually exponential in the size of the original system.

[64] improves the efficiency of checking livelock freedom by constructing a stuttering insensitive extended Büchi automaton, called a testing automaton, that represents all the livelocked behavior. Such a testing automaton is usually smaller than the normal automaton constructed to represent livelocked behavior, and therefore reduces the size of the global state space. Furthermore, a single depth first search can be used to explore the global state space in order to detect any livelocked execution. It further reduces the complexity of livelock freedom checking. However, it still relies on the construction of a global state space which is exponential in the size of the system even if a testing automaton is used.

Process Algebra

The concept of livelock in Communicating Sequential Processes (CSP) is similar to ours. Livelocked executions are interpreted as divergences, i.e., traces in which a process will eventually perform only internal actions and never engage in a visible event. [103] gives a model checking approach to check divergence freedom for systems with a finite number of states. This is the theory of livelock checking behind the FDR (or FDR2) model checker [56, 29]. Given a CSP process, the proposed method first constructs a labeled transition system from the process, and then detects any cycle consisting of only internal actions. If no such cycle can be found, then the system is free of divergences. The detection of cycles can be done by a nested depth first search whose complexity is linear in the size of the checked transition system. However, the size of the labeled transition may be exponential in the size of the original process. Moreover, this method fails on infinite state CSP processes whose livelock freedom problem is undecidable. Note furthermore that our analysis focuses on asynchronous systems, which are difficult to be modeled concisely in CSP.

Petri Nets

[31] checks livelock freedom for concurrent Ada programs. These are synchronous systems and supposed to terminate if correctly designed and implemented. Therefore, any infinite execution of an Ada program is a livelocked execution. It corresponds to a situation in which the executions of some components (i.e., Ada tasks) are never completed while some groups of components continuously communicate internally. The concept of livelock freedom for Ada programs differs from our definition only in that progress actions[1] are required to be executed only once, not infinitely often. So, livelock freedom for Ada programs is not a fairness property.

[31] proposes a verification method for checking the above mentioned notion of livelock freedom for concurrent Ada programs. An Ada program is first transformed to a Petri net. The places in the Petri net represent the local control points of all the components of the program, and the transitions represent the control flow and the synchronization communication of the components. The effects of transitions on tokens in the places can be represented as an incidence matrix[2] D. Given any vector X that represents the linear combination of the number of times that each transition is fired, if $XD = \bar{0}$ has no solutions, then no firing sequence can be repeated forever and the system must be either terminating or deadlocked. Note that $XD = \bar{0}$ is a homogeneous ILP problem. Consequently, the approach has a low complexity linear in the size of the Petri net into which the original program is transformed. However, it is also incomplete and coarse as it abstracts from orders of transition firings. It is also not clear from [31] how efficient and precise the approach works in practice.

[1] Usually the termination of an Ada program is the only progress action.

[2] The input of a transition is the number of tokens in each place that the transition consumes to be fired. The output of a transition is the number of tokens in each place that the transition adds after being fired. Let the matrix D^- denote the inputs of all the transitions and D^+ denote the outputs of all the transitions in a Petri net. Then, the incidence matrix $D = D^+ - D^-$ represents the effects of all the transitions on the number of tokens in each place.

6.2 Overview of the Livelock Freedom Test

We propose an incomplete method to prove livelock freedom for asynchronous reactive systems based on ILP solving. The incompleteness follows from the undecidability of livelock freedom as stated in Theorem 6.1.

We outline the method as follows. Given a reactive system and a set of predetermined progress actions, we first carry out a series of abstractions, the same as the abstraction steps in the boundedness test, that transforms the system into a set of independent control flow cycles labeled with their message passing effects. A cycle is a *progress cycle* if it contains one of the progress transitions, and we identify the set of all the local progress cycles in the system. We give a *necessary condition* which ensures the existence of a livelocked execution, i.e., an infinite execution in which all the progress cycles are repeated only a finite number of times. We translate this condition into a homogeneous ILP problem. If the resulting ILP problem has no solution then the necessary condition cannot hold, which implies livelock freedom. On the other hand, if the resulting ILP has solutions then the system may or may not be livelock free, which corresponds to the incomplete side of our test.

6.2.1 Abstraction

In asynchronous reactive systems, concurrent processes coordinate their actions by exchanging messages. Thus, the message passing behavior is a major factor to decide how cycles in the control flow are executed. This observation underlies the conservative abstraction approach for our livelock freedom analysis, which results in the same abstraction steps used in our previous work on buffer boundedness analysis. Given the program code of a reactive system, we abstract from arbitrary program code, message orders, activation conditions of cycles, and cycle dependencies. The resulting system is an independent cycle system in which each cycle has a summary effect vector. Since the abstraction steps were already explained in the previous chapter, we leave out the details here. We also omit the complexity results for each intermediate models during the abstractions.

In the example system shown in Figure 6.1, there are 4 cycles: one from the process $client_1$, one from $client_2$, and two from $server$. Their effect vectors are respectively $\langle -1, 1, -1, 0, 0, 0\rangle$, $\langle 0, 0, 0, -1, 1, -1\rangle$, $\langle 1, -1, 1, 0, 0, 0\rangle$, and $\langle 0, 0, 0, 1, -1, 1\rangle$.

6.3 Livelock Freedom Test

A reactive system is livelock free if at least one progress cycle can be repeated infinitely often in any infinite execution. Let $C_1, \ldots, C_n$ be the set of control flow cycles in the corresponding independent cycle system, and $C_{j_1}, \ldots, C_{j_m}$ ($j_1, \ldots, j_m \in \{1, \ldots, n\}$) be the set of progress cycles. We use the following ILP problem to characterize a necessary condition for the existence of a livelocked execution, i.e., an infinite execution in which any progress cycle can be repeated only a finite number of times.

$$x_i \geq 0 \quad \text{for each } i \tag{6.1}$$

$$\sum_{i=1}^{n} x_i \cdot \mathit{eff}(C_i) \geq \bar{0} \tag{6.2}$$

$$\sum_{i=1}^{n} x_i > 0 \tag{6.3}$$

$$\sum_{i=1}^{m} x_{j_i} = 0 \tag{6.4}$$

In the above inequalities, each integer variable x_i denotes the number of times that the cycle C_i is repeated in a linear combination of cycles. These variables may have only non-negative values as imposed by the inequalities (6.1). The inequality (6.2) requires a linear combination of cycles to consume no messages. Thus, an infinite repetition of such a combination is possible since it does not run out of any type of messages. The inequality (6.3) excludes a trivial combination in which no cycle is executed at all. The inequalities (6.2) and (6.3) together give a necessary condition for the existence of infinite executions. The inequality (6.4) then excludes any progress cycle C_{j_i} from a linear combination. Consequently, this condition excludes any progress cycle from being repeated infinitely often in any infinite execution. The arguments in Section 6.4 ensure that the ILP problem defined by the inequalities (6.1–6.4) gives indeed a necessary condition for the existence of livelocked executions.

If the ILP problem has no solutions, then the necessary condition cannot hold. In such a case, at least one progress cycle C_{j_i} has to be repeated infinitely often in any infinite execution. This proves livelock freedom for the system. On the other hand, if the ILP problem has solutions, then we do not know whether the system is livelock free since the ILP problem gives a necessary but not sufficient livelock existence condition.

In the system in Figure 6.1, let the only progress action be the local computation of one client, say client_1. We use x_1 to correspond to the cycle in client_1, x_2 to the cycle in client_2, and x_3 and x_4 to the two cycles given as the two nondeterministic choices within the `do` loop in *server*. The resulting livelock freedom determination ILP problem is given as below.

$$-x_1 + x_3 \geq 0 \tag{6.5}$$

$$x_1 - x_3 \geq 0 \tag{6.6}$$

$$-x_1 + x_3 \geq 0 \tag{6.7}$$

$$-x_2 + x_4 \geq 0 \tag{6.8}$$

$$x_2 - x_4 \geq 0 \tag{6.9}$$

$$-x_2 + x_4 \geq 0 \tag{6.10}$$

$$x_1 + x_2 + x_3 + x_4 > 0 \tag{6.11}$$

$$x_1 = 0 \tag{6.12}$$

$$x_1, x_2, x_3, x_4 \geq 0 \tag{6.13}$$

The inequalities (6.5–6.10) restrict the aggregate effect vector of a linear

combination to be positive. The inequality (6.11) excludes an all-zero combination. The inequality (6.12) excludes the only progress cycle in $client_1$. There is one solution satisfying these inequalities: $x_2 = x_4 = 1$ while assigning 0 to all other variables. As a consequence we cannot prove livelock freedom for the example system. However, we show in the following that we may retrieve the information of potential livelocked executions from a particular solution to the livelock freedom determination ILP problem.

Recall that we defined the concept of a counterexample to the buffer boundedness property for CFSM systems (see Definition 5.11 in Section 5.5). Since we use the same abstraction steps for the livelock freedom test as for the boundedness test, we can directly reuse Definition 5.11 to describe counterexamples in the livelock freedom analysis, i.e., from a solution to the livelock freedom determination ILP problem, we construct a counterexample to be a set of cycles whose corresponding variable receives a nonzero value in the solution. In the above mentioned example, we can construct a counterexample that consists of the cycle in $client_2$ and the right cycle in *server*. A manual check of the counterexample reveals a real livelock scenario in which the server decides to accept only requests from $client_2$. So, the counterexample that we found for the example system is real. However, not all the counterexamples constructed from our test are necessarily real, due to the over-approximation of the abstractions that the test makes use of. It is also hard in general to determine manually the spuriousness of a counterexample. We will describe a common automated counterexample spuriousness determination method for both the boundedness test and the livelock freedom test in Chapter 12.

To eliminate the source of livelock that we uncovered above, we modify the model by removing the nondeterministic behavior of the server. We fix an order in which the server alternatively handles requests from the two clients as shown in Figure 6.2.

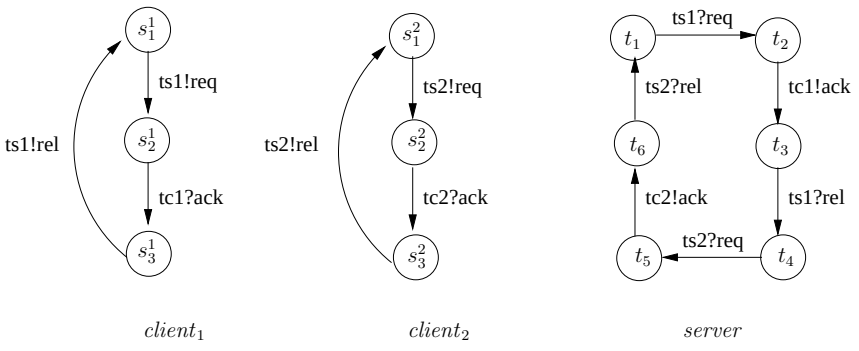

Figure 6.2: The modified model of the client-server CFSM system in Figure 6.1 to remove livelock sources

For the above revised model, the ILP problem to determine its livelock freedom, as given in Inequalities (6.14–6.22), has no solutions. This implies livelock freedom for the revised system as we expect since the only source of livelocks has been removed.

$$-x_1 + x_3 \geq 0 \tag{6.14}$$
$$x_1 - x_3 \geq 0 \tag{6.15}$$
$$-x_1 + x_3 \geq 0 \tag{6.16}$$
$$-x_2 + x_3 \geq 0 \tag{6.17}$$
$$x_2 - x_3 \geq 0 \tag{6.18}$$
$$-x_2 + x_3 \geq 0 \tag{6.19}$$
$$x_1 + x_2 + x_3 > 0 \tag{6.20}$$
$$x_1 = 0 \tag{6.21}$$
$$x_1, x_2, x_3 \geq 0 \tag{6.22}$$

6.4 Soundness

The soundness proof of our livelock freedom test relies on the fact that the ILP problem described in Inequalities (6.1–6.4) gives a necessary condition for the existence of livelocked executions.

Theorem 6.2 (Soundness)**.** *Given a CFSM system and a predetermined set of progress transitions, if the livelock freedom test determines the system to be free of livelock, then the system is actually free of livelock.*

Proof. If the system is proved to be free of livelock by our method, then there exists no positive linear combination of effect vectors of non-progress cycles since there is no solution to the ILP problem described by the inequalities (6.1–6.4). By Proposition 5.4, we can see that the any execution of the system in which only non-progress cycles are repeated infinitely often is bounded.

By contradiction we assume that the CFSM system has a livelocked execution r. So, all the progress cycles are repeated only a finite number of times in r. Then, there exists a particular point of time t in r after which only non-progress cycles are executed. Furthermore, we have shown above that r is bounded. Note that any process in the CFSM has only finitely many local states. Consequently, there will be only finitely many reachable configurations of the system after t in r. Furthermore, since r is an infinite execution, there must be two distinct points of time t_1 and t_2 after t at which the system reaches one same configuration. The finite segment of execution between t_1 and t_2 can be represented as a linear combination of executions of non-progress cycles. The aggregate effect vector of this segment is however an all-zero vector. This contradicts the fact that no solution exists for the ILP problem described by the inequalities (6.1–6.4), i.e., there is no non-negative linear combination of non-progress cycles. □

Note that the above proof does not use any assumption about buffer lengths. Consequently, if a system with unbounded buffers is proved to be livelock free, then the same system with bounded buffers of predefined lengths is also livelock free.

6.5 Complexity

Given a CFSM system, if there are m cycles in its independent cycle system and n types of messages, then there are no more than $m + n + 2$ linear inequalities in the constructed ILP problem to determine livelock freedom. Each inequality contains no more than m items. The size of the constructed ILP problem is bounded therefore by $(m + n + 2) \times m$. Furthermore, the ILP problem that our method constructs is always homogeneous, which can be solved in polynomial time as discussed previously. To summarize, the complexity of the livelock freedom test that we propose is polynomial in the number of control flow cycles in the CFSM system being verified.

Part II

Code Abstraction

Chapter 7

Abstracting Promela Code

In the previous chapters we presented two incomplete but efficient methods to verify buffer boundedness and livelock freedom for CFSM systems. Models written in other modeling languages can be checked using these verification methods when they are first abstracted into CFSM systems. We refer to this first step of abstraction as *code abstraction*. Since each modeling language has its distinct features and expressiveness, the code abstraction techniques for each language must be individually and differently designed. We focus in this chapter on the abstraction of Promela models, and will consider some specific abstraction issues related to UML RT models in the next chapter.

Given a Promela model, the goal of a code abstraction procedure is to construct a CFSM system that approximates the behavior of the original Promela model. The abstraction must be property preserving in the sense that any property violating execution of the original model should also be present in the abstract system. This can be guaranteed by over-approximation, i.e., the abstract system preserves all possible behavior of the original model. Furthermore, the abstract CFSM system usually includes spurious behavior that is *not* allowed in the original model, since CFSM is less expressive than Promela. As we will show later, the spurious behavior introduced in the code abstraction procedure is a major source of imprecision in our verification framework. In order to address this imprecision problem, we have designed refinement techniques that effectively rule out spurious behavior, which will be explained in Chapter 12.

We follow the principle of over-approximation in the design of code abstraction techniques in order to ensure the soundness of the buffer boundedness and livelock freedom tests. Since CFSM systems only allow message passing behavior, the focus of our abstraction approach is naturally on the specific problems to determine the message passing effects of Promela code. Issues that we will consider include

- the identification of message types in Promela models (see Section 7.2);
- the replication of identical proctype instances (see Section 7.3);
- the assignment to buffer variables (see Section 7.4);
- the use of buffer arrays, in particular, when an array element is indexed by a variable whose value is statically unknown (see Section 7.5);

```
1  mtype = {question, answer};
2
3  chan ch = [1] of {mtype};
4
5  proctype P(mtype m1; mtype m2){
6      do
7      :: ch?m1 -> ch!m2
8      od
9  }
10
11 init{
12     ch!question;
13     run P(question, answer);
14     run P(answer, question);
15 }
```

Listing 7.1: A simple Promela model with parameterized proctype definitions.

- and the impact of unbounded proctype instantiations (see Section 7.6).

Structure. In Section 7.1 we give an overview of the proposed code abstraction procedure, and address each of the above listed issues in the remainder of the chapter.

7.1 Overview

In this section we give an overview of the code abstraction procedure that we propose, and point out where the difficulties lie.

The code abstraction procedure that we propose takes two stages: (1) first constructing the control flow graph of every actually running process (i.e. proctype instance) of the original model; and (2) constructing an abstract CFSM system based on the previously obtained control flow graphs by determining the message passing effects of each Promela statement. We show each stage in details in the following subsections.

7.1.1 Constructing Control Flow Graphs of Actually Running Promela Processes

Let us first recall the syntactical structure of a Promela model. A Promela model consists of a set of `proctype` definitions. Each proctype represents a class of processes whose common behavior is described by the sequential code in the definition of the proctype. A proctype may have formal parameters such as the proctype `P` in the Promela model in Listing 7.1. A proctype may have none, one, or multiple instances at any point of the runtime. Instances of a same parameterized proctype may be instantiated with different actual parameters. In the above example, the two instances of `P` are created in Line 13 and Line 14, with different mtype values passed to the formal parameters `m1` and `m2`.

The first step of abstracting a Promela model is to determine statically which processes will be created at runtime. Once we have such information, we can

easily derive a control flow graph for each of the actually running processes of the model as follows.

Step 1: generating control flow graphs of proctypes. The SPIN model checker can be used to automatically generate a control flow graph for each proctype definition in the model, using the following command lines:

`spin -a MODEL_FILE`	(generating the source code of a verifier program for the model)
`gcc -o pan pan.c`	(compiling the generated source code using a C compiler such as `gcc`)
pan -d	(using the verifier to generate a textual representation of the control flow information of the model)

We can easily generate the control flow graph of each proctype definition from the textual control flow information as obtained above. In the generated control flow graph for a proctype, each edge corresponds to one single Promela statement in the code of the proctype. As an example, Figure 7.1 shows the control flow graph (left) constructed for the proctype `P` in the model in Listing 7.1. In the next step, we will construct the control flow graph of an actually running process based on the control flow graph that we obtain for its corresponding proctype.

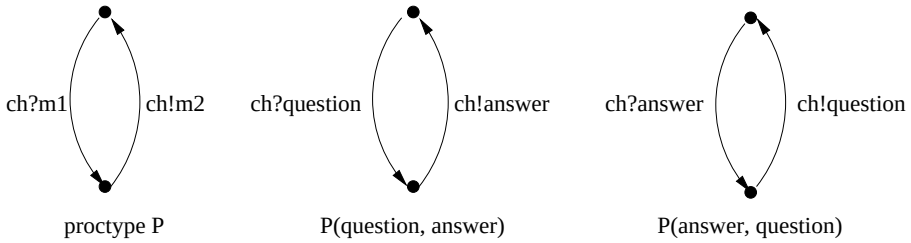

Figure 7.1: The control flow graphs of the proctype `P` and its running instances of the Promela model shown in Listing 7.1

Step 2: constructing control flow graphs of actually running processes. For each actually running process, its control flow graph is identical to the control flow graph of its proctype with the following exception. If the proctype is parameterized, then all the occurrences of the formal parameters in the control flow graph of the proctype are replaced with the respective actual parameters in the control flow graph of the process. Figure 7.1 shows the control flow graphs (middle and right) for the two running instances of the proctype `P`. Note that the two control flow graphs differ only in the mtype values of the actual parameters passed to `m1` and `m2` in the message sending and receiving statements.

One premise of the above construction steps is that we can statically determine all actually running processes for a Promela model. Although we observed

in practice that it is usually easy to determine running processes for most realistic models, it is generally not the case. In particular, the number of proctype instantiations can be unbounded, which we will discuss in Section 7.6. In the scope of this thesis, we assume that we can always determine statically running processes for a Promela model.

7.1.2 Constructing CFSM systems

Now we explain how to construct an abstract CFSM system for a Promela model based on the control flow graphs that we obtain for all actually running processes of the model. The central idea is to build a CFSM process for each actually running Promela process by determining the message passing effects of each Promela statement in the Promela process. Therefore, we first need to determine (1) which message buffers exist and are used in the model to exchange messages; and (2) which types of messages are exchanged in these buffers.

Determining Message Buffers

The message buffers used in a Promela model are declared using `chan` declarations. For each global `chan` variable, a separate concrete FIFO message queue is initialized at the beginning of the execution of the model. For each `chan` variable locally defined in a proctype `P`, a separate concrete message queue is initialized for each instance of `P` when the instance is created. It seems that each `chan` variable, or more precisely the concrete queue that it points to, can be directly mapped to one distinct message buffer in the abstract CFSM system to be constructed. However, this simple solution of direct mapping is unsound due to potential assignments to `chan` variables in Promela models. We will address the buffer assignment problem later in Section 7.4.

Determining Message Types

The Promela language does not have the concept of message types. Even though `mtype` constants can be used to indicate types of messages, a message does not need to contain an mtype field. In the Promela language, the format of messages that can be exchanged in a buffer is defined in the declaration of the buffer. A naive solution is then to map the message format of a buffer directly to a message type in the constructed abstract CFSM system. However, we will show later in Example 7.1 (Section 7.2.2) that this solution may result in the imprecision of the buffer boundedness test. We will explain in Section 7.2.3 how to identify an optimal set of message types from the Promela code so as to make the future analysis as accurate as possible.

Constructing State Machines of CFSM Processes

Given a Promela model, we assume that we have determined the set of message buffers, the set of message types, and the control flow graphs for all actually running processes of the model. The next step is to construct CFSM processes and their state machines. For each actually running Promela process, we build one CFSM process whose state machine can be constructed by determining the message passing effects of Promela statements: Given the control flow graph G of a Promela process, let P be its corresponding CFSM process. Every

```
1  mtype msg;
2
3  if
4  :: x>0 -> msg = msg1;
5  :: else -> msg = msg2;
6  fi;
7
8  ... ...
9
10 ch!msg;
```

Listing 7.2: A piece of Promela code.

control point in G corresponds to one distinct local state in the state machine of P. A control flow edge in G may however correspond to more than one local transition in P. This is because the Promela statement along a control flow edge may have more than one possible message passing effect, due to the use of variables in message sending and receiving statements. An example is shown in Listing 7.2. In the code, `msg` is a variable of mtype, and may contain different values at runtime (see Line 4 and Line 5). As a result, the message sending statement at Line 10 may send a message containing either `msg1`, or `msg2`, or other mtype values. Consequently, we need to have more than one corresponding CFSM transition to capture all possible message passing effects, following the over-approximation principle that the abstract CFSM system must preserve all possible behavior of the original Promela model. This is shown in Figure 7.2 in which each CFSM transition is labeled with one distinct possible message passing effect.

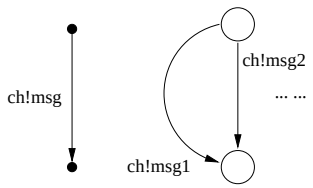

Figure 7.2: The left shows the control flow edge corresponding to the statement at Line 10 in the Promela code in Listing 7.2; the right shows at least two transitions in the constructed CFSM system that correspond to the control flow edge on the left.

In general, taking G and P as defined in the previous paragraph, let e be an edge from a control point p_1 to another control point p_2 in G. Let s_1 and s_2 be the local states in P that correspond to p_1 and p_2, respectively. If we determine that the Promela statement along e has n possible message passing effects, then there are n transitions in P from s_1 to s_2, each of which is labeled with one possible message passing effect. We will explain in details inside Section 7.2.1 how to determine message passing effects for a statement.

Sizes of Generated Abstract CFSM Systems

Naturally, we prefer a smaller size of the CFSM system that we generate for a Promela model, because the future analysis will be performed on the CFSM system. A smaller CFSM system means a more efficient analysis.

Given a Promela model, we assume that we determine the actually running processes to be $p_1, \ldots, p_k$. We further assume that we identify n message types in the model. For each Promela process p_i, if its control flow graph contains m_i edges, then there are at most $n \times m_i$ transitions in the corresponding CFSM process. This is because each Promela statement may have no more than n message passing effects. Therefore, the size of the constructed CFSM system is linear in the number of message types and also linear in the size of the control flow graph of each actually running Promela process. Even if the size of the constructed CFSM system is only linearly increased when the number of identified message types is increased, the number of control flow cycles in the CFSM system may however be polynomially increased. Consider Figure 7.3 as an example. If the Promela statement along each control flow edge on the left has two message passing effects, then there are two CFSM transitions corresponding to each edge, as shown on the right. While there is only one cycle on the left, there are 2^4 corresponding cycles in the CFSM process on the right. In general, given a control flow cycle of length m in a Promela process, if there are n message types, then the number of corresponding control flow cycles in the respective CFSM process is bounded by n^m.

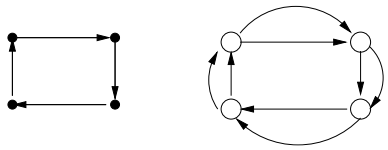

Figure 7.3: The left shows a control flow cycle of a Promela process whose corresponding CFSM process is partly shown on the right.

Based on the discussion above, we know that the size of the constructed CFSM system for a Promela model largely depends on the number of message types that we identify in the model. While this implies a desire to have a small number of message types, we will show in the next section that having too few message types may result in an imprecise analysis of the Promela model.

7.2 Identifying Message Types

In this section we formally define message types for Promela models, and propose a method to determine message passing effects for Promela statements based on this concept of message types. We also discuss how the identification of message types may influence the precision of the future analysis of a model. Finally, we give an optimal message type identification method to make the future analysis as accurate as possible while still obtaining a small number of message types.

For the subsequent discussion, we fix a formal denotation of messages in Promela models. We intentionally include buffer names in the denotation of messages to distinguish messages stored in different buffers. More precisely, given

a buffer declaration `chan ch = [l] of t1,...,tn`, we use $\langle \texttt{ch}, v_1, \ldots, v_n \rangle$ to denote a runtime message stored in `ch` whose i-th field contains the value v_i of the type `ti`.

7.2.1 Message Types

We formally define the concept of message types for Promela models as follows.

Definition 7.1 (Message Types of Promela Models)**.** Given a Promela model, we use $\mathcal{M}$ to denote the set of all possible messages exchanged in the model. A *message type* is a subset of $\mathcal{M}$.

We use the concept of a *message type definition* to denote the result of a message type identification.

Definition 7.2 (Message Type Definitions)**.** Given a Promela model, a *message type definition* D is a collection of message types such that

- for any two distinct message types m_1 and m_2 in D, m_1 and m_2 are disjoint, i.e., $m_1 \cap m_2 = \emptyset$;
- the union of all message types in D contains all messages that can be exchanged in the model, i.e., $\bigcup_{m \in D} m = \mathcal{M}$.

We also define the *refinement* relation between message type definitions as follows.

Definition 7.3 (Refinement of Message Type Definitions)**.** A message type definition D_1 *refines* another one D_2, denoted by $D_2 \lhd D_1$, if for any message type m such that $m \in D_2$ and $m \notin D_1$, there exists a set of message types $m_1, \ldots, m_n \in D_1$ such that $\bigcup_{i=1}^{n} m_i = m$.

In the above definition, we say that D_1 *refines* the message type m in D_2 with message types $m_1, \ldots, m_n$. We use $D_2 \lhd^1 D_1$ to denote that D_1 refines only one message type in D_2.

Determining Message Passing Effects

Based on the concept of message types in Definition 7.1, we propose a method to determine possible message passing effects for a Promela statement. We first need an auxiliary concept called *coverage sets* before we explain the method.

Message sending or receiving statements may contain variables, such as the statement `c!5,x` where `x` is an integer variable. The runtime value of `x` is unknown statically. If we assume that `x` may take *any* value at runtime, then the set of all possible messages sent by this statement is $\{\langle c, x, 5\rangle \mid x \text{ is an integer}\}$. We call this set as the *coverage set* of the statement.

Definition 7.4 (Coverage Sets of Promela Statements)**.** Given a Promela statement s, we define its *coverage set*, denoted by M_s, as follows:

- if s is neither a message sending statement nor a message receiving statement, then $M_s = \emptyset$;

- if s is a message sending or receiving statement in form of $\texttt{ch}\backsim e_1, \ldots, e_n$, where $\backsim$ is either `!` or `?`, then M_s contains all messages $\langle q, v_1, \ldots, v_n \rangle$ satisfying the following properties:
  - for each v_i, if e_i is a variable, then v_i can be any value of the same type as the type of e_i;
  - Otherwise, $v_i = e_i$.

Intuitively, a message sending or receiving statement may send or receive messages of a certain type m only if its coverage set has a non-empty intersection with m. Therefore, given a message type definition D, we determine the set of types of messages that a Promela statement s may send or receive to be $\{m \mid m \in D \wedge m \cap M_s \neq \emptyset\}$. As an example, we consider the Promela code in Listing 7.2. The message sending statement at Line 10 has a coverage set $M_{s_{10}} = \{\langle \texttt{ch}, msg \rangle \mid msg \text{ is any mtype constant.}\}$. If we have identified $m_1 = \{\langle \texttt{ch}, \texttt{msg1} \rangle\}$ and $m_2 = \{\langle \texttt{ch}, \texttt{msg2} \rangle\}$ as two message types, then the set of potential types of messages that the statement can send should include at least m_1 and m_2 since $M_{s_{10}} \cap m_1 = m_1 \neq \emptyset$ and $M_{s_{10}} \cap m_2 = m_2 \neq \emptyset$.

7.2.2 Message Types and the Precision of Verification

We have known that fewer identified message types may result in a smaller abstract CFSM system being constructed, which implies a more efficient analysis that we will perform on the CFSM system. However, the following example suggests that if too few message types are identified then the precision of the future analysis may be severely compromised.

Example 7.1. Consider the Promela model as shown in Listing 7.3. In the model, a client asks a consultant a question about A (`askA`) and waits for the answer. The client is forgetful such that if she ever gets an answer she will forget it immediately and has to ask the same question again. Moreover, she apparently visits a wrong consultant since the consultant only knows how to answer questions about B. The consultant also documents every consultation, which is represented in the model by sending a `record` message to the buffer `log` (see Line 18).

In the example model, if we identify 5 types of messages as

$$\begin{aligned} \{&m_1 = \{\langle \texttt{toConsultant}, \texttt{askA} \rangle\}, \\ &m_2 = \{\langle \texttt{toClient}, \texttt{answerA} \rangle\}, \\ &m_3 = \{\langle \texttt{toConsultant}, \texttt{askB} \rangle\}, \\ &m_4 = \{\langle \texttt{toClient}, \texttt{answerB} \rangle\}, \\ &m_5 = \{\langle \texttt{log}, \texttt{record} \rangle\}\} \end{aligned}$$

then we obtain the abstract CFSM system for the model as shown in Figure 7.4. We can determine either using our boundedness test or even by manual inspection that this CFSM system is bounded. Therefore, the original Promela model is bounded as well. This is because none of the two control flow cycles in the CFSM system can be ever completely executed – the consultant can never answer a question about A.

```
1  mtype = {askA, answerA, askB, answerB, record};
2
3  chan toConsultant = [1] of {mtype};
4  chan toClient = [1] of {mtype};
5  chan log = [10] of {mtype};
6
7  active proctype forgetfulClient(){
8     do
9     :: toConsultant!askA;
10       toClient?answerA;
11    od;
12 }
13
14 active proctype Consultant(){
15    do
16    :: toConsultant?askB ->
17       toClient!answerB;
18       log!record;
19    od;
20 }
```

Listing 7.3: A Promela model describing a consulting scenario.

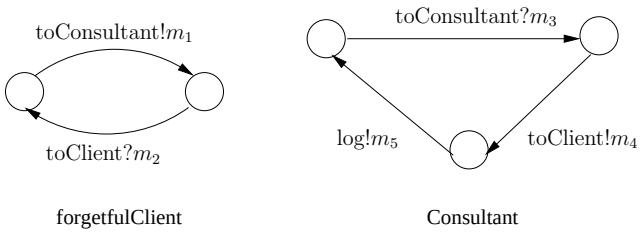

Figure 7.4: The abstract CFSM system for the Promela model shown in Listing 7.3 when we identify 5 types of messages in the model.

Alternatively, we may decide to discriminate only messages in different buffers. This is in fact an example of mapping the formats of messages defined in buffer declarations directly to message types. In this way, we only identify three message types as follows:

$$\begin{aligned}\{m_1 &= \{\langle \texttt{toConsultant}, \texttt{askA}\rangle, \langle \texttt{toConsultant}, \texttt{askB}\rangle\},\\ m_2 &= \{\langle \texttt{toClient}, \texttt{answerA}\rangle, \langle \texttt{toClient}, \texttt{answerB}\rangle\},\\ m_3 &= \{\langle \texttt{log}, \texttt{record}\rangle\}\}.\end{aligned}$$

The resulting abstract CFSM system is depicted in Figure 7.5. This CFSM system is unbounded due to the introduction of the spurious behavior that the consultant answers the client's questions and can therefore add an unbounded number of `record` messages of the message type m_3. The presence of the spurious behavior is caused by being no longer able to distinguish between messages regarding A and messages regarding B. The boundedness analysis based on such a CFSM system is imprecise and returns "UNKNOWN".

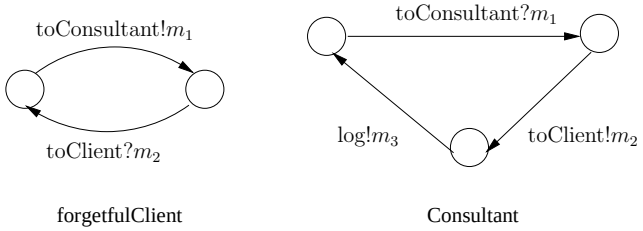

Figure 7.5: The abstract CFSM system for the Promela model shown in Listing 7.3 when we discriminate messages in different buffers.

The above example shows that the precision of the buffer boundedness test is reduced when we decide a too coarse message type definition. In the following we formally prove that by refining message types we always obtain no less precise results of the boundedness test.

Proposition 7.1. *Given a CFSM system, let D_1 and D_2 be two message type definitions such that $D_2 \lhd D_1$. Let S_1 and S_2 be the independent cycle systems that we derive respectively based on D_1 and D_2, using the message passing effect determination method described in Section 7.2.1. If S_2 is bounded, then S_1 is also bounded.*

Proof. It is sufficient to prove that, given two message type definitions D and D' such that $D \lhd^1 D'$, if the resulting independent cycle system based on D is bounded then the resulting cycle system based on D' is also bounded.

Let $S = (n, C, \mathit{eff})$ be the independent cycle system based on D, and $S' = (n', C', \mathit{eff}')$ be the independent cycle system based on D'. Without loss of generality, we assume that D' refines the message type m in D with message types $m_1, \ldots, m_k$. Therefore, we have $n' = n - 1 + k$. For the sake of convenience, we re-order the effect vector components for both independent cycle systems such that (1) for any i such that $1 \leq i \leq n-1$, the i-th effect vector components in both systems correspond to one same message type; and (2) the

n-th components in D correspond to the message type m; and (3) for any i such that $1 \leq i \leq k$, the $(n-1+i)$-th components in S' correspond to the message type m_i.

Furthermore, we assume without loss of generality that there are p cycles in C: $c_1, \ldots, c_p$. Assume that in some cycle c_i there is a transition t that potentially sends or receives a message of type m. Let s be the Promela statement along the control flow edge that corresponds to t. As the result of refining the message type m, the transition t may correspond to a set of transitions in the abstract CFSM system based on D'. Each of these corresponding transitions sends or receives a message of a distinct type m_j such that $M_s \cap m_j \neq \emptyset$. The cycle c_i may contain one or more transitions such as t. Consequently, each cycle c_i corresponds to a set of cycles in C': $c'_{i1}, \ldots, c'_{iq_i}$ as the result of the refinement of the message type m. Furthermore, we can easily see the following property for c_i and one of its corresponding cycles c'_{il}:

$$\forall j.(1 \leq j \leq n-1) \rightarrow \mathit{eff}(c_i)^j = \mathit{eff}'(c'_{il})^j \tag{7.1}$$

and

$$\mathit{eff}(c_i)^n = \sum_{j=1}^{k} \mathit{eff}'(c'_{il})^{n-1+j} \tag{7.2}$$

Now we prove that if S is bounded then S' is also bounded. By contradiction we assume that S is bounded while S' is unbounded. Since S' is unbounded, the sufficient and necessary condition as stated in Proposition 5.4 implies a solution to the following ILP problem:

$$\sum_{i=1}^{q_1} \mathit{eff}'(c'_{1i})^j \cdot x_{1i} + \cdots + \sum_{i=1}^{q_p} \mathit{eff}'(c'_{pi})^j \cdot x_{pi} \geq 0 \quad \text{for each} \quad 1 \leq j \leq n' \tag{7.3}$$

$$\sum_{i=1}^{q_1} (\sum_{j=1}^{n'} \mathit{eff}'(c'_{1i})^j) \cdot x_{1i} + \cdots + \sum_{i=1}^{q_p} (\sum_{j=1}^{n'} \mathit{eff}'(c'_{pi})^j) \cdot x_{pi} > 0 \tag{7.4}$$

Let one of the solutions to the above ILP problem be $x_{11} = v_{11}, \ldots, x_{pq_p} = v_{pq_p}$. We show in the following that from this solution we can construct a solution to the ILP problem representing the sufficient and efficient condition of the unboundedness of S, as shown in Inequalities (7.5–7.6), thereby contradicting the assumption.

$$\mathit{eff}(c_1)^j \cdot x_1 + \cdots + \mathit{eff}(c_p)^j \cdot x_p \geq 0 \quad \text{for each} \quad 1 \leq j \leq n \tag{7.5}$$

$$\sum_{j=1}^{n} \mathit{eff}(c_1)^j \cdot x_1 + \cdots + \sum_{j=1}^{n} \mathit{eff}(c_p)^j \cdot x_p > 0 \tag{7.6}$$

We show that $x_i = \sum_{r=1}^{q_i} v_{ir}$ is a solution to the above ILP problem.

- For each j-th $(1 \leq j \leq n-1)$ inequality in Inequalities (7.5), replacing

each x_i with its respective value $\sum_{r=1}^{q_i} v_{ir}$, we have that:

$$\mathit{eff}(c_1)^j \cdot \sum_{r=1}^{q_1} v_{1r} + \cdots + \mathit{eff}(c_p)^j \cdot \sum_{r=1}^{q_p} v_{pr} \tag{7.7}$$

$$= \sum_{r=1}^{q_1} \mathit{eff}(c_1)^j \cdot v_{1r} + \cdots + \sum_{r=1}^{q_p} \mathit{eff}(c_p)^j \cdot v_{pr} \tag{7.8}$$

(pushing effect vectors inside summations)

$$= \sum_{r=1}^{q_1} \mathit{eff}'(c'_{1r})^j \cdot v_{1r} + \cdots + \sum_{r=1}^{q_p} \mathit{eff}'(c'_{pr})^j \cdot v_{pr} \tag{7.9}$$

(following the property as Inequality (7.1))

$$\geq 0 \tag{7.10}$$

(following the fact that $x_{ir} = v_{ir}$ is a solution to Inequalities (7.3))

- For the n-th inequality in Inequalities (7.5), also replacing each x_i with its respective value, we have that:

$$\mathit{eff}(c_1)^n \cdot \sum_{r=1}^{q_1} v_{1r} + \cdots + \mathit{eff}(c_p)^n \cdot \sum_{r=1}^{q_p} v_{pr} \tag{7.11}$$

$$= \sum_{r=1}^{q_1} \mathit{eff}(c_1)^n \cdot v_{1r} + \cdots + \sum_{r=1}^{q_p} \mathit{eff}(c_p)^n \cdot v_{pr} \tag{7.12}$$

$$= \sum_{r=1}^{q_1} (\sum_{j=1}^{k} \mathit{eff}'(c'_{1r})^{n-1+j}) \cdot v_{1r} + \cdots + \sum_{r=1}^{q_p} (\sum_{j=1}^{k} \mathit{eff}'(c'_{pr})^{n-1+j}) \cdot v_{pr} \tag{7.13}$$

(following the property as Inequality (7.2))

$$\geq 0 \tag{7.14}$$

(By adding all j-th Inequalities (7.3) where $n \leq j \leq n'$)

- For Inequality (7.6):

$$\sum_{j=1}^{n} \mathit{eff}(c_1)^j \cdot \sum_{r=1}^{q_1} v_{1r} + \cdots + \sum_{j=1}^{n} \mathit{eff}(c_p)^j \cdot \sum_{r=1}^{q_p} v_{pr} \tag{7.15}$$

$$= \sum_{j=1}^{n-1} \mathit{eff}(c_1)^j \cdot \sum_{r=1}^{q_1} v_{1r} + \cdots + \sum_{j=1}^{n-1} \mathit{eff}(c_p)^j \cdot \sum_{r=1}^{q_p} v_{pr} + \mathit{eff}(c_1)^n \cdot \sum_{r=1}^{q_1} v_{1r} + \cdots + \mathit{eff}(c_p)^n \cdot \sum_{r=1}^{q_p} v_{pr} \tag{7.16}$$

$$= \sum_{r=1}^{q_1} (\sum_{j=1}^{n'} \mathit{eff}'(c'_{1r})^j) \cdot v_{1r} + \cdots + \sum_{r=1}^{q_p} (\sum_{j=1}^{n'} \mathit{eff}'(c'_{pr})^j) \cdot v_{pr}$$

$$> 0$$

(following the fact that $x_{ir} = v_{ir}$ is a solution to Inequalities (7.4))

□

Proposition 7.1 shows that the boundedness analysis returns no less precise results by refining message type definitions. The same holds for the livelock freedom analysis that we proposed. We leave out the formal proof here. However, a refined message type definition causes more identified message types, which may lead to a growing number of control flow cycles and hence render the future analysis to be less efficient. Therefore, there is a trade-off between efficiency and precision for the determination of message types. In the following subsection, we will propose a method to identify message types optimally such that the precision of the future analysis cannot be improved by further refining the obtained optimal message type definition.

7.2.3 An Optimal Message Type Identification Method

Before we introduce our message type identification method, we first review the executability semantics of message sending and receiving statements in the Promela language.

A message sending statement is executable if the receiving buffer is not full. However, since we interpret buffers in Promela models to have unbounded capacities, a message sending statement is never blocked. On the contrary, a message receiving statement may be blocked due to the following two potential reasons. First, if a statement tries to receive messages from an empty buffer, then it is not executable. Second, a message receiving statement is blocked if it tries to receive a certain kind of messages that is not available in the respective buffer. More precisely, a message receiving statement is not executable if the top message in the respective buffer is not in its coverage set. As an example, consider a message receiving statement `ch?5,x` where `x` is a boolean variable. This statement contains a constant message field 5. It is not executable if the top message in the buffer `ch` is $\langle \texttt{ch}, 4, \texttt{true} \rangle$. The statement is executable if the top message contains 5 as its first field. On the contrary, which boolean value the second field contains has no effect on the executability of the receiving statement.

Based on the above argument, we know that a Promela model discriminates messages in the following two ways. First, the model discriminates messages in different message buffers. Second, the constant message fields in the message receiving statements in the model also cause the discriminations of the messages stored in one same buffer. Such discriminations are the intuition behind the message type identification method that we will propose, i.e., we only need to distinguish two messages if they are distinguished by the model. In the following we explain our method in details, and then prove that it results in an optimal message type definition such that further refining the message type definition cannot improve the precision of the buffer boundedness test. The proposed method applies only to models in which there are no buffer assignments. An adaption of the method with respect to buffer assignments is discussed in Section 7.4.

Given a Promela model, we first collect all the message receiving statements that contain constant message fields. We denote this collection of message receiving statements as the set R. For each buffer `ch` in the model, we identify the types of messages exchanged in `ch` in the following way. We use M_{ch} to denote

the set of all messages that can be stored in `ch` at runtime. Let $\mathtt{s}_1, \ldots, \mathtt{s}_k \in R$ be all the statements receiving messages from `ch`. In practice we observed that in most cases the coverage sets of these statements $M_{s_1}, \ldots, M_{s_k}$ are pairwise disjoint. If this is the case, then we can directly identify the set of message types for `ch` as $\{M_{s_1}, \ldots, M_{s_k}, M_{ch} - \bigcup_{i=1}^{k} M_{s_i}\}$. In particular, if there is no statement in R to receive messages from `ch`, then we identify only one message type for `ch` as M_{ch}. As an example, for the Promela model in Listing 7.3, we collect all the receiving statements with constant message fields as follows:

$$\mathtt{s}_1 : \mathtt{toClient?answerA}$$
$$\mathtt{s}_2 : \mathtt{toConsultant?askB}$$

Therefore, we identify for the buffer `toClient` two message types as

$$M_{s_1} = \{\langle \mathtt{toClient}, \mathtt{answerA} \rangle\},$$
$$M_{toClient} - M_{s_1} = \{\langle \mathtt{toClient}, \mathtt{answerB} \rangle\}$$

for the buffer `toConsultant` two message types as

$$M_{s_2} = \{\langle \mathtt{toConsultant}, \mathtt{askB} \rangle\},$$
$$M_{toConsultant} - M_{s_2} = \{\langle \mathtt{toConsultant}, \mathtt{askA} \rangle\}$$

and for the buffer `log` only one message type as

$$M_{log} = \{\langle \mathtt{log}, \mathtt{record} \rangle\}.$$

The resulting message type definition is the same as the first message type definition used in Example 7.1, which results in a precise buffer boundedness analysis. While in this example the coverage sets of the collected message receiving statements for each buffer are pairwise disjoint, this cannot generally be assumed, as the following example illustrates.

Example 7.2. Consider a Promela model in which there is a buffer `a = [10] of {int, bool}`. Suppose that there are two message receiving statements with constant message fields to receive messages from `a` as the following:

$$\mathtt{t}_1 : \mathtt{a?5, b}$$
$$\mathtt{t}_2 : \mathtt{a?x, true}$$

where `b` is a boolean variable and `x` is an integer variable. Obviously M_{t_1} and M_{t_2} overlap, e.g., the message $\langle \mathtt{a}, 5, \mathtt{true} \rangle$ belongs to both sets. In this case, if we identify M_{t_1} and M_{t_2} as two message types, then they violate the disjointness property in Definition 7.2.

In order to maintain the disjointness of the identified message types, we identify message types for a buffer in the following way: Given R, `ch`, $\mathtt{s}_1, \ldots, \mathtt{s}_k$ as defined previously, $M_{s_1}, \ldots, M_{s_k}$ form a partition over the set M_{ch}. More precisely, the partition consists of the following pairwise disjoint subsets of M_{ch}:

$$\overline{M_{s_1}} \cap \overline{M_{s_2}} \cap \cdots \cap \overline{M_{s_k}}$$
$$M_{s_1} \cap \overline{M_{s_2}} \cap \cdots \cap \overline{M_{s_k}}$$
$$\overline{M_{s_1}} \cap M_{s_2} \cap \cdots \cap \overline{M_{s_k}}$$
$$M_{s_1} \cap M_{s_2} \cap \cdots \cap \overline{M_{s_k}}$$
$$\ldots\ldots$$
$$M_{s_1} \cap M_{s_2} \cap \cdots \cap M_{s_k}$$

Each of the above subsets corresponds to one identified message type for the buffer `ch`. In the above example, we identify the following four message types for the buffer `a`: $\overline{M_{t_1}} \cap \overline{M_{t_2}}$ that contains all the messages whose first field is not 5 and whose second field is `false`; $M_{t_1} \cap \overline{M_{t_2}}$ of messages whose first field is 5 and whose second field is `false`; $\overline{M_{t_1}} \cap M_{t_2}$ of messages whose first field is not 5 and whose second field is `true`; and $M_{t_1} \cap M_{t_2}$ of messages whose first field 5 and whose second field is `true`.

Now we argue formally that the above method always constructs an optimal message type definition such that further refining it does not improve the accuracy of the buffer boundedness test.

Proposition 7.2. *Given a Promela model, let D be the message type definition identified by the method described previously. Let D' be a message type definition such that $D \lhd^1 D'$. Let S and S' be the independent cycle systems constructed respectively based on D and D'. If S' is bounded, then S is also bounded.*

Proof. Let $S = (n, C, \mathit{eff})$ and $S' = (n', C', \mathit{eff}')$. Without loss of generality, we assume that D' refines the message type m in D with message types $m_1, \ldots, m_k$. Therefore, we have that $n' = n - 1 + k$. For the sake of convenience, we re-order the effect vector components for both independent cycle systems such that (1) for any i such that $1 \leq i \leq n-1$, the i-th effect vector components in both systems correspond to one same message type; and (2) the n-th components in S correspond to the message type m; and (3) for any i such that $1 \leq i \leq k$, the $(n-1+i)$-th components in S' correspond to the message type m_i.

Let t be a local transition in some cycle in C. Suppose that t corresponds to a message receiving statement s in the original Promela code, which may receive one message of the type m from the buffer `ch`. Then, we have that $M_s \cap m \neq \emptyset$. We show that in fact $m \subseteq M_s$, by checking the following two possible cases: (1) If s contains no constant message fields, then $m \subseteq M_{ch} = M_s$ follows; (2) Otherwise, s must be one of the message receiving statements used to identify message types for `ch`. Recall how M_{ch} is partitioned to identify message types as described in the optimal identification method. We can easily see that $m \subseteq M_s$. Because $m \subseteq M_s$ and $m_i \subset m$ for each message type m_i, we have that $m_i \subset M_s$. This implies that s may receive messages of any of the message types $m_1, \ldots, m_k$.

Furthermore, we assume without loss of generality that there are p cycles in C: $c_1, \ldots, c_p$. Similar to the argument in the proof of Proposition 7.1, any cycle c_i in C corresponds to a set of cycles in C': $c'_{i1}, \ldots, c'_{iq_i}$ as the result of the refinement of the message type m. We also have the same properties as the ones stated in Inequalities (7.1) and (7.2) for each cycle c_i and any of its corresponding cycles in C'.

Besides, for any cycle $c_i \in C$, we can show that there exists one of its corresponding cycles $c'_{il} \in C'$ such that c'_{il} satisfies the following property: If $\mathit{eff}(c_i)^n \geq 0$ ($\mathit{eff}(c_i)^n \leq 0$ respectively), then $\mathit{eff}'(c'_{il})^j \geq 0$ ($\mathit{eff}'(c'_{il})^j \leq 0$ respectively) holds for each $n \leq j \leq n'$. Moreover, if $\mathit{eff}(c_i)^n \neq 0$, then there exists at least one component $\mathit{eff}'(c'_{il})^j$ ($n \leq j \leq n'$) such that $\mathit{eff}'(c'_{il})^j > 0$ ($\mathit{eff}'(c'_{il})^q < 0$ respectively). We only prove here the case when $\mathit{eff}(c_i)^n > 0$. The proof for other cases can be similarly derived. Without loss of generality, we assume that c_i has a message sending statements $s_1, \ldots, s_a$ that send messages of the type m; and b message receiving statements $r_1, \ldots, r_b$ that receive messages of the type m. Because $\mathit{eff}(c_i)^n > 0$, we have that $a > b$. We select the first b message sending statements $s_1, \ldots, s_b$. Among all the cycles in C' that

correspond to c_i, we can find such a cycle c'_{il} to satisfy the following property of the message passing effects of $s_1, \ldots, s_b$ and $r_1, \ldots, r_b$: If any s_u $(1 \leq u \leq b)$ sends a message of some type $m' \subset m$, then r_u receives a message of the same type m'. This is possible because $m' \subset m \subseteq M_{r_u}$ and r_u may therefore receive a message of the type m'. In this way, s_u cancels the r_u's negative effect that consumes one message of the type m'. Therefore, all the negative effects of $r_1, \ldots, r_u$ can be compensated by $s_1, \ldots, s_u$, which implies that $\mathit{eff}'(c'_{il})$ has no negative component corresponding to any of the message types $m_1, \ldots, m_k$. Furthermore, since $a > b$, we still have some remaining message sending statements $s_{b+1}, \ldots, s_a$ that would make some components $\mathit{eff}'(c'_{il})^j$ $(n \leq j \leq n')$ positive.

We now start to prove the proposition based on the above discussion. By contradiction, we assume that S' is bounded while S is unbounded. Since S is unbounded, the sufficient and necessary condition as stated in Proposition 5.4 implies a solution to the following ILP problem:

$$\mathit{eff}(c_1)^j \cdot x_1 + \cdots + \mathit{eff}(c_p)^j \cdot x_p \geq 0 \quad \text{for each} \quad 1 \leq j \leq n \tag{7.17}$$

$$\sum_{j=1}^{n} \mathit{eff}(c_1)^j \cdot x_1 + \cdots + \sum_{j=1}^{n} \mathit{eff}(c_p)^j \cdot x_p > 0 \tag{7.18}$$

Let one of the solutions to the above ILP problem be $x_1 = v_1, \ldots, x_p = v_p$. We show in the following that from this solution we can construct a solution to the ILP problem representing the sufficient and efficient condition of the unboundedness of S', as shown in Inequalities (7.19–7.20), thereby contradicting the assumption.

$$\sum_{i=1}^{q_1} \mathit{eff}'(c'_{1i})^j \cdot x_{1i} + \cdots + \sum_{i=1}^{q_p} \mathit{eff}'(c'_{pi})^j \cdot x_{pi} \geq 0 \quad \text{for each} \quad 1 \leq j \leq n' \tag{7.19}$$

$$\sum_{i=1}^{q_1} (\sum_{j=1}^{n'} \mathit{eff}'(c'_{1i})^j) \cdot x_{1i} + \cdots + \sum_{i=1}^{q_p} (\sum_{j=1}^{n'} \mathit{eff}'(c'_{pi})^j) \cdot x_{pi} > 0 \tag{7.20}$$

We construct a solution to the above ILP problem from the solution $x_i = v_i$ as follows. Let $C^{>} \subseteq C$ contain all cycles c_i such that $v_i > 0$ and $\mathit{eff}(c_i)^n > 0$, $C^0 \subseteq C$ contain all cycles c_i such that $v_i > 0$ and $\mathit{eff}(c_i)^n = 0$, and $C^{<} \subseteq C$ contain all cycles c_i such that $v_i > 0$ and $\mathit{eff}(c_i)^n < 0$. We describe in the following three different selection procedures to select corresponding cycles in C' for the cycles in these three subsets of C respectively. The selected cycles will receive positive coefficients in the constructed ILP solution.

Selecting cycles for $C^{>}$. For each cycle $c_i \in C^{>}$, we select nondeterministically a cycle $c'_{il} \in C'$, which corresponds to c_i, as long as $\mathit{eff}'(c'_{il})^j \geq 0$ holds for each $n \leq j \leq n'$. We use $\mathit{sel}(c_i)$ to denote the selected cycle for each $c_i \in C^{>}$. We calculate the summary effect $E^{>}$ of all selected cycles from C', assuming that each selected cycle is executed as many times as its corresponding cycle in $C^{>}$, as follows.

$$E^{>} = \sum_{c_i \in C^{>}} \mathit{eff}'(\mathit{sel}(c_i)) \cdot v_i \tag{7.21}$$

We can easily see that $(E^{>})^j \geq 0$ holds for each $n \leq j \leq n'$, and that there exists at least one j ($n \leq j \leq n'$) such that $(E^{>})^j > 0$. Furthermore, following the property stated in Inequality (7.2), we have that

$$\sum_{j=n}^{n'}(E^{>})^j = \sum_{c_i \in C^{>}} \mathit{eff}(c_i)^n \cdot v_i > - \sum_{c_i \in C^{<}} \mathit{eff}(c_i)^n \cdot v_i. \tag{7.22}$$

Selecting cycles for C^0. For each $c_i \in C^0$, we select a cycle $c'_{il} \in C'$ if it corresponds to c_i and $\mathit{eff}'(c'_{il})^j = 0$ holds for any $n \leq j \leq n'$. We also use $sel(c_i)$ to denote selected cycles.

Selecting cycles for $C^{<}$. The principle of this selection procedure is to guarantee that, for each message type $m_i \in \{m_1, \ldots, m_k\}$, the summary effect of the selected cycles from C' should not over-consume the number of messages of the type m_i generated by $E^{>}$. We use a counter $z(j)$ for each j-th component in $E^{>}$ where $n \leq j \leq n'$, which is initialized to $(E^{>})^j$. During the selection procedure, each counter $z(j)$ records how many messages of the type m_{j-n+1} have not been consumed by already selected cycles. Moreover, we assign a positive number $co(c')$ to each cycle $c' \in C'$ being selected to denote its respective coefficient in the constructed solution. We also use a counter $co(i)$ for each cycle $c_i \in C^{<}$, which is initialized to v_i. During the selection procedure, $co(i)$ shows the upper bound on the coefficient that can be assigned to the next selected cycle corresponding to c_i. The selection procedure is described as follows.

We look for the smallest index j, where $n \leq j \leq n'$, such that $(E^{>})^j > 0$. Note that the j-th component in $E^{>}$ corresponds to the message type m_{j-n+1}. Then, we start the selection with j and the cycle in C^{-} with the smallest subscript. Suppose that this cycle is c_i. Then, this cycle consumes $-(\mathit{eff}(c_i)^n \cdot v_i)$ messages of the type m in the solution to the ILP problem (7.17–7.18). The idea is to select some cycles for c_i to consume as many messages of the type m_{j-n+1} as possible. The selection is an iterative procedure. Each iteration is in one of the following three cases:

- If $z(j) \geq -(\mathit{eff}(c_i)^n \cdot co(i))$, then we select the cycle c'_{il} such that $\mathit{eff}'(c'_{il})^j = \mathit{eff}(c_i)^n$. We set $co(c'_{il}) = co(i)$. By now we have finished the selection of cycles corresponding to c_i. If there is no other cycle in $C^{<}$ for which we need to select cycles, then the selection procedure terminates. Otherwise, we look for the next cycle to process, i.e., the cycle $c_u \in C^{<}$ whose subscript is the smallest one satisfying $u > i$. Furthermore, if $z(j) = -(\mathit{eff}(c_i)^n \cdot co(i))$, then we have consumed all messages of the type m_{j-n+1}. In this case, we continue the selection with c_u and the w-th component of $E^{>}$ such that w is the smallest index satisfying $w > j$ and $(E^{>})^w > 0$. Such an index w must exist following the Inequality (7.22). Otherwise, if $z(j) > -(\mathit{eff}(c_i)^n \cdot co(i))$, then we reduce $z(j)$ by $-(\mathit{eff}(c_i)^n \cdot co(i))$ and continue the selection with c_u and the j-th component.

- If $z(j) < -(\mathit{eff}(c_i)^n \cdot co(i))$ and $z(j) \geq -\mathit{eff}(c_i)^n$, then we select the cycle c'_{il} such that $\mathit{eff}'(c'_{il})^j = \mathit{eff}(c_i)^n$. We then set $co(c'_{il})$ to the largest number u such that $z(j) \geq -\mathit{eff}(c_i)^n \cdot u$. We also reduce $co(i)$ by u. Moreover, if $z(j) = -\mathit{eff}(c_i)^n \cdot u$, then we have consumed all messages of

the type m_{j-n+1}. In such case, we continue the selection with c_i and the w-th component of $E^>$ such that w is the smallest index satisfying $w > j$ and $(E^>)^w$. Otherwise, if $z(j) > -\mathit{eff}(c_i)^n \cdot u$, then we reduce $z(j)$ by $-(\mathit{eff}(c_i)^n \cdot u)$ and continue the selection with c_i and the j-th component.

- If $z(j) < -\mathit{eff}(c_i)^n$, then we construct an effect vector e as follows. Initially, $e^u = \mathit{eff}(c_i)^u$ for each $1 \leq u \leq n-1$, and $e^u = 0$ for each $n \leq u \leq n'$. We add a new counter z and set it to $-\mathit{eff}(c_i)^n$. First, we set $e^j = -z(j)$. We reduce z by $z(j)$. Next, we look for the u-th component of $E^>$ such that u is the smallest index satisfying $u > j$ and $(E^>)^u > 0$. Then, we set $e^u = -max\{z(u), z\}$. Next, we reduce z by $-e^u$, and also reduce $z(u)$ by $-e^u$. If $z > 0$, then we look for the next smallest index u' such that $u' > u$ and $(E^>)^{u'} > 0$ and set $e^{u'}$ to the proper value. We repeat this construction until $z = 0$. We can easily see that there is a cycle $c'_{il} \in C'$ with $\mathit{eff}'(c'_{il}) = e$. We select c'_{il} with $co(c'_{il}) = 1$. We reduce $co(i)$ by 1. If $co(i) = 0$, we look for the cycle $c_u \in C^<$ whose subscript is the smallest one satisfying $u > i$. If no such c_u exists, then the selection procedure terminates. Otherwise, let the w-th component be the last component of $E^>$ used in the previous construction of e. If $z(w) > 0$, then we continue the selection procedure with c_u and the w-th component. Otherwise, we continue the selection with c_u and the the next smallest index w' such that $w' > w$ and $(E^>)^{w'} > 0$.

From the selection of cycles for $C^>$, C^0 and $C^<$ above, we construct the solution to the ILP problem (7.19–7.20) as follows. For any $c_i \in C^> \cup C^0$, we set the coefficient of $sel(c_i)$ to v_i. For any $c_i \in C^<$ and for each c' of the selected cycles for c_i, we set its respective coefficient to $co(c')$. For any other x_{il} in the ILP problem, we set its value to 0. We can easily prove that the constructed solution is a real solution, following the selection procedures and the properties as stated in Inequalities (7.1) and (7.2). We omit this proof here, which can be derived in a similar way as in the end of the proof of Proposition 7.1. □

The above proposition only states that the buffer boundedness test returns no better result when we refine one message type in the optimal message type definition obtained by our proposed method. However, we can easily see that any finer message type definition would not lead to a more precise boundedness analysis.

7.3 Replication of Identical Processes

In this section we discuss the impact of having replications of identical processes on the buffer boundedness and livelock freedom analysis. Intuitively, two identical instances of a proctype contribute the same set of control flow cycles. Because we abstract from dependencies among control flow cycles in both verification methods, the replications of identical processes do not provide more information for the analysis than having just one copy of the processes.

Given a Promela model, suppose that there are two identical instances p_1 and p_2 of a same proctype. According to the code abstraction procedure that we proposed in Section 7.1, there are two processes p'_1 and p'_2 in the constructed CFSM system that correspond to p_1 and p_2 respectively. Obviously, p'_1 and

p'_2 are defined identically. Consequently, in the resulting independent cycle system $S = (n, C, \mathit{eff})$, there are two subsets of cycles $\{c_1, \ldots, c_n\} \subset C$ and $\{c'_1, \ldots, c'_n\} \subset C$, corresponding to p'_1 and p'_2 respectively, such that $\mathit{eff}(c_i) = \mathit{eff}(c'_i)$ for each $1 \leq i \leq n$. The buffer boundedness determination ILP problem for the Promela model is then given as follows:

$$\cdots + \sum_{i=1}^{n} \mathit{eff}(c_i) \cdot x_i + \sum_{i=1}^{n} \mathit{eff}(c'_i) \cdot x'_i + \cdots > \bar{0} \tag{7.23}$$

On the contrary, if we decide to construct a CFSM process only for p_1 and discard p_2, then the resulting ILP problem is the same as the ILP problem in Inequalities (7.23) except that the $\sum_{i=1}^{n} \mathit{eff}(c'_i)$ part is missing on the left hand sides of the inequalities, as shown below.

$$\cdots + \sum_{i=1}^{n} \mathit{eff}(c_i) \cdot x_i + \cdots > \bar{0} \tag{7.24}$$

We show that there is a solution to the ILP problem in Inequalities (7.23) if and only if there is a solution to the ILP problem in Inequalities (7.24). If there is a solution to the former ILP problem in which $x_i = v_i$ and $x'_i = v'_i$, then we can construct a solution to the latter ILP problem by assigning $(v_i + v'_i)$ to each x_i and copying the values of other variables in the solution to the former ILP problem. If there is a solution to the latter ILP problem in which $x_i = v_i$, then we can construct a solution to the former ILP problem by assigning v_i to x_i, assigning 0 to x'_i, and copying the values of other variables in the solution to the latter ILP problem. Therefore, if we are interested in no more than just determining the buffer boundedness of a Promela model, then we need only to construct one CFSM process for all identical copies of same processes. This improves the efficiency of the analysis as it reduces the size of the boundedness determination ILP problem. We can easily see a similar argument for the determination of livelock freedom.

However, if we want to estimate buffer bounds for a Promela model, then we should not disregard any of the identical instances of a proctype. This is because more identical instances contribute more message passing effects of acyclic paths of the respective proctype. Nevertheless, we can still construct only one CFSM process for all identical processes of some proctype p, and remember the replication factor of p, i.e., the number of identical copies of the Promela processes. In this way, we can just multiply the acyclic path contribution of p by its replication factor in the over-approximation of the acyclic path effect of the overall system.

7.4 Buffer Assignments

The buffer names declared in a Promela model are actually variables of the type `chan`. Each buffer variable keeps a pointer to an actual FIFO message queue at runtime. The queue that a buffer variable currently points to can be changed through buffer assignments. Consider the Promela model in Listing 7.4. The buffers `ch1` and `ch2` initially point to two different message queues that are assigned to them at the beginning of the execution. However, after the

```
1  mtype = {msg1};
2
3  chan ch1 = [2] of {mtype};
4  chan ch2 = [2] of {mtype};
5
6  active proctype P(){
7      ch1 = ch2;
8      ch1!msg1;
9
10     do
11     :: ch1?msg1 ->
12        ch2!msg1;
13        ch2!msg1;
14     od
15 }
```

Listing 7.4: A Promela model with buffer assignments.

assignment at Line 7, `ch1` points to the same queue as the one that `ch2` points to, and the queue that `ch1` originally pointed to is simply discarded. We can easily see that the remaining queue that both buffer variables point to is then flooded by messages `msg1`.

A simple abstraction dealing with buffer assignments works as follows. Wherever we find a buffer assignment `ch1 = ch2` *anywhere* in the model, we distinguish no longer messages exchanged in `ch1` and `ch2`. More precisely, let R be the set of message receiving statements with constant message fields that we collect in the model. Let $R_{\texttt{ch1}}$ and $R_{\texttt{ch2}}$ contain all statements in R that receive messages from `ch1` and `ch2` respectively. We modify the coverage set of each statement in $R_{\texttt{ch1}}$ and $R_{\texttt{ch2}}$ in the following way: For any statement $s \in R_{\texttt{ch1}}$, we construct a set of messages $M_s^{\texttt{ch2}}$ from M_s by replacing the buffer name `ch1` in all messages in M_s with the buffer name `ch2`. Then, the new coverage set of s is the union of $M_s^{\texttt{ch2}}$ and M_s. The new coverage sets of statements in $R_{\texttt{ch2}}$ can be obtained in a similar way. For identifying message types *jointly* for `ch1` and `ch2`, we use the new coverage sets of statements in $R_{\texttt{ch1}} \cup R_{\texttt{ch2}}$ to partition the set $M_{\texttt{ch1}} \cup M_{\texttt{ch2}}$. Any subset in the partition is identified as a message type. Furthermore, consider a statement s that sends a message to (or receives a message from respectively) from `ch1` or `ch2`. In order to determine the message passing effects of s, we also construct its new coverage set in the same way as in the message type identification. A type m of messages can be sent (or received) by s if the intersection of the new coverage set and m is non-empty. As an example, the new coverage set of the only message receiving statement (Line 11) in Listing 7.4 is $M_s = \{\langle ch1, msg1\rangle\} \cup \{\langle ch2, msg1\rangle\}$. As a result, there is only one identified message type as M_s. The message sending statement at Line 12 has a new coverage set as the same as M_s, and it can certainly send a message of the type M_s.

Since we abstract from message orders, the above abstraction is an over-approximation. We omit the formal proof for this argument. The intuition is that not distinguishing messages in two message buffers results in the merging of certain inequalities in the resulting buffer boundedness or livelock freedom determination ILP problem.

```
1  mtype = {msg1};
2
3  chan ch1 = [2] of {mtype};
4  chan ch2 = [2] of {mtype};
5
6  active proctype P(){
7      ch1!msg1;
8
9      do
10     :: ch1?msg1 ->
11        ch2!msg1;
12        ch2!msg1;
13     od
14
15     ch1 = ch2;
16 }
```

Listing 7.5: A Promela model with buffer assignments.

The proposed abstraction is also relatively coarse because a buffer assignment does not necessarily affect all parts of the model. Consider the Promela model in Listing 7.5, which is the same as the model in Listing 7.4 except that the buffer assignment occurs after the `do` loop. Apparently, the buffer assignment `ch1 = ch2` can never be executed, and therefore does not affect the loop where `ch1` and `ch2` still point to separate queues.

We propose a finer over-approximation based on strongly connected components (or SCCs). An SCC in a directed graph is a subgraph in which any vertex is reachable from any other vertex. By collapsing all the vertices in the same SCC into a single vertex, we obtain a directed acyclic graph (or DAG). In the DAG each vertex denotes an SCC in the original graph. Let C_1 and C_2 be two strongly connected components in the original graph. If there is an edge from one of the states in C_1 to one of the states in C_2, then there is an edge from the vertex in the DAG, that corresponds to C_1, to the vertex corresponding to C_2.

Given a Promela process that contains buffer assignments, we derive the DAG from the control flow graph of the process. It's obvious that a buffer assignment in some SCC can only affect those SCCs reachable from it in the DAG. For parallel processes, a buffer assignment in one process can affect every part of every other process running in parallel. Based on this idea, we can use a finer abstraction in which we still identify message types separately for two buffers `ch1` and `ch2` that may point to a same message queue in the model. Consider a buffer assignment `ch1 = ch2` in a strongly connected component C. For any statement s that sends messages to or receives messages from `ch1` or `ch2`, if s is in a SCC that can be affected by C, then we construct the new coverage set of s, in the same way as in the previous coarser abstraction approach, in order to determine its message passing effects. Otherwise, we still use the original coverage set M_s to determine message passing effects.

7.5 Buffer Arrays

An array of buffers can be declared in the following way

```
chan ch[3] = [5] of {mtype}.
```

The buffer array `ch` consists of three buffers indexed from 0 and 2. For instance, the statement `ch[1]!msg` sends a message to the buffer indexed at 1. However, if a buffer index is a variable, then its value is in general unknown statically. As an example, the statement `ch[x]!msg` uses an integer variable `x` to index the array element. Then, we do not know at compile time to which buffer element this statement will send messages at runtime.

A simple solution is to model the above statement as nondeterministically sending messages to any element of `ch`, assuming that the run-time values of `x` always fall inside the range of the buffer array indices.

The above solution may cause a too coarse over-approximation. In practice, we observe in many realistic Promela models that a variable used as a buffer array index is initialized to a constant value and never changed again. If this is the case, then we can substitute the occurrence of the variable in the model with its initialized constant. This results in a much finer abstraction.

7.6 Unbounded Process Creations

The SPIN model checker limits the number of parallel processes to an implementation dependent constant which is in most installations 255. If one takes such a limit for granted then process creation alone could not lead to buffer unboundedness. However, we drop the restriction on the number of running processes in the Promela language in order to investigate the impact of unbounded process instantiations on the checking of buffer boundedness.

Unbounded process creations can result in unbounded buffers. The first example is demonstrated by the Promela model in Listing 7.6. The instance of the proctype `P` repeatedly creates instances of the proctype `Q`. There is an execution of the model in which the instance of `P` immediately creates a new instance of `Q` after the previously created instance of `Q` sends a message `msg1` to the buffer `ch`. This execution apparently floods the buffer `ch` with messages `msg1`.

However, the unboundedness of the buffer `ch` in the above example cannot be detected by our buffer boundedness test: According to Section 7.3, since all the instances of `Q` are identical, we need only one CFSM process for the proctype `Q` in the constructed CFSM system. It is easy to see that there is no positive linear combination of cycles and the boundedness test thus returns "BOUNDED". A straightforward solution to this problem is to add an extra backward edge in the control flow graph of `Q` from each control point to the initial point. We call these backward edges as *replication edges*. The replication edges result in the CFSM process for `Q` as shown in Figure 7.6. The dotted transitions correspond to replication edges. Unboundedness can be detected now because there are two cycles in the process of `Q` and one of them contains nothing but a message sending statement. This cycle alone would cause the system to be unbounded.

The second example of unbounded process creations is through self-creations as shown in the Promela model in Listing 7.7. The buffer `ch` is unbounded

```
1  mtype = {msg1, msg2};
2  chan ch = [2] of {mtype};
3
4  proctype Q(){
5      ch!msg1;
6      ch?msg2;
7  }
8
9  active proctype P(){
10     do
11     :: run Q()
12     od;
13 }
```

Listing 7.6: A Promela model with unbounded process instantiations.

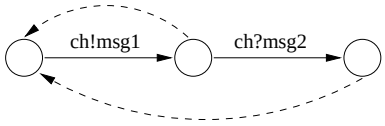

Figure 7.6: The replication edge resulting CFSM process for the proctype `Q` in Listing 7.6

because every new instance of `P` creates a new copy of itself and every instance sends a message to the buffer. Similarly, the unboundedness is also not detected by the buffer boundedness test. In this case we may also use replication edges to detect self-creations. However, we only need to add replication edges from the source control points of the edges of self-creations. This results in the CFSM process for `P` as shown in Figure 7.7. Similarly, unboundedness can be detected solely by the only cycle in the process as a consequence of having an extra dotted transition that corresponds to the replication edge.

The last example of unbounded process creations is through mutual instantiations as demonstrated by the Promela model in Listing 7.8. An instance of `P` creates an instance of `Q` which creates an instance of `R`. This instance of `R` creates in turn another new instance of `P`. The buffer `ch` is obviously flooded with messages `msg1` at runtime. The buffer boundedness test cannot detect this unboundedness either.

One way of dealing with this is to use replication edges again to detect the

```
1  mtype = {msg1};
2  chan ch = [2] of {mtype};
3
4  active proctype P(){
5      ch!msg1;
6      run P();
7  }
```

Listing 7.7: A Promela model with self-instantiations.

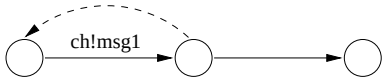

Figure 7.7: The replication edge resulting CFSM process for the proctype `P` in Listing 7.7

```
1  mtype = {msg1};
2  chan ch = [2] of {mtype};
3
4  active proctype P(){
5      ch!msg1;
6      run Q();
7  }
8
9  active proctype Q(){
10     ch!msg1;
11     run R();
12 }
13
14 active proctype R(){
15     ch!msg2;
16     run P();
17 }
```

Listing 7.8: A Promela model with mutual instantiations.

unboundedness caused by mutual creations. In this case, however, a replication edge does not go back to the respective initial control point. Instead, for instance, the replication edge in the control flow graph of `P` goes across the process boundary to the initial control point of the control flow graph of `Q`. This results in a united state machine as shown in Figure 7.8, instead of having an individual state machine for each of `P`, `Q`, and `R`. Thereby, our buffer boundedness analysis would no longer be local to individual processes, but has to consider the complete system as a whole, which would significantly increase its complexity. Unlike previously, the runtime required by the boundedness test is now exponential in the number of parallel processes.

A second solution is to add local replication edges in the control flow graphs of proctypes that can, directly or indirectly, instantiate themselves. This is certainly a coarser over-approximation. However, using this solution avoids inter-process control flow edges and therefore keeps the boundedness analysis local and efficient.

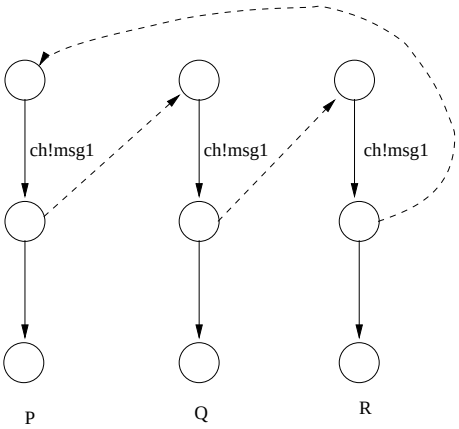

Figure 7.8: The replication edge resulting CFSM system for the model in Listing 7.7

Chapter 8

Abstracting UML RT Models

In the previous chapter we discussed how to abstract Promela models into CFSM systems for the buffer boundedness and livelock freedom tests. In this chapter we concern ourselves with the specific issues related to the abstraction of UML RT models.

The UML RT language is much more expressive than the Promela language. For instance, it allows for hierarchical state machines in the behavioral specification of a model. Moreover, UML RT uses program code written in high-level programming languages, such as Java or C++, for the specification of transition actions. While these advanced features give users more power in modeling real life systems, they also make it difficult to abstract UML RT models.

Although UML RT has richer features than Promela, UML RT has many aspects in common with Promela. Some of these common aspects are listed below. Therefore, all abstraction techniques treating these aspects in Promela, as we proposed in the previous chapter, can be easily adapted for the abstraction of UML RT models.

- A UML RT model is defined through a set of *capsule* classes. Each capsule class may have none, one, or more instances at runtime. The concepts of capsule classes and capsule instances are similar to the concepts of proctypes and proctype instances in Promela, except that there are no parameterized capsule classes in UML RT. However, because of the use of hierarchical structures, instances of one same capsule class may be embedded as sub-capsules inside different capsules. Furthermore, they may be connected with different capsules and exhibit therefore different message passing behaviors.

- The ports of a capsule are the communication interface of the capsule. As a proper abstraction of all possible scheduling mechanisms, we assume in Section 3.4 that each port corresponds to one unique message queue to store incoming messages. Thus, a port is not different from a `chan` variable in Promela, except that a port cannot be re-assigned to a different queue at runtime. Moreover, a port may be replicated, i.e., it corresponds to a set of ports indexed by natural numbers. This is similar to the concept of

buffer arrays in Promela.

- Variables can be used in message sending statements in UML RT models, whose runtime values are in general unknown statically. We take the same approach as for Promela models and assume that these variables can contain any runtime values of the respective type.

This chapter will leave out all the above listed aspects, and only focus on the specific issues related to the abstraction of UML RT models. These include the identification of message types, the treatment of hierarchical state machines, and the abstraction of the action code of transitions.

Structure. We first discuss how to identify message types in Section 8.1. The abstraction of hierarchical state machines is then considered in Section 8.2. Section 8.3 concentrates on the automated abstraction of the action code of transitions in UML RT state machines.

8.1 Identifying Message Types

The UML RT language supports a concept of message types, which can be straightforwardly mapped to message types in CFSM systems. We first recall some message related concepts in UML RT.

In UML RT, every message consists of a mandatory field, called *message signal*, and an optional data field. A *message type* in UML RT is determined by a pair of a message signal symbol and a valid data type. A message belongs to a certain message type T if and only if the message contains (1) the same message signal symbol as in T and (2) a data field of the same data type as specified in T. Every port is associated with a *protocol* that stipulates which messages the port is allowed to send or receive. A protocol consists of two respective sets of message types for incoming messages and outgoing messages. Every transition in a state machine has a trigger that specifies which types of messages must be received from which port for firing the transition.

Based on the above discussion, we can map message types from UML RT to CFSM in the following way: For each port p in the model, let P be the protocol that p is associated with. Then, we regard every pair in $\{(p, T) \mid T \text{ is an incoming message type in } P\}$ as a message type to be included in the corresponding abstract CFSM system. Taking the example system of the Alternating Bit Protocol in Figure 3.1, the protocol `AB_Protocol` has the set of incoming message types as $\{(\texttt{ACK}, \texttt{int})\}$. The port `sp` is the only port associated with `AB_Protocol`. Therefore, we have a message type as $(\texttt{sp}, \texttt{ACK}, \texttt{int})$. Another port `rp` is associated with the conjugated version of `AB_Protocol`, i.e., the set of incoming message types and the set of outgoing message types are reversed. The conjugated protocol has the set of incoming message types as $\{(\texttt{BIT}, \texttt{int})\}$, and we have another message type as $(\texttt{rp}, \texttt{BIT}, \texttt{int})$.

8.2 Hierarchical State Machines

The state machine of a UML RT capsule may have a hierarchical structure as illustrated in Figure 8.1. The figure shows a state machine modeling a simple

handheld device display. The device has two function units: a phone call manager and a music player. When the device is switched on, the display is in the standby mode (the state `standBy`). Pressing any key on the device triggers the display to go to the active mode (taking the transition `anyKey`). The active mode is represented in the state machine as a *composite* state `active` which contains an inner state machine. It is the first time that the state `active` is entered, so the main menu is displayed. This is captured in the state machine by passing the control from the composite state `active` to its initial sub-state `mainMenu`. Now, the user can switch to the phone book display, or to the music player display, or back to the main menu by pressing proper keys. This corresponds to firing the transitions inside the state `active`. Moreover, if no key has been pressed for a certain period of time, the device switches to the standby mode as represented by the firing of the transition `timeout`. The device returns to the active mode as soon as any key is pressed (`anyKey`). In this case, the device should remember what was displayed at the last time when the active mode was exited. For instance, if the active mode was disabled while the user was listening to music, then the display shows the music player interface when the device is activated again. The knowledge of the most recently active display is maintained by the history node on the boundary of `active` which connects to the transition `anyKey`.

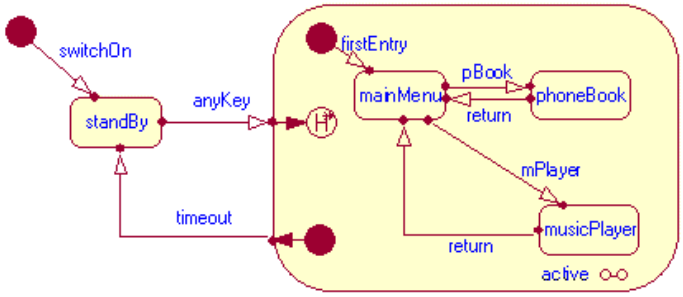

Figure 8.1: A UML RT hierarchical state machine.

In summary, a state in a hierarchical state machine may be either non-composite or *composite*. A composite state contains inside an inner state machine. A group transition is a transition leaving or entering a composite state such as the transitions `anyKey` and `timeout` in the above example. There are two cases when a group transition entering a composite state s is taken. If s is entered for the first time, then the control is passed to the initial state inside s. Otherwise, the control is re-directed to the most recently visited sub-state in s, i.e., the sub-state that was active when s was exited at the last time.

CFSM systems have only flat state machines. In order to approximate hierarchical state machines using CFSM systems, we may take any of the following approaches: Either (1) flatten the hierarchical state machines in the UML model, or (2) modify the CFSM formalism to allow for hierarchies. The two approaches are discussed in the subsequent subsections in the context of the buffer boundedness and the livelock freedom tests, during which we will argue that the non-flattening approach is more efficient.

8.2.1 Flattening Hierarchical State Machines

A simple way to flatten UML hierarchical state machine is as follows. Recall that, for any group transition t entering a composite state s, firing t may lead to entering one of the sub-states in s. Moreover, which sub-state is to be entered depends on the history and therefore unknown statically. As a safe over-approximation, we may assume that any sub-state of s can be entered. Then, for each of the sub-states s' of s, we add a transition from the source state of t to s' and then delete t from the state machine. This procedure is repeated until there is no more group transition entering a composite state. Next, we deal with all group transitions leaving a composite state in a similar way. Given an outgoing transition t from a composite state s, for each sub-state s' in s, we add a transition from s' to the destination state of t. When there is no more group transition in the state machine, we can then remove all the composite states. For the UML RT state machine in Figure 8.1, a flattening approach as described above will leave us with a flat state machine as in Figure 8.2. Note that each of the states `mainMenu`, `phoneBook`, and `musicPlayer` has two respective transitions to (or from) the state `standBy`. The size of the resulting flat state machine is not drastically increased. We even have fewer states in the flat state machine than in the hierarchical state machine. The number of transitions is increased in the flat state machine but bounded by $t \cdot n^2$ where t is the number of transitions in the hierarchical state machine and n is the number of non-composite states: the worst case is when all n non-composite states are within one composite state and all transitions are self-transitions of this composite state.

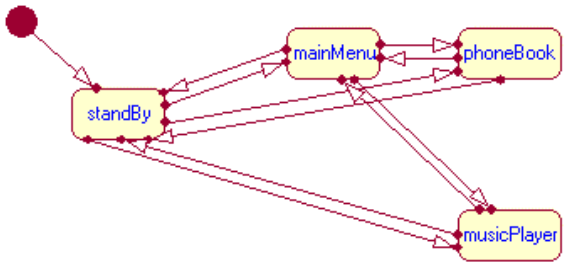

Figure 8.2: The flattened state machine of the state machine in Figure 8.1.

However, the above flattening approach results in a coarse flat state machine. In the above example, the flat state machine allows the device to awake and switch back to the phone book display even if the music player interface was shown when the device left the active mode at the last time. Such spurious behavior is introduced as a consequence of disregarding history information in the flattening procedure. Certainly, the introduction of spurious behavior may make the future analysis to become less imprecise.

Based on the above discussion, we can overcome the imprecision problem by maintaining the history information in the flat state machine. Based on this idea, a finer approach can be as follows: We add a new type of variables whose domain is the set of all non-composite states in the hierarchical state machine. For each composite state s, we add a new variable h_s of this type to the respective capsule class. Such h_s variable stores the history information related to the state s, and is initially set to the initial non-composite sub-state

inside s. Furthermore, for any transition t' leaving a state s' in the flat state machine, if t' corresponds to a group transition leaving a composite state s, then we include an extra action for t' to set the variable h_s to s'. For any transition t' entering a state s' in the flat state machine, if t' corresponds to a group transition entering a composite state s, then we include an extra condition for the firing of t' as $h(s) = s'$.

The above finer approach does not lose the history information and therefore results in a flat state machine behaviorally equivalent to the original hierarchical state machine. However, as explained in Chapter 5 and Chapter 6, the variables in a model are completely abstracted away in the code abstraction procedure in both the buffer boundedness and livelock freedom tests. The resulting CFSM system is consequently not different than the CFSM system obtained using the coarser flattening approach. Certainly, we may always come back to the flattened original model to regain the information of h_s variables in the refinement procedure. However, it causes extra overhead in the future analysis.

As we see from the above discussion, flattening approaches either result in imprecise abstractions, or cause extra overhead to fix the imprecision problem. As an alternative approach, we may consider revising the definition of CFSM systems to allow for hierarchical structures. In this way, transforming a UML RT state machine into a CFSM state machine becomes straightforward. While taking this approach, the only remaining question is how to collect control flow cycles in a hierarchical CFSM state machine, which will be discussed in 13.2.1.

8.2.2 Hierarchical Communicating Finite State Machines

[15] has extended the notion of CFSM systems, called *Communicating Hierarchical State Machines*, to have hierarchical structures. The extension does not only allow for hierarchies but also nested concurrency, i.e., a composite state may contain a set of concurrent regions and each region is a state machine. The state machines of all concurrent regions are executed in parallel once the containing composite state is entered. However, the concept of nested concurrency is not supported in the UML RT language. On the contrary, transitions across state boundaries and different hierarchical levels and the concept of histories in UML RT are not supported by Communicating Hierarchical State Machines. Therefore, we devise our notion of *Hierarchical Communicating Finite State Machines* to better support our analysis.

Definition 8.1 (Hierarchical Communicating Finite State Machines)**.** A system of *hierarchical communicating finite state machines* (HCFSM) is a quadruple

$$(P, M, B, succ)$$

where

- M, B, and $succ$ are defined as the same as in Definition 3.1.
- P is a finite set of *processes* p_i. Each p_i is a triple (S_i, sub_i, I_i, s_0^i) such that
  - S_i is a finite set of *states* of p_i.

- sub_i is a function $S_i \rightarrow \mathcal{P}(S_i)$. If $s' \in sub_i(s)$, then s' is a *child* of s and s is the *parent* of s'. A state s' is an *offspring* of s if (1) s' is a child of s, or (2) there is an offspring s'' of s such that s' is a child of s''. If s' is an offspring of s, then s is an *ancestor* of s'. We have the following restrictions on sub_i: (1) no two states share a same child; (2) no state is an offspring of itself; and (3) all states have a unique common ancestor s_t such that s_t is not a child of any other states. Furthermore, we call a state s a *composite* state if $sub_i(s) \neq \emptyset$. Otherwise, s is a non-composite state.
- $I_i \subseteq S_i$ is a set of non-composite states as initial states. We put the restrictions on I_i that for every composite state s there exists one and only one child s' of s such that $s' \in I_i$.
- $s_0^i \in I_i$ is the initial state of s_t.
- Finally, we put the restriction that $S_i \cap S_j = \emptyset$, i.e., S_i and S_j are disjoint.

The semantics of HCFSM systems is defined by making several modifications to the semantics of CFSM systems with respect to the newly introduced sub_i functions. First, a *configuration* of a HCFSM system $(P, M, B, succ)$ is a tuple $\langle s^1, \ldots, s^{|P|}, q^1, \ldots, q^{|B|} \rangle$ where each s^i is a non-composite state of the process $p_i \in P$ and each q^i is a queue of messages exchangeable in the buffer $b_i \in B$. A configuration $c = \langle s^1, \ldots, s^{|P|}, q^1, \ldots, q^{|B|} \rangle$ is an initial configuration if (1) for each s_i we have that $s_i = s_0^i$, and (2) each q_i is an empty queue. Let the set $f(s) = \{s\} \cup \{s' \mid s' \text{ is an ancestor of } s\}$ contains the non-composite state s and all its ancestors. An *execution* of the system is defined recursively as follows: (1) $\langle c_0 \rangle$ is an execution of the system if c_0 is an initial configuration; (2) If r is an execution, then $r.\langle c_2 \rangle$ is an execution if and only if the following are satisfied, assuming that the last configuration in r is $c_1 = \langle s_1^1, \ldots, s_1^{|P|}, q_1^1, \ldots, q_1^{|B|} \rangle$ and $c = \langle s_2^1, \ldots, s_2^{|P|}, q_2^1, \ldots, q_2^{|B|} \rangle$:

- For some i, there exist two states $s \in f(s_1^i)$ and $s' \in f(s_2^i)$ such that $(s, l, s') \in succ$ for some label l. Moreover, if s' is a composite state, then $s_2^i \in I_i$ when there is no offspring of s' in r. This corresponds to the case when s' is entered for the first time. Otherwise, s_2^i is the same as the last non-composite offspring of s' that occurs in r.
- For any other $i \neq j$, we have that $s_1^j = s_2^j$.
- Message queues are updated as in the semantics of CFSM systems.

In our implementation of the buffer boundedness and livelock freedom tests, we take the approach of using HCFSM systems. It is efficient because the hierarchical structures of the original UML RT model can be directly mapped to the hierarchical structures of the abstract HCFSM system. We also give an efficient cycle detection algorithm for HCFSM systems, which is presented in Section 13.2.1.

8.3 Abstracting Transition Action Code

A transition in a UML RT state machine may be labeled with arbitrary program code that specifies its action. Transition action code can be written in high-level programming languages such as C++ and Java. While the use of these programming languages greatly increases the modeling power of UML RT, they also introduce many difficulties in the design of a code abstraction procedure. Especially, we want to fully automate the abstraction procedure because manual abstractions are prone to human errors and require expertise in understanding the model and the language used to implement the transition action code.

The purpose of abstracting action code is to determine the message passing effects of the respective transition. As message receiving events are specified in transition triggers, action code may only contain message sending statements. Among many others, we list some of the difficulties in the abstraction of transition action code in the following.

Conditional statements. The use of `if-then-else` statements and `switch` (or `case`) statements results in several possible execution paths within a piece of program code. Each execution path may send a different number and different types of messages. Since the values of boolean conditions in the conditional statements are unknown statically, we cannot determine which execution paths are taken at runtime. As an over-approximating solution, we determine statically all possible execution paths of the action code of the considered transition t. For each path, we determine a set of message passing effects. Note that each path may have several possible effects because of the use of variables in the message sending statements. In this way, we determine all possible message passing effects of the transition t. Consequently, t has as many corresponding transitions in the abstract HCFSM system as the determined message passing effects. Each corresponding HCFSM transition is labeled with one distinct effect.

Loops. When the action code of a transition contains only conditional statements, it is still possible to determine all possible message passing effects of the transition. However, when the code contains loops, we may not even be able to give any over-approximation of the message passing behavior of the respective transition. Consider a loop in the action code which contains a message sending statement. How many messages are sent by this statement depends on the iteration times of the loop. However, even the problem of whether a loop is terminating or not is in general undecidable. The estimation of loop iteration times is more difficult. When the maximal iteration time of the considered loop cannot be determined, any estimation of the number of messages sent within the loop may not be safe. We will address the problem of program loops in Chapter 9.

Procedures. When the action code of a transition contains a call to a procedure f, we need to check if f contains any message sending statements. If this is the case, then we need to determine which types of messages and how many of them are sent in the execution of f. Moreover, there may be nested procedure calls, i.e., f may call another procedure f' which again may contain procedure

```
1  \\ The transition action code starts.
2  ... ...
3
4  f();
5  h();
6
7  ... ...
8  \\ The transition action code ends.
9
10 void f(){
11   ... ...
12
13   if (x > 0)
14     g1();
15   else
16     g2();
17
18   h();
19 }
```

Listing 8.1: A piece of transition action code that contains nested procedure calls.

calls. To deal with nested procedure calls, we can construct directed graphs called *procedure call graphs* (or PCGs) as illustrated in the following example.

The transition action code in Listing 8.1 calls first a procedure `f` and then a procedure `h`. The procedure `f` may call either `g1` or `g2` before it makes another call to `h`. Which of `g1` and `g2` is called depends on the runtime value of the variable `x`. We assume that the procedures `g1`, `g2`, and `h` do not make further procedure calls. We need to build two PCGs for the action code in this example, and each corresponds to one possible execution path within `f` as shown in Figure 8.3. The left PCG corresponds to the case when `g1` is called within `f`, and the right one corresponds to the case when `g2` is called. The root of each PCG corresponds to the considered action code, which is denoted by a solid black circle. Every other node denotes a distinct call to some procedure. Therefore, in each PCG in the example there are two nodes labeled with `h` because `h` is called twice respectively within the action code and within `f`. Given any node n in each PCG, for each procedure call made within the procedure that n represents, there is a child n' of n such that n' represents the callee procedure. For instance, the procedure `f` calls `h`, therefore the node labeled with `f` in each PCG has a child labeled with `h`. All children of a node are not necessarily ordered since we will later abstract from message orders.

If all constructed PCGs have finitely many nodes, then we can determine the set of possible message passing effects for the considered action code based on PCGs as follows. Consider the above example. For each PCG, we list all occurrences of procedure calls in the PCG: a call to `f`, a call to `g1` (or to `g2`), and two calls to `h`. We then determine the set of message passing effects for the transition action code and for each occurrence of the procedure calls. Suppose the determined sets are $E_1, \ldots, E_5$. Then, the set of message passing effects of the considered PCG contains all possible combinations of the concatenations of

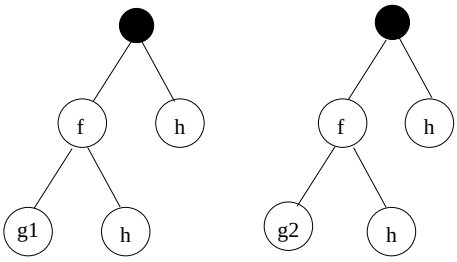

Figure 8.3: The procedure call graphs constructed for the transition action code in Listing 8.1

the message passing effects in each segment: $E_1.E_2.\ldots.E_5$. Finally, the set of message passing effects of the action code is the union of the sets of message passing effects of all constructed PCGs. This method can be straightforwardly generalized to any action code whose constructed PCGs are all finite.

The above method does not work when a piece of action code has PCGs that have infinitely many nodes. This may happen when a procedure call is in a loop, or when there are recursive calls or cyclic mutual calls, e.g., `f` calls `g` and `g` calls `f`.

Summary. We show above only some of the difficulties in the abstraction of UML RT transition action code. There are many other issues such as the use of pointers or aliases, the use of real-time controls, e.g., timeouts, and the dynamic construction and destruction of communication channels. All these difficulties currently impede the fully automatic abstraction and verification of UML RT models. In the next chapter we will tackle one of these difficulties – program loops. The other problems will be left for future work.

Chapter 9

Automated Termination Proofs

In the previous chapter we have shown the importance of developing automated code abstraction techniques for the analysis of UML RT models. It is however a difficult task as the action code of transitions can be written in a high-level programming language such as Java or C++. Therefore, one has to deal with many advanced programming features, such as pointers, program loops, and procedure calls, among others, which add tremendous complexities and difficulties to the design of automated abstraction methods. As a first step toward a fully automated code abstraction procedure for UML RT models, we consider here the abstraction of program loops, which are difficult to analyze due to the undecidability of their termination problem.

Within our verification framework, we need at least two kinds of information from a program loop in the action code of a transition. First, we need to determine whether the program loop always terminates. If the loop does not terminate, then the firing of the respective transition will never be completed, which causes the execution of the respective state machine stuck at the current state forever. Moreover, if the loop contains message sending statements, then we also need the information of how many times the loop iterates in order to determine the message passing effects of the respective transition. In this chapter we focus on the problem of *automatically* proving termination for program loops. In future work we will develop methods to estimate loop iteration times based on the automated termination proving techniques that we propose here.

We decide not to design termination proving methods for any particular programming language. Instead, we consider a generic and abstract form of loops that real program loops constructed in any high-level programming language can be easily transformed into. Furthermore, we do not consider the whole class of arbitrary loops, for which it is unlikely to build a common termination proving method. On the contrary, we consider an important class of loops, namely *deterministic multiple-path linear numerical loops with conjunctive conditions* or G^*P^*, to which a large number of loops in real life programs belong. Several subclasses of G^*P^* were already studied in [99, 115].

The basic idea of our methods is the following. Given a loop, we construct one or more region graphs in which regions define subsets of variable valuations

that have different effects on loop termination. In order to establish termination, we check for some of the generated region graphs whether (1) any region will be exited once it is entered, and (2) no region is entered an infinite number of times. We show the effectiveness of our proving method by experiments with Java code using a prototype implementation of our approach.

Structure. We first review some existing automated loop termination proving techniques in Section 9.1. We define loops, regions, and region graphs in Sections 9.2 and 9.3. The region graph based termination proving methods are explained for three subclasses of loops: (1) G^1P^1 in Section 9.4, (2) G^1P^* in Section 9.5, and (3) G^*P^1 in Section 9.6. We generalize these methods to handle the whole G^*P^* class in the end of Section 9.6. Experimental results are reported in Section 9.7.

9.1 Existing Automated Techniques for Proving Loop Termination

Automated termination proving has recently received intensive attention [99, 115, 24, 23, 41, 35]. Most of the recent work [99, 23] focuses on the construction of linear ranking functions. However, loops may not always possess linear ranking functions such as the loop in Example 9.1 (see Section 9.2).

[99] gives an efficient linear ranking function synthesis method for loops that can be represented as a linear inequality system. It considers nondeterministic update of variable values to allow for abstraction. However, the proposed analysis does not apply to multiple-path loops.

[23] can discover linear ranking functions for any linear loops over integer variables based on building ranking function templates and checking satisfiability of template instantiations that are Presburger formulas. The method is neither efficient nor guaranteed to terminate when applied to non-terminating loops.

[24] gives a novel solution to proving termination for polynomial loops based on finite difference trees. In fact it applies only to those polynomial loops whose behavior is also polynomial, i.e., the considered guarding function value at any time can be represented as a polynomial expression in terms of the initial guarding function value. Note that the simple linear loop in Example 9.1 does not have a polynomial behavior, for which this method would fail to prove termination.

[115] proves the decidability of termination for linear single-path loops over real variables. However, the decidability of termination for integer loops remains a conjecture.

[41] suggests a constraint solving based method of synthesizing nonlinear ranking functions for linear and quadratic loops. The method is incomplete due to the Lagrangian Relaxation of verification conditions that it takes advantage of.

The *Terminator* research project proposes a framework to prove termination for program loops and procedures based on the concept of transition invariants in a counterexample-guided abstraction refinement fashion [35, 36]. A transition invariant consists of boolean predicates describing the relation of variable values

before and after the execution of a certain program path. Proving termination for a given loop requires a transition invariant $\mathcal{P}$ and a set of well-founded relations $\mathcal{R}$ on variable values, both of which may be iteratively refined during the proving procedure. For each cyclic path p_i within the loop, an abstract boolean relation r_i on variable values is computed with respect to the current version of the transition invariant $\mathcal{P}$. Each resulting relation is a conjunction of predicates in $\mathcal{P}$ which over-approximates the behavior of the respective cyclic path. If every computed r_i is a subset of some well-founded relation in $\mathcal{R}$ then the termination of the loop is proved. Otherwise, there exists at least one cyclic path p_i whose computed relation r_i is not subsumed in any relation in $\mathcal{R}$. This cyclic path is called a counterexample whose spuriousness is determined automatically. If this counterexample is determined to be spurious, then either (1) $\mathcal{P}$ does not contain enough transition predicates, which results in coarse abstract relations to be computed for cyclic paths, or (2) $\mathcal{R}$ does not contain enough well-founded relations. In this case, refinement can be done by discovering new transition predicates or new well-founded relations. Transition predicates can be derived from the transition relations in the considered loop. The discovery of well-founded relations for $\mathcal{R}$ still relies on the synthesis of ranking functions using the above mentioned synthesis methods. The Terminator approach is certainly incomplete and cannot prove termination for all loops. In particular, it is not able to prove the termination of the loop in Example 9.1 since none of the above synthesis methods can construct a ranking function for the loop.

9.2 Loops

We formalize the class of loops that we consider in this chapter. Note that we only consider numerical loops, i.e., we will not consider the rounding and overflow problems with program variables as usually considered while analyzing programs.

Definition 9.1 (Deterministic Multiple-Path Linear Numerical Loops with Conjunctive Conditions)**.** The class *deterministic multiple-path linear numerical loops with conjunctive conditions*, or G^*P^* (multiple-guard-multiple-path), contains loops that have the following syntactic form:

while lc **do**
  $pc_1 \rightarrow \bar{x}' = U_1\bar{x} + \bar{u}_1$
  ...
  $pc_p \rightarrow \bar{x}' = U_p\bar{x} + \bar{u}_p$
od

where

- $\bar{x} = [x_1, ..., x_n]^T$ is a column variable vector where T denotes the transposition of matrices. $x_1, ..., x_n$ can be either integer variables or real variables. We use $\bar{x}' = [x'_1, ..., x'_n]^T$, the primed version of $\bar{x}$, to denote the new variable values after one loop iteration.
- $lc = \bigwedge_{i=1}^{m} lc_i$ is the loop condition. Each conjunct lc_i is a linear inequality in the form $\bar{a}_i\bar{x} \geq b_i$ where $\bar{a}_i = [a^i_1, ..., a^i_n]$ is a constant row vector of coefficients of variables and b_i is a constant. We call $\bar{a}_i\bar{x}$ a *guard*. The values of $\bar{a}_i\bar{x}$ are always bounded from below during loop iterations.

- Each $pc_i \rightarrow \bar{x}' = U_i\bar{x} + \bar{u}_i$ is a path with a path condition pc_i which is a conjunction of linear inequalities. We require that $\bigvee_{i=1}^{p} pc_i = \mathit{true}$, which guarantees a complete specification of the loop body. We further require that, for any i and j such that $i \neq j$, $pc_i \wedge pc_j = \mathit{false}$. This means that only one path can be taken at any given point in time.

- Each U_i is a constant matrix of dimension $n \times n$. Each $\bar{u}_i$ is a constant column vector of dimension n. They together describe how values of variables are updated along the i-th path.

If a loop has only one single path, then the loop body can be written as $\bar{x}' = U_1\bar{x} + \bar{u}_1$, in which we leave out the path condition *true*. Here are some examples of G^*P^* loops.

Example 9.1. This loop is an example of a loop without linear ranking functions [99]:
while $x \geq 0$ **do**
$x' = -2x + 10$
od

Example 9.2. This is a loop with two paths:
while $x \geq -4$ **do**
$x \geq 0 \rightarrow x' = -x - 1$
$x < 0 \rightarrow x' = -x + 1$
od

Example 9.3. This loop has more than one inequality in its loop condition:
while $x_1 \geq 1 \wedge x_2 \geq 1$ **do**
$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
od

The three examples above represent three interesting subclasses of G^*P^* that are studied in this chapter: (1) G^1P^1 are single-guard-single-path loops such as Example 9.1; (2) G^1P^* are single-guard-multiple-path loops such as Example 9.2; and (3) G^*P^1 are multiple-guard-single-path loops such as Example 9.3.

Definition 9.2 (Termination of Loops)**.** A loop is *terminating* if the loop always terminates starting with *any* initial assignment of variable values.

9.3 Region Graph

We define regions, positive and negative regions, standstill regions, and region graphs.

Definition 9.3 (Regions)**.** Given a loop, a *region* is a set of vectors of variable values such that

- all the vectors in the region satisfy the loop condition.

- it forms a convex polyhedron, i.e., it can be expressed as a system of linear inequalities.

We will also call a vector of variable values a *point*. We say that the loop iteration is *at some point* when the variables have the same values as in the point.

Definition 9.4 (Positive/Standstill/Negative Regions)**.** Given a loop and a guard in the loop condition, a *positive (standstill, negative, respectively) region with respect to the guard* is a region such that, starting at any point in the region, the value of the guard is decreased (unchanged, increased, respectively) after one iteration.

For instance, a positive region of Example 9.1 with respect to the guard x is $\{v \mid v > 10/3\}$, a negative region with respect to x is $\{v \mid 0 \leq v < 10/3\}$, and the only standstill region with respect to x is $\{10/3\}$. Moreover, if x is an integer variable, then there is no standstill region with respect to x. In the remainder of this chapter, when we mention a positive (or negative or standstill) region, we will omit the respective guard if it is clear from the context.

Definition 9.5 (Transitions of Regions)**.** Given a loop and two regions R_1 and R_2 of the loop, there is a *transition* from R_1 to R_2 if and only if, starting at some point p in R_1, a point p' in R_2 is reached after one iteration. R_1 is the *origin* of the transition. R_2 is the *target* of the transition.

In the definition, if R_1 and R_2 are distinct, then we say that R_1 is *exited* at p and R_2 is *entered* at p'. A transition is a *self-transition* if it starts and ends in one same region. We define that a self-transition on a region means that the region is neither exited nor entered.

For instance, there is a transition from the positive region $\{v \mid v > 10/3\}$ to the negative region $\{v \mid 0 \leq v < 10/3\}$ of Example 9.1 because $-2 \times 4 + 10 = 2$ while 4 is in the positive region and 2 is in the negative region.

Definition 9.6 (Region Graph)**.** Given a loop, a *region graph* is a pair $< \mathbb{R}, \mathbb{T} >$ such that

- $\mathbb{R}$ is a finite set of pairwisely disjoint regions such that the union of all the regions is the complete set of points satisfying the loop condition.
- $\mathbb{T}$ is the complete set of transitions among regions in $\mathbb{R}$.

In general, a region graph may contain regions that are neither positive, nor negative, nor standstill. However, the region graphs constructed by our termination proving methods contain only positive, negative, or standstill regions. A loop may have infinitely many region graphs.

In the chapter, we define a cycle in a region graph to be an elementary cycle that contains at least two transitions. In this way, we exclude self-transitions to be cycles. A region graph of Example 9.1 is illustrated in Figure 9.1, assuming that x is an integer variable. There is one cycle passing the two regions.

The basic idea of our region graph based termination proofs is stated in the following proposition.

Proposition 9.1. *Given a loop and one of its region graphs, the loop is terminating, if and only if, during loop iterations starting with any variable values, we have*

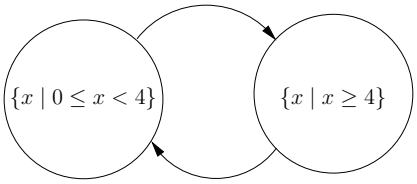

Figure 9.1: A Region Graph of the loop in Example 9.1.

- *once a region is entered, it will be exited eventually; and*
- *no region is entered infinitely often.*

Proof. For the "if" part, we assume by contradiction that the above two conditions are satisfied while the loop does not terminate with some initial variable values. Therefore, we can construct an infinite sequence δ of points, by recording the variable values, during the infinite loop iterations. Because there are only finitely many regions, there must be at least one region R such that an infinite number of points in δ are in R. If R will always be exited later after it has been entered, then R must be visited infinitely often. This contradicts the assumption that both conditions above are satisfied.

For the "only if" part, we assume by contradiction that the loop is terminating while one of the above conditions is not satisfied. If the first condition is not satisfied, then we have an infinite sequence δ of points such that (1) for each two adjacent points p_1 in Region R_1 and p_2 in Region R_2, there is a transition from R_1 to R_2; and (2) there is an infinite suffix of δ such that all the points in the suffix are in some same region. We can easily see that there is an non-terminating execution of the loop corresponding to δ. A similar argument can be made if we assume the second condition is violated. □

In the next sections we will show how to construct region graphs for proving termination.

9.4 Proving Termination for G^1P^1

We first show how to prove termination based on region graphs for loops in the simplest class G^1P^1. The concepts and methods described in this section can also apply to more general subclasses with little adaption as explained in the subsequent sections.

9.4.1 Constructing Region Graphs

Given a G^1P^1 loop as below,

while $\bar{a}\bar{x} \geq b$ **do**
  $\bar{x}' = U\bar{x} + \bar{u}$
od

we construct a region graph as follows in a straightforward way:

- The only positive region is defined by the system of the following linear inequalities if it has solutions. Otherwise, there is no positive region.

$$\bar{a}\bar{x} \geq b \tag{9.1}$$
$$\bar{x}' = U\bar{x} + \bar{u} \tag{9.2}$$
$$\bar{a}\bar{x} > \bar{a}\bar{x}' \tag{9.3}$$

- The only negative region is defined by the system of the following linear inequalities if it has solutions. Otherwise, there is no negative region.

$$\bar{a}\bar{x} \geq b \tag{9.4}$$
$$\bar{x}' = U\bar{x} + \bar{u} \tag{9.5}$$
$$\bar{a}\bar{x} < \bar{a}\bar{x}' \tag{9.6}$$

- The only standstill region is defined by the system of the following linear inequalities if it has solutions. Otherwise, there is no standstill region.

$$\bar{a}\bar{x} \geq b \tag{9.7}$$
$$\bar{x}' = U\bar{x} + \bar{u} \tag{9.8}$$
$$\bar{a}\bar{x} = \bar{a}\bar{x}' \tag{9.9}$$

- For a region R_1 defined by an inequality system I_1 and a region R_2 defined by I_2, there is a transition from R_1 to R_2 if the following system of inequalities has solutions:

$$\bigwedge_{e \in I_1} e \wedge \bigwedge_{e \in I_2} e[\bar{x} \mapsto \bar{x}', \bar{x}' \mapsto \bar{x}''] \tag{9.10}$$

  where $e[\bar{x} \mapsto \bar{x}', \bar{x}' \mapsto \bar{x}'']$ is the same inequality as e except that $\bar{x}$ is substituted with $\bar{x}'$ and $\bar{x}'$ is substituted with $\bar{x}''$ simultaneously.

The constructed region graph for Example 9.1 is exactly the one in Figure 9.1, assuming that x is an integer variable. The right region is positive and defined by the inequalities (9.11–9.13). The left region is negative and defined by the inequalities (9.14–9.16). There is a transition from the positive region to the negative region because the system of the inequalities (9.17–9.22) has solutions.

$$x \geq 0 \tag{9.11}$$
$$x' = -2x + 10 \tag{9.12}$$
$$x > x' \tag{9.13}$$

$$x \geq 0 \tag{9.14}$$
$$x' = -2x + 10 \tag{9.15}$$
$$x < x' \tag{9.16}$$

$$x \geq 0 \tag{9.17}$$
$$x' = -2x + 10 \tag{9.18}$$
$$x > x' \tag{9.19}$$
$$x' \geq 0 \tag{9.20}$$
$$x'' = -2x' + 10 \tag{9.21}$$
$$x' < x'' \tag{9.22}$$

Construction of region graphs can be fully automated since feasibility of linear inequality systems can be checked using linear optimization tools such as a linear programming problem solver.

Next, we propose a method of proving termination by studying region graphs.

9.4.2 Checking Regions

One of the two termination conditions in Proposition 9.1 is that any region will eventually be exited once it is entered. For any region without a self-transition, after it is entered, it will be exited after one iteration. For any positive region with a self-transition, the runtime values of variables cannot stay in the region forever. This is because the respective guard value is always decreased during self-transitions and also bounded from below as imposed by the loop condition. On the contrary, negative and standstill regions with self-transitions introduce the potential of staying in one region forever.

Every time that the self-transition of a negative region is taken, the respective guard value is increased. However, if the guard value has an upper bound within the region, then the self-transition cannot be continuously taken forever. In such a case, we call this region a *bounded* region.

The boundedness of a negative (or positive or standstill, respectively) region can be checked at the same time when the region is created during region graph construction. For instance, the system of the inequalities (9.14–9.16) defines the negative region of Example 9.1. We can use an optimizer to determine the maximum of guard values under the constraint of the inequalities (9.14–9.16) while checking feasibility, by adding the objective function $max : x$. In this example, the negative region is bounded since x has an upper bound 3 within the region.

Having an unbounded negative region, however, does not imply that the runtime values of variables can stay in the region forever. Consider Example 9.4 whose negative region is unbounded and defined by the inequalities (9.23–9.26). Note that the difference of the guard values before and after one iteration

is the value of x_2 before the iteration. By Inequality (9.25) we know that the value of x_2 is always decreased in this region and cannot remain positive forever. This implies eventual leaving of the region. We call such a region a *slowdown* region.

Example 9.4. This loop has an unbounded negative region.

while $x_1 \geq 0$ **do**

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

od

$$x_1 \geq 0 \tag{9.23}$$
$$x_1' = x_1 + x_2 \tag{9.24}$$
$$x_2' = x_2 - 1 \tag{9.25}$$
$$x_1' > x_1 \tag{9.26}$$
$$x_2' \geq x_2 \tag{9.27}$$

Checking whether a negative region R is a slowdown region can be done by checking the feasibility of a linear inequality system. The checked inequality system describes a subregion of R in which the difference of the respective guard value is increased or unchanged after one iteration. If no such a subregion exists, then R is a slowdown region. For instance, the negative region of Example 9.4 is a slowdown region because the system of the inequalities (9.23–9.27) has no solutions.

We generalize the concept of slowdown regions using an idea similar to the concept of finite difference trees [24]. For an unbounded negative region and an arbitrary natural number n, we build a finite chain $d_0, d_1, ..., d_n$ where the root d_0 is the difference of the respective guard values before and after one loop iteration within the region, and d_1 is the difference of d_0 before and after one iteration within the region, i.e, the "difference of difference", and so forth. When any d_i of the $d_0, d_1, ..., d_n$ is decreased within the region, the region is a slowdown region since d_i dominates the change of d_0, making it impossible to remain positive forever.

9.4.3 Checking Cycles

Eventual exiting of regions is not enough to show termination. We must make sure that no region is entered an infinite number of times.

In a region graph, if there are no cycles, then no region is entered infinitely often. The region graph in Figure 9.1 of Example 9.1 does not have this property. There is a cycle passing the positive region and the negative region. If this cycle can be taken forever, then both regions are entered infinitely often.

We observe that, for Example 9.1, if the negative region is entered at some point p, then it will be entered at the next time at such a point p' that the value of the guard x at p is greater than the value of x at p'. Because of the loop condition $x \geq 0$, we know that the cycle cannot be taken forever. So, no region is entered infinitely often.

We generalize the above idea by the following definition.

Definition 9.7 (Progressive Cycles)**.** A cycle is *progressive on a region* R if one of the following is satisfied:

- Along the cycle, every time that R is entered, the respective guard value is greater than the guard value at the last time that R is entered. In such a case, we say that the cycle is *upward progressive* if R is bounded.
- Along the cycle, every time that R is entered, the respective guard value is smaller than the guard value at the last time that R is entered. In such a case, we say that the cycle is *downward progressive*.

It is easy to prove that the following cycles are progressive: (1) a cycle passing the positive region and the standstill region, and (2) a cycle passing the negative region and the standstill region if the negative region is bounded.

For other types of cycles, we can check their progressiveness by checking feasibility of a set of linear inequality systems. We have at most six choices: checking whether the cycle is upward (or downward) progressive on the positive (or negative or standstill) region. For the purpose of illustration, we show how to check downward progressiveness on negative regions. The idea can be easily adapted for other choices and other cases.

Given a G^1P^1 loop as below,

while $\bar{a}\bar{x} \geq b$ **do**

$\bar{x}' = U\bar{x} + \bar{u}$

od

we assume that there is a cycle passing the positive region and negative region in its constructed region graph. If both regions have no self-transitions, then we can use the linear inequality system (9.28–9.37) to describe the behavior in which the respective guard value is not decreased every time that the negative region is entered along the cycle. The inequalities (9.28–9.30) define that the negative region is entered at a point $\bar{x}$. The inequalities (9.31–9.33) define that the positive region is then entered at $\bar{x}'$. The inequalities (9.34–9.36) define that the negative region is re-entered at $\bar{x}''$. Inequality (9.37) imposes that the guard value at $\bar{x}''$ is no smaller than the guard value at $\bar{x}$. If the inequality system has no solutions, then the guard value is always decreased and the cycle is downward progressive on the negative region.

$$\bar{a}\bar{x} \geq b \tag{9.28}$$
$$\bar{x}' = U\bar{x} + \bar{u} \tag{9.29}$$
$$\bar{a}\bar{x}' > \bar{a}\bar{x} \tag{9.30}$$
$$\bar{a}\bar{x}' \geq b \tag{9.31}$$
$$\bar{x}'' = U\bar{x}' + \bar{u} \tag{9.32}$$
$$\bar{a}\bar{x}' > \bar{a}\bar{x}'' \tag{9.33}$$
$$\bar{a}\bar{x}'' \geq b \tag{9.34}$$
$$\bar{x}''' = U\bar{x}'' + \bar{u} \tag{9.35}$$
$$\bar{a}\bar{x}'' > \bar{a}\bar{x}''' \tag{9.36}$$
$$\bar{a}\bar{x} \leq \bar{a}\bar{x}'' \tag{9.37}$$

If one of the regions above has a self-transition, then we do not know precisely at which point this region is exited after being entered. In such a case, we have to

overapproximate the exit point. Assume that both regions have a self-transition. The linear inequalities to check downward progressiveness are Inequalities (9.38–9.57). Note that the negative region is entered at a point $\bar{x}$ as defined by the inequalities (9.38–9.40), and it is exited at $p_{\bar{x}'}^{-}$ as defined by the inequalities (9.42–9.44). An additional inequality (9.41) guarantees that the successor $s_{\bar{x}}^{-}$ of $\bar{x}$ satisfies the loop condition because loop iterations cannot continue otherwise. Inequality (9.45) relates the entry point and the exit point by imposing that the guard value at $\bar{x}$ is no larger than the guard value at $p_{\bar{x}'}^{-}$ due to the effect of self-transitions of a negative region. Note that the "equal" part cannot be dropped since it is still possible to leave the negative region immediately without taking the self-transition. The inequalities (9.46–9.53) describe the entering and the exiting of the positive region similarly.

$$\bar{a}\bar{x} \geq b \tag{9.38}$$
$$s_{\bar{x}}^{-} = U\bar{x} + \bar{u} \tag{9.39}$$
$$\bar{a}s_{\bar{x}}^{-} > \bar{a}\bar{x} \tag{9.40}$$
$$\bar{a}s_{\bar{x}}^{-} \geq b \tag{9.41}$$
$$\bar{a}p_{\bar{x}'}^{-} \geq b \tag{9.42}$$
$$\bar{x'}^{-} = Up_{\bar{x}'}^{-} + \bar{u} \tag{9.43}$$
$$\bar{a}\bar{x}' > \bar{a}p_{\bar{x}'}^{-} \tag{9.44}$$
$$\bar{a}p_{\bar{x}'}^{-} \geq \bar{a}\bar{x} \tag{9.45}$$
$$\bar{a}\bar{x}' \geq b \tag{9.46}$$
$$s_{\bar{x}'}^{-} = U\bar{x}' + \bar{u} \tag{9.47}$$
$$\bar{a}\bar{x}' > \bar{a}s_{\bar{x}'}^{-} \tag{9.48}$$
$$\bar{a}s_{\bar{x}'}^{-} \geq b \tag{9.49}$$
$$\bar{a}p_{\bar{x}''}^{-} \geq b \tag{9.50}$$
$$\bar{x}'' = Up_{\bar{x}''}^{-} + \bar{u} \tag{9.51}$$
$$\bar{a}p_{\bar{x}''}^{-} > \bar{a}\bar{x}'' \tag{9.52}$$
$$\bar{a}\bar{x}' \geq \bar{a}p_{\bar{x}''}^{-} \tag{9.53}$$
$$\bar{a}\bar{x}'' \geq b \tag{9.54}$$
$$\bar{x}''' = U\bar{x}'' + \bar{u} \tag{9.55}$$
$$\bar{a}\bar{x}'' > \bar{a}\bar{x}''' \tag{9.56}$$
$$\bar{a}\bar{x} \leq \bar{a}\bar{x}'' \tag{9.57}$$

The progressiveness of each individual cycle is sufficient to show no infinite number of entering of any region only if any two cycles do not pass a same region (see the proof of Proposition 9.4). Otherwise, this condition is insufficient.

Definition 9.8 (Interfered Regions)**.** Given a region graph, if two cycles pass one same region, then we say that these two cycles *interfere* with each other on this region. The region is called an *interfered region* of both cycles.

Consider the region graph in Figure 9.2 where transitions are distinctly named for convenience. Two cycles[1] $\langle t_1, t_2 \rangle$ and $\langle t_1, t_3, t_4 \rangle$ interfere with each other on R_1 and R_2.

[1]In this chapter we denote a cycle by the sequence of transitions along the cycle.

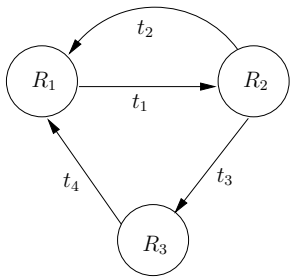

Figure 9.2: Two interfering cycles.

We say that a cycle is *completed* when, starting from a region in the cycle, the region is re-entered along the cycle. Furthermore, a cycle c is *uninterruptedly completed* if no other cycle is completed during the completion of c. If a cycle c_1 interferes with some other cycle c_2 on a region R, then a completion of c_1 can be interrupted at R to enter c_2 and resumed from R after c_2 is completed. In such a case, even if c_1 is progressive on some region R', R' may still be entered infinitely often since the respective guard value can be arbitrary when the completion of c_1 is resumed from R after one interruption. However, the following case deserves special attention.

Definition 9.9 (Base Regions and Orbital Cycle Sets)**.** A region R is a *base region* if the following is satisfied: For any cycle c that passes R, all the cycles that interfere with c also pass R. The set of cycles $\{c \mid c \text{ passes } R\}$ is called an *orbital cycle set*.

An orbital cycle set can have more than one base region. For instance, in Figure 9.2 both R_1 and R_2 are base regions of the orbital set consisting of two cycles. In contrast no region in Figure 9.3 is a base region.

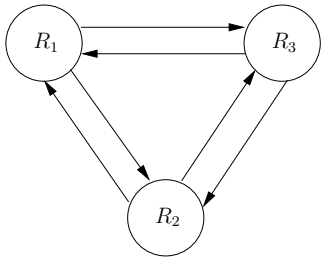

Figure 9.3: Three interfering cycles.

Orbital sets have an interesting property as follows.

Proposition 9.2. *Given a base region and its corresponding orbital set, between two successive times that the base region is entered, some cycle in the orbital set is uninterruptedly completed.*

Proof. We can easily see that a cycle is completed between two successive times that the base region is entered. Assume that this completion is interrupted at

some region R and resumed after some other cycle c is completed. Therefore, c is also in the same orbital set. During the completion of c, the base region must be entered, which contradicts the assumption that there is no entering of the base region in-between. □

Lemma 9.3. *Given an orbital cycle set O, any region in any cycle in O is entered only a finite number of times during loop iterations if all the cycles in O are uniformly upward or uniformly downward progressive on some base region.*

The proof of the above lemma is given in the proof of the soundness of the termination checking algorithm (see Proposition 9.4).

9.4.4 Determining Termination

Based on the previous discussion, we suggest a termination proving algorithm for loops in G^1P^1 as follows. Given a loop,

1. Check the existence of a standstill region[2]. If it exists, then check whether it has a self-transition. If the self-transition exists, then return "UNKNOWN".
2. Check the existence of a negative region. If neither a negative region nor a standstill region exists, then return "TERMINATING". In such a case, the loop has linear ranking functions (see Theorem 9.5).
3. If the negative region exists, then check whether it has a self-transition. If the self-transition exists and the region is unbounded, then check whether it is a slowdown region. If it cannot be determined to be a slowdown region, then return "UNKNOWN".
4. Complete construction of the region graph by constructing the positive region and the rest of the transitions.
5. Check if there are any cycles. If no cycle exists, then return "TERMINATING".
6. Construct all the orbital cycle sets. If there is any interfering cycle that does not belong to any orbital set, then return "UNKNOWN".
7. Check if all cycles are progressive. If there is one cycle whose progressiveness cannot be determined, then return "UNKNOWN". For each orbital set, check whether all cycles in the set are progressive on one base region and agree on the direction of progress (upward or downward). If it is satisfied, then return "TERMINATING". Otherwise, return "UNKNOWN".

All the steps in this algorithm are arranged in an optimal order so that no unnecessary step is taken. Since all the constructions and checks are performed by automatic translation into linear inequality systems and automated solving of these systems, the algorithm requires no human intervention.

Proposition 9.4. *The above given termination proving algorithm for G^1P^1 loops is sound.*

[2]Remember that the boundedness of a region is checked at the same time that the region is created.

Proof. There are three cases when the algorithm returns "TERMINATING": (1) there is no negative region and no standstill region; (2) there is a positive region, no negative region or a bounded negative region or a slowdown region or a negative region without a self-transition, no standstill region or a standstill region without a self-transition, and no cycle; (3) there is a positive region, no negative region or a bounded negative region or a slowdown region or a negative region without a self-transition, no standstill region or a standstill region without a self-transition, and all the cycles are progressive, and all the mutually interfering cycles form an orbital set and agree on the direction of progress on some base region. In order to prove that the checked loop is terminating, it is sufficient to show that in each case the two termination conditions in Proposition 9.1 are satisfied.

The cases (1) and (2) are trivial. We show the proof for the third case as follows. We first prove that any region will be exited once it is entered. It is obvious as discussed in Section 9.4.2. Next, we prove that no region is entered infinitely often by contradiction. Assume that there is one region being entered an infinite number of times with some initial variable values. Then, the region must be in a cycle. We have two possibilities. (1) The cycle does not interfere with other cycles. Then, the cycle can only be uninterruptedly completed. Because the cycle is progressive, it cannot be taken forever. So, the runtime values of variables will leave the cycle some time and cannot return to the cycle since the cycle will interfere with some other cycles otherwise. This contradicts the assumption. (2) The cycle interferes with other cycles. Then, the cycle is in some orbital set. So, all the cycles in the orbital set are progressive on some base region and agree on the direction of progress. Furthermore, this base region is also entered an infinite number of times. However, this base region cannot be entered infinitely often due to progressiveness. This is a contradiction. □

Complexity. Let N be a parameter to the algorithm as the upper bound on the length of finite difference chains built to check slowdown regions. The number of the linear inequality systems constructed by the algorithm is no more than $16 + N$. Each constructed inequality system has a size linear in the number of variables. If all the variables used in the loop are real variables, then solving of a linear inequality system is polynomial. Otherwise, it is NP-complete. However, in practice constructed inequality systems are usually very small. For the class of loops that have linear ranking functions, the algorithm in [99] needs to construct only one linear inequality system to determine termination, which seems much more efficient than our method. However, we can show that, for any G^1P^1 loop with linear ranking functions, its constructed region graph contains only one positive region as stated in Proposition 9.5. So, for any G^1P^1 loop that has linear ranking functions, our algorithm only generates 2 inequality systems to check the existence of a negative region and a standstill region.

Proposition 9.5. *A G^1P^1 loop over rational-valued variables has linear ranking functions if and only if its constructed region graph contains no negative region and no standstill region.*

Proof. The "if" part is simple. We show the proof of the "only if" part by contradiction as follows. For any G^1P^1 loop as below,

while $\bar{a}\bar{x} \geq b$ **do**

$\bar{x}' = U\bar{x} + \bar{u}$

od

is represented as suggested in [99], yielding the following form[3]:

$$\begin{bmatrix} B & B' \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{x}' \end{bmatrix} \leq \bar{c} \tag{9.58}$$

where

$$B = \begin{bmatrix} -\bar{a} \\ -U \\ U \end{bmatrix} \quad B = \begin{bmatrix} \bar{0} \\ I \\ -I \end{bmatrix} \quad \bar{c} = \begin{bmatrix} -b \\ \bar{u} \\ -\bar{u} \end{bmatrix}$$

where $\bar{0}$ is the all-zero row vector of dimension n if there are n variables in the loop, and I is the identity matrix of dimension $n \times n$.

By Theorem 2 in [99], if the loop has linear ranking functions, then there exist nonnegative row vectors $\bar{\lambda} = [\lambda_1, ..., \lambda_{2n+1}]$ and $\bar{\lambda}' = [\lambda'_1, ..., \lambda'_{2n+1}]$ such that the following inequalities hold:

$$\bar{\lambda}B' = 0 \tag{9.59}$$
$$(\bar{\lambda} - \bar{\lambda}')B = 0 \tag{9.60}$$
$$\bar{\lambda}'(B + B') = 0 \tag{9.61}$$
$$\bar{\lambda}'\bar{c} < 0 \tag{9.62}$$

Let $\bar{u} = [u_1, ..., u_n]^T$. Then, we have that

$$\bar{\lambda}'\bar{c} = -b\lambda'_1 + u_1\lambda'_2 + ... + u_n\lambda'_{n+1} - u_1\lambda'_{n+2} - ... - u_n\lambda'_{2n+1} < 0 \tag{9.63}$$

Now, assume that the constructed region graph has either a negative region or a standstill region, which implies that the following system of inequalities has solutions:

$$\bar{a}\bar{x} \geq b \tag{9.64}$$
$$\bar{x}' = U\bar{x} + \bar{u} \tag{9.65}$$
$$\bar{a}\bar{x}' \geq \bar{a}\bar{x} \tag{9.66}$$

This inequality system can be transformed to the form

$$D \begin{bmatrix} \bar{x} \\ \bar{x}' \end{bmatrix} \leq \bar{e} \tag{9.67}$$

where

$$D = \begin{bmatrix} -\bar{a} & \bar{0} \\ -U & I \\ U & -I \\ \bar{a} & -\bar{a} \end{bmatrix} \quad \bar{e} = \begin{bmatrix} -b \\ \bar{u} \\ -\bar{u} \\ 0 \end{bmatrix}.$$

By Corollary 7.1e (a variant of Farkas' Lemma) in [105], we know that, for any row vector $\bar{y} > 0$ such that $\bar{y}D = 0$, $\bar{y}\bar{e} \geq 0$. Let $\bar{y} = [\lambda'_1, ..., \lambda'_{2n+1}, \lambda_1]$, we can show that $\bar{y}D = 0$. So, $\bar{y}\bar{e} = -b\lambda'_1 + u_1\lambda'_2 + ... + u_n\lambda'_{n+1} - u_1\lambda'_{n+2} - ... - u_n\lambda'_{2n+1} \geq 0$. This is a contradiction. □

[3]Note that although [99] considers loops over integer variables, the theoretical results in [99], especially Theorem 2, are actually correct for loops over rational-valued variables.

Incompleteness. The algorithm is incomplete and may return "UNKNOWN". Although termination for G^1P^1 loops in which all the variables are real variables is decidable, the decidability of termination for G^1P^1 loops that have integer variables remains a conjecture [115]. Furthermore, our algorithm can prove termination for a large set of G^1P^1 loops whose iterations change the guard value in one of the patterns as informally illustrated in Figure 9.4. The horizontal axes represent passage of time and the vertical axes represent change of guard values. The left pattern corresponds to existence of linear ranking functions. The middle one corresponds to existence of slowdown regions. The right one corresponds to progressiveness of cycles.

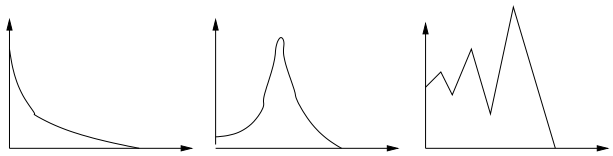

Figure 9.4: Patterns in which the guard value changes.

In the next two sections, we will generalize the idea of determining termination for G^1P^* and G^*P^1 loops.

9.5 Proving Termination for G^1P^*

All the ideas in the previous section can be used for G^1P^* loops without too much adaption except that some concepts need to be generalized with path conditions.

9.5.1 Constructing Region Graphs

Given a G^1P^* loop as below,

while $\bar{a}\bar{x} \geq b$ **do**
  $pc_1 \rightarrow \bar{x}' = U_1\bar{x} + \bar{u}_1$
  ...
  $pc_p \rightarrow \bar{x}' = U_p\bar{x} + \bar{u}_p$
od

the construction of region graphs is similar to the construction for G^1P^1 loops as follows:

- For each i-th path, we create a positive region, a negative region and a standstill region if their respective defining inequality system has solutions. Let the path condition be $pc_i = \bar{c}_1\bar{x} \geq d_1 \wedge ... \wedge \bar{c}_q\bar{x} \geq d_q$. The system of the linear inequalities (9.68–9.71) defines the positive region. The linear inequality systems to define the negative and the standstill region differ

only in the relational operator in Inequality (9.71) accordingly.

$$\bar{a}\bar{x} \geq b \tag{9.68}$$

$$\bigwedge_{j=1}^{q} \bar{c}_j\bar{x} \geq d_j \tag{9.69}$$

$$\bar{x}' = U_j\bar{x} + \bar{u}_j \tag{9.70}$$

$$\bar{a}\bar{x} > \bar{a}\bar{x}' \tag{9.71}$$

- Transitions are built in exactly the same way as for G^1P^1.

9.5.2 Using Path Conditions

Path conditions can be used to determine eventual exiting of standstill regions and negative regions with self-transitions.

Consider Example 9.5. If the first path is taken, the guard value x_1 remains unchanged. However, the path cannot be taken forever. This is because the value of x_2 is always decreased every time that the path is taken and is bounded by 0 as imposed by the path condition.

Example 9.5. This is a loop with two paths.

while $x_1 \geq 0$ **do**

$$x_2 \geq 0 \rightarrow \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$x_2 < 0 \rightarrow \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

od

To generalize the idea, we define *drag regions* as follows.

Definition 9.10 (Drag Regions)**.** A negative region or a standstill region is a *drag region* with respect to the respective path condition $pc = \bar{c}_1\bar{x} \geq d_1 \wedge ... \wedge \bar{c}_q\bar{x} \geq d_q$ if, for some $\bar{c}_j\bar{x}$ in pc, the value of $\bar{c}_j\bar{x}$ is always decreased within the region.

Drag regions can be checked by solving a linear inequality systems. The construction is similar to the linear inequality system for checking slowdown regions. We omit the full detail here.

Progressiveness of cycles can also be generalized when taking path conditions into consideration. For a region R with respect to a path condition $pc = \bar{c}_1\bar{x} \geq d_1 \wedge ... \wedge \bar{c}_q\bar{x} \geq d_q$, a cycle is progressive on R if, along the cycle, every time that R is entered, the value of some $\bar{c}_j\bar{x}$ in pc is smaller than the value of $\bar{c}_j\bar{x}$ at the last time that R is entered.

9.5.3 Determining Termination

The algorithm in Subsection 9.4.4 is modified for proving termination for G^1P^* loops as follows.

- Positive, negative, and standstill regions are created for all paths.
- When a standstill region has a self-transition, instead of returning "UNKNOWN", check whether it is a drag region. If not, return "UNKNOWN".

- For an unbounded negative region, check whether it is a drag region. If not, check whether it is a slowdown region. If not, return "UNKNOWN".
- Progressiveness is checked also with respect to path conditions.

Since the number of cycles is exponential in the number of loop paths, so is the number of linear inequality systems constructed by the modified algorithm. The size of each constructed inequality system is linear both in the number of loop paths and in the number of variables. The algorithm is sound and incomplete. In fact, termination of G^1P^* has been shown to be undecidable [115].

9.6 Proving Termination for G^*P^1

The basic idea to prove termination for a G^*P^1 loop is to check whether termination can be proved by the region graph constructed with respect to some guard in the loop condition. While analyzing the region graph with respect to a chosen guard, we also consider other guards in the loop condition as explained below.

Construction of region graphs. Choosing a guard in the loop condition, the construction of the region graph is similar to the construction for G^1P^1. The linear inequality system to define the positive region contains (1) all the inequalities in the loop condition, (2) variable update equations, and (3) the inequality that expresses the decrease of the chosen guard value. The inequality systems defining the negative region and the standstill region are constructed similarly.

Generalization of concepts. A negative or a standstill region is a drag region with respect to some guard that is not chosen for constructing the region graph if the value of the considered guard is decreased within the region. For a region R and some guard g that is not chosen for constructing the region graph, a cycle is progressive on R also if, along the cycle, every time that R is entered, the value of g is smaller than the value of g at the last time that R is entered.

Determining termination. The algorithm to determine termination for a G^*P^1 loops is as follows. Given a G^*P^1 loop, a guard in the loop condition is chosen nondeterministically. The algorithm in Subsection 9.4.4 is then used to construct and check the region graph with respect to the chosen guard, with a slight modification which allows for checking generalized drag regions and generalized progressiveness. If termination cannot be determined, then another guard is chosen. This procedure is repeated until termination is proved or all the guards have been checked.

Complexity, soundness and incompleteness. Let m be the number of guards in the loop condition and N be the parameter as the upper bound on the length of finite difference chains. In the worst case m region graphs are constructed and checked. For each region graph, the number of constructed linear inequality systems is no more than $14+2m+N$. The size of each inequality

system is linear in both m and the number of variables. The algorithm is sound and incomplete. In fact it remains a conjecture that termination of G^*P^1 loops that have integer variables is decidable [115].

Proving termination for G^*P^*. In this chapter we present incomplete approaches to proving termination for G^1P^* and G^*P^1. These two methods are orthogonal and can be easily combined to yield an approach to proving termination for the G^*P^* class.

9.7 Experimental Results

We implemented our method in a prototype tool named "PONES" (positive-negative-standstill). Finding a representative sample of realistic software systems that exhibit a large number of non-trivial loops that fall into our categorization is not easy, as it was also observed in [24]. Also, automated extraction of loop code and the resulting loop information has not yet been but will be implemented in the future. For the experiments described here, we manually collected program loops from the source code of Azureus[4] which is a peer-to-peer file sharing software written in Java. The software contains 3567 while- and for-loops. We analyzed the 1636 loops that fall into our categorization. There were only 3 loops in G^1P^* and 4 in G^*P^1. In fact, most of the loops were of the form "`while (i<j) i++`". The prevalent simplicity of the loops encountered corresponds to the desire of programmers to code loops that are easy to comprehend.

PONES failed to prove termination for 14 of the analyzed loops and proved termination within 65 milliseconds for each of all other loops on a Pentium IV 3.20GHz machine with 2GB memory. Manual inspection revealed that the 14 loops that PONES failed on are not terminating on arbitrary initial variable values but do terminate in the context of the Azureus software system which limits the range of the initial variable values.

We cannot give a direct comparison with other termination proof methods because other works use different extraction and abstraction techniques than our method to collect loops from programs. It should also be noted that our method can be considered as being complementary to linear ranking function based approaches.

Future work. We propose that our analysis method can be improved by incorporating value analysis [43] to generate linear inequalities over variables as loop invariants. These inequalities are then used to shrink some regions in the constructed region graph in order to exclude those points that will never be reached during loop iterations.

As further future work we propose to generalize the concept of program loops as explicitly constructed by the *while* or *for* constructs to control flow cycles resulting from mutual and recursive function calls. These control flow cycles are usually more complex but we expect that our analysis can handle them nonetheless.

[4] Available from sourceforge.net.

Part III

Abstraction Refinement

Chapter 10

Sources of Imprecision

The buffer boundedness and livelock freedom tests that we have presented are both incomplete. In case buffer boundedness or livelock freedom cannot be established for a given model, both tests will return an inconclusive verdict of "UNKNOWN". In this case both tests also return a counterexample consisting of a collection of control flow cycles in the CFSM system that over-approximates the original model. We can easily identify those control flow cycles in the original model which correspond to the cycles in the counterexample. A counterexample represents those executions of the model in which only the cycles in the counterexample are repeated infinitely often. Any other cycles are either executed a finite number of times, or not executed at all. However, not every counterexample returned by our tests may correspond to valid executions that the original model permits. This is due to the introduction of spurious behavior in the abstraction procedures that both tests make use of – recall that we abstract from arbitrary program code, message orders, the activation conditions of cycles, and cycle dependencies.

One solution to the imprecision of our tests is that, once a counterexample is obtained, we determine whether the counterexample is spurious or not. In case it is spurious, we also check the source of imprecision that causes the presence of this particular spurious counterexample. Using such information, we can refine the abstraction by removing the spurious behavior that the detected counterexample corresponds to. The above described abstraction refinement results in an iterative analysis that starts with a relatively coarse abstraction for efficiency and gradually increases the imprecision of the abstraction guided by the counterexample found in each iteration. Such an iterative analysis must be fully automated to be practically useful. It is therefore our objective to give users automated support to achieve the abstraction refinement procedure.

In this thesis we will mainly consider and develop methods for the abstraction refinement procedure in the analysis of Promela models. Our choice of modeling languages is motivated by the reason of convenience that a large number of Promela models are available in the public domain. Moreover, some of the features of the SPIN model checker greatly facilitate our analysis. However, the choice would not compromise the generality of the developed methods since Promela has most of the modeling features that an advanced modeling language should include. We conjecture that applying our analysis ideas to other modeling and programming languages based on communicating finite state machines, such

```
1  mtype = {msg1, msg2};
2
3  chan ch1 = [1] of {mtype};
4  chan ch2 = [1] of {mtype};
5
6  active proctype P1(){
7    do
8    :: ch2 ? msg2 ->
9       ch1 ! msg1;
10   od
11 }
12
13 active proctype P2(){
14   int x;
15   x = 0;
16   do
17   :: (x == 0) ->
18      ch2 ! msg2;
19      x = 1;
20   :: (x == 1) ->
21      ch1 ? msg1;
22      x = 0;
23   od
24 }
```

Listing 10.1: A simple Promela model.

as UML RT or SDL, could easily be accomplished.

In this chapter we will briefly re-consider the abstraction procedures used in the buffer boundedness and livelock freedom tests. We study the sources of imprecision introduced during the abstraction of a Promela model, and check the significance of each source in reducing the accuracy of future analyses. Based on the discussion in this chapter, we will present methods in the next two chapters for the determination of spurious counterexamples and the refinement of the abstract systems.

Structure. In Section 10.1 we illustrate through a simple Promela model how the imprecision of the buffer boundedness analysis is caused by the abstraction procedure. Then, we discuss the sources of imprecision introduced in each abstraction step in Section 10.2.

10.1 Abstractions and Spurious Counterexamples

We use the following example to illustrate how the imprecision of the buffer boundedness test can be caused by the abstraction steps used in the test. The imprecision of the livelock freedom test can be similarly caused.

Example 10.1. Consider the Promela model shown in Listing 10.1 to which we apply the buffer boundedness test proposed in Chapter 5. We identify four

message types as $m_1 = \{\langle \texttt{ch1}, \texttt{msg1}\rangle\}$, $m_2 = \{\langle \texttt{ch1}, \texttt{msg2}\rangle\}$, $m_3 = \{\langle \texttt{ch2}, \texttt{msg1}\rangle\}$, and $m_4 = \{\langle \texttt{ch2}, \texttt{msg2}\rangle\}$, using the optimal message type identification method explained in Section 7.2.3. As the first abstraction step, we abstract from arbitrary Promela code and retain only the control flow structure and the message passing behavior. In particular, we discard the boolean conditions at Line 17 and Line 20, and the assignments to the variable `x` at Line 15, Line 19, and Line 22. The resulting CFSM system is shown in Figure 10.1. We further abstract from message orders, and the message passing behavior of each transition in the CFSM system is denoted by effect vectors. We use the i-th ($i = 0, 1, 2, 3$) effect vector component to denote the message type m_{i+1}. The effect vector of each transition is indicated in the label of the transition in the figure. Next, we abstract from the activation conditions of the control flow cycles in the model and also cycle dependencies. The resulting independent cycle system consists of three cycles: c_1 as the only cycle in the CFSM process corresponding to the proctype `P1`, and c_2 and c_3 as the left cycle and the right cycle respectively in the CFSM process corresponding to the proctype `P2`. Furthermore, we have that $\mathit{eff}(c_1) = (1, 0, 0, -1)$, $\mathit{eff}(c_2) = (0, 0, 0, 1)$, and $\mathit{eff}(c_3) = (-1, 0, 0, 0)$. From this independent cycle system, the boundedness test will construct the following boundedness determination ILP problem:

$$x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} > \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{10.1}$$

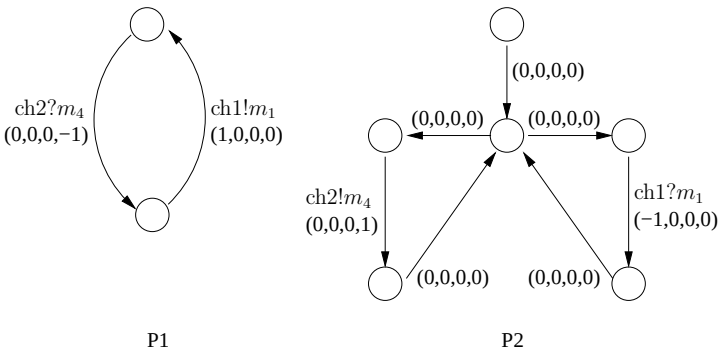

Figure 10.1: The abstract CFSM system of the Promela model in Listing 10.1.

The above ILP problem has apparently a solution as $x_1 = x_3 = 0$ and $x_2 = 1$. The boundedness test returns an outcome of "UNKNOWN". From the above obtained ILP solution, we can construct a counterexample that consists of only the cycle c_2. Recall that a counterexample represents the executions of the model in which only the cycles in the counterexample are repeated infinitely often. This means that c_2 is the only cycle being executed an infinite number of times in the unbounded executions that the constructed counterexample corresponds to. However, simply by manual inspection, we can see that c_2 alone cannot be executed infinitely often while c_3 is not. This is because c_2

carries a condition statement `x==0` (Line 17) in the original model. It imposes a condition on any execution of c_2 to have the runtime value of `x` as 0. However, one execution of c_2 changes the value of `x` to 1 (Line 19). Without the cycle c_3 to change the value of `x` back to 0, c_2 cannot be executed further. Therefore, the counterexample consisting of solely c_2 is spurious, whose presence is a consequence of abstracting away the condition statement at Line 17 and its aftermath of losing the dependency relation between c_2 and c_3.

The above example only shows one cause of the imprecision of the buffer boundedness test. We will discuss in the following sections all possible sources of imprecision and to what extent each cause affects the precision of our analyses.

10.2 Counterexamples and Spuriousness

The introduction of spurious counterexamples is a consequence of the conservative abstraction steps that we perform in the course of our buffer boundedness or livelock freedom test. We reconsider each of these abstraction steps to examine which information is removed from models during the step and how significantly it affects the precision of the analysis.

Step 1: Code Abstraction. In this step the program code in a model is abstracted away. The resulting CFSM system retains only the finite control structure and the message passing behavior of the model. We lose all the information about how the behavior of the model is constrained by the conditions on variables that are imposed by the program code. Losing such information is very significant because it often depends on the runtime value of a variable whether to send or receive a message, which message to send or receive, where messages are to be sent or from where messages are to be received. We will therefore consider this source of imprecision in more detail.

Step 2: Abstraction from Message Orders. In this step we neglect all information regarding the order of messages in message buffers. In particular, we assume that a message is always available to trigger a transition wherever it is in the buffer. This can be too coarse an overapproximation for a model that employs strict FIFO message buffers. However, models in practice usually have a message deferral/recall mechanism that stores an arriving message which cannot immediately be processed by the system into a special buffer so that it can be recalled when it is later needed, as specified in UML RT and implemented concretely in Rational Rose RealTime. This is consistent with the semantics of our abstraction. In other words, this abstraction step does not introduce imprecision in most practical situations and we will therefore not address it in the thesis.

Step 3: Abstraction from Activation Conditions. In this step the activation conditions of control flow cycles are abstracted away. We assume that there are always enough messages of the right type available for a cycle to be reachable from the initial configuration of the model. In this way we abstract from the dependency between the acyclic part and the cyclic part of an execution. This may result in imprecision as shown in the following example.

Example 10.2. Consider the CFSM system in Figure 10.2 which models a simple boolean value calculator. It consists of a `User` process and a `Calculator` process. The user can decide to calculate either the negation of a boolean value or the conjunction of two boolean values, by sending either a `not` message or an `and` message that will direct the calculator process into one of the two lower cycles for the proper type of calculations. A negation calculation needs only one operand while a conjunction calculation needs two operands. The user always provides the exact number of operands required by the calculation that she has chosen. The user can only change the type of calculation when the calculator let her to choose by sending a `restart` message. By manual checking we can easily see that the system is bounded. This is because no surplus operands are provided by the user – the user will never provide two operands for a negation calculation. This corresponds to a dependency of the acyclic part and the cyclic part of an execution, which is completely lost in this abstraction step. Therefore, the boundedness test will find a positive linear combination of cycle effects, i.e., one execution of the cycle in the `User` process to do conjunction calculations and one execution of the cycle in the `Calculator` process to do negation calculations. By this combination the boundedness test will return an "UNKNOWN" verdict.

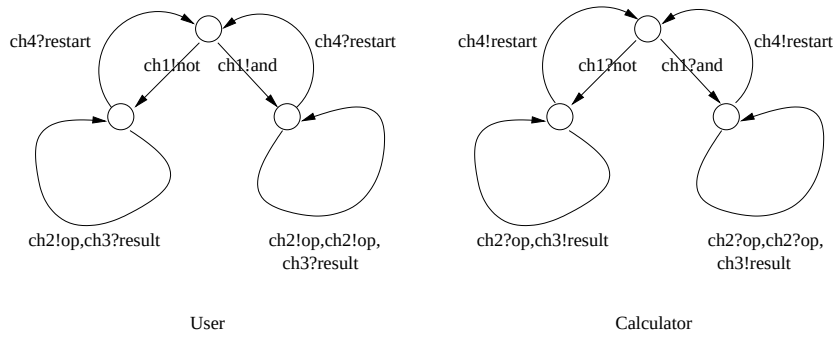

Figure 10.2: A CFSM system modeling a simple boolean value calculator.

The loss of the activation conditions of cycles is also significant in the estimation of buffer bounds, as we allow any combination of acyclic path effects and cyclic effects.

Step 4: Abstraction from Cycle Dependencies. In this step we abstract from dependencies between control flow cycles. There are different types of such dependencies: mutual exclusion or mutual inclusion, global dependencies or local dependencies. A dependency of mutual exclusion forbids a set of cycles to be jointly repeated infinitely often, while a dependency of mutual inclusion requires some cycles being repeated infinitely often to enable other cycles to be repeated infinitely often. A global dependency specifies a dependency among cycles in several concurrent processes, while a local dependency relates cycles in one same process. Cycle dependencies may be caused by many reasons, such as by the program code along control flow cycles, or by the structural characteristics of control flow graphs. Example 10.1 shows a dependency of mutual inclusion caused by condition statements on control flow cycles. One simple example of a graph structure imposed cycle dependency is when two

cycles in the control flow graph of one CFSM process reside in two different strongly connected components. Since at least one cycle is not reachable from the other, they cannot be jointly repeated infinitely often. Strongly connected components induce dependencies of mutual exclusion. There is another kind of graph structure imposed cycle dependencies that are less obvious, as shown in the following example.

Example 10.3. Consider the control flow graph of some CFSM process as shown in Figure 10.3. Assume that a counterexample contains the cycles c_1 and c_3 and does not contain the cycle c_2. Note that c_1 and c_3 do not share a common state, which implies that it is impossible to repeat c_1 and c_3 infinitely often without repeating c_2 infinitely often. Since c_2 is not included in the above counterexample it is spurious. The resulting dependency is a dependency of mutual inclusion.

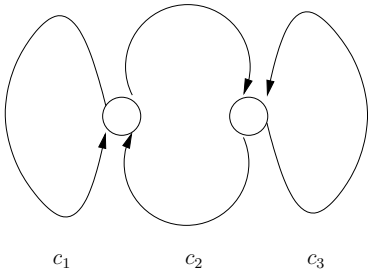

Figure 10.3: A CFSM process containing three control flow cycles.

Disregarding cycle dependencies means that arbitrary cyclic executions can be combined to form a potentially spurious counterexample. Therefore, this is also a significant source of imprecision.

Summary. We have argued above that disregarding program code, activation conditions of cycles, and cycle dependencies are significant causes of imprecision. In practice we observed that the spuriousness of counterexamples returned by the buffer boundedness and livelock freedom tests is mostly due to the loss of information of cycle dependencies. Moreover, program code plays a major role in imposing dependencies between cycles. Therefore, we will pay special attention to the discovery of cycle dependencies by reconsidering the program code in the original Promela model. We will address in future work the discovery of the dependencies between cycles and acyclic paths.

In the next chapter we will formalize the concept of cycle dependencies and present static analysis methods to discover cycle dependencies. In Chapter 12 we will use the cycle dependency information obtained to refine the abstraction.

Chapter 11

Discovering Cycle Dependencies

In the previous chapter we have shown that the loss of cycle dependencies during the abstraction procedure is a major cause of an imprecise buffer boundedness or livelock freedom analysis. As a consequence, cycle dependencies stand at the very core of the refinement procedure to be used in our verification framework, as proposed in Chapter 12.

We focus here on cycle dependencies of mutual inclusion, i.e., the infinitely repeated execution of some cycle relies on the infinite repeated executions of a certain set of cycles. The formal notion of cycle dependencies given in this chapter is only for mutual inclusion, and does not include mutual exclusion situations. The mutual exclusions of control flow cycles are mainly caused by strongly connected components and rarely observed in practice. This is because it is meaningful for any part of a reactive system to be always executable. Therefore, the control flow graph of every process in a system is usually strongly connected such that any cycle can be reached from any others when certain conditions are satisfied. However, we will still propose methods to handle mutual exclusions of cycles in Section 12.3.

Furthermore, we also inspect different causes of cycle dependencies and develop static analysis methods to discover dependencies with respect to each cause. The information of discovered cycle dependencies can be used to remove spurious behavior in the abstract system of the original model, as explained in Chapter 12.

Structure. Section 11.1 considers the executions of cycles in infinite executions of a Promela model. Section 11.2 defines the concept of cycle dependencies. We propose in Sections 11.3 and 11.4 several static analysis methods for cycle dependency discovery.

11.1 Cycle Executions

An infinite execution of a Promela model amounts to the repeated executions of cycles in some processes of the model. In this section we examine the executions of cycles in the context of infinite executions.

A control flow graph may contain an infinite number of cycles. However, the number of elementary cycles is always finite and in the worst case exponential in the number of transitions. Since any non-elementary cycle can be decomposed into elementary cycles, our analysis considers only elementary cycles, like what we have done in the boundedness and livelock freedom analysis. Unless otherwise specified, all the cycles mentioned in the following are elementary.

In order to study cycle executions, we need first to define some concepts related to control flow cycles as follows. If two control flow cycles share control points, then they are *neighbors* of each other. Any such shared control point is an *exit point* of the cycles that contain it, because one can exit one cycle at that control point and enter another cycle.

For an execution r of a Promela model, let $r/_p$ denote the projection of r on the set of transitions in a process p. Thus, $r/_p$ corresponds to the local execution of p in r. Any $r/_p$ can be decomposed into two parts: (1) an acyclic path from the initial control point, and (2) repeated executions of cycles. Given a cycle c in p, one execution of c in $r/_p$ may be interrupted by the executions of other cycles in p: Some part of c is executed until some exit point s is reached where it starts to execute other cycles. The execution of c is later resumed from s after the executions of those interrupting cycles are completed. If r is an infinite execution, then at least one cycle in the model is repeated infinitely often. We denote by $IRC(r)$ (*infinitely repeated cycles*) the set of cycles that are executed infinitely often in r. For a process p, $IRC(r/_p)$ is the subset of $IRC(r)$ consisting of only cycles in p. We have the following property for $IRC(r/_p)$.

Proposition 11.1. *Given a Promela model and an infinite execution r of the model, for any process p in the model, $IRC(r/_p)$ is either empty or forms a strongly connected subgraph of the control flow graph of p.*

Proof. By contradiction, we assume that, for some process p, the subgraph formed by $IRC(r/_p)$ is neither empty nor strongly connected. Then there exist two cycles c_1 and c_2 in $IRC(r/_p)$ such that c_1 is not reachable from c_2 in the subgraph. Then, we have two cases: (1) If c_1 is also not reachable from c_2 in the whole control flow graph of p, then after c_2 is executed c_1 can never be executed again. This contradicts the assumption that both cycles are repeated infinitely often. (2) Now suppose that c_1 is reachable from c_2 in the control flow graph of p. Then, it must traverse from c_2 to c_1 an infinite number of times. Along each such traversal, some transitions in p must be executed such that they are not contained in the subgraph formed by $IRC(r/_p)$. As there are only finitely many such transitions, at least one of them, say t, is repeated an infinite number of times. It implies that some cycle containing t has to be repeated infinitely often in r. This cycle is apparently not in $IRC(r/_p) \subseteq IRC(r)$, which contradicts the fact that $IRC(r)$ includes all the cycles that are repeated infinitely often in r. □

The above proposition justifies our previous argument in Section 10.2 that two cycles from one same process must be in a same strongly connected component to be both repeated infinitely often. The proposition also shows that not any arbitrary set of cycles can be an IRC. This is because the executions of cycles are *not* independent of each other. For instance, an infinite number of executions of one cycle may rely on an infinite number of executions of some

other cycles. We show in the following theorem that it is impossible to decide in general whether a set of cycles can be an *IRC*.

Proposition 11.2. *Given a Promela model and a set C of cycles in the model, it is undecidable in general whether there exists an infinite execution r of the model such that $C = IRC(r)$.*

Proof. We can prove that the problem described in the proposition is even undecidable for CFSM systems. Promela models with unbounded buffers can simulate CFSM systems. Therefore, the undecidability result also holds for Promela models. The proof is by a reduction from the undecidable message reception executability problem (Problem 1) of CFSM systems.

The reduction is as follows. Consider a CFSM system S that contains a local state s. There is an outgoing transition t from s, which is labeled with a message receiving event $b?m$. We construct a CFSM system S' such that, after the execution of $b?m$ on t, a certain set of cycles and only these cycles can be repeated infinitely often. More precisely, S' can be obtained from S by (1) introducing a new state s' in the same process as s is; (2) adding at s' a self-transition labeled with an empty string, i.e., it neither sends nor receives messages; and finally (3) changing the destination state of the transition t to the newly introduced state s'. Moreover, let C be the singleton cycle set consisting of the self-loop at s'.

We now prove that $b?m$ can be executed at s in S if and only if there exists an infinite execution r of S' such that $C = IRC(r)$.

For the "if" part, assume that there is an infinite execution r such that $C = IRC(r)$. From the construction of S', any such r has a finite prefix ρ leading to the execution of $b?m$ at s and an infinite suffix that executes the self-loop at s' forever. We can simulate ρ in S.

For the "only if" part, assume that $b?m$ can be executed in some execution r of s. We take the prefix ρ of r which ends after the first execution of $b?m$. After simulating ρ in S', we execute the self-loop at s' forever. In this way, we obtain an infinite execution r' such that $C = IRC(r')$. □

11.2 Cycle Dependencies

We now define the concept of cycle dependencies. We first need the concept of *correlated cycle sets*. We call a cycle set C a correlated cycle set for a cycle c if the infinite execution of c requires at least one of the cycles in C to be executed infinitely often.

Definition 11.1. Given a Promela model and a cycle c in the model, a set C of cycles is a *correlated cycle set* (CCS) of c if it satisfies the following conditions:

- $c \notin C$, and
- $\forall r.c \in IRC(r) \rightarrow (\exists c'.c' \in C \wedge c' \in IRC(r))$ where r is an execution of the model.

A cycle may have none, one or many CCSs. A cycle has no CCS if and only if it is the only one being repeated infinitely often in some execution of the model. For any cycle c, if C is a CCS of c, then any superset C' of C is

```
1  mtype = {request, reply}
2
3  chan toServer = [10] of {mtype}
4  chan fromServer = [10] of {mtype}
5
6  active proctype Client(){
7    int x = 0;
8
9    do
10   :: (x < 3) ->
11      toServer!request;
12      x++;
13   :: (x == 3) ->
14      fromServer?reply;
15      x--;
16   od
17 }
18
19 active proctype Server(){
20   do
21   :: toServer?request ->
22      fromServer!reply;
23   od
24 }
```

Listing 11.1: A simple client-server Promela model.

also a CCS of c when $c \notin C'$. A CCS of a cycle c is *minimal* (MCCS) if none of its proper subsets is also a CCS of c. A cycle may also have more than one MCCS. Intuitively, a MCCS of a cycle c gives a set of indispensable cycles that c depends on in all executions in which c is repeated infinitely often.

Definition 11.2. Given a cycle c and a MCCS C of c, we call the pair (c, C) a *cycle dependency*. In this case, we say that c *depends on* C.

In the above definition, if all the cycles in C are in the same process as c is, then (c, C) is a *local* dependency. Otherwise, (c, C) is a *global* dependency.

Example 11.1. Consider the Promela model in Figure 11.1. The model describes a simple client-server protocol. The `Client` process may only have at most 3 unprocessed requests at any time, and therefore must wait for a reply from the `Server` process when the counter `x` equals 3. The control flow graphs of the model are shown in Figure 11.1. We use c_l and c_r respectively to denote the left and the right cycle in the process `Client`, and c_s to denote the only cycle in the process `Server`. $\{c_l, c_s\}$ is a CCS of c_r. $\{c_l\}$ and $\{c_s\}$ are two MCCSs of c_r. Then, we obtain two cycle dependencies: $(c_r, \{c_l\})$ is a local dependency, and $(c_r, \{c_s\})$ is a global dependency. Intuitively, c_r depends on c_s because whenever c_r is executed it needs a `reply` message sent only by c_s. We will later discuss the causes of cycle dependencies in depth.

Definition 11.3. Given a cycle c and a CCS C of c, we call the pair (c, C) an *approximate cycle dependency* (ACD).

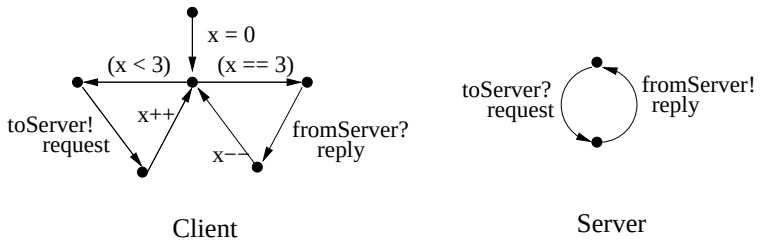

Figure 11.1: The control flow graphs of the Promela model in Listing 11.1.

For any cycle dependency (c, C) and a CCS C' of c such that $C \subset C'$, the ACD (c, C') is an over-approximation of (c, C).

If we interpret all buffers in a Promela model to have only finite capacities and all variables to have finite domains, then the Promela model possesses a finite global state space. In this case, it is decidable whether (c, C) is an ACD as follows. We construct the global state space for the model and then look for any (either non-elementary or elementary) cycle in the global state space which contains c but no cycles from C. If no such global cycles exist, then (c, C) is an ACD. However, we have assumed that buffers in Promela models have infinite capacities and variables may have infinite domains such as integer variables, since we want to generalize the Promela language to consider the more interesting class of infinite state systems. With this assumption, a Promela model may have an infinite global state space, for which we show in the following proposition that the above problem becomes undecidable.

Proposition 11.3. *Given a cycle c and a set C of cycles, it is undecidable in general whether C is a CCS of c.*

Proof. Similarly, we prove the proposition for CFSM systems, by a reduction from the undecidable Problem 1.

The reduction is as follows. Consider a CFSM system S that contains a local state s. There is an outgoing transition t from s, which is labeled with a message receiving event $b?m$. We construct a CFSM system S' from S by (1) introducing a new state s' in the same process as s is; (2) adding at s' a self-transition labeled with an empty string; (3) changing the target state of the transition t to the newly introduced state s'; and finally (4) adding a new process whose state machine consists of a single state s'' and a self-transition at s''. Moreover, let c be the self-loop at s' and C be the singleton cycle set consisting of the self-loop at s''.

We prove that $b?m$ can be executed at s in S if and only if C is *not* a CCS of c in S'.

For the "if" part, assume that C is not a CCS of c. Then, there exists an infinite execution of S' in which c is executed infinitely often while the self-loop at s'' is not. From the construction of S', c can be executed only if $b?m$ can be executed at s in S', which means that $b?m$ can be executed also in S.

For the "only if" part, assume that $b?m$ can be executed at s in S. Then, $b?m$ can be also executed in S'. After $b?m$ is executed, c can be repeated alone forever, which means c has no CCSs.

Since it is impossible to determine whether C is not a CCS of c, it is also undecidable whether C is a CCS of c. □

Although it is impossible in general to determine ACDs, we propose several static analysis methods to derive some types of ACDs according to how they are caused. The root cause for cycle dependencies lies in the executability of Promela statements. Given a cycle, if the executability of every statement along the cycle is unconditional, then the cycle can be repeated without interruption forever once the cycle is entered. Such a cycle has no CCSs and its repetition does not depend on any other cycles. On the contrary, consider a cycle c that contains a statement s whose executability is conditional. If s cannot be continuously enabled forever by only repeating c, then some other cycles need to be executed in order to re-enable s by, e.g., modifying the values of some variables, sending a message etc. In Promela there are two kinds of statements with conditional executability: condition statements and message receiving statements[1]. In the following we explain how cycle dependencies may be imposed by these two kinds of statements.

Condition statements. Consider the right cycle c_r in the proctype `Client` in Figure 11.1, which contains a condition statement `(x == 3)`. The condition `x` $= 3$ cannot remain true after c_r is executed because `x` is decremented by 1 in the cycle. Then, c_r can be repeated infinitely often only if the left cycle c_l is also repeated infinitely often to modify the value of `x` such that `x` can always acquire the value 3 again. In such a case, we say that c_r is *terminating* on the condition statement `(x == 3)`.

Definition 11.4 (Termination of Control Flow Cycles)**.** Given a Promela model and a control flow cycle c in the model, we assume that c contains a condition statement s. The cycle c is *terminating* on s if the boolean condition in s cannot remain true forever when c is repeated without interruption.

Note that a cycle may be terminating on several condition statements contained in the cycle. Since we focus here on discovering cycle dependencies, it is out of the scope of this chapter how to determine whether a cycle is terminating, which is a well-known undecidable problem. We will propose an incomplete procedure to prove termination for control flow cycles in Section 12.2. There are also many existing techniques [99, 35, 41, 24] to prove termination for certain kinds of loops in programs, which can be adapted to prove termination for control flow cycles. In Section 11.3 we will show how to determine cycle dependencies from a condition statement on which a cycle is terminating.

Message receiving statements. The above mentioned cycle c_r contains a message receiving statement `fromServer?reply`. Thus, the cycle c_s sending `reply` messages has to be repeated infinitely often when c_r is to be repeated infinitely often. In Section 11.4 we will present a method to determine cycle dependencies from message receiving statements, which are usually global dependencies.

[1]Recall that, by the assumption that buffers have infinite capacities, message sending statements are always executable.

```
1 if
2 :: y > 0 -> x = 5;
3 :: y <= 0 -> x = 4;
4 fi
```

Listing 11.2: A piece of Promela code.

11.3 Discovering Dependencies from Condition Statements

We show some types of cycle dependencies imposed by condition statements on which a cycle is terminating. In order to derive them, we need to discriminate between different ways in which the variables in a condition statement are modified in the cycle. A variable is *local* if its value can be referenced and modified only by one process. Otherwise, it is a *global* variable. However, the runtime value of a local variable may still depend on the executions of other processes. For instance, given a local variable x, if there is an assignment `x =` $e(y)$ where e is an arithmetic expression containing a global variable y, then the runtime value of x may depend on how y is modified in other processes.

Definition 11.5 (Globally and Locally Modified Variables)**.** For a cycle c and a variable x, x is *globally modified in the cycle* c if one of the following is satisfied:

- x is global, or
- there is a message receiving statement `b?msg(`x_1`,...,`x_n`)` in c where x is some x_i, or
- there is an assignment `x =` $e(y)$ in c where y is globally modified in c.

If a variable is not globally modified in a cycle c, then we say that it is *locally modified* in c.

Note that in the above definition we disregard the dependency of the runtime value of a local variable on a condition statement, such as the Promela code in Listing 11.2 shows, where `y` is a global variable. The reason is that, even though each branching statement results in several branches in the code, each control flow cycle may contain at most one branch of this condition statement. For instance, a control flow cycle may either contain the branch with the condition `y` > 0 in the example code, or the branch with the condition `y` ≤ 0, but not both. Therefore, inside a particular cycle, which branch is taken is fixed and the impact of the runtime value of the respective boolean condition is also fixed. Note that we are not interested in how a variable is modified or influenced in the whole model. We are only interested in how a variable is modified inside a particular cycle when the cycle is repeated without interruption.

For a boolean condition B in a cycle c, we denote by $var(B)$ the set of variables occurring in B. If all the variables in $var(B)$ are locally modified in c, then B is a *locally determined* condition. Otherwise, it is *globally determined*. In the subsections 11.3.1 and 11.3.2, we show how to determine cycle dependencies from these two kinds of conditions.

11.3.1 Locally Determined Conditions

We first prove that, if a cycle is terminating on a locally determined condition, then it depends on some of the cycles in the same process for an infinite number of executions.

Proposition 11.4. *Given a cycle c in a process p such that c is terminating on a locally determined condition B, if c is repeated infinitely often in an execution r, then one of the cycles c' in p where $c \neq c'$ is also repeated infinitely often in r.*

Proof. By contradiction, we assume that there is an execution in which c is repeated infinitely often and no other cycles in p are repeated infinitely often. Then, after some point of time, c is repeated without interruption forever. Because c is terminating on B, cycles in some concurrently running processes other than p must be executed such that they can influence the value of B to remain it true. However, this contradicts the fact that B is locally determined in c. □

Corollary 11.5. *Given a cycle c in a process p such that c is terminating on a locally determined condition B, if c is repeated infinitely often in an execution r, then one of the neighbors of c is also repeated infinitely often in r.*

Proof. From Proposition 11.4, if c is repeated an infinite number of times, then some other cycle c' in p must be repeated also infinitely often. On every path from a control point in c to a control point in c', there must be a transition t from an exit point of c such that t is not contained in c. There are only finitely many such transitions, so one of them is taken an infinite number of times and it belongs to at least one of the neighbors of c. □

Let C_p denote the set of cycles in p, and N_c denote the set of the neighbors of c. Proposition 11.4 and Corollary 11.5 give two ACDs, namely

$$(c, C_p - \{c\})$$

and

$$(c, N_c).$$

In the following, we propose several methods to refine the ACD $(c, C_p - \{c\})$.

Refinement 1. The ACD $(c, C_p - \{c\})$ is usually coarse because not all the cycles in p will necessarily contribute to the re-satisfaction of B. However, it is in general impossible to determine which cycles make a contribution. We define $E_c(B)$ as the set of variables occurring in c such that at least one of the variables in $E_c(B)$ must be modified in order to make B true again. Then, we can reduce $(c, C_p - \{c\})$ by removing all the cycles that do not modify any variable in $E_c(B)$. $E_c(B)$ subsumes but not necessarily equals $var(B)$. Consider the Promela code in Listing 11.3 and its control flow graph in Figure 11.2. An infinite number of repetitions of the left cycle relies on an infinite number of repetitions of the right one that resets the value of `y`. However, the enabling condition of the left cycle contains only the variable `x` which is not modified by the right cycle.

We propose the following method to compute $E_c(B)$ for a condition B on which a cycle c is terminating. A variable v is in $E_c(B)$ if one of the following conditions is satisfied

```
1  active proctype P(){
2    int x = 5;
3    int y = 5;
4
5    do
6    :: (x > 0) ->
7       y--;
8       x = y;
9    :: y = 5;
10   od
11 }
```

Listing 11.3: A piece of Promela code.

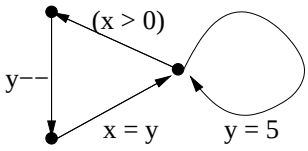

Figure 11.2: The control flow graph of the proctype in Listing 11.3.

- $v \in var(B)$, or
- there is an assignment v' = $e(v)$ in c such that $v' \in E_c(B)$.

Intuitively, the variables in $E_c(B)$ are the ones that may influence the runtime value of a variable in $var(B)$ in the cycle. For a set V of variables, we denote by $MC_p(V)$ the set of cycles in p which modify at least one variable in V. The following proposition gives a finer ACD

$$(c, MC_p(E_c(B)) - \{c\}).$$

Proposition 11.6. *Given a cycle c in a process p such that c is terminating on a locally determined condition B, if c is repeated infinitely often in an execution r, then one of the cycles in $(MC_p(E_c(B)) - \{c\})$ is also repeated infinitely often in r.*

Proof. By contradiction, we assume that there is an execution in which c is repeated infinitely often and no cycle in $(MC_p(E_c(B)) - \{c\})$ is repeated infinitely often. Then, after some point of time, no cycle except c is executed to modify any variable in $E_c(B)$. Because c is terminating, the control constantly leaves c, executes some other cycles, and comes back to continue the repetition of c. At each time that c is re-entered, each variable in $E_c(B)$ retains the same value that it had when c was exited last time. Thus, all the variables in $var(B)$ are modified in the same way as if c were repeated without interruption. So, eventually, B will become false, and the control is either blocked or exits c at the source control point of the transition corresponding to the condition statement `(`B`)`. If c is exited, then it will never be executed again. This is because no cycle will be executed to modify the variables in $var(B)$ and B will remain false forever. This contradicts the assumption that c is repeated infinitely often. □

```
1  active proctype P(){
2    int x;
3
4    do
5    :: x = 5;
6
7       do
8       :: (x > 0) ->
9          x--;
10      :: break;
11      od;
12
13      x = 5;
14
15      do
16      :: (x > 0) ->
17         x--;
18      :: break;
19      od;
20   od
21 }
```

Listing 11.4: A piece of Promela code.

Refinement 2. The above refined ACD may still be coarse. This is because not necessarily all the cycles that modify some variables in $E_c(B)$ have an effect that renders B true. This can be illustrated by the Promela code in Listing 11.4, whose control flow graph is shown in Figure 11.3 where we name the three cycles in the control flow graph c_1, c_2, and c_3. From Proposition 11.6, we can get an ACD $(c_1, \{c_2, c_3\})$. However, it is easy to see that c_1 only depends on c_2 for an infinite number of repetitions. c_3 needs not to be executed at all even though it also modifies the variable `x`, for it has no effect to make the condition `x` > 0 true. Consequently, c_3 can be safely removed from the previously computed ACD. However, it is impossible in general and very hard in practice to determine which cycle that modifies some variables in $E_c(B)$ has no effect to make B true.

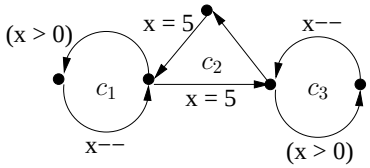

Figure 11.3: The control flow graph of the Promela code in Listing 11.4.

Figure 11.3 shows yet another reason why $(c, MC_p(E_c(B)) - \{c\})$ may be coarse. Note that, whenever leaving c_1 to execute c_3, c_2 is always executed. So, in any execution in which c_1 is repeated infinitely often, no matter whether c_3 is repeated infinitely often or not, c_2 is always repeated infinitely often. Based solely on this observation we can safely remove c_3 from the ACD $(c_1, \{c_2, c_3\})$

```
1  proc compute_acd(cycle c_0, condition B_0)
2    set[cycle] visited = {}
3    set[cycle] ccs = {}
4    queue[cycle] open = {}
5    search_for_preemptive_cycles(c_0, B_0)
6    return (c_0, ccs)
7
8  proc search_for_preemptive_cycles(cycle c, condition B)
9    add c to visited
10
11   for each nc in neighbors(c)
12     if (nc not in visited) and (nc not in open)
13     then
14       if (nc modifies some variables in E_c(B))
15       then
16         add nc to visited
17         add nc to ccs
18       else
19         enqueue(open, nc)
20
21   if (open not empty)
22   then
23     c' = dequeue(open)
24     search_for_preemptive_cycles(c', B)
```

Listing 11.5: An algorithm to determine cycle dependencies from locally determined conditions.

to obtain a finer ACD.

Definition 11.6 (Preemptive and Preempted Cycles)**.** Given a cycle c in a process p such that c is terminating on a locally determined condition B, and a cycle $c' \in MC_p(E_c(B))$ such that c and c' are reachable from each other, c' is *preemptive* with respect to c and B if there exist one exit point s in c and one exit point s' in c' such that

- there is an acyclic path from s to s' that does not modify any variables in $E_c(B)$, and
- there is an acyclic path from s' to s that does not modify any variables in $E_c(B)$.

Otherwise, c' is *preempted*.

In the previous example, c_2 is preemptive and c_3 is preempted. It is easy to prove that, on the way from any cycle c to execute one of its preempted cycles and then back to c, at least one preemptive cycle must be executed. $(c, MC_p(E_c(B)) - \{c\})$ can be refined by removing all the preempted cycles from the CCS $(MC_p(E_c(B)) - \{c\})$.

Whereas Definition 11.6 can be used to determine whether a cycle is preempted, Listing 11.5 gives an efficient algorithm to collect preemptive cycles during the computation of ACDs. In a Breadth First Search manner, the algorithm visits each cycle at most once, and thus is linear in the number of cycles. We show in the following that this algorithm is both terminating and sound.

Proposition 11.7 (Termination)**.** *The algorithm in Listing 11.5 always terminates.*

Proof. It is easy to see that no cycle can be added to `visited` more than once. Hence, each call to `search_for_preemptive_cycles` results in a new cycle being added to `visited` (Line 9). Note that our algorithm never removes any cycle from `visited`. Since there are only finitely many cycles, the algorithm must terminate. □

Proposition 11.8 (Soundness)**.** *Given as an input a cycle c in process p such that c is terminating on a locally determined condition B, the algorithm in Listing 11.5 returns an ACD (c, C) such that a cycle $c' \in MC_p(E_c(B))$ is preemptive if and only if $c' \in C$.*

Proof. We assign a natural number $level(c)$ to each cycle c that is added to `visited` as follow: (1) $level(c) = 0$; (2) if c_1 is enqueued (Line 19) or added to `ccs` (Line 17) inside the call to `search_for_preemptive_cycles(`c_2`, B)` and $level(c_2) = n$, then $level(c_1) = n + 1$. In the second case, we say that c_2 is the *parent* of c_1 and c_1 is a *child* of c_2. Then, we can build a *parent-child tree* (PCT).

For the "if" part, we prove that if $c' \in C$ then it is preemptive. It is easy to see that, in the path from the root c to c' in the PCT, no cycle except c and c' modifies any variable in $E_c(B)$. From this path, we can easily construct an acyclic path θ from an exit point t in c to an exit point t' in c', and an acyclic path θ' from t' to t. θ and θ' apparently do not modify any variable in $E_c(B)$.

For the "only if" part, assume that c' is preemptive. Then, there is an exit point t in c, an exit point t' in c', an acyclic path θ from t to t', and an acyclic path θ' from t' to t such that θ and θ' do not modify any variable in $E_c(B)$. The path $<\theta, \theta'>$ can be decomposed into a set of cycles, from which we can construct a sequence of pairwise distinct cycles $c_1, \ldots, c_n$ such that (1) c_1 is a neighbor of c, (2) c_n is a neighbor of c', and (3) each c_i and c_{i+1} are neighbors. It is easy to see that no cycle in such a sequence modifies any variable in $E_c(B)$. Let *SEQ* be the set of shortest sequences of cycles as constructed in this way. Assume that the sequences in *SEQ* are of length n. For each sequence in *SEQ*, we add c to its head and attach c' to the end. We prove that there is one sequence $seq \in SEQ$ that is a path in the PCT, which implies that c' is added to *ccs*. The proof is by showing that, for any $k \leq n$, there is a sequence $seq \in SEQ$ such that its prefix of length i is a path in the PCT (*), by an induction on the length i of prefixes of sequences in *SEQ*.

Induction base: The prefix of length 1 of any sequence in `SEQ` is c, which is a path in the PCT.

Induction step: Assume that (*) holds for k. Let P be the set of sequences in `SEQ` such that their prefixes of length k are paths in the PCT. Let C_k be the set of cycles $\{d \mid d$ is the k-th element in a sequence in $P\}$, and C_{k+1} be $\{d \mid d$ is the $(k+1)$-th element in a sequence in $P\}$. By contradiction, we assume that there is no sequence in P such that its prefix of length $(k+1)$ is a path in the PCT. Then, inside the call to `search_for_preemptive_cycles(`c_k`, B)` for each $c_k \in C_k$, none of the neighbors of c_{k+1} in C_{k+1} is enqueued or added to `visited`. This happens only when c_{k+1} is already in `open` or in `visited`. Let p be the parent of c_{k+1}. So, $p \notin P$. We have either that (1) $level(p) = k$, or that (2) $level(p) < k$. When $level(p) = k$, the path from c to p must be the

```
1  int y = 5;
2
3  active proctype P(){
4    int x = 5;
5    do
6    :: (x > 0) ->
7       y--;
8       x = y;
9    od
10 }
11
12 active proctype Q(){
13   do
14   :: y = 5;
15   od
16 }
```

Listing 11.6: A piece of Promela code.

prefix of length k of some sequence in SEQ, which means that $p \in P$. This leads to a contradiction. When $level(p) < k$, we construct a sequence of cycles from any sequence in P whose $(k+1)$-th element is c_{k+1}, by replacing the prefix of length k by p. The new sequence is shorter than the sequences in `SEQ`, which contradicts that `SEQ` contains the shortest sequences of pairwise distinct cycles connecting c and c'. □

11.3.2 Globally Determined Conditions

If a cycle is terminating on a globally determined condition, then it may not only depend on cycles in the same process, because cycles in other concurrent processes can possibly influence the runtime value of the condition. This can be illustrated in the example in Listing 11.6 whose control flow graphs are shown in Figure 11.4. The cycle in the proctype `P` is actually the only cycle in `P`, and it depends on the cycle in `Q`.

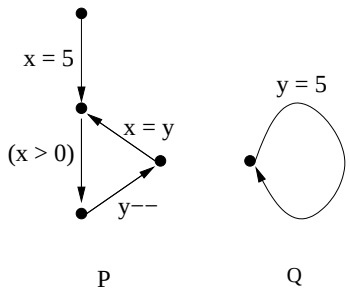

Figure 11.4: The control flow graphs of the Promela code in Listing 11.6.

We will not consider any globally determined condition whose value is influenced by a message receiving statement, which will be discussed in Section 11.4. We modify Proposition 11.6 to accommodate globally determined conditions by

considering the influence of cycles from all processes. For a Promela model M, we denote by $proc(M)$ the set of processes in M.

Proposition 11.9. *Given a Promela model M and a cycle c in a process in M such that c is terminating on a globally determined condition B, if c is repeated infinitely often, then one of the cycles in $(\bigcup_{q \in proc(M)} MC_q(E_c(B)) - \{c\})$ is also repeated infinitely often.*

The above proposition gives an ACD

$$(c, \bigcup_{q \in proc(M)} MC_q(E_c(B)) - \{c\}).$$

We may refine this ACD by using the algorithm in Listing 11.5 to rule out all the preempted cycles in $MC_p(E_c(B))$ if c is in the process p.

11.4 Discovering Dependencies from Message Receiving Statements

When a cycle c contains a message receiving statement `b?msg(`$x_1, \ldots, x_n$`)`, it needs an infinite number of `msg` messages to be repeated infinitely often. Consequently, c depends on some cycles that send such messages. Let $SC_{b,msg}$ be the set of the cycles sending messages `msg` to b. If $c \notin SC_{b,msg}$, then $(c, SC_{b,msg})$ is an ACD. In the remainder of the section, we assume that a cycle never receives messages sent by itself, which is the normal case for real life Promela models.

A cycle that receives messages may contain a condition statement in which the condition contains some variables used to store components of received messages. Usually, the cycle can be executed only if the received message contains such components that make the condition true. Consider a cycle that contains a message receiving statement s_1 and a condition statement s_2 such that the condition in s_2 contains variables used in s_1. The following pattern for s_1 and s_2 are observed in most real life Promela models: (1) all the variables in s_1 are local; (2) the condition in s_2 contains only variables used in s_1; (3) no variable in the condition is modified between s_1 and s_2 in the cycle. We call such a condition a *message determined* condition. Listing 11.7 shows two proctypes `GIOPClient` and `GIOPAgent`[2]. In the control flow graph of `GIOPClient` as shown in Figure 11.5, there is a cycle depicted using only solid lines that contains a message determined condition `reply_status` = 4. We show in the remainder of the section how to derive ACDs from such a message determined condition.

In Figure 11.5, let c_1 denote the solid-lined cycle in the proctype `GIOPClient`, and c_2 and c_3 denote the cycles that respectively assign 4 and 5 to `rs` in the proctype `GIOPAgent`. We have an ACD $(c_1, SC_{\texttt{toClientL,Reply}})$ and both c_2 and c_3 are in $SC_{\texttt{toClientL,Reply}}$. However, this ACD is coarse because not necessarily every cycle in $SC_{\texttt{toClientL,Reply}}$ may send a `Reply` message to make `reply_status` = 4 true in c_1. As an example, c_3 assigns 5 to `rs` whose value is passed to `reply_status` in c_1 through message passing. Thus, c_3 cannot make `reply_status` = 4 true, and it can be safely removed from $SC_{\texttt{toClientL,Reply}}$ to obtain a finer ACD. Now the question is how to determine which cycle *cannot* send messages to make `reply_status` = 4 true.

[2]This example is taken from a Promela model of the GIOP protocol published in [74].

```
1  active proctype GIOPClient(){
2    ...
3
4    do
5    :: toClientL?Reply(..., reply_status, ...) ->
6         ... // replay_status is not modified here
7
8         if
9         :: (usedReqId[reqId] == 1) ->
10
11           if
12           :: (reply_status == 4) -> ...
13           ...
14           fi;
15
16        ...
17        fi;
18   ...
19   od
20 }
21
22 active proctype GIOPAgent()7
23   ...
24
25   do
26   :: toAgent?Request(...) ->
27      ...
28
29      if
30      :: (registered[objKey] == 255) ->
31         ...
32         rs = 5;
33      :: else ->
34         ...
35         rs = 4;
36      fi;
37
38      toClientL!Reply(..., rs, ...);
39
40   ...
41   od
42 }
```

Listing 11.7: A part of the GIOP model.

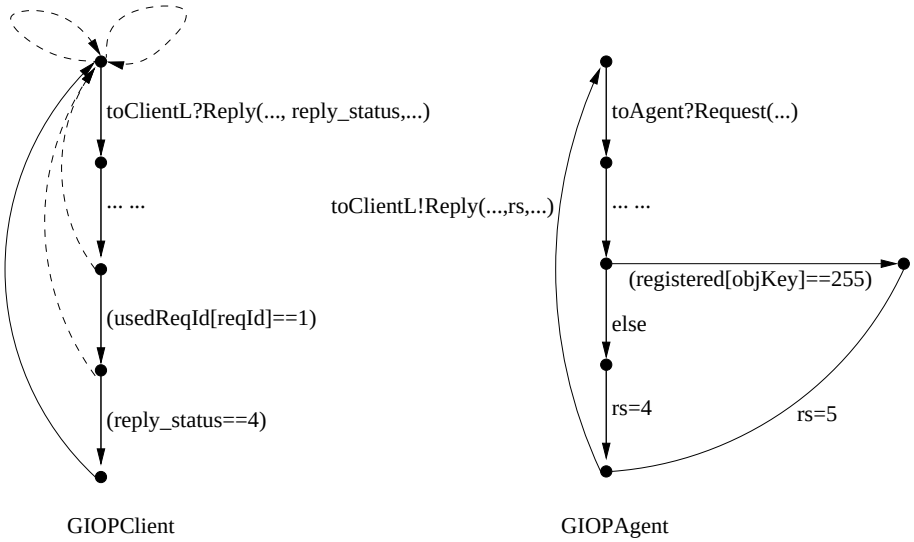

Figure 11.5: The control flow graphs of the Promela code in Listing 11.7.

First, we need to determine which kind of `Reply` messages must be received by c_1 to make `reply_status` $= 4$ true. More precisely, we need to know which condition must be satisfied by the components of such a message. According to the definition of message determined conditions, the variable `reply_status` is not modified in c_1 between the message receiving statement and the condition statement `(reply_status == 4)`. However, after a message is received, `reply_status` may still be modified before the statement `(reply_status == 4)` is reached. This is because the execution of c_1 can be interrupted, e.g., at the source control point of the transition corresponding to the statement `(usedReqId[reqId] == 1)`. Then, when the execution of c_1 is resumed, the variable `reply_status` may be already modified by other cycles. However, in this concrete example, if c_1 is interrupted, then before c_1 is resumed the last completed interrupting cycle always receives a `Reply` message. Moreover, this message contains a component whose value is passed to `reply_status`. The value of `reply_status` is afterward unchanged before reaching the message determined condition. This is because c_1 and its neighbors satisfy the following structural property named *fastened cycles*.

Definition 11.7 (Fastened Cycles)**.** Given a Promela model and one of the processes in the model, assume that there is a control flow cycle c in the considered process which contains a message receiving statement s_1 and a condition statement s_2. We denote by t_1 the transition corresponding to s_1, by t_2 the transition corresponding to s_2, and by p the path from the source control point of t_1 to the source control point of t_2. For each neighbor c' of c, if c' and c contain a common control point s within p, then c' contains also the path in c from the source control point of t_1 to s. In this case, we call c and its neighours as a set of *fastened cycles*.

The pattern in the fastened cycles property results from nested `if` statements inside `do` loops which are a common control structure of concurrent processes

in an asynchronous reactive system.

Proposition 11.10. *Let c be a cycle that contains a condition statement* `(`B`)` *in which the condition B is determined by messages received via the statement* `b?msg`$(x_1, \ldots, x_n)$ *in c. If the fastened cycles property is satisfied by c and all of its neighbors, then one execution of c needs a* `msg`$(d_1, \ldots, d_n)$ *message such that $B[x_i \mapsto d_i]$*[3] *is true.*

The proof of the above proposition needs the following auxiliary lemma.

Lemma 11.11. *Using the notation of the fastened cycles property in Definition 11.7, the following is satisfied: For any path p_1 that ends at an exit point s within p, the path p_2 in c from the source control point of t_1 to s is consecutively executed*[4] *in the end of p_1.*

Proof. We suppose that there are q exit points in p: $es_1, \ldots, es_q$. We prove the lemma by an induction on the index k of es_k.

Induction base: es_1 is the source control point of t_1. The path from es_1 to es_1 is an empty path which is always consecutively executed.

Induction step: Assume the lemma holds for es_m where $m < k$. Let p' denote the path from the source control point of t_1 to es_k. Suppose that es_j is the last exit point at which the execution of p' is interrupted. We have that $j < k$. From the induction assumption, immediately before the execution p' is resumed at es_j, the path from the source control point of t_1 to es_j is consecutively executed. Furthermore, after the execution of p' is resumed, the remaining part of p' is also consecutively executed. So, p' is consecutively executed. □

Using the above lemma, we now prove Proposition 11.10 as follows.

Proof. We denote by s_1 the statement `b?msg`$(x_1, \ldots, x_n)$, by s_2 the statement `(`B`)`, by t_1 the transition corresponding to s_1, by t_2 the transition corresponding to s_2, and by p the path from the source control point of t_1 to the source control point of t_2.

We denote by s_l the last exit point within p at which the execution of c is interrupted in an execution. We denote by p' the path from the source control point of t_1 to s_l in c, and by p'' the path from s_l to the source control point of t_2 in c . So, $p = < p', p'' >$. Following Lemma 11.11, before the execution of c is resumed, p' is consecutively executed. Because s_l is the last control point at which c is exited, p'' is also consecutively executed after c is re-entered. So, p is consecutively executed before the condition statement s_2 is reached. In this consecutive execution of p a message `msg`$(d_1, \ldots, d_n)$ is received and each variable $x_i \in var(B)$ is assigned with d_i. After p is executed, the execution of c can continue if and only if B is true. Since any variable $x_i \in var(B)$ is not modified in p after receiving the message, we have that $B[x_i \mapsto d_i]$ is true. □

Using Proposition 11.10, if we can determine that the execution of c requires a message `msg(`$d_1, \ldots, d_n$`)` such that $B[x_i \mapsto d_i]$ is true, then we can use the

[3] $B[x_i \mapsto d_i]$ is a boolean expression obtained from B by substituting simultaneously each occurrence of x_i with d_i.

[4] Given two paths p_1 and p_2, we say that p_2 is executed in p_1 if p_2 is a subsequence of p_1. If p_2 is a consecutive subsequence of p_1, then we say that it is consecutively executed in p_1. In particular, an empty path is always consecutively executed.

following method to determine whether a cycle c' may *not* send such a message. Given a cycle c' that contains a message sending statement `b!msg`$(d_1, \ldots, d_n)$, if all d_i's are constant values, then we directly evaluate $B[x_i \mapsto d_i]$ which is a constant truth value. If it is false, then we can exclude c' from $SC_{b,msg}$. When some d_i is a variable, we traverse backward in c' from the source control point s' of the transition t corresponding to `b!msg`$(d_1, \ldots, d_n)$, and locate the first control point $s \neq s'$ such that s has an incoming transition outside c' but within other cycles. If no such s exists, then we take as s the predecessor of s' in c'. The path p from s to s' is then the longest acyclic path within c that must be consecutively executed immediately before reaching the message sending statement. We compute the postcondition *Post(p)* of p by Floyd-Hoare-style forward inference starting with the precondition `true`[5]. This assumes that all the variables initially contain arbitrary values before p is consecutively executed. If $Post(p) \wedge B[x_i \mapsto d_i]$ is unsatisfiable, then c' can be removed from $SC_{b,msg}$. If the Promela model contains only linear arithmetic expressions in assignments and conditions, then the satisfiability of $Post(p) \wedge B[x_i \mapsto d_i]$ can be decided fully automatically using either an automated theorem prover or a linear programming solver. In the example in Figure 11.5, we illustrate how to determine that c_3 cannot send a message to satisfy `reply_status = 4`. The longest consecutively executed path p in this example starts from the source control point of the transition corresponding to the message receiving statement, i.e., the topmost control point in the control flow graph of `GIOPAgent`. Then $Post(p) = (\cdots \wedge (\texttt{rs} = 5))$. Since $Post(p) \wedge (\texttt{reply\_status} = 4)[\texttt{reply\_status} \mapsto \texttt{rs}]$ is false, c_3 can be safely removed from $(c_1, SC_{\texttt{toClientL,Reply}})$.

Summary. In this chapter we have defined the concept of cycle dependencies and proposed several static analysis methods to discover cycle dependencies. In the next chapter, we will use the information of cycle dependencies obtained using the methods in this chapter to refine abstractions in the buffer boundedness and livelock freedom tests.

[5]Since the path p is acyclic, *Post(p)* can be computed fully automatically.

Chapter 12

Abstraction Refinement

The core ideas underlying the abstraction refinement in our verification framework are (1) the discovery of cycle dependencies and (2) the use of the dependency information to remove spurious behavior in the abstract system of the original model. In the previous chapter we have addressed the first problem of obtaining the knowledge of cycle dependencies. We will look at now the second problem – how to use the obtained knowledge in the refinement procedure.

The basic idea of our abstraction refinement procedure is as follows. Given a counterexample, we determine whether the counterexample violates any cycle dependency that we have discovered, i.e., whether there is any cycle in the counterexample such that all the cycles that it depends on are not in the same counterexample. If this is the case, then the counterexample is certainly spurious. The discovered cycle dependencies will also be encoded into a set or several sets of linear inequalities. Each such set of inequalities is used to refine the original property checking ILP problem.

The above described refinement procedure is guided by counterexamples and results in an iterative analysis. In each iteration of the refinement procedure, we do not intend to remove all or arbitrary spurious behavior from the abstract system. Instead, we remove only those spurious behavior that the considered counterexample corresponds to. We argue that such counterexample-guided strategy is both efficient and effective.

First, our refinement procedure is efficient. Note that the static analysis methods used for abstraction refinements are much more expensive than the ILP-based property checking methods explained in Chapter 5 and Chapter 6. Our refinement procedure allows to use the more costly methods only when it is necessary to use them, i.e., when an imprecise verification result is returned.

Our refinement procedure is also effective. Note that not necessarily all spurious behavior introduced in the abstraction steps would violate the checked property. The spurious behavior that satisfies the property does not cause the analysis to be imprecise, and therefore needs not to be removed. On the contrary, a spurious counterexample always corresponds to some property violating spurious behavior that must be removed in order to improve the accuracy of the analysis.

Structure. Section 12.1 explains how to use a discovered cycle dependency to refine the abstract system for livelock freedom analyses. We will also show that

the concept of cycle dependencies proposed in the previous chapter is however insufficient to refine abstractions for buffer boundedness analyses. The boundedness test requires a finer concept of cycle dependencies, so-called numerical cycle dependencies, with the information of cycle iteration times, which will be formally defined in Section 12.2. We also present in this section an incomplete method to discover numerical cycle dependencies. Besides the refinement based on cycle dependencies, we also employ structural criteria on the control flow cycle graphs to determine counterexample spuriousness and derive abstraction refinements, which we discuss in Section 12.3.

12.1 Refinement by Cycle Dependencies

In this section we first outline our abstraction refinement procedure based on counterexample analysis, and then show in detail how to use cycle dependency information to refine an abstract system.

Given a Promela model, the buffer boundedness or livelock freedom test may return "UNKNOWN" and a counterexample consisting of a set of control flow cycles. We select randomly a cycle c from the counterexample. Let c' be the control flow cycle in the original model which corresponds to c. We use the methods explained in the previous chapter to determine cycle dependencies for c'. If there exists any discovered ACD (c', D') such that no cycle in D' has a corresponding cycle in the counterexample, then the counterexample is spurious. This is because, for c' to be repeated infinitely often, one of the cycles in D' must also be repeated infinitely often according to the definition of cycle dependencies. Otherwise, if the counterexample respects all discovered cycle dependencies for c', then we select another cycle from the counterexample and apply the same analysis. This procedure is repeated until the spuriousness of the counterexample is established or there is no remaining cycle to be analyzed. In the latter case, our refinement procedure fails and the boundedness or livelock freedom analysis also terminates with an "UNKNOWN" verdict.

For the livelock freedom test, in case the spuriousness of a counterexample is proved, we can use the cycle dependencies that we have discovered so far to refine the abstract system of the original model: Any ACD (c', D') that we obtained imposes restrictions on the potential executions of the model, resulting in two possibilities, namely (1) the cycle c' is not repeated infinitely often, or (2) c' is repeated infinitely often and one of the cycles in D' is also repeated infinitely often. Each of the two above possibilities can be expressed as a set of inequalities and hence easily used to further constrain the original set of ILP constraints. Suppose that c' corresponds a set of cycles C_c in the respective independent cycle system, and the cycles in C_c correspond to the set of variables X_c in the livelock freedom determination ILP problem. Furthermore, we assume that the cycles in D' correspond to a set of cycles C_d in the independent cycle system. The cycles in C_d correspond to the set of variables X_d in the livelock freedom determination ILP problem. Then, we can encode the first possibility to the following set of inequalities:

$$\sum_{x_i \in X_c} x_i = 0 \tag{12.1}$$

and the second possibility to

$$\sum_{x_i \in X_c} x_i > 0 \tag{12.2}$$

$$\sum_{x_i \in X_d} x_i > 0 \tag{12.3}$$

We use the above obtained linear inequalities to constrain the livelock freedom determination ILP problem in order to refine the abstraction. However, linear inequality systems cannot express alternative possibilities – disjunctions – in general. As a consequence, we must build two new ILP problems, each augmenting the original ILP problem with one of the two sets above. In order to prove livelock freedom for the original Promela model, we must show that both newly constructed ILP problems have no solutions.

Example 12.1. Consider the simple Promela model in Example 10.1 in Section 10.1, whose corresponding abstract CFSM system is shown in Figure 10.1. Let the progress transition be the transition labeled with `ch2?`m_4 in the CFSM process of `P1`. Then, the following ILP problem is obtained for the determination of livelock freedom:

$$x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{12.4}$$

$$x_1 + x_2 + x_3 > 0 \tag{12.5}$$

$$x_1 = 0 \tag{12.6}$$

The above ILP problem has a solution $x_1 = x_3 = 0, x_2 = 1$. Therefore, we have obtained a counterexample that consists of only the cycle that corresponds to the left cycle in the control flow graph shown in Figure 12.1. Moreover, the left cycle is terminating on the boolean statement `(x == 0)`. Using the cycle dependency discovery methods that we presented in the previous chapter, we can easily determine that the left cycle depends on the right cycle in Figure 12.1. By this dependency information, we can determine that the counterexample is spurious.

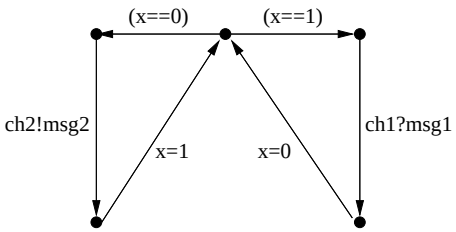

Figure 12.1: The two control flow cycles of the proctype `P2` in the Promela model in Listing 10.1.

Furthermore, from the above mentioned cycle dependency, we obtain two sets of linear inequalities

$$x_2 = 0 \tag{12.7}$$

and

$$x_2 > 0 \tag{12.8}$$
$$x_3 > 0 \tag{12.9}$$

From the livelock freedom determination ILP problem above, we construct the following two ILP problems:

$$x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{12.10}$$
$$x_1 + x_2 + x_3 > 0 \tag{12.11}$$
$$x_1 = 0 \tag{12.12}$$
$$x_2 = 0 \tag{12.13}$$

and

$$x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{12.14}$$
$$x_1 + x_2 + x_3 > 0 \tag{12.15}$$
$$x_1 = 0 \tag{12.16}$$
$$x_2 > 0 \tag{12.17}$$
$$x_3 > 0 \tag{12.18}$$

Both above ILP problems have no solution. Therefore, the original model is proved to be free of livelock.

Soundness. We need to formally prove that the above described refinement preserves the soundness of the livelock freedom test.

Proposition 12.1. *Given a Promela model, suppose that we have identified a set of cycles PC in its independent cycle system as progress cycles. Moreover, we have determined an ACD (c, D) for some cycle c and a set of cycles D in the control flow graphs of the original Promela model. If the livelock freedom test decides that the model is free of livelock after an abstraction refinement based on (c, D), then the model is actually free of livelock.*

Proof. Without loss of generality, we assume that the livelock freedom determination ILP problem for the model without refinements is as follows

$$x_1 \cdot \bar{e}_1 + \cdots + x_n \cdot \bar{e}_n \geq \bar{0} \tag{12.19}$$
$$\sum_{i=1}^{n} x_i > 0 \tag{12.20}$$
$$\sum_{c_i \in PC} x_i = 0 \tag{12.21}$$

Suppose that c corresponds a set of cycles C_c in the respective independent cycle system, and the cycles in C_c correspond to the set of variables X_c in the above

ILP problem. Furthermore, we assume that the cycles in D correspond to a set of cycles C_d in the independent cycle system. The cycles in C_d correspond to the set of variables X_d in the above ILP problem. The refinement based on the ACD (c, D) then results in the following two ILP problems:

$$x_1 \cdot \bar{e}_1 + \cdots + x_n \cdot \bar{e}_n \geq \bar{0} \tag{12.22}$$

$$\sum_{i=1}^{n} x_i > 0 \tag{12.23}$$

$$\sum_{c_i \in PC} x_i = 0 \tag{12.24}$$

$$\sum_{x_i \in X_c} x_i = 0 \tag{12.25}$$

and

$$x_1 \cdot \bar{e}_1 + \cdots + x_n \cdot \bar{e}_n \geq \bar{0} \tag{12.26}$$

$$\sum_{i=1}^{n} x_i > 0 \tag{12.27}$$

$$\sum_{c_i \in PC} x_i = 0 \tag{12.28}$$

$$\sum_{x_i \in X_c} x_i > 0 \tag{12.29}$$

$$\sum_{x_i \in X_d} x_i > 0 \tag{12.30}$$

Because livelock freedom is determined for the model, the two ILP problems above have no solutions.

It suffices to prove that, if the two ILP problems above have no solutions, then the abstract CFSM system is free of livelock with the restriction that if some cycle in C_c is executed infinitely often then some cycle in C_d is also executed infinitely often. By contradiction, we assume that the CFSM system has a livelocked execution r. So, after some point of runtime t_0, no cycle in PC is executed in r. There are two possible cases for r as follows.

First, if after t_0 all cycles in C_c are executed at most a finite number of times, then there exists another point of runtime $t_1 > t_0$ such that no cycle in C_c is executed after t_1. Because all processes in the CFSM system have a finite control flow structure, the infinite suffix of r after t_1 must contain an infinite number of configurations such that they agree on the local state of each process. We order these configurations by their occurrence in r and obtain an infinite sequence $\langle s_1, s_2, \ldots \rangle$. Let $\mathit{eff}(s_i)$ denote the effect vector that describes the number of messages of each type at s_i. Note that each $\mathit{eff}(s_i)$ is bounded below by the all-zero vector. Following Lemma 5.5, there exist two configurations s_i and s_j such that $i < j$ and $\mathit{eff}(s_i) \leq \mathit{eff}(s_j)$. The path between s_i and s_j can be decomposed into a linear combination of cycles in which at least one cycle is executed and no cycle in $C_c \cup PC$ is executed. Apparently, this linear combination is non-negative, which contradicts the fact that there is no solution to the ILP problem (12.22–12.25).

Second, if after t_0 some cycle $c_p \in C_c$ and some cycle $c_q \in C_d$ are executed infinitely often, then we select randomly a configuration s_1 at which one execution of c_p is started. After s_1, we may select another configuration s_2 such that (1) s_1 and s_2 agree on the local state of each process; (2) another execution of c_p is started at s_2; and (3) between s_1 and s_2 at least one execution of c_q is completed. This selection procedure can continue forever because c_p and c_q are executed infinitely often, which results in an infinite sequence $\langle s_1, s_2, \dots \rangle$. Similar to the argument for the first case, there exist two configurations s_i and s_j such that $i < j$ and $\mathit{eff}(s_i) \leq \mathit{eff}(s_j)$. The path between s_i and s_j can be decomposed into a linear combination of cycles in which at least one cycle in C_c is executed, at least one cycle in C_d is executed, and no cycle in PC is executed. Apparently, this linear combination is non-negative, which contradicts the fact that there is no solution to the ILP problem (12.26–12.30). □

Incompleteness. Because the methods explained in the previous chapter can only approximate cycle dependencies and cannot find all cycle dependencies, our refinement procedure is inevitably incomplete. As a consequence, it may be the case that a counterexample is spurious, but our refinement method fails to determine its spuriousness. In this case, the imprecision of the livelock freedom analysis cannot be remedied.

We have shown above that cycle dependencies can be used to refine abstractions in livelock freedom analyses. However, using the following example, we can see that, while cycle dependencies are sufficient to determine counterexample spuriousness, they are insufficient to refine abstractions for the buffer boundedness test.

Example 12.2. Consider a Promela model consisting of only one process whose control flow graph is given in Figure 12.2. Apparently the boundedness test will return a counterexample containing only the bottom cycle in the graph since it sends a message without receiving any messages. We can determine a cycle dependency that the bottom cycle depends on its only neighbor – the right cycle. Refining the abstraction with this dependency, the boundedness test cannot find any positive linear combination of cycles and would thus return "BOUNDED". However, we can easily see that the system is unbounded. This is because the top cycle that increases the variable `y` can be repeated without constraints. Consequently, the number of messages `m1` that can be sent by the bottom cycle is also unbounded.

In the next section, we will propose a finer concept of cycle dependencies that contain the information of cycle iteration times. With this additional information, cycle dependencies can be used to refine abstractions for buffer boundedness analyses. In addition, we show in the following that the use of the finer concept of cycle dependencies can also result in a more efficient livelock freedom analysis.

The previously described refinement method doubles the number of ILP problems for the determination of livelock freedom. This may result in an exponential growth of the number of ILP problems along the iterative refinement procedure. This problem can be overcome when we have knowledge about cycle iteration times. Consider the above example, we can easily see that the left cycle in Figure 12.1 can be repeated without interruption at most once. This is

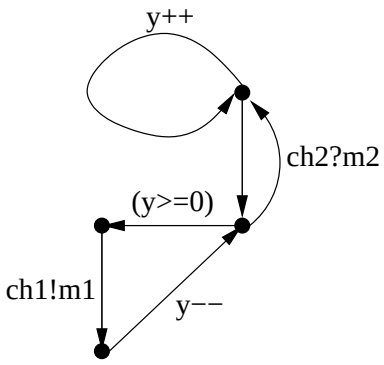

Figure 12.2: A control flow graph.

because the assignment statement `x = 1` immediately renders the condition `(x == 0)` false. With such information, we know that, in any infinite execution of the model, every one time that the left cycle is executed, the right cycle has to be executed at least once. Therefore, it suffices to use only one linear inequality to express the two possibilities mentioned previously:

$$x_2 \leq 1 \times x_3 \tag{12.31}$$

Now, we need only to add the above linear inequality to the original livelock freedom determination ILP problem to refine the abstraction. The obtained ILP problem is as follows:

$$x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + x_3 \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{12.32}$$

$$x_1 + x_2 + x_3 > 0 \tag{12.33}$$

$$x_1 = 0 \tag{12.34}$$

$$x_2 \leq x_3 \tag{12.35}$$

The above ILP problem has no solution and the original model is therefore free of livelock. Note that Inequality (12.35) can be written as $x_2 - x_3 \leq 0$. The introduction of this inequality still preserves the homogeneity of the ILP problem.

In the next section we give a variant definition of cycle dependencies to include the information of cycle iteration times.

12.2 Numerical Cycle Dependencies

We propose a finer definition of cycle dependencies as follows.

Definition 12.1 (Numerical Cycle Dependencies)**.** A *numerical cycle dependency* (or NCD) is a triple (n, c, C), consisting of a natural number n, a control flow cycle c and a set of cycles C in a Promela model, such that every n times that c is executed one of the cycles in C is also executed at least once in any infinite execution of the model.

Given an NCD (n, c, C), it is easy to see that (c, C) is also an ACD according to Definition 11.3.

Given a global cycle dependency (c, C) caused by a message determined condition, we can easily see that $(1, c, C)$ is an NCD. This is because every execution of the cycle c needs to consume a message that only the cycles in C can supply.

It is however difficult to determine NCDs caused by locally determined conditions or the globally determined conditions that are not influenced by any message receiving statements, since it involves the estimation of control flow cycle iteration times. We propose the following approximative method to determine NCDs caused by locally determined conditions. NCDs caused by globally determined conditions can be similarly derived.

12.2.1 Determining Numerical Cycle Dependencies from Locally Determined Conditions

The core of our NCD determination method is to estimate, for each condition statement `(`B`)` in a cycle c, the maximal number of times $max(B)$ that c can be successively repeated during which (1) the variables in $E_c(B)$ are only modified within c and (2) the condition B still remains true. Having determined such $max(B)$, we know that every $max(B)$ times that the cycle is executed some cycles must be executed at least once to modify the variables in $E_c(B)$. However, it is in general impossible to precisely determine $max(B)$ and we will hence resort to computing an upper bound n for $max(B)$. It is easy to see that this is a safe approximation: The restriction that other cycles have to be executed every n times that c is executed does not exclude the possibility that other cycles are executed every $max(B)$ times that c is repeated.

Our method can only apply to a condition statement in a cycle such that (1) the boolean condition in the condition statement contains only linear expressions; and (2) inside the cycle only linear expressions can be assigned to any variable in the considered condition statement. We notice that this is not overly constraining since Promela code respecting the above restrictions is commonly observed in real life models. We currently also do not consider complex data structures such as arrays and records, among others, to simplify our analysis. These complex data structures will be considered in future work.

Given a condition statement `(`B`)` in a cycle c, suppose that B is locally determined and in its disjunctive normal form $d_1 \vee \cdots \vee d_n$ where each disjunct d_i is a conjunction of several linear inequalities $a_{i1}, \ldots, a_{im_i}$. We further assume that each inequality is in the form $e_{ij} \geq f_{ij}$ where e_{ij} is a linear combination of the variables in B and f_{ij} is a constant.

For each disjunct d_i of B, we denote by $max(d_i)$ the maximal number of times that c can be repeated during which the variables $E_c(d_i)$ are modified within c and d_i remains true. If for any d_i and any inequality $e_{ij} \geq f_{ij}$ in d_i the value of e_{ij} is changed monotonically within c, then $max(B)$ is bounded by the sum of all $max(d_i)$ values – once some d_i becomes false, it cannot become true again without the variables in $E_c(d_i)$ to be modified outside c. Therefore, the problem of estimating $max(B)$ can be reduced to the problem of determining each $max(d_i)$ value. Similarly, for each inequality a_{ij} in d_i, we can estimate the $max(a_{ij})$ value. The $max(d_i)$ value is the minimum of all $max(a_{ij})$ values, because if any inequality a_{ij} is not satisfied then d_i is not satisfied. We can

therefore further reduce the computation of $max(d_i)$ to the computation of $max(a_{ij})$ values.

Example 12.3. Consider as an example the cycle in the control flow graph in Figure 12.3 where `x` and `y` are integer variables. Before the cycle is entered, `x` is initialized to 6 and `y` is initialized to -1. During each execution of the cycle, the variable `x` is decremented by 2 and `y` is incremented by 1. We use B to denote the boolean condition $(\mathtt{x}+\mathtt{y} \geq 3) \vee (-\mathtt{y} \geq 0)$ in the condition statement in the cycle. Let $d_1 = a_{11} = \mathtt{x}+\mathtt{y} \geq 3$, and $d_2 = a_{21} = -\mathtt{y} \geq 0$. It can be manually determined that $max(d_1) = max(a_{11}) = 3$, $max(d_2) = max(a_{21}) = 2$, and $max(B) = 3$. However, our method would determine an upper bound for $max(B)$ as 5 $(3+2)$. Such an upper bound is based on the assumption that only one disjunct is true at any time, and when it becomes false another disjunct immediately becomes true to make the whole boolean condition true. We can easily see that it is not the case in this particular example, which indicates the potential coarseness of our method when dealing with boolean conditions with disjunctions.

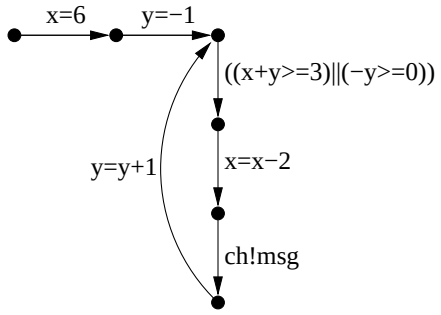

Figure 12.3: A control flow graph.

In the following we show how to determine $max(a_{ij})$ for an inequality $a_{ij} = e_{ij} \geq f_{ij}$. An upper bound on $max(a_{ij})$ can be obtained if (1) an upper bound on the initial value of e_{ij} before the cycle is entered can be determined; and (2) it can be determined that the value of e_{ij} is always decreased, therefore changed monotonically, in each iteration of the cycle.

Determining Initial Values. It is certainly impossible in general to determine the possible initial values of an expression before the cycle is entered. We use an incomplete solution that employs backward depth first searches to check each acyclic path leading to one of the exit points of the cycle whether e_{ij} receives a constant value on that path before the cycle is entered. If there exists an acyclic path along which e_{ij} does not receive a constant value, then our attempt to determine the initial value for e_{ij} fails. Otherwise, we take the maximum of all determined constant values as the initial value of e_{ij} in the determination of $max(a_{ij})$.

Determining Step Values. The Promela code in the cycle determines how the value of e_{ij} is changed during cycle executions. We refer to the maximal difference of the values of e_{ij} before and after one cycle execution as the *step*

value of e_{ij}. If the step value is negative, i.e., the value of e_{ij} is always decreased, then the linear inequality a_{ij} cannot remain true without the variables in $E_c(a_i)$ being modified outside the cycle. Determining the step value of e_{ij} is also impossible in general. We propose an incomplete method based on linear program (LP) solving.

We generate a set of LP problems from the cycle code, whose constraints reflect how the runtime values of variables are changed and constrained by the cycle code. The objective functions of these LP problems are the difference of the values of e_{ij} before and after one cycle execution. We maximize the objective function to get the step value of e_{ij} under the constraints of the linear inequalities to which the cycle code is transformed. The generation of the LP problem constraint involves the transformation of each Promela statement in the cycle code into one or more inequalities. The transformation is described as follows. We introduce a subscripted variable for each of the variables used in the cycle code. All subscripts have an initial value as 0. We start the transformation from the considered condition statement `(`B`)`. We also create a set of LP problems L consisting of one LP problem containing currently no constraint. We treat different kinds of statements in different ways as explained below:

- For the considered statement `(`B`)`, if it is the first time to process this statement, then we construct a linear inequality as the same as e_{ij} by substituting each variable with its corresponding subscripted variable. We add this linear inequality to the LP problem in L. The subscripts remain unchanged, and we move to the next statement in the cycle. If it is the second time to encounter this statement, then we terminate the transformation procedure.

- For any other condition statement `(`B'`)`, we convert B' to its disjunctive normal form $d'_1, \ldots, d'_n$. For each d'_k, we construct a set of linear inequalities. Each constructed inequality is obtained from one inequality l in d'_k by substituting each variable in l with its corresponding subscripted variable. Next, we construct n new LP problem for each LP problem P in L such that each k-th new LP problem is obtained by adding to P the inequalities that we construct for d'_k. The subscripts remain unchanged, and we move to the next statement in the cycle.

- For an assignment `x =` e, we construct an equation $x_{k+1} = e'$ where k is the current subscript for the variable `x` and e' is obtained from e by substituting each variable with its corresponding subscripted variable. We add this equation to every LP problem in L. All subscripts except the one for `x` remain unchanged. The subscript for `x` is incremented by 1, i.e., the new subscript is $k + 1$. We move to the next statement in the cycle.

- For a message sending statement, we simply move to the next statement in the cycle without doing anything.

- For a message receiving statement, we increase the subscripts for all the variables in the statement by 1. In this way, for any variable in the statement, we do not have any connection between the current version of the variable and the previous version. This corresponds to the fact that the

variable may receive an arbitrary value by the message receiving statement. This results in imprecision and may fail the cycle iteration time estimation. We then move to the next statement in the cycle.

After the transformation procedure ends, we add the same objective function to every LP problem in L: We use e_{ij}^{init} to denote the version of e_{ij} where all the variable subscripts are 0, and e_{ij}^{final} to denote the version of e_{ij} with the final version of each variable subscript. The objective function is then $e_{ij}^{final} - e_{ij}^{init}$.

Example 12.4. Consider the example in Figure 12.3. The cycle code is transformed into the following LP problem for determining step values for $\mathtt{x}+\mathtt{y}$, the first inequality in the condition statement.

$$\mathtt{max}: x_1 + y_1 - x_0 - y_0 \tag{12.36}$$

$$x_0 + y_0 \geq 3 \tag{12.37}$$
$$x_1 = x_0 - 2 \tag{12.38}$$
$$y_1 = y_0 + 1 \tag{12.39}$$

We solve all the LP problems for determining the step value for e_{ij}. If the optimal solutions to the objective function of all LP problems are negative, then we know that the value of e_{ij} is always decreased during cycle executions. In this case, we take the maximum of all optimal solutions as the step value of e_{ij}. Otherwise, if some optimal solution is positive or 0, then it is possible that the value of e_{ij} can be increased or unchanged after one cycle execution. In this case, we cannot determine if a_{ij} will eventually become false or not, and the estimation of $max(B)$ also fails because we lose the monotonicity property. In the above example, we can easily compute that the optimal solution to the objective function (12.36) is -1. Therefore, the step value of $\mathtt{x}+\mathtt{y}$ is -1.

Computing $max(a_{ij})$. Once we obtain the step value sv and the initial value iv for e_{ij}, an upper bound on $max(a_{ij})$ is simply $\lceil (f_{ij} - (iv+1))/sv \rceil$[1]. In the above example, if the initial value of $\mathtt{x}+\mathtt{y}$ is 5, then $max(\mathtt{x}+\mathtt{y} \geq 3) = 3$.

Determining Numerical Cycle Dependencies After we successfully determine $max(B)$, we can construct two NCDs as

$$(max(B), c, C)$$

where (c, C) is the ACD that we obtain using the algorithm in Listing 11.5 with respect to the condition B, and

$$(max(B), c, N_c)$$

where N_c is the set of neighbors of c.

[1] $\lceil \cdot \rceil$ is the ceiling function that returns the smallest integer greater than the input real number.

Complexity. The method for determining NCDs requires the estimation of cycle iteration times by transforming cycle code into a set of LP problems and the solving of these LP problems. If a control flow cycle contains integer variables, the constructed LP problems are then ILP problems or mixed LP problems. Since these ILP or mixed LP problems are not homogeneous in general, the solving of them is NP-hard. However, we usually obtain small LP problems from cycle code in practice, which can be solved in short time. In order to further improve the efficiency, we may slice the cycle code to remove all the statements irrelevant to the satisfaction of the considered boolean condition [114]. More precisely, given a cycle c and a condition statement `(`B`)`, when we transform the code in c to LP problems for the estimation of $max(B)$, we can disregard all condition statements that contain no variables in $E_c(B)$, and all assignment statements and message receiving statements that do not change any variables in $E_c(B)$. This is because no variables outside $E_c(B)$ have influence on the value of the boolean condition B as explained in Section 11.3.1. The slicing strategy may reduce the size of the constructed LP problems as observed in practice.

The method proposed in this section can be used to prove termination for control flow cycles and estimate cycle iteration times. Given a cycle, if we can determine $max(B)$ for every condition statement `(`B`)`, then the cycle is terminating. Let m be the minimum of all determined $max(B)$ values. Then, the cycle cannot be repeated without interruption more than m times.

12.2.2 Refinement by Numerical Cycle Dependencies

Given an NCD (n, c, D), suppose that c corresponds a set of cycles C_c in the respective independent cycle system, and their corresponding variables in the boundedness determination ILP problems are in the set X_c. Furthermore, we assume that the cycles in D correspond to a set of cycles C_d in the independent cycle system. The cycles in C_d correspond to the set of variable X_d in the boundedness determination ILP problems. Then, we can encode the NCD to the following linear inequality:

$$\sum_{x_i \in X_c} x_i \leq n \cdot \sum_{x_i \in X_d} x_i \tag{12.40}$$

The above constraint can be then added to the boundedness determination ILP problems for refining the abstraction. After one refinement based on an NCD, the number of ILP problems to solve remains unchanged. Therefore, it is more efficient than abstraction refinements based on non-numerical cycle dependencies.

We formally prove now that the above described refinement based on NCDs preserves the soundness of the buffer boundedness test. The soundness result for livelock freedom analyses can be similarly obtained and is omitted here.

Proposition 12.2. *Given a Promela model, suppose that we have determined an NCD (m, c, D) for some cycle c and a set of cycles D in the control flow graphs of the model. If the buffer boundedness test returns "BOUNDED" for the model after one refinement based on (m, c, D), then the model is actually bounded.*

Proof. Without loss of generality, we assume that the boundedness determination ILP problem for the model without refinements is as follows

$$x_1 \cdot \bar{e}_1 + \cdots + x_n \cdot \bar{e}_n > \bar{0} \tag{12.41}$$

Suppose that c corresponds to a set of cycles C_c in the respective independent cycle system, and the cycles in C_c correspond to the set of variables X_c in the above ILP problem. Furthermore, we assume that the cycles in D correspond to a set of cycles C_d in the independent cycle system. The cycles in C_d correspond to the set of variables X_d in the above ILP problem. The refinement based on the NCD (m, c, D) then results in the following ILP problem:

$$x_1 \cdot \bar{e}_1 + \cdots + x_n \cdot \bar{e}_n > \bar{0} \tag{12.42}$$

$$\sum_{x_i \in X_c} x_i \leq m \sum_{x_i \in X_d} x_i \tag{12.43}$$

Because buffer boundedness is proved for the model, the above ILP problem has no solutions. Given any vector $\bar{e}$, we now prove that (*) if the above ILP problem has no solutions then there exists a vector $\bar{b}$ such that $f(x_1, \ldots, x_n) = \bar{e} + \sum_{i=1}^{n} x_i \cdot \bar{e}_i \geq 0 \rightarrow f(x_1, \ldots, x_n) \leq \bar{b}$ for all $x_1, \ldots, x_n$ where $\sum_{x_i \in X_c} x_i \leq m \sum_{x_i \in X_d} x_i$. By contradiction, we assume that there exists no bound. Then, according to the proof of Proposition 5.4, we can construct a sequence of strictly increasing non-negative integer vectors $\langle \bar{c}_1, \bar{c}_2, \ldots \rangle$ such that $f(\bar{c}_i)$ is also strictly increasing and the following is satisfied: Let I_c denote the set of indices j such that $x_j \in X_c$, I_d denote the set of indices j such that $x_j \in X_d$. For any $\bar{c}_i$, we have that $\sum_{j \in X_c} \bar{c}_i^j \leq m \sum_{j \in X_d} \bar{c}_i^j$. Now we select randomly two $\bar{c}_i$ and $\bar{c}_j$ such that $\bar{c}_i < \bar{c}_j$. It is easy to see that $\bar{d} = \bar{c}_j - \bar{c}_i$ has the property that $\sum_{j \in X_c} \bar{d}^j \leq m \sum_{j \in X_d} \bar{d}^j$. Therefore, $\bar{d}$ is a solution to the ILP problem (12.42–12.43). This leads to a contradiction.

Now we start to prove the proposition. It suffices to prove that, if the ILP problem above has no solutions, then the abstract CFSM system is bounded with the following restriction: Every m times that the cycles in C_c are repeated some cycle in C_d is executed at least once. By contradiction, we assume that the CFSM system is unbounded.

Any finite prefix of an execution of the CFSM system can be decomposed into an acyclic part and a cyclic part. Because there are only finitely many possible acyclic parts for execution prefixes, there is an upper bound $\bar{b}_a$ on the effect vectors generated by acyclic parts. Furthermore, the cyclic part of an execution prefix can be decomposed into a linear combination of cycles. Consider any finite execution prefix r in which the cycles in C_c are executed i times and the cycles in C_d are executed j times such that $i > m \cdot j$. We decompose the cyclic part of r into the following two linear combinations of cycles: (1) a linear combination containing $m \cdot j$ cycles from C_c, j cycles from C_d, and all other cycles outside $C_c \cup C_d$; and (2) a linear combination containing the rest $(i - m \cdot j)$ cycles from C_c. Note that $(i - m \cdot j) \leq m$ because every m times that the cycles in C_c are repeated one cycle in C_d has to be executed at least once. Therefore, the effect vector of the second linear combination is bounded by some vector $\bar{b}_2$. In the statement (*) above, let $\bar{e} = \bar{b}_a + \bar{b}_2$, and we can see that the first linear combination constructed from r satisfies the condition $\sum_{x_i \in X_c} x_i \leq m \sum_{x_i \in X_d} x_i$. We denote by $\bar{b}_1$ the effect vector generated by the

first linear combination. Then, $\bar{b}_a + \bar{b}_2 + \bar{b}_1$ is a buffer bound for any execution prefix r. Consequently, the CFSM system is bounded, which contradicts the assumption. □

12.3 Refinement by Graph-Structures

We have discussed so far the abstraction refinement procedure based on the notion of cycle dependencies that are actually mutual inclusion relations among cycles imposed by condition statements. As mentioned previously, there are other types of dependencies among cycles which are caused by graphic structures of control flow graphs. Examples were given in the end of Section 10.2, including Example 10.3. In this section we consider the discovery of graph structure imposed dependencies and the use of such information to refine abstractions. Unlike previously, the analysis described in this section can be performed at the level of CFSM systems since we are only interested in the graphic characteristics and CFSM systems preserve the structures of their original Promela models. Based on reasons similar to the argument in Section 12.1, the refinement methods described in this section apply only to the livelock freedom test.

Given a counterexample in which two cycles in one same process do not share common states, some other cycles in the same process have to be included in the counterexample to "bridge" them. We introduce the concept of *self-connected cycle sets*.

Definition 12.2 (Self-Connected Cycle Sets)**.** A set of control flow cycles in a CFSM process is *self-connected* if and only if these cycles form a strongly connected subgraph of the state machine of the process.

A counterexample is spurious if, for some process p, the set of cycles from p in the counterexample is not self-connected.

We propose the following refinement. Given a counterexample, suppose that there are two cycles c_1 and c_2 from one same process p in the counterexample such that they are not reachable from each other by traversing through only the cycles in the counterexample. We determine all the self-connected sets of cycles of p that contain both c_1 and c_2. If no such set exists, then c_1 and c_2 belong to different strongly connected components. This implies that c_1 and c_2 mutually exclude each other. Therefore, there are three possible cases: (1) c_1 is executed infinitely often while c_2 is not; (2) c_2 is executed infinitely often while c_1 is not; and (3) both cycle are not executed infinitely often. Let x_1 and x_2 be the variables corresponding respectively to c_1 and c_2 in the livelock freedom determination ILP problems. The above possibilities can be expressed as three sets of linear inequalities

$$x_1 > 0 \tag{12.44}$$
$$x_2 = 0 \tag{12.45}$$

and

$$x_1 = 0 \tag{12.46}$$
$$x_2 > 0 \tag{12.47}$$

and

$$x_1 = 0 \tag{12.48}$$
$$x_2 = 0 \tag{12.49}$$

The sets above triplicates the number of livelock freedom determination ILP problems as each ILP problem can only be augmented with one of the three sets.

Alternatively, suppose that there exists at least one self-connected set of cycles containing both c_1 and c_2. Let n denote the number of determined self-connected cycle sets. Then, there are n different ways to "connect" the cycles c_1 and c_2. For each determined self-connected set S, we can generate a set of linear inequalities as below, assuming that each x_i is the variable corresponding to c_i in the livelock freedom determination ILP problem

$$\bigwedge_{c_i \in S} x_i > 0 \tag{12.50}$$

Besides, we also have the three possibilities as expressed respectively in Inequalities (12.44–12.49). In this way, we obtain $n+3$ sets of linear inequalities. Each set will be used to modify the livelock freedom determination ILP problems in the same way as previously explained.

Example 12.5. Consider the state machine in Figure 12.4. All the self-connected cycle sets containing c_1 and c_2 are $S_1 = \{c_1, c_2, c_3, c_4\}$ and $S_2 = \{c_1, c_2, c_5\}$. Let x_i be the variable corresponding to the cycle c_i in the livelock freedom determination problems. Then, we generate the following sets of linear inequalities

$$x_1 > 0 \tag{12.51}$$
$$x_2 > 0 \tag{12.52}$$
$$x_3 > 0 \tag{12.53}$$
$$x_4 > 0 \tag{12.54}$$

and

$$x_1 > 0 \tag{12.55}$$
$$x_2 > 0 \tag{12.56}$$
$$x_5 > 0 \tag{12.57}$$

and

$$x_1 > 0 \tag{12.58}$$
$$x_2 = 0 \tag{12.59}$$

and

$$x_1 = 0 \tag{12.60}$$
$$x_2 > 0 \tag{12.61}$$

and

$$x_1 = 0 \tag{12.62}$$
$$x_2 = 0 \tag{12.63}$$

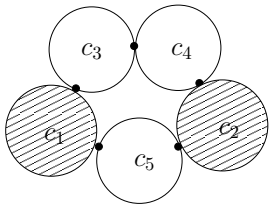

Figure 12.4: The state machine of a CFSM process.

A caveat of the above described method is that the number of newly generated ILP problems can grow exponentially, since the number of determined self-connected sets for two cycles can be exponential in the size of the respective process. As a solution, we propose an alternative method that only generates 4 sets of linear inequalities instead of $n + 3$ sets. Consider the state machine in Figure 12.4. To reach c_2 from c_1, one of the neighbors of c_1 must be entered. Similarly, one of the neighbors of c_2 must be entered in order to reach c_1 from c_2. Based on this observation, we can generate only one set of linear inequalities to restrict the behavior in which both cycles are repeated infinitely often, given as follows

$$x_1 > 0 \tag{12.64}$$
$$x_2 > 0 \tag{12.65}$$
$$x_3 + x_5 > 0 \tag{12.66}$$
$$x_4 + c_5 > 0 \tag{12.67}$$

To generalize this idea we assume that two cycles c and c' from some process p in a counterexample are not reachable from each other by traversing only the cycles in the counterexample. We compute a sequence of pairs of cycle sets: $\langle (N_c^0, N_{c'}^0), \ldots, (N_c^n, N_{c'}^n) \rangle$ where (1) N_c^0 is the set of neighbors of c, (2) N_c^{i+1} contains all cycles that neighbor some cycle in N_c^i and that are not in any N_c^j if $j \leq i$, and (3) $N_{c'}^i$ is defined likewise. We terminate the computation of the sequence for some number n such that either (1) N_c^n or $N_{c'}^n$ is empty, or (2) there are a cycle in N_c^n and a cycle in $N_{c'}^n$ such that these two cycles are neighbors. If the first termination condition is true, then c and c' must be in different strongly connected components. The treatment in this case is the same as in the previous refinement method. If the second termination condition is true, then we know that one cycle from each set in the sequence $\langle N_c^0, \ldots, N_c^n, N_{c'}^1, \ldots, N_{c'}^n \rangle$ has to be entered when traveling between c and c'. This can be expressed using the following set of inequalities, assuming that the variable x corresponds to c, x' corresponds to c', and each x_j corresponds to c_j in the livelock freedom

determination ILP problems

$$x > 0 \tag{12.68}$$
$$x' > 0 \tag{12.69}$$
$$\bigwedge_{i=1}^{n} \Big(\sum_{c_j \in N_c^n} x_j > 0 \Big) \tag{12.70}$$
$$\bigwedge_{i=1}^{n} \Big(\sum_{c_j \in N_{c'}^n} x_j > 0 \Big) \tag{12.71}$$

Consider the state machine in Figure 12.5. Let x_i be the variable corresponding to the cycle c_i in the livelock freedom determination ILP problems. If a counterexample contains c_1 and c_2 but no other cycles in the state machine, then the counterexample is spurious. Furthermore, we can obtain the following set of linear inequalities to restrict the behavior in which both c_1 and c_2 are repeated infinitely often

$$x_1 > 0 \tag{12.72}$$
$$x_2 > 0 \tag{12.73}$$
$$x_3 + x_5 > 0 \tag{12.74}$$
$$x_4 + x_6 > 0 \tag{12.75}$$

However, this constraint cannot exclude a potentially spurious counterexample that contains c_1, c_2, c_3, and c_6.

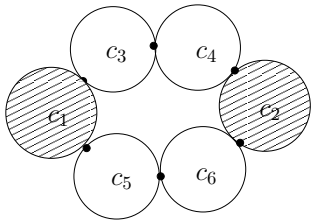

Figure 12.5: The state machine of a CFSM process

Summary We have proposed in this chapter several methods to refine abstractions based on discovering cycle dependencies. Some methods may increase exponentially the number of ILP problems for the determination of livelock freedom. A practical strategy of abstraction refinements is therefore to apply the refinement methods based on graph structure only when the abstract system cannot be refined anymore by the methods based on cycle dependencies. Finally, note that any linear inequality constructed in the refinement procedure is homogeneous such that any boundedness or livelock freedom determination ILP problem augmented with this inequality is still homogeneous and can be solved in polynomial time. The efficiency and effectiveness of our abstraction refinement procedure will be evaluated using case studies as described in Chapter 14.

Part IV

Implementations and Experiments

Chapter 13

Verification Tools

In the previous chapters we have presented the foundation and the technical details of our verification framework and its applications to the checking of the buffer boundedness and livelock freedom properties. We have implemented the verification methods for these two properties into two prototype tools: *IBOC*, a boundedness checker; and *aLive*, a livelock freedom checker. In this chapter we present the tool architecture and discuss some specific issues related to the implementation of these tools, such as cycle detections.

IBOC. The buffer boundedness test explained in Chapter 5 has been implemented in the IMCOS Boundedness Checker or, in short, IBOC. IBOC can check boundedness for both Promela models and UML RT models. When it successfully proves boundedness for a model without going through the refinement procedure, it also delivers an estimated bound for each buffer in the model. The code abstraction of Promela models is fully automated in IBOC, except that the actually running processes of a model must be manually determined for the time being. Manual abstraction is currently required as a preprocessing step for UML RT models. IBOC accepts only UML RT models in which the action code of transitions contains nothing more than a sequence of message sending statements. Moreover, the refinement procedure is only implemented for Promela models.

aLive. The livelock freedom test explained in Chapter 6 has been implemented in the aLive tool. Currently, aLive checks livelock freedom only for Promela models. Users can define their set of progress actions in a model by adding progress labels in front of the proper Promela statements in the model. In the Promela language, a label is a progress label if it has the prefix `progress`.

Structure. In Section 13.1 we show the common software architecture of the IBOC and aLive tools. Section 13.2 addresses the problem of enumerating elementary cycles in a control flow graph. Section 13.3 checks the feasibility to reduce the number of effect vectors used to construct ILP problems for the sake of efficiency. Finally, a refinement strategy for buffer bound estimation is suggested in Section 13.4.

13.1 Tool Architecture

As explained in the previous chapters, the buffer boundedness and the livelock freedom tests share the common abstraction techniques, the ILP solving techniques, and the refinement techniques. As a consequence, the IBOC and aLive tools may reuse the same modules for abstractions, ILP solving, refinement, and also cycle enumerations.

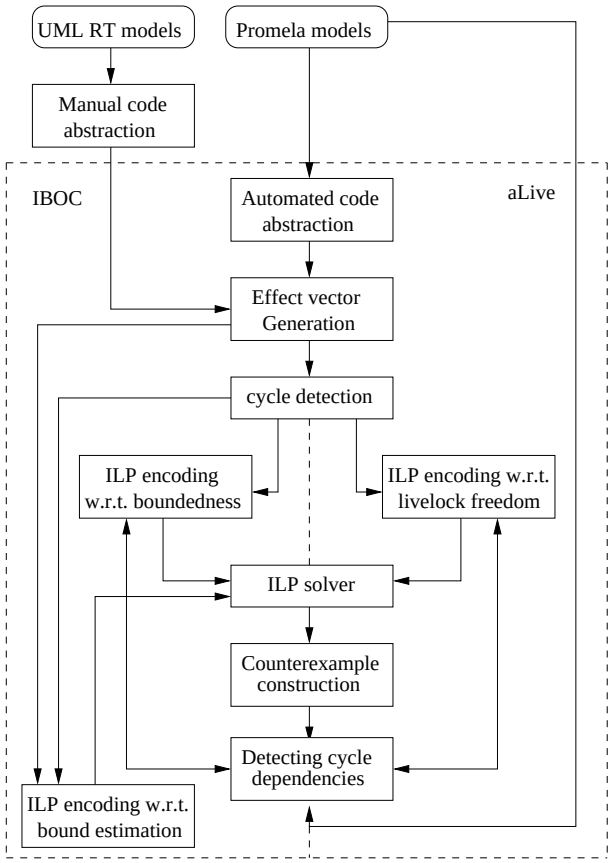

Figure 13.1: The common software architecture of IBOC and aLive.

Figure 13.1 shows the common software architecture of IBOC and aLive. The only difference in the implementations of the two tools is the different ILP encodings needed for the analysis of the respective property. Arrows denote data flow between different modules. We briefly explain the functionality of each module in the figure.

Automated code abstraction. This module can only abstract Promela models for the time being. UML RT models need manual abstraction of transition action code as a preprocessing step. The code abstraction of Promela models is detailed in Chapter 7. The output of this module is a CFSM system. In this step, message types are also determined for Promela models.

Effect vector generation. For the CFSM system generated by code abstraction, the message passing effect is computed for each transition. The output of this module is essentially a VASS system, although for reasons of efficiency this is not explicitly constructed.

Cycle enumeration. Control flow cycles are enumerated in this module and the summery effect vector is also calculated for each cycle. The output of this module is an independent cycle system. In IBOC the least upper bound on acyclic path effects is also computed along with the the detection of cycles. As shown in the next section, for efficiency reasons aLive may use a different cycle enumeration algorithm. However, in our implementation the same algorithm in Listing 13.1 is used in both IBOC and aLive. With this step the abstraction is completed.

ILP encoding. From the effect vectors in the independent cycle system, an ILP problem is constructed in this module for the checking of the property. IBOC and aLive use different ILP encodings as explained in Chapter 5 and Chapter 6, respectively. This module also translates the cycle dependencies discovered in the refinement procedure into linear inequalities, and refine the property checking ILP problems with these inequalities.

ILP solver. We currently use an open source linear optimization tool named `lp_solve` [5] for the solving of ILP problems. It uses a variant of the simplex algorithm to solve linear programming problems, and the combinatorial Branch-and-Bound algorithm to solve ILP problems. `lp_solve` does not take advantage of the polynomial solving method for homogeneous ILP problems as described in Section 5.7.2. However, we observed that it worked still efficiently in practice as the experimental results suggested in Chapter 14.

Counterexample construction. If the ILP solver returns a solution to a property checking ILP problem, then it indicates a potential violation of the checked property. This module finds all the control flow cycles in the original Promela model such that these cycles correspond to those cycles in the independent cycle system whose variables receive non-zero values in the returned ILP solution. The found cycles form a counterexample that will be subject to the cycle dependency analysis in the refinement procedure.

Detecting cycle dependencies. This module checks each cycle in the constructed counterexample for the discovery of cycle dependencies. For each cycle c, it tries to determine three sets of cycles that c may depend on: (1) neighboring cycles (see Corollary 11.5 in Section 11); (2) local supplementary cycles (see Listing 11.5); and (3) the cycles sending the types of messages that c needs (see Section 11.4). The first two kinds of dependencies are caused by locally determined conditions. Therefore, the discovery of them requires termination proving and cycle iteration times estimation that were implemented based on the idea in Section 12.2. The third kind of dependencies is caused by message determined conditions, whose discovery is detailed in Section 11.4. This module

currently does not support the discovery of dependencies caused by globally determined conditions. However, it can be easily implemented and we will include it in the future development of the IBOC and aLive tools.

13.2 Cycle Enumeration

Our asynchronous reactive systems verification framework centers around the analysis of control flow cycles. Therefore, it is practically important to use an efficient algorithm to collect elementary cycles in a control flow graph. Note that a control flow graph may contain an exponential number of elementary cycles. Therefore, any cycle enumeration algorithm has an exponential complexity lower bound. To the best of our knowledge, [111] gives the most efficient algorithm for collecting cycles based on depth first search, which is bounded by $O(n+m(c+1))$ time where n is the number of vertices, m is the number of edges, and c is the number of elementary cycles. Although the algorithm is devised for directed graphs in which at most one edge is allowed from a certain vertex to another, it can be easily adapted straightforwardly to apply to the control flow graphs or state machines in our setting – there may be multiple transitions from one state to another as a consequence of conservative code abstraction. In the adaption, while visiting each vertex, all outgoing transitions of the vertex are explored one after another, instead of exploring all successors of the vertex.

We also propose a cycle enumeration algorithm as shown in Listing 13.1, which explores all acyclic paths in a graph. Therefore, it can meanwhile compute the least upper bound on the effect vectors of acyclic paths for the estimation of buffer bounds. The proposed algorithm is a variant of Tiernan's cycle collecting algorithm in [113]. There are two main differences between our algorithm and Tiernan's algorithm: (1) Tiernan's algorithm applies to graphs in which at most one edge is allowed from one point to another, while our algorithm does not have this restriction; (2) our algorithm constructs each cycle exactly once and therefore avoids the need to check duplicate cycles in the end. Checking duplicate cycles can be very time consuming. Consequently, our algorithm greatly improves the efficiency of cycle enumeration. Furthermore, as the termination proof (Proposition 13.1) shows, each acyclic path of a graph is explored only once by the algorithm. This proves that our algorithm is the most efficient one for collecting cycles and computing the least upper bound on acyclic path effect vectors at the same time.

Proposition 13.1 (Termination)**.** *The algorithm in Listing 13.1 is terminating.*

Proof. The termination of the algorithm can be shown if we prove that (1) the stack `current_path` contains always an acyclic path; and (2) the contents of the stack before two distinct calls to `explore_acyclic_path` are always different. Because there are finitely many acyclic paths, the algorithm cannot run forever.

We first prove that the stack `current_path` contains always an acyclic path. By contradiction, we assume that it contains a cyclic path $\langle t_1, \ldots, t_n \rangle$ at some point of time. Then, there exist two transitions t_i and t_j in the path such that the source state s of t_i is the destination state of t_j. Let s' be the source state of t_j. Note that t_j must be pushed to the stack inside a call to `explore_acyclic_path(`s'`)` (Line 21). Moreover, it is possible only if the stack

```
1  proc collect_cycles(state_machine sm)
2    vector[integer] lub = <0,...,0>
3    vector[integer] eff = <0,...,0>
4    set[state] visited = {}
5    set[state] closed = {}
6    stack[transition] current_path = <>
7    set[cycle] cycles = {}
8    explore_acyclc_path(sm.initial_state)
9
10 proc explore_acyclic_path(state s_1)
11   add s_1 to visited
12
13   for each tr in outgoing_transitions(s_1)
14     state s_2 = tr.destination_state
15     if (exists tr_p in current_path
16           such that tr_p.source_state = s_2)
17       if (s_2 not in closed)
18         cycle c = construct_cycle(current_path, tr_p)
19         add c to cycles
20     else
21       push tr to current_path
22       eff = eff + tr.effect_vector
23       update(lub,eff)
24       explore_acyclic_path(s_2)
25       eff = eff - tr.effect_vector
26       pop current_path
27
28   add s_1 to closed
29
30 proc update(vector[integer] lub, vector[integer] eff)
31   for each i-th component of lub
32     if (lub[i] < eff[i])
33       lub[i] = eff[i]
```

Listing 13.1: An algorithm to collect all elementary cycles from a CFSM state machine.

`current_path` contains no transition whose source state is s (Line 20). However, the source state of t_i is s, which leads to a contradiction.

Second, we prove that the content of the stack is unique before each call to `explore_acyclic_path`. Consider a call to `explore_acyclic_path`(s_1) before a call to `explore_acyclic_path`(s_2). Note that it is possible that $s_1 = s_2$. The proof is based on an induction on the number k of transitions in the stack `current_stack` before these two calls.

Induction base: When $k = 0$, the stack is empty. We can easily see that only the call to `explore_acyclic_path` at Line 8 can be made with an empty stack. Before any other calls, some transition is pushed to the stack. Therefore, there cannot exist two calls before which the stack is empty.

Induction step: We assume that the content of the stack is unique before each call to `explore_acyclic_path` if the number of transitions in the stack is $k-1$. Let the content of the stack before the call to `explore_acyclic_path`(s_1) be $\langle t_1^1, \ldots, t_k^1 \rangle$, and the content before the call to `explore_acyclic_path`(s_2) be $\langle t_1^2, \ldots, t_k^2 \rangle$. Note that the call to `explore_acyclic_path`(s_1) must be made inside a call to `explore_acyclic_path`(s_3) such that t_k^1 is a transition from s_3 to s_1. Similarly, the call to `explore_acyclic_path`(s_2) must be made inside a call to `explore_acyclic_path`(s_4) such that t_k^2 is a transition from s_4 to s_2. There are then two possible cases. (1) If the two calls to `explore_acyclic_path`(s_3) and `explore_acyclic_path`(s_4) are one same call, then s_1 and s_2 must be different. This implies that $t_k^1 \neq t_k^2$. (2) If the two calls to `explore_acyclic_path`(s_3) and `explore_acyclic_path`(s_4) are distinct, then there exists some $1 \leq i \leq k-1$ such that $t_i^1 \neq t_i^2$. □

Proposition 13.2 (Soundness)**.** *The algorithm in Listing 13.1 collects all cycles in a CFSM state machine. Moreover, the cycles in the set* `cycles` *are pairwise different.*

The proof of the above proposition needs the following auxiliary lemmas.

Lemma 13.3. *During the execution of the algorithm in Listing 13.1, the stack* `current_path` *contains each of all acyclic paths of the checked state machine before a call to* `explore_acyclic_path`*.*

Proof. Consider an arbitrary acyclic path $\langle t_1, \ldots, t_k \rangle$. We prove the lemma based on an induction on k.

Induction base: When $k = 0$, the acyclic path is empty. It is contained in the stack before the call at Line 8.

Induction step: Suppose that every acyclic path of length $k-1$ is contained in the stack before some call to `explore_acyclic_path`. Then, $\langle t_1, \ldots, t_{k-1} \rangle$ is contained in the stack before a call to `explore_acyclic_path`(s_k) where s_k is the destination state of t_{k-1}. Inside this call, t_k is pushed into the stack before a call to `explore_acyclic_path`(s_{k+1}) where s_{k+1} is the destination state of t_k. This is because $\langle t_1, \ldots, t_k \rangle$ is acyclic and the condition at Line 15 is consequently not satisfied. □

Corollary 13.4. *In the algorithm in Listing 13.1, we drop the conditional at Line 17 such that a cycle is constructed as long as the condition at Line 15 is satisfied. The modified algorithm collects all cycles in a CFSM state machine.*

Proof. By contradiction, we assume that some cycle $\langle t_1, \ldots, t_k \rangle$ is not detected by the algorithm. Without loss of generality, let the source state s of t_1 be the first state in the cycle to be included into the set `visited`. Let p be the content of the stack `current_path` before `explore_acyclic_path(`s`)` is called for the first time. Then, we can easily see that $p.\langle t_1, \ldots, t_k \rangle$ is an acyclic path. Following Lemma 13.3, there exists a call to `explore_acyclic_path(`s'`)` such that s' is the destination state of t_k and the content of the stack before the call is $p.\langle t_1, \ldots, t_k \rangle$. Inside this call, the cycle $\langle t_1, \ldots, t_k \rangle$ will be detected when t_k is examined against the condition at Line 15. This leads to a contradiction. □

Lemma 13.5. *In the modified algorithm in Corollary 13.4, suppose that at some point of time the content of the stack* `current_path` *contains an acyclic path in which a state s is not closed. Then, all states in the path before s are not closed at that time.*

Proof. Assume that the predecessor of s in the acyclic path is s'. It is sufficient to prove that s' is also not closed. By contradiction, we assume that s' is closed. Then, s' must have been added to the set `closed` earlier inside a call to `explore_acyclic_path(`s'`)` (Line 28). Inside this call, a call to `explore_acyclic_path(`s`)` was made. Moreover, s' was added into `closed` only after the call to `explore_acyclic_path(`s`)` was completed by which s should have been included into `closed`. This leads to a contradiction. □

Note that the above lemma also holds for the algorithm in Listing 13.1.

Lemma 13.6. *In the modified algorithm in Corollary 13.4, suppose that a cycle* $\langle t_1, \ldots, t_n \rangle$ *is found when the content of the stack is* $p.\langle t_1, \ldots, t_{n-1} \rangle$*. Let* s_i *be the source state of each* t_i *where* $1 \leq i \leq n$*. If* s_1 *is closed, then the cycle must have been discovered by the modified algorithm before.*

Proof. Because s_1 is closed and still in the stack, there must be an earlier call to `explore_acyclic_path(`s_1`)` in which s_1 is closed for the first time. Let p' denote the content of the stack before this earlier call. We can easily see that $p \neq p'$. There are two possible cases with p' as follows.

First, if there is no common state in p' and $\langle t_1, \ldots, t_{n-1} \rangle$, then $p'.\langle t_1, \ldots, t_n \rangle$ must be acyclic. Inside the earlier call to `explore_acyclic_path(`s_1`)`, the transition t_1 would be pushed to the stack since $p'.\langle t_1 \rangle$ is acyclic. Similarly, inside the earlier call to `explore_acyclic_path(`s_1`)` there was a nested call to `explore_acyclic_path(`s_n`)` before which $p'.\langle t_1, \ldots, t_{n-1} \rangle$ was in the stack. Inside this call, t_n would be pushed to the stack and the cycle $\langle t_1, \ldots, t_n \rangle$ would be constructed.

Second, if there are common states in p' and $\langle t_1, \ldots, t_{n-1} \rangle$. Let the source state s_i of some transition t_i be the first common state in p'. Let p'' be the segment of p' before s_i. The earlier call to `explore_acyclic_path(`s_1`)` must be made inside a call to `explore_acyclic_path(`s_i`)`. Moreover, following Lemma 13.5, s_i was not closed since s_1 was not closed and s_i occurred earlier than s_1 in p'. Since $p''.\langle t_i, \ldots, t_n, t_1, \ldots, t_{i-2} \rangle$ is acyclic, the cycle $\langle t_1, \ldots, t_n \rangle$ would be detected inside a call to `explore_acyclic_path(`s_{i-1}`)` where s_{i-1} is the destination state of t_{i-2}, based on the same argument as for the first case. □

Lemma 13.7. *In the modified algorithm in Corollary 13.4, suppose that a cycle* $\langle t_1, \ldots, t_n \rangle$ *is found when the content of the stack is* $p.\langle t_1, \ldots, t_{n-1} \rangle$*. Let* s_i *be*

the source state of each t_i where $1 \leq i \leq n$. If s_1 is not closed, then the cycle $\langle t_1, \ldots, t_n \rangle$ must have not been discovered by the algorithm before.

Proof. It suffices to prove that once a cycle is detected, it will not be detected again before all states in the cycle become closed.

The first detection of the cycle $\langle t_1, \ldots, t_{n-1} \rangle$ must occur inside a call to `explore_acyclic_path(`s_i`)` for some state s_i in the cycle. Moreover, before the call, the content of the stack `current_path` is $p'.\langle t_{i+1}, \ldots, t_n, t_1, \ldots, t_{i-1} \rangle$ for some path p'. After the cycle is detected, the transition t_i will not be checked again before the call to `explore_acyclic_path(`s_i`)` is completed. By the time the call is completed, s_i is closed. Afterward, the transition t_{i-1} will be popped out of the stack. The content of the stack becomes $p'.\langle t_{i+1}, \ldots, t_n, t_1, \ldots, t_{i-2} \rangle$, and we are inside a call to `explore_acyclic_path(`s_{i-1}`)`. Since t_{i-1} was already checked before, it will not be checked again before the call is completed. The similar argument then apply to s_{i-1} and all other states in the cycle. □

The soundness of the algorithm follows the above lemmas.

13.2.1 Cycle Enumeration in Hierarchical State Machines

In Section 8.2.2 we introduced hierarchical CFSMs for the abstraction of hierarchical state machines in UML RT models. While it greatly eases the code abstraction procedure, it also poses the problem as how to explore a hierarchical structure for collecting cycles.

Let us recall the structure of a hierarchical state machine. A state machine may have non-composite states and composite states. A composite state contains an inner state machine in which a non-composite initial state exists. Any transition leaving or entering a composite state is a group transition. A group transition leaving a composite state s may be taken at any inner state of s at any nested level. When a group transition entering a composite state s is taken, it depends on the history of the previous execution which non-composite state will be reached by the transition. If it is the first time to enter s, then the control will re-transits to the initial state of s. Otherwise, the control is passed to the last active non-composite sub-state inside s.

Examining the algorithm in Listing 13.1 reveals that the stack `current_path` contains the acyclic path having been explored so far. This acyclic path can offer useful history information as which states were encountered along this path. Therefore, we can use this information to decide which inner state should be reached after a group transition is taken. A simple adaption of the algorithm to treat hierarchical state machines is given in Listing 13.2 and Listing 13.3. In the modified algorithm, the procedure `explore_acyclic_path` only applies to non-composite states. Moreover, we introduce a new stack `history` containing all the non-composite states reached in the currently explored acyclic path. At Line 15, while extending the currently explored path, we consider not only the outgoing transitions of the non-composite state `s_1` but also the outgoing group transitions of all ancestors of `s_1`. At Line 16, when a transition is taken, we determine if its destination state `s_2` is a composite state or not. If `s_2` is composite, then we re-locate the destination state to the last active non-composite offspring state of `s_2` by checking the stack `history` in a top-down way. If no such non-composite offspring exists, then it is re-located to the initial state of `s_2`.

```
1  proc collect_cycles(state_machine sm)
2    vector[integer] lub = <0,...,0>
3    vector[integer] eff = <0,...,0>
4    set[state] visited = {}
5    set[state] closed = {}
6    stack[transition] current_path = <>
7    stack[state] history = <>
8    set[cycle] cycles = {}
9    explore_acyclc_path(sm.initial_state)
10
11 proc explore_acyclic_path(non-composite-state s_1)
12   push s_1 to history
13   add s_1 to visited
14
15   for each tr in outgoing_transitions(ancestors(s_1))
16     state s_2 = re-locate(tr.destination_state)
17     if (exists tr_p in current_path
18           such that tr_p.source_state = s_2)
19       if (s_2 not in closed)
20         cycle c = construct_cycle(current_path, tr_p)
21         add c to cycles
22     else
23       push tr to current_path
24       eff = eff + tr.effect_vector
25       update(lub,eff)
26       explore_acyclic_path(s_2)
27       eff = eff - tr.effect_vector
28       pop current_path
29
30   add s_1 to closed
31   pop history
32
33 proc update(vector[integer] lub, vector[integer] eff)
34   for each i-th component of lub
35     if (lub[i] < eff[i])
36       lub[i] = eff[i]
```

Listing 13.2: An algorithm to collect all elementary cycles from a hierarchical CFSM state machine.

```
1  func ancestors(non-composite-state s) as set[state]
2    set[state] ancestors = {s}
3
4    while (s has a parent)
5      add parent(s) to ancestors
6
7    return ancestors
8
9  func re-locate(state s_1) as non-composite-state
10   if (s_1 is non-composite)
11     return s_1
12   else
13     int i = length(history) - 1
14     boolean found = false
15     while (i >= 0)
16       state s_2 = history[i]
17       if (s_2 is non-composite and s_2 is an offspring of s_1)
18          found = true
19          break
20       i--
21     if (found)
22       return s2
23     else
24       return s_1.initial_state
```

Listing 13.3: An algorithm to collect all elementary cycles from a hierarchical CFSM state machine.

Example 13.1. Consider the HCFSM process in Figure 13.2. In the following we list all acyclic paths during the exploration of which a cycle is constructed by our algorithm. Acyclic paths are given in the order in which they are explored during the depth first search.

$$\begin{aligned}
\langle t_1, t_2, t_3 \rangle &\text{ leading to the discovery of } \langle t_2, t_3 \rangle \\
\langle t_1, t_2, t_4, t_3 \rangle &\text{ leading to the discovery of } \langle t_2, t_4, t_3 \rangle \\
\langle t_1, t_2, t_4, t_5, t_3 \rangle &\text{ leading to the discovery of } \langle t_2, t_4, t_5, t_3 \rangle \\
\langle t_1, t_2, t_4, t_5, t_6 \rangle &\text{ leading to the discovery of } \langle t_5, t_6 \rangle \\
\langle t_1, t_2, t_4, t_7, t_3 \rangle &\text{ leading to the discovery of } \langle t_2, t_4, t_7, t_3 \rangle \\
\langle t_1, t_2, t_4, t_7, t_8 \rangle &\text{ leading to the discovery of } \langle t_7, t_8 \rangle
\end{aligned} \tag{13.1}$$

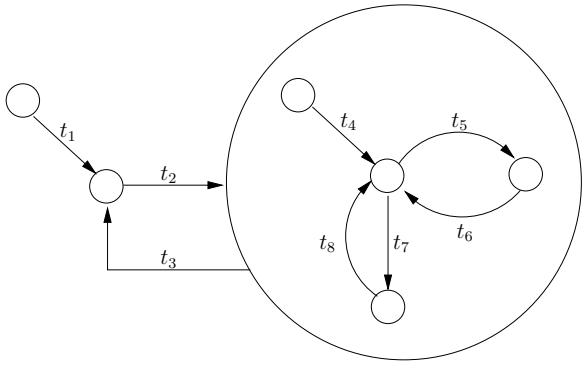

Figure 13.2: The HCFSM process of the UML RT capsule in Figure 8.1.

The soundness and completeness can be proved for the modified cycle collection algorithm similarly as for the original algorithm in Listing 13.1.

Our treatment of hierarchical CFSMs is different from other hierarchical structure exploration methods such as [16] mainly because of the use of histories in UML RT state machines. A hierarchical state machine in [16] does not have the concept of history, and a group transition entering a composite state always reaches the initial state inside the composite state. Furthermore, every composite state has special exit states in its inner state machine. Any transition leaving the composite state must leave at one of the exit states. Therefore, during the exploration of the state machine, the internal structure of a composite state needs only to be traversed once. However, in our algorithm the information of histories must be preserved to determine which sub-state of a composite state should be visited while a group transition is taken. Finally, the ideas used in our algorithm to deal with histories and group transitions can be also used in the development of model checking algorithms for UML RT state machines.

13.3 Reduction of Effect Vector Sets

Both the IBOC and aLive tools first transform a system into an independent cycle system in which each cycle carries an integer vector as its effect vector. An

ILP problem is then constructed from these integer vectors for the checking of the respective property. Therefore, by reducing the number of integer vectors, we can construct a smaller ILP problem whose checking can be done faster.

One way to reduce the number of integer vectors is to reduce the size of the independent cycle system constructed for an original system. This can be achieved by improving the abstraction techniques that we use in both tests, which is rather difficult. Consider a Promela statement `ch?msg` in a model where `msg` is an mtype variable. To safely over-approximate the effect of this statement, we build as many transitions in the corresponding CFSM process as the number of values that `msg` can take at runtime. Each transition assumes a distinct value to be taken by `msg`. If the considered statement is contained in a control flow cycle in the Promela process, then the cycle corresponds to several cycles in the CFSM process. The set of potential runtime values V cannot be precisely determined in general. Usually we over-approximate V by a superset $V' \supseteq V$. If V' is coarser, then we will have more cycles in the independent cycle system. By employing data flow analysis techniques [117, 43], we may compute a more precise approximation of V and thus reduce the number of cycles. However, data flow analyses are expensive, especially when taking global variables into consideration. Then, the additional overhead needed for using data flow analyses may counteract the gain that we obtain from a reduced number of cycles, and sometimes make the analysis even more inefficient.

Another possible way to reduce the number of integer vectors is the following. Given a set of integer vectors V as the effect vectors of an independent cycle system, V spans an integer vector space (or called a lattice) by finite additions and scalar multiplications. The basis of the lattice is the smallest set of integer vectors such that all vectors in the vector space can be constructed as a linear combination of vectors in the basis. Clearly, the basis is a non-empty subset of V. For example, if $V = \{\langle 1, 2\rangle, \langle 2, 4\rangle\}$, then the basis of the vector space spanned from V is $\{\langle 1, 2\rangle\}$. The vector $\langle 2, 4\rangle$ is removed from the basis because it is equivalent to $2 \times \langle 1, 2\rangle$. Once we obtain the vector space basis for V, we can construct a smaller property checking ILP problem from the basis. However, computing bases for integer vector spaces is NP-complete [116]. On the contrary, the property checking ILP problems constructed by our tests are always homogeneous and can be solved in polynomial time. Therefore, computing vector space bases can only result in polynomial gain. Furthermore, the application of vector space basis reduction technique can also complicate the construction of counterexamples and the abstraction refinement procedure. Finally, we observed in practice that the ILP problems constructed for realistic models always have a size manageable by the linear optimization tools and can be solved efficiently. Considering all the above factors, we decided not to implement any effect vector set reduction method in IBOC and aLive.

13.4 An Improvement to Buffer Bound Estimation

In this section we propose a refinement strategy for computing buffer bounds, which was implemented in the IBOC tool. For estimating buffer bounds, we need to compute the least upper bound on the local acyclic path effects of each process

in the CFSM system. The sum of all the obtained local bounds is then used to approximate the message passing effects of the acyclic part of the execution of the system. Consider the CFSM system as shown in Figure 13.3. Let us assume that the buffer B contains the type of messages corresponding to the first component of effect vectors. The acyclic paths of p_1 are $r_1 = \langle\rangle$, $r_2 = \langle t_1 \rangle$, and $r_3 = \langle t_1, t_2 \rangle$. Their effect vectors are $\mathit{eff}(r_1) = (0, 0)$, $\mathit{eff}(r_2) = (1, -1)$, and $\mathit{eff}(r_2) = (3, 0)$. So, the least upper bound of acyclic path effects of p_1 is $(3, 0)$. The least upper bound of acyclic path effects of p_2 is $(0, 1)$. The sum of these two upper bounds is $(3, 1)$. The ILP problem for estimating bounds for B is the following:

$$max : 3 + 3x_1 \tag{13.2}$$
$$3 + 3x_1 \geq 0 \tag{13.3}$$
$$1 - x_1 \geq 0 \tag{13.4}$$
$$x_1 \geq 0 \tag{13.5}$$

It is easy to see that the computed bound for B is 6 when $x_1 = 1$. The actual bound for B is 3.

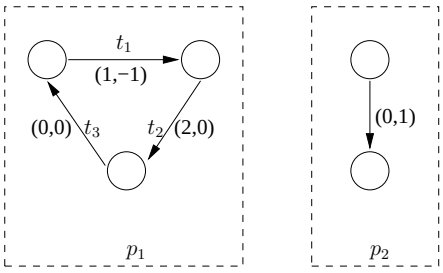

Figure 13.3: A simple CFSM system.

The coarseness of the above computed bound for B is partly due to the fact that we abstract from dependencies between acyclic paths and control flow cycles. Besides this source of imprecision, we show that if we disregard the acyclic path r_3 in p_1 then we can refine the buffer bound for B. By ruling out r_3, the least upper bound on acyclic path effects of p_1 becomes $(1, 0)$. Consequently, the computed bound for B becomes 4 instead of 6. We can safely disregard r_3 because the effect of r_3 is the same as the effect of the cycle $c = \langle t_1, t_2, t_3 \rangle$ due to the fact that t_3 has an all-zero effect. Assume that an execution of the whole system can be decomposed into an acyclic part and a cyclic part such that the acyclic part contains the acyclic path r_3. Let r be the acyclic path of p_2 contained in the acyclic part and E be the linear combination of cycles in the cyclic part. Then, the execution that we consider has a summery effect $\mathit{eff}(r_3) + \mathit{eff}(r) + E = \mathit{eff}(r) + E + \mathit{eff}(c)$. Note that $E + \mathit{eff}(c)$ can be still regarded as a linear combination of cycles. This justifies that we can disregard the effect of r_3 since it can be combined into the linear combination of cycles in the cyclic part.

We generalize the above strategy as follows. Consider any acyclic path $\langle t_1, \ldots, t_i, \ldots, t_j \rangle$ satisfying the following conditions: (1) there exists a transition t_{j+1} such that $\mathit{eff}(t_{j+1})$ is an all-zero vector and $\langle t_1, \ldots, t_{j+1} \rangle$ is a cycle;

and (2) t_i is the last transition with a non-zero effect vector. Then, we can disregard all effect vectors $\mathit{eff}(t_k)$ where $i \leq k \leq j$. In the cycle collecting algorithm in Listing 13.1, this strategy can be easily achieved by delaying the update of the least upper bound `lub` until the next transition with a non-zero effect is reached.

Chapter 14

Case Studies

We report experimental results on realistic Promela and UML RT models with the prototype tools IBOC and aLive. All experiments were performed on an Intel Core2 6400 2.13GHz machine with 2GB memory.

Due to intellectual property concerns real-life UML RT models are notoriously difficult to get hold of. On the contrary, a large number of realistic Promela models are available in the public domain [9, 3, 2].

14.1 Buffer Boundedness

14.1.1 PBX

A Private Branching eXchange (PBX) is a telephony switching system for private telephone networks inside organizations. We obtained a UML RT model of a wireless PBX system for mobile phone networks, which IBM/Rational Canada constructed for presenting the Rational Rose RealTime tool. We briefly describe the PBX model, explain the tactics of manual code abstraction that we conducted before IBOC was used to check its buffer boundedness, and finally report the experimental result.

Overview of the PBX Model

The PBX model consists of four subsystems as shown in Figure 14.1. Each subsystem is represented as a capsule in the model. Each capsule is hierarchically decomposed further into sub-capsules, as indicated by the small symbol at the right lower corner of each capsule. The subsystem `OAMSystem` maintains a record of which phone number is assigned to which mobile handset. The subsystem `CallController` establishes and tears down call connections. The subsystem `DeviceManager` allows mobile phones to dynamically join and leave the network. The subsystem `WirelessProxyManager` manages a set of phone proxies for mobile handsets.

Let us take a closer look into the above mentioned subsystems. As an example, we show the state machine of the subsystem `CallController` in Figure 14.2. When a new call is requested, it checks whether the number of active calls reaches the maximum allowed. If not, the request is accepted. Otherwise, any request can only be accepted after some currently active call has terminated.

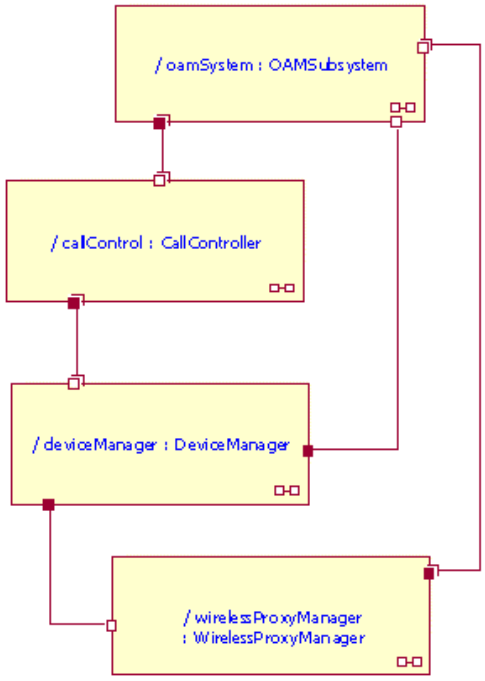

Figure 14.1: The structure of the PBX model.

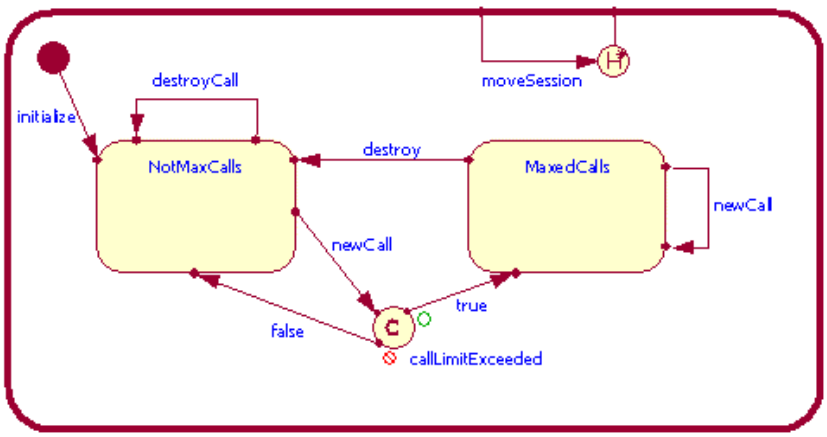

Figure 14.2: The state machine of the capsule `CallController` in the PBX model in Figure 14.1.

The PBX model contains 29 different classes of capsules, some of which are abstract classes from which other classes inherit. There are 10 capsule classes that may have multiple instances created at runtime. The maximal number of instances that a capsule class may have is specified in the model as adjustable constants. During the boundedness checking of the model, IBOC generated 27 CFSM processes that have altogether 808 local states and 335 transitions. IBOC identified 75 message buffers and 293 types of messages. With this size the complexity of the PBX model is comparable to medium-size industrial models. However, as we can easily see, the model possesses a huge global state space whose size is not directly manageable by any state-of-the-art model checker.

Manual Code Abstraction

IBOC is currently incapable of performing automated abstraction of transition code. It expects transition code to consist of a sequence of message sending statements. To make the PBX model analyzeable, we resort to manual code abstraction, during which we apply the following abstraction strategies.

Variables in message sending statements. In a message sending statement both the sending port and the sent message can be variables. Generally their values are computed at runtime and statically unknown. Consider as an example the statement `RTOutSignal(outport, msgsignal,...).send()` in the model which sends the message `msgsignal` through the port `outport`, where both `outport` and `msgsignal` are variables. We inspected the model and found out that the variable `outport` is only initialized once with the concrete port `guiInterface` and never changed again at runtime. There are 20 outgoing message types allowed by the protocol of this port. The runtime value of `msgsignal` depends on the contents of incoming messages from other capsules. Therefore, it is hard or impossible to determine its runtime values statically. In this case, we assumed the easy-to-compute and conservative approximation that a message of any of these 20 types can be sent by this statement.

Procedures. Procedures and functions are defined in the PBX model through `operations` that can be called while executing the action code of transitions. If an operation contains a message sending statement, then the statement will be executed when the operation is called during the execution of some transition action code. We checked the code inside all operations in the model. Since there is no recursive operation and no mutual invocation is present, we did nothing more than an inlining of operations.

Conditionals. If there is any message sending statement in a branch of an `if-then` statement, then it is generally not decidable statically whether and when the message sending statement is executable. We over-approximate it as usual by nondeterminism, i.e., any of the two branches of the `if-then` statement can be taken at the same time.

Capsule and port replications. Figure 14.3 shows two arrays of capsule instances `devices` and `phoneProxy` connected with each other. In Rational Rose RealTime, an array of capsule instances can be defined using replicated

capsule references. The elements of a capsule instance array may share the same ports such as the port `phoneInterface~` of the capsules `devices`. They may also possess their own copies of ports through the use of replicated ports such as `callControl~` of the capsules `devices`. A replicated port is in fact an array of ports under the same name. Each element of a replicated port is referenced using an index. There are various ways of connecting replicated ports. As an example, for two connected replicated ports, any element of a replicated port can be connected to any element of the other port. Alternatively, every i-th element of a port can be connected only with the i-th element of the other port. As a safe over-approximation, we consider each replicated port as a single port. In this way, all instances of `devices` behave identically and we need therefore to construct only one CFSM process for all instances of `devices`.

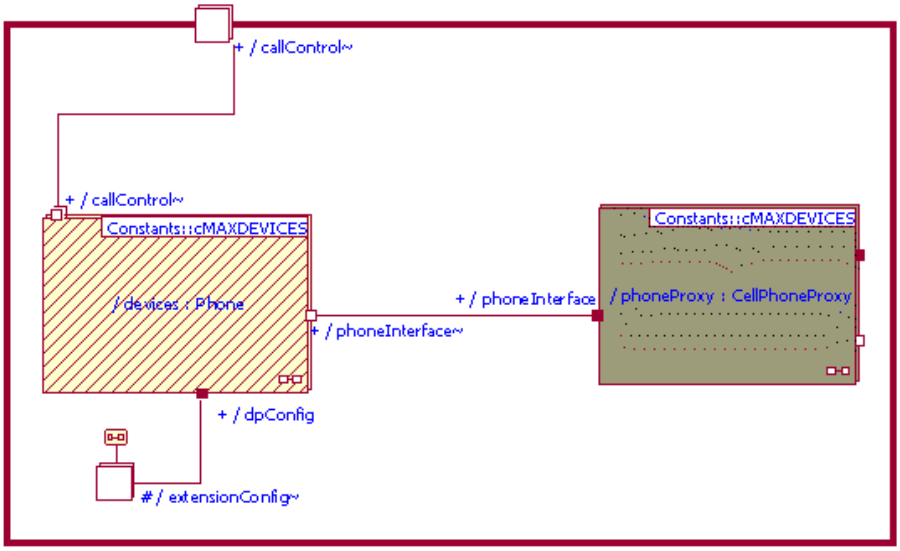

Figure 14.3: Two replicated capsule roles with replicated and non-replicated ports in the PBX model in Figure 14.1.

Loops. There is no *while*-loop or *until*-loop in the PBX model. Only a *for*-loop is used where a message sending statement occurs. This loop is used to send a user's dialed phone number to the call controller subsystem. The number of its iteration times depends on an integer variable whose runtime value is the length of the dialed phone number and thus statically unknown. Apparently, setting any upper bound on this variable, no matter how large it is, is not safe. In our experiment we set a reasonably large upper bound on the variable, e.g., 100 because a user would hardly dial a phone number containing more than 100 digits. Then our experimental result only holds when the runtime value of the variable never exceeds the bound that we have chosen.

Timeout. We also removed all timer-relevant details, mainly the use of the timeout mechanism, from the PBX model. A timer is set to mask message dropping. A timeout causes retransmission of a message. This may flood mes-

sage buffers when, for instance, all parts of the PBX model halt except a cycle is forever repeated where a timer is set, a timeout is issued and a message is retransmitted. After removing the timer and the timeout event, this cycle alone would cause the PBX model unbounded. However, the scenario that we mentioned above can rarely occur in reality. As a solution, we remove not only the timer-relevant part in the cycle but also the retransmission of messages.

Experimental Results

IBOC proved buffer boundedness and estimated all buffer bounds for the PBX model after a runtime of 72 seconds, detecting 2345 cycles and solving an LP problem with 981 variables and 294 constraints. Most of the buffer bound estimates that IBOC provided are only small numbers while some are large. It is hard to assess the quality of the estimated bounds because the high complexity of the PBX model makes it very difficult to obtain the precise buffer bounds. We could only use the Rational Rose RealTime tool to simulate the PBX model, and observe the growth of buffer lengths during the simulation. In particular, we observed that a port `configureDialPlan` never stored more than 5 messages at runtime while our estimate for the port is 7.

14.1.2 MVCC

The Model View Concurrency Control protocol, or MVCC in short, is one of the underlying protocols of the *Clock* toolkit [62] for the development of groupware applications. It supports multi-user server-client communication and the synchronization of concurrent updates of information. We took a Promela implementation of the protocol that handles 2 clients [112]. There are 8 concurrently running processes of the model at runtime. IBOC visited 70 states and 83 transitions in the state machines of the processes, and constructed 46 cycles and identified 16 types of messages.

Within 5.312 seconds, IBOC found 4 counterexamples and determined the first 3 of them to be spurious. One of the three spurious counterexamples consists of a single cycle that corresponds to the code segment between from Line 2 until Line 6 in Listing 14.1. We call this cycle c. As a consequence of code abstraction, the boolean variable `waitingForLock` is abstracted away. Disregarding the condition at Line 2 and the assignment statement at Line 6, the message sending statement at Line 3 can be repeated without constraints. IBOC determined that the cycle c can be repeated without interruption at most once. IBOC then determined a set of neighbors and a set of supplementary cycles for c to obtain two cycle dependencies. One such supplementary cycle corresponds to the code from Line 10 until Line 15. Two linear inequalities were encoded to represent the detected cycle dependencies. With the inclusion of these two inequalities in the boundedness determination ILP problem, the spurious behavior in which the cycle c is exclusively repeated forever is removed.

The last counterexample that IBOC reported contains only one cycle in a `User` process (as a client) that sends a message without any constraints. It represents the environmental behavior in which a user can send an update information whenever the user wants. IBOC failed to determine its spuriousness and consequently terminated and reported "UNKNOWN". It is easy to see that this counterexample is real.

```
1  do
2    :: (canProcessInput && waitingForLock == false) ->
3       updaterToCC!requestLock,id;
4       atomic{
5         canProcessInput = false;
6         waitingForLock = true}
7
8    ... ...
9
10   :: ccToUpdater[id]?msg ->
11      if
12      :: (msg == grantLock) ->
13         userToUpdater[id]?input;
14         updaterToVC[id]!input;
15         waitingForLock = false
16      ... ...
17      fi
18 od
```

Listing 14.1: An excerpt of a Promela model of the MVCC protocol.

14.1.3 HTTPR Exchange

The Promela model *http-exchange* [102] describes how a client and a server exchange messages using the *PUSH* and *PULL* commands in the *Reliable HTTP* (HTTPR) protocol. The model consists of a client process, a server process, and a net process. It contains 348 local states, 409 local transitions, and 5 message buffers. Within 2 seconds, IBOC identified 12 message types, collected 144 cycles, proved boundedness without going through an abstraction refinement procedure, and finally delivered an estimated bound for each message buffer. As a coincidence, the estimates for all buffers are equal to 14. In order to evaluate the accuracy of the estimates, we ran several random or guided simulations using the SPIN model checker's simulation capabilities. From our observation, all buffers in the system usually contain only 1 or 2 messages during these simulation runs. The maximal buffer length that we observed is 3. Certainly, the precise buffer bounds cannot be derived this way. However, we conjecture that the actual bounds are small numbers. This reflects the coarseness of the current buffer bound computation method, which needs to be improved in future work. One potential solution is to borrow the idea of counterexample-guided abstraction refinement. The estimated bound for a buffer is the optimal value of the objective function of an ILP problem, and corresponds to an optimal solution in the solution space of the ILP problem. This solution indicates how many times a certain cycle is repeated in a system execution that maximizes the occupancy of the considered buffer. We can then determine numerical cycle dependencies for those cycles whose corresponding variable has a non-zero value in the solution. If any cycle dependency is not respected by the solution then we can encode the dependency into linear equalities to refine the buffer bound estimation ILP problem.

14.2 Livelock Freedom

14.2.1 GARP

The *Group Address Registration Protocol* (GARP) is a network protocol allowing users to dynamically register to and detach themselves from a multicast group. A progress action is either for a user to join or leave a multicast group, or for the system to remove all the users from a group. The Promela implementation of GARP [94] consists of 7 concurrent processes with 131 local states, 212 local transitions, and 10 communication buffers. SPIN proved livelock freedom for the model within 16.8 seconds and visited 5×10^6 global states during the check. aLive used only 2.14 seconds to return the same result after 7 abstraction refinement steps. We contend that the almost eight fold speedup compared to SPIN is possible because, as opposed to SPIN, aLive does not require the exploration of a comparably large state space. During the verification aLive identified 29 message types and collected 86 control flow cycles. aLive reported 7 counterexamples and automatically determined all of these to be spurious. One of these spurious counterexamples suggests the following scenario: While no other process moves on, one process keeps executing a cycle in which it sends a *join* message to inform a service process of some user's decision to join a multicast group. However, after the message is sent, the user is included in the group, and the process cannot send another *join* message. This cycle can be repeated only if another cycle of the same process has been executed in which the process receives a message through which the user announces that he is leaving the group. aLive successfully detected this dependency between these two cycles and refined the abstraction accordingly.

#processes	7	11	19
#message types	29	50	90
#cycles	86	152	280
#reported counterexamples	7	13	25
runtime (sec)	2.14	4.11	11.3

Table 14.1: The statistical experimental results of the application of aLive on three versions of the GARP model with different numbers of processes.

In order to show the scalability of aLive, we increased the number of actually running processes in the model by allowing more users and more multicast groups. Note that, as discussed before, our verification methods may construct only one CFSM process for all identical instances of a certain proctype. As a consequence, the number of CFSM processes in the abstraction may remain unchanged even if more instances of a proctype are created in the Promela model. However, we intentionally create one CFSM process for each actually running process of the Promela model to check the scalability of the tool. The statistical results are shown in Table 14.1. The version with 11 processes is obtained by doubling both the number of user processes and the number of multicast groups in the original model. The 19-process version further doubles the numbers of users and groups. The number of identified message types, the number of collected cycles, and the number of reported counterexamples are clearly increased linearly with the increased number of processes. However, the runtime required for determining livelock freedom has a faster-than-linear

growing trend which we suspect to be exponential, or at least cubical, for the following two reasons: First, when the number of message types and the number of cycles are increased, the sizes of generated ILP problems are quadratically increased. Moreover, the number of ILP problems that aLive has to solve is also linearly increased with the number of discovered counterexamples. Therefore, the runtime of aLive is expected to increase at least cubically. Second, as we have mentioned, aLive uses `lp_solve` for the solving of ILP problems, which does not take advantage of the homogeneity of the ILP problems constructed by aLive. It still uses a combinatorial solving method that has an exponential complexity. Nevertheless, the SPIN model checker ran out of memory already on the 11-process version.

14.2.2 Mobile Handover

We also checked livelock freedom for a Promela model of the *Handover* procedure in the GSM protocol. The model is included in the SPIN distribution. In this case a progress action is to hand over the control of communication from one base region to another one. The model allows messages to contain communication buffers so that a communication channel can be dynamically established between two processes by passing around message buffers. However, such buffer sending behavior would spoil our analysis since aLive had to merge all buffers into a single one as a safe over-approximation. Therefore, we carefully revised the original model to remove the buffer passing behavior while still preserving the behavior of the original model.

The revised model consists of 6 processes with 49 local states, 62 local transitions, and 7 communication buffers. For the revised model, SPIN immediately reported an error trail with a length of 36 steps. aLive also found one counterexample in the first checking iteration and returned `unknown` after it failed to determine spuriousness for the counterexample. The counterexample consists of 6 control flow cycles and indicates the situation in which a base station is continuously forwarding messages between a mobile user and the system without handing over the control to another base region. Guided by this counterexample, we replayed the indicated scenario by a manual simulation of the *original* model within exactly 36 steps. Thus, the counterexample that aLive found is a real counterexample.

14.2.3 CORBA GIOP

The *General Inter-ORB Protocol* (GIOP) protocol supports message exchange and server object migration between Object Request Brokers (ORBs) in the CORBA architecture [10]. [74] provides several versions of Promela models of the GIOP protocol, among which we took a simplified version that does not support object migration. The model that we chose contains two users, two servers, one GIOP client, and two GIOP agents. The progress action defined in the model is that one of the two users gets a reply from the server. In the original model, any user sends only one request and will terminate after receiving a reply. Therefore, the concept of livelock freedom that the designers of the model intended to check is the following: the model is free of livelock if one of the users gets a reply. Since we are interested in a different notion of livelock freedom, i.e., the users should get replies infinitely often, we adapt the

model to allow any user to repeat the procedure of sending a request and waiting for a reply. Before the adaption, SPIN proved livelock freedom for the model in short time. However, after the adaption, SPIN always ran out of memory on our machine and was unable to complete the verification.

There are 4 proctypes defined in the model and 7 processes created at runtime. aLive identified 9 message buffers and 25 types of messages. During the checking, it visited 150 local states and 217 local transitions.

aLive found 9 counterexamples during the analysis. The first 7 of the counterexample were determined to be spurious by the detection of local cycle dependencies resulted from locally determined conditions. The eighth counterexample was determined to be spurious by the discovery of a global cycle dependency caused by a message determined condition. aLive failed to determine spuriousness for the last counterexample that it found. The last counterexample is spurious and introduced into the abstraction because we lose the information of dependencies between acyclic paths and control flow cycles.

Chapter 15

Conclusion

Concerning the checking of high-level properties for asynchronous reactive systems, this thesis describes an efficient verification framework based on ILP solving, and its applications to the checking of two important concrete properties: buffer boundedness and livelock freedom. Since these two properties are undecidable for infinite state systems, the verification methods that we proposed are inevitably incomplete. Both verification methods make use of a common abstraction strategy that takes advantage of the two main characteristics of the class of systems that we consider: asynchronous message passing and reactivity. As a result, we concentrate on the local cyclic message passing behavior of each component in a system and avoid considering the interleaving of different components. Interactions between components are abstracted by the combinational message passing effects produced by control flow cycles in all system components, as asynchronous message passing is the only or the main way of inter-component communication. This abstraction strategy results in efficient checking of properties. However, imprecision arises as we lose important information such as the dependencies between the executions of control flow cycles. The imprecision would affect our analysis when spurious property violating behavior is introduced. As a solution, we designed a counterexample-guided refinement procedure that relies on the re-discovery of cycle dependencies.

We have applied our verification framework to the analysis of Promela and UML RT models. The analysis of Promela models has been mostly automated thanks to the code abstraction and static analysis techniques that we have developed. The automated code abstraction for UML RT is much more difficult since UML RT models use high-level programming language to specify transition action code. In this thesis we addressed one difficulty in the abstraction of program code as the termination proofs of program loops.

For future work, we suggest the following possible directions to pursue. First, the usability of our framework can be greatly enhanced when the precision of the verification methods can be improved. While not hurting the efficiency of the methods, the key to improving precision is a more accurate counterexample analysis and refinement procedure. This can be achieved either (1) if we can detect more kinds of cycle dependencies and also capture dependencies more precisely, or (2) we can deal with other types of information lost during the abstraction, e.g., the dependency between the acyclic part and the cyclic part of an execution.

Second, both the buffer boundedness test and the livelock freedom test are incomplete. In case they fail to prove the considered property, they return an "UNKNOWN" result. In this case, we have a counterexample whose spuriousness cannot be determined by the refinement procedure. This counterexample may be either real or spurious. If the counterexample is real, then the property is violated by the system in question. However, the current framework is not able to use this information to debug the system. Remember that our counterexample analysis cannot determine a counterexample to be real. A complementary solution is to construct a property violating execution of the original system using the information provided by a counterexample when the spuriousness of the counterexample cannot be determined. We may use the idea of directed model checking [48]: We derive heuristics from a counterexample and use this heuristics to guide the search in the state space of the original system to find an erroneous execution. Since counterexamples are in the form of a set of cycles, useful heuristics can be the shortest distance from a state to any state contained in the cycles of the counterexample such that during searching we always tend to move closer to the cycles in the counterexample.

Third, the automated support for the verification of UML RT models is yet to be developed. To achieve this goal it largely depends on the development of an automated code abstraction procedure, for which we may borrow the ideas of existing static program analysis and abstraction techniques. One such technique that we can take advantage of is slicing [114] which removes the program code irrelevant to the checking of the considered property in order to reduce the size of the code that we must analyze. Other techniques that we can make use of include value analysis [43] and constant propagation [117] that can be used to determine more precisely the set of potential values that a variable can attain at runtime. These techniques mainly apply to sequential programs. We must extend them to effectively handle concurrent systems. We will also extend our region-graph based termination proving methods to handle a more general class of program loops, and to achieve the estimation of loop iteration counts.

Fourth, we assume the most general scheduling mechanism in our analysis, i.e., we allow for any possible interleaving of component executions. However, some interleavings can rarely happen in reality. Therefore, we can safely exclude such interleavings from our abstraction. Furthermore, a particular implementation of a system usually employs some special scheduling policy. While a property is not satisfied by the system in general, it may still hold if only the allowed scheduling mechanism is considered. Therefore, we may integrate the scheduling mechanism into our abstraction, e.g., by encoding it into linear constraints, and thereby exclude unrealistic executions.

Last, we attempt to generalize our verification framework and methods for the checking of more properties. The abstraction strategy focusing on cyclic message passing behavior that we use in the thesis becomes incapable if we consider reachability properties such as deadlock freedom. We may use the idea of the state equation technique [38] in our setting combined with cycle analysis for the treatment of more safety and liveness properties.

Bibliography

[1] ACL2. http://www.cs.utexas.edu/users/moore/acl2/index.html.

[2] BEEM: Bemchmarks for explicit model checkers. http://anna.fi.muni.cz/models/index.html.

[3] Database of Promela models. http://www.albertolluch.com/index.html?x=promelamodels.html.

[4] Java message service. http://java.sun.com/products/jms/.

[5] lp_solve. http://tech.groups.yahoo.com/group/lp_solve/.

[6] Microsoft .NET framework developer center. http://msdn2.microsoft.com/en-us/netframework/default.aspx.

[7] PVS specification and verification system. http://pvs.csl.sri.com/.

[8] Rational Rose technical developer. http://www.ibm.com/software/awdtools/developer/technical/.

[9] The SPIN website. http://spinroot.com.

[10] Common object request broker architecture: Core specification. Object Management Group, 2004.

[11] Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 146–159. Springer, 1999.

[12] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[13] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[14] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[15] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings of 26th International Colloquium on Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer, 1999.

[16] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):273–303, 2001.

[17] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172, 2002.

[18] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer*, 5(1):49–58, 2003.

[19] William R. Bevier. Towards an operational semantics of PROMELA in ACL2. In *Proceedings of 3rd SPIN Workshop*, 1997.

[20] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega-regular model checking. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 561–575. Springer, 2004.

[21] Ahmed Bouajjani and Richard Mayr. Model checking lossy vector addition systems. volume 1563 of *Lecture Notes in Computer Science*, pages 323–333. Springer, 1999.

[22] Robert S. Boyer and J. Strother Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985.

[23] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of 16th International Conference on Concurrency Theory*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502. Springer, 2005.

[24] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Radhia Cousot, editor, *Proceedings of 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 113–129. Springer, 2005.

[25] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. Technical Report RZ 1053, IBM Zurich Research Lab, 1981.

[26] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[27] Franz-Josef Brandenburg. On the intersection of stacks and queues. *Theoretical Computer Science*, 58:69–80, 1988.

[28] Luca Breveglieri, Alessandra Cherubini, and Stefano Crespi-Reghizzi. Real-time scheduling by queue automata. In Jan Vytopil, editor, *Proceedings of Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 1992.

[29] Bettina Buth, Jan Peleska, and Hui Shi. Combining methods for the livelock analysis of a fault-tolerant system. In Armando Martin Haeberer, editor, *Proceedings of 7th International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 1998.

[30] Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. In Friedrich Ludwig Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specifications*, NATO Advanced Science Institutes Series, pages 143–202. Springer-Verlag, 1991.

[31] Jingde Cheng and Kazuo Ushijima. Analyzing Ada tasking deadlocks and livelocks using extended Petri nets. In Dimitris Christodoulakis, editor, *Ada: The Choice for '92, Proceedings of Ada-Europe International Conference*, volume 499 of *Lecture Notes in Computer Science*, pages 125–146. Springer, 1991.

[32] Edmund M. Clarke. SAT-based counterexample guided abstraction refinement in model checking. In Franz Baader, editor, *Proceedings of 19th International Conference on Automated Deduction Miami Beach*, volume 2741 of *Lecture Notes in Computer Science*, page 1. Springer, 2003.

[33] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[34] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.

[35] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *Proceedings of 12th International Symposium on Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.

[36] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In Thomas Ball and Robert B. Jones, editors, *Proceedings of 18th International Conference on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.

[37] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM Press, 1971.

[38] James C. Corbett. *Automated Formal Analysis Methods for Concurrent and Real-Time Software.* PhD thesis, 1992. Available as Technical Report 92-48, Department of Computer Science, University of Massachusetts at Amherst.

[39] James C. Corbett and George S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, 1995.

[40] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* Clifford Stein, Columbia University, 2 edition, 2002.

[41] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Radhia Cousot, editor, *Proceedings of 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2005.

[42] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[43] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of Fifth Annual ACM Symposium on Principles of Programming Languages 1978*, pages 84–96, 1978.

[44] Werner Damm and Bengt Jonsson. Eliminating queues from RT UML model representations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Proceedings of 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems.*, volume 2469 of *Lecture Notes in Computer Science*, pages 375–394. Springer, 2002.

[45] George B. Dantzig. Maximization of linear function of variables subject to linear inequalities. In Tjalling C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347, 1951.

[46] María del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. A generalized semantics of PROMELA for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, 2004.

[47] Yifei Dong, Xiaoqun Du, Gerard J. Holzmann, and Scott A. Smolka. Fighting livelock in the GNU i-protocol: a case study in explicit-state model checking. *International Journal on Software Tools for Technology Transfer*, 4(4):505–528, 2003.

[48] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2-3):247–267, 2004.

[49] Javier Esparza and Stephan Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design*, 16(2):159–189, 2000.

[50] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets - a survey. *Elektronische Informationsverarbeitung und Kybernetik*, 30(3):143–160, 1994.

[51] Alain Finkel. A new class of analysable CFSMs with unbounded FIFO channels. In Sudhir Aggrawal and Krishan K. Sabnani, editors, *Proceedings of the IFIP WG6.1: 8th International Conference on Protocol Specification, Testing and Verification*. North-Holland, 1988.

[52] Alain Finkel and Annie Choquet. Simulation of linear FIFO nets by Petri nets having a structured set of terminal markings. In *Proceedings of the 8th International Conference on Applications and Theory of Petri Nets*, 1987.

[53] Alain Finkel and Louis E. Rosier. A survey on the decidability questions for classes of FIFO nets. In Grzegorz Rozenberg, editor, *Selected papers of 8th European Workshop on Applications and Theory of Petri Nets*, volume 340 of *Lecture Notes in Computer Science*, pages 106–132. Springer, 1987.

[54] Alain Finkel and Grégoire Sutre. Decidability of reachability problems for classes of two counters automata. In Horst Reichel and Sophie Tison, editors, *Proceedings of 17th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1770 of *Lecture Notes in Computer Science*, pages 346–357. Springer, 2000.

[55] Alain Finkel and Guy Vidal-Naquet. *Structuration des Systemes de Transitions - Applications au Controle du Parallelisme par Files FIFO.* These Science, Paris 11, 1986. NewsletterInfo: 30.

[56] Formal Systems (Europe) Ltd. *Failures-divergences refinement: FDR 2 user manual*, 6 edition, June 2005.

[57] Marc Geilen and Twan Basten. Requirements on the execution of Kahn process networks. In Pierpaolo Degano, editor, *Proceedings of 12th European Symposium on Programming Languages and Systems*, volume 2618 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2003.

[58] Ralph E. Gomory. Solving linear programming problems in integers. In R. Bellman and M. Hall, editors, *Combinatorial analysis, Proceedings of Symposia in Applied Mathematics*, volume 10, pages 211–215. American Mathematical Society, 1960.

[59] Mohamed G. Gouda and Chung-Kuo Chang. Proving liveness for networks of communicating finite state machines. *ACM Transactions on Programming Languages and Systems*, 8(1):154–182, 1986.

[60] Mohamed G. Gouda, Eitan M. Gurari, Ten-Hwang Lai, and Louis E. Rosier. On deadlock detection in systems of communicating finite state machines. *Computers and Artificial Intelligence*, 6(3):209–228, 1987.

[61] Mohamed G. Gouda and Louis E. Rosier. On deciding progress for a class of communication protocols. In *Proceedings of the 18th Annual Conference on Information Sciences and Systems*, pages 663–667, 1984.

[62] T. C. Nicholas Graham, Tore Urnes, and Roy Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *ACM Symposium on User Interface Software and Technology*, pages 1–10, 1996.

[63] Eitan M. Gurari and Oscar H. Ibarra. The complexity of decision problems for finite-turn multicounter machines. *Journal of Computer and System Sciences*, 22(2):220–229, 1981.

[64] Henri Hansen, Wojciech Penczek, and Antti Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. *Electronic Notes in Theoretical Computer Science*, 66(2), 2002.

[65] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[66] Jr. Hartley Rogers. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, USA, 1987.

[67] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. Technical Report UCAM-CL-TR-633, Computer Laboratory, University of Cambridge, 2005.

[68] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[69] Jan Huus and Hasan Ural. Language-based analysis of communicating finite state machines. In *Proceedings of first International Conference on Network Protocols*, pages 384–393. IEEE, 1993.

[70] Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1):116–133, 1978.

[71] Oscar H. Ibarra, Jianwen Su, Zhe Dang, Tevfik Bultan, and Richard A. Kemmerer. Counter machines and verification problems. *Theoretical Computer Science*, 289(1):165–189, 2002.

[72] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[73] Thierry Jéron and Claude Jard. Testing for unboundedness of FIFO channels. *Theoretical Computer Science*, 113(1):93–117, 1993.

[74] Moataz Kamel and Stefan Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.

[75] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[76] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[77] Leonid Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[78] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[79] Leslie Lamport. What good is temporal logic? In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, pages 657–668, 1983.

[80] A.H. Land and A.G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[81] Stefan Leue, Alin Ştefănescu, and Wei Wei. A livelock freedom analysis for infinite state asynchronous reactive systems. In Christel Baier and Holger Hermanns, editors, *Proceedings of 17th International Conference on Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2006.

[82] Stefan Leue, Richard Mayr, and Wei Wei. A scalable incomplete test for message buffer overflow in Promela models. In Susanne Graf and Laurent Mounier, editors, *Proceedings of 11th International SPIN Workshop on Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 2004.

[83] Stefan Leue, Richard Mayr, and Wei Wei. A scalable incomplete test for the boundedness of UML RT models. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2004.

[84] Stefan Leue and Wei Wei. Counterexample-based refinement for a boundedness test for CFSM languages. In Patrice Godefroid, editor, *Proceedings of 12th International SPIN Workshop on Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 58–74. Springer, 2005.

[85] Stefan Leue and Wei Wei. A region graph based approach to termination proofs. In Holger Hermanns and Jens Palsberg, editors, *Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2006.

[86] Richard J. Lipton. The reachability problem requires exponential space. Technical Report 62, Department of Computer Science, Yale University, 1976.

[87] Cong Liu, Alex Kondratyev, Yosinori Watanabe, Alberto L. Sangiovanni-Vincentelli, and Jorg Desel. Schedulability analysis of Petri nets based on structural properties. pages 69–78. IEEE Computer Society, 2006.

[88] Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, 1984.

[89] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[90] Olivier Maréchal, Pascal Poizat, and Jean-Claude Royer. Checking asynchronously communicating components using symbolic transition systems. In Robert Meersman and Zahir Tari, editors, *Proceedings of OTM Confederated International Conferences On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE, Part II*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer, 2004.

[91] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[92] Gérard Memmi and Gérard Roucairol. Linear algebra in net theory. volume 84 of *Lecture Notes in Computer Science*, pages 213–223. Springer, 1980.

[93] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[94] Tadashi Nakatani. Verification of group address registration protocol using PROMELA and SPIN. In *Proceedings of 3rd SPIN Workshop*, 1997. Available at `http://spinroot.com/spin/Workshops/ws97/nakatani.pdf`.

[95] Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[96] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1995.

[97] Doron Peled. Ten years of partial order reduction. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer-Verlag, 1998.

[98] Wuxu Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(3):399–442, 1991.

[99] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

[100] Pascal Poizat, Jean-Claude Royer, and Gwen Salaün. Bounded analysis and decomposition for behavioural descriptions of components. In *Proceedings of 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2006.

[101] ITU-T Recommendation. ITU-T Recommendation Z.100: Specification and description language (SDL). ITU Telecommunication Standardization Sector, 2002.

[102] Paolo Romano, Milton Romero, Bruno Ciciani, and Francesco Quaglia. Validiation of the sessionless mode of the HTTPR protocol. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *Proceedings of 23rd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, volume 2767 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2003.

[103] A. William Roscoe. Model-checking CSP. pages 353–378, 1994.

[104] Louis E. Rosier and Hsu-Chun Yen. Boundedness, empty channel detection, and synchronization for communicating finite automata. *Theoretical Computer Science*, 44:69–105, 1986.

[105] Alexandeer Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[106] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling.* John Wiley & Sons, 1994.

[107] Bran Selic and James Rumbaugh. Using UML for modeling complex real-time systems. `http://www.ibm.com/developerworks/rational/library/139.html`, March 1998.

[108] Stephen F. Siegel and George S. Avrunin. Improving the precision of inca by eliminating solutions with spurious cycles. *IEEE Transactions on Software Engineering*, 28(2):115–128, 2002.

[109] A. Prasad Sistla. On characterization of safety and liveness properties in temporal logic. In *Proceedings of the Fourth Annual ACM Symposium on Princiles of Distributed Computing*, pages 39–48, 1985.

[110] ISO Standards. ISO 9074: Information technology – open systems interconnection – Estelle: A formal description technique based on an extended state transition model. International Organization for Standardization, 1997.

[111] Jayme L. Szwarcfiter and Peter E. Lauer. A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics*, 16(2):192–204, 1976.

[112] Maurice H. ter Beek, Mieke Massink, Diego Latella, Stefania Gnesi, Alessandro Forghieri, and Maurizio Sebastianis. A case study on the automated verification of groupware protocols. In Gruia-Catalin Roman,

William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of 27th International Conference on Software Engineering*, pages 596–603. ACM, 2005.

[113] James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, 1970.

[114] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[115] Ashish Tiwari. Termination of linear programs. In Rajeev Alur and Doron Peled, editors, *Proceedings of 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 70–82. Springer, 2004.

[116] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report report 81-04, Department of Mathematics and Computer Science, University of Amsterdam, 1981.

[117] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[118] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In David A. Watt, editor, *Proceedings of 9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2000.

[119] Hsu-Chun Yen. A unified approach for deciding the existence of certain Petri net paths. *Information and Computation*, 96(1):119–137, 1992.

Index