INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

Université d'Ottawa · University of Ottawa

Improving state exploration techniques for the automatic verification of concurrent systems

Johannes J.M. van der Schoot

A thesis submitted to
the School of Graduate Studies and Research of the University of Ottawa
in partial fulfillment of the requirements for the degree of Ph.D. in Computer Science

School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario

Thesis Supervisor:

Dr. Hasan Ural, University of Ottawa

Thesis Examiners:

Dr. Luigi Logrippo, University of Ottawa

Dr. Gregor von Bochmann, University of Ottawa

Dr. Danny Krizanc, Carleton University

Dr. Mohamed Gouda, University of Texas at Austin

Abstract

The research underlying this thesis concerns the verification of concurrent systems. In particular, strategies are studied to tackle the state explosion problem which arises during the verification of concurrent systems by state space exploration. State (space) exploration, commonly known also as reachability analysis, amounts to exploring in a systematic manner the entire state space of a system, i.e. all states and transitions of the system that can be reached from a given initial state. It is a simple and fully automatic technique that can be employed in principle to verify many different types of correctness requirements of concurrent systems, ranging from various general consistency requirements to more system-specific, functional requirements. However, the state explosion problem severely limits the applicability of reachability analysis in practice. The state spaces of most realistic industrial-strength systems are excessive in size (comprising hundreds of thousands or even millions of states and transitions), and thereby surpass any conceivable amount of memory and time available for analysis.

One of the main causes of the state explosion problem in the verification of concurrent systems is the modeling of concurrency by interleaving or, more accurately, the exploration of all possible interleavings of concurrent events/transitions. During reachability analysis, each possible way in which the execution of concurrent transitions can be ordered in time is examined. Yet, it has been recognized for quite some time that many interesting correctness properties of concurrent systems are insensitive to the interleaving order of concurrent transitions. Consequently, such systems often manifest a large number of reachable states and transitions that are redundant for verification purposes. For nearly two decades, this apprehension has inspired researchers to devise improved state exploration techniques that relieve the state explosion problem. These state exploration based relief strategies reduce the complexity of conventional reachability analysis by examining just part of the state space of a system, a part that is provably sufficient to verify one or several desirable properties. In effect, they enable the verification of concurrent systems without incurring (most of) the cost of modeling concurrency by interleaving.

Several of the strategies proposed to relieve the state explosion problem in the verification of concurrent systems have proved promising indeed, enabling in many cases substantial memory and time savings for state exploration. With the advent of these relief strategies, the applicability of verification by state exploration has thus certainly been widened to "larger" concurrent systems. Nevertheless, since concurrent systems are inherently complex and since this complexity is there to stay, pursuing further performance improvements in verification clearly remains of utmost importance. This thesis contributes by proposing improvements of three existing relief strategies, namely fair reachability analysis, simultaneous reachability analysis and partial-order reduction methods. These are among the most evolved state exploration based relief strategies to date. First, the technique of fair reachability analysis (FRA) is generalized from cyclic protocols to multi-cyclic protocols modeled as networks of communicating finite state machines (CFSMs). A multi-cyclic protocol constitutes one or more cyclic protocols, i.e. unidirectional rings, which are interconnected such that no two component cyclic protocols share more than one CFSM. FRA is established as an effective and efficient relief strategy for the detection of deadlocks in multi-cyclic protocols with a finite fair reachable state space (i.e. the reduced state space of the protocol explored by FRA). Furthermore, it is also shown that FRA is infeasible as a relief strategy beyond the class of multi-cyclic protocols.

Secondly, a technique called leaping reachability analysis (LRA) is presented as an incremental improvement of simultaneous reachability analysis (SRA), which was proposed as a relief strategy for the verification of logical correctness properties of protocols specified in the CFSM model without topological or structural constraints. For any protocol in this model, LRA is proven to maintain the power of SRA to detect all deadlocks, all non-executable transitions, all unspecified receptions and all buffer overflows of the protocol, while enabling further reductions in the size of the state space that needs to be analyzed. These analytical results are complemented by an empirical evaluation of the performance of LRA and SRA, which reveals that LRA can indeed yield important extra savings in space and time over SRA.

Lastly, an enhancement of partial-order reduction methods is proposed for linear-time temporal logic (LTL) model-checking. More precisely, the concepts underlying LRA are integrated with those underlying partial-order reduction methods to enable further savings in both space and time for the verification of linear-time temporal properties of general, finite-state concurrent systems (i.e. any system whose behavior can be defined as a finite transition system). This approach is subsequently fine-tuned for protocols specified in the CFSM model, whereby LRA emerges as an effective and uniform relief strategy for the verification of both logical (i.e. syntactic) and functional (i.e. semantic) correctness properties of protocols. Empirical results are provided which attest that the proposed approach to LTL model-checking is indeed a notable enhancement of the partial-order reduction approach.

Acknowledgements

A number of people have contributed to this thesis, either directly or indirectly, and I am pleased to thank them at this point. First of all, I am very grateful to my thesis advisor Dr. Hasan Ural for his instrumental guidance, support and encouragement throughout my doctoral studies. I also wish to express my appreciation for his friendship, fostered in particular by our many fruitful discussions over a cup of his excellent coffee. Although his role as my thesis advisor has come to an end, the friendship will remain.

I would like to thank the members of my advisory committee, Dr. Luigi Logrippo and Dr. Danny Krizanc, for their constructive criticism on the progress of my thesis research.

Many thanks also to Tuong Nguyen for implementing the various algorithms proposed in this thesis into the research tool package RELIEF (v3.5), and for conducting the experiments.

I gratefully acknowledge financial support by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Ontario Graduate Scholarship (OGS) program, the "fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR)", and the School of Graduate Studies and Research (SGSR) of the University of Ottawa.

Last but certainly not least, I wish to thank my wife Patrice for her love, for sharing the ups and downs, and for her faith in me. My deepest love and respect to her for her tremendous patience and constant moral support, which were invaluable in completing this thesis.

Contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Background

Sophisticated computer and information systems have become of the essence in today's society, and they are being deployed at an ever increasing rate. Most contemporary computing systems are typically composed of entities that operate concurrently and cooperate through communication. Examples of such *concurrent systems* are computer and communication networks and protocols, operating systems, asynchronous circuits and many other embedded systems with application areas like process control, telephony and air traffic control to just name a few.

The correct design of *concurrent systems* is known to be a problem of considerable depth. One major source of difficulties lies in the fact that the functionality of these systems tends to be very large and complex. Traditionally, a sequential system (or program) is transformational and can be thought of as a function: given an input, it may produce an output. The specification of all input-output pairs defines the precise meaning of the system. A concurrent system is yet hard to describe in this way as it operates within an environment over an indefinite period of time. Its functional behavior is defined by the many ongoing interactions with its environment, and these interactions often exhibit complex interdependencies. For this reason, it is difficult to adequately specify, understand and predict the behavior of concurrent systems and, hence, to assess whether they meet their requirements.

Another source of difficulties lies in the distributed nature of concurrent systems. As the constituent entities of a concurrent system are dispersed over different locations, they must also interact with each other in order to realize the functionality of the system as a whole. An important example is found in *communication protocols*, where protocol entities interact according to strict rules. Indeed, the mere purpose of a communication protocol is to govern the orderly exchange of messages among communicating entities. Both the design of communication protocols and the

assessment of their correctness are certainly delicate tasks, and rigorous automated analysis methods are required to support these tasks.

The work described in this thesis pertains to the (design) *verification* of concurrent systems, and of communication protocols in particular. Verification refers thereby to the act of *proving* (or disproving) formally that a system design meets its expected properties, which can range from several types of general consistency requirements to more specific functional requirements asserted in, for instance, a logical language. What is strictly not meant is testing (unless it is exhaustive), or any other method which may indicate that a system design is "probably" correct. In order to prove that a system satisfies some property, *all possible executions* of the system must be checked to determine whether each and every one of them complies to the property. As such, verification is thus the means to guarantee the correctness of the design of a concurrent system or communication protocol.

1.2 Scope, objective, contributions

Throughout the past twenty years or so, various formal models have been proposed and studied to facilitate the specification and validation of (designs of) concurrent systems. These models differ in their expressiveness in terms of specification and in their tractability in terms of validation (i.e. verification and testing). However, most of the models have in common that their individual semantics renders a translation of the pure syntactic description of a concurrent system into some kind of a transition system, consisting of a set of states, a designated initial state, and a (labeled) transition relation among these states. This transition system represents the behavior of the concurrent system as a whole, i.e. the joint behavior of all the concurrent entities in the system.

A model particularly suited for specifying communication protocols is the *communicating finite state machine* (CFSM) model [Boc78, ZW+80, BZ81, BZ83]. In the CFSM model, a protocol is specified as a collection of processes (i.e. the protocol entities) that exchange messages over error-free simplex channels. Each process is modeled as a finite state machine (FSM) and each simplex channel is a FIFO queue. A (global) state of the protocol consists of a state for each FSM and a content for each simplex channel. A state transition can occur only when some process is ready to either send a message to one of its output channels, or receive a message from one of its input channels. The CFSM model is well-defined, elegant and rather easy to understand. These features make it attractive for both academia and industry. Indeed, the CFSM model has become a widely established means for specifying, verifying and testing communication protocols. Furthermore, it underlies two standardized specification languages, namely Estelle [BD89] and SDL [BH88]. For these reasons, and because there is no "best" formal model, we have chosen to study primarily the

verification of protocols specified in the CFSM model, although we will extend our scope later in the thesis to the verification of any (concurrent) system that can be viewed as a (labeled) transition system. In particular, state transitions may then represent system events other than just message transmissions and receptions, and system entities may communicate not only by asynchronous message passing but also by synchronous "handshaking".

One of the most prevalent techniques for the verification of protocols, and concurrent systems in general, is *state space exploration*. State (space) exploration, which is widely known also as *reachability analysis*, amounts to exploring in a systematic manner the complete state space of a system, i.e. all states and transitions of the system that can be reached from a given initial state. Many different types of system properties can be verified by reachability analysis. It was originally proposed for verifying so-called *logical correctness properties* of protocols specified in the CFSM model, namely freedom of deadlocks, non-executable transitions (cf. dead code in a computer program), unspecified receptions, and buffer overflows or unbounded channel growth [Wes78, WZ78, ZW+80, BZ83]. These are general correctness properties that concern concurrent systems at large, albeit that unspecified receptions and buffer overflows or unbounded channel growth are particular to models featuring some form of asynchronous message passing. Reachability analysis can further be employed for the verification of individual, *functional correctness properties* of concurrent systems and protocols, like temporal safety and liveness properties [Lam80, Lam83, AS87, MP92]. This has emanated in the past decade from the development of model-checking methods for various temporal logics [LP85, CES86, VW86].

Reachability analysis is a simple and easy-to-automate verification technique. Moreover, it is fully automatic and thus no user-intervention is required. The effectiveness of reachability analysis has been witnessed by various notable success stories about its application to complex, industrial-size systems (see e.g. [Rud92]): it revealed several subtle design errors that had previously been missed by ad hoc approaches. However, a main limiting factor of reachability analysis is the swift explosion of the state spaces to be analyzed. Simple combinatorics affirm that the size of the state space of a system can be exponential in the size of the description of the system. This phenomenon is well-known as the *state explosion problem*. It severely hampers the practical usefulness of reachability analysis to industrial-strength applications. Indeed, the state spaces of most realistic systems are excessive in size (hundreds of thousands, or even millions of states) and thereby surpass any conceivable amount of memory available for analysis. Certain systems have in fact infinite state spaces, which causes most of their properties to be undecidable altogether (see e.g. [BZ83]).

Fortunately, the state explosion problem is not entirely inherent. It has long been recognized that many concurrent systems manifest a large number of reachable states and transitions that are redundant for verification purposes. One of the leading causes of this redundancy is the modeling

of concurrency by interleaving or, more accurately, the exploration of all possible interleavings of concurrent events. For instance, the execution of k concurrent events is examined by exploring all $k!$ possible orderings of these events. Many interesting properties of concurrent systems are yet insensitive to the interleaving order of concurrent events. Consequently, for nearly two decades, researchers have put much effort into the development of improved state exploration techniques in order to relieve the state explosion problem [LCL87, Yua88]. Such state exploration based *relief strategies* reduce the complexity of reachability analysis by examining just part of the state space of a system, a part that is provably sufficient to verify certain properties. Practically, they enable the verification of properties of systems while avoiding most of the cost of modeling concurrency by interleaving.

Despite the various formal models around, virtually all state exploration based relief strategies proposed in the literature adhere in fact to the CFSM model, or a slight variation thereof [RW82, YG82, II83, GY84, GH85, GCL85, KI+85, GC86, ZB86, CR93, LM94, ÖU94, ÖU95, LM96]. The earlier strategies are rather limited in their applicability, as they were devised for protocols with just two communicating processes [RW82, YG82, GY84, GH85, GCL85, GC86, ZB86, CR93], or with otherwise strong conditions on the structural attributes of the individual processes [YG82, II83, KI+85]. It was only recently that several researchers innovated ideas to handle protocols with more complex communication structures. Liu & Miller [LM94, LM96] contributed by generalizing the technique of *fair reachability analysis* (FRA) proposed for two-process protocols in [RW82, GH85] to n-process *cyclic* protocols, where $n \geq 2$ processes form a unidirectional ring. FRA proves to be a very powerful relief strategy for the detection of deadlocks in cyclic protocols [LM94a], and it also provides a good basis for the efficient detection of non-executable transitions, unspecified receptions and unbounded channel growth [LM94b]. Özdemir & Ural [ÖU94, ÖU95] went further by proposing a relief strategy, called *simultaneous reachability analysis* (SRA), that is applicable to n-process protocols without topological or structural constraints. SRA can significantly reduce the number of stored states and explored transitions for the detection of deadlocks, non-executable transitions, unspecified receptions and buffer overflows in protocols with arbitrary communication structures. Yet another relief strategy developed in recent years is partial-order state exploration, which actually captures a collection of cognate algorithms better known as *partial-order reduction methods* [God90, Val90, HGP92, KP92a, Val92, Val93, GW93, GW94, HP95, God96, Pel96]. Unlike FRA and SRA, which are intended explicitly for protocols specified in the CFSM model, these methods are largely independent of the model used for specifying concurrent systems. They apply in principle to all specification models whose semantics induce (labeled) transition systems [HP95, God96]. Partial-order reduction methods are also effective as a relief strategy for verifying deadlock-freedom and freedom of non-executable transitions. Moreover, they lend themselves as

an efficient means for LTL model-checking, i.e. for model-checking system properties asserted in linear-time temporal logic (LTL) [Pnu77, Lam80].

The use of FRA, SRA or partial-order reduction methods does in many cases yield substantial savings in the memory and time requirements for state exploration [LM96, ÖU95, HP95, God96, Pel96]. Hence, with the advent of these techniques, the applicability of state exploration based verification has certainly been widened to "larger" concurrent systems and protocols. Nevertheless, since concurrent systems are inherently complex, and since this complexity is there to stay, pursuing additional performance improvements in verification clearly remains of utmost importance. It is this awareness that has motivated us to investigate the possibility of further increasing the effectiveness and efficiency of existing state exploration based relief strategies. Respecting the current state of the art, we have focused in particular on improving FRA, SRA and partial-order methods. While FRA appears to be a very powerful relief strategy indeed for the verification of cyclic protocols, the unidirectional ring topology of these protocols is still very restricted. The natural question is whether the effectiveness of FRA can be extended to protocols in the CFSM model with more complex, or even arbitrary communication topologies. Both SRA and partial-order reduction methods already enjoy such generality, but possible improvements of these two relief strategies may still be found in their performance. The critical issue thereby is the characteristic trade-off in computing between space and time. Although space is the major concern in verification, due to the state explosion problem, extreme care must be taken that potential extra savings in space are not attended by unacceptable expenses in time.

As a thesis, this manuscript presents in detail the results obtained with respect to the above research objective. Our main contributions can be summarized as follows:

- We generalize FRA from cyclic protocols to so-called *multi-cyclic protocols* in the CFSM model. A multi-cyclic protocol consists of a collection of unidirectional rings, or component cyclic protocols, which are interconnected such that no two rings share more than one process. The communication topology underlying multi-cyclic protocols has a rather wide applicability in practical protocol modeling. It captures not only protocols with a multi-ring topology (in particular all cyclic protocols), but also protocols with other common network topologies like a daisy-chain, a star, and a tree, as well as many combinations of these elementary topologies. We establish that FRA is an effective and efficient relief strategy for the detection of deadlocks of multi-cyclic protocols with a finite fair reachable state space (i.e. the reduced state space of the protocol explored by FRA), which follows the same result obtained for cyclic protocols. Furthermore, we also advocate that FRA is infeasible as a relief strategy beyond the class of multi-cyclic protocols. Albeit a negative result, recognizing the fundamental limitations of a technique is certainly of benefit as well.

- We present a relief strategy called *leaping reachability analysis* (LRA), which we propose as an incremental improvement of SRA for verifying logical correctness properties of protocols defined in the CFSM model. We prove that, for any protocol in this model, LRA maintains the power of SRA to detect all deadlocks, all non-executable transitions, all unspecified receptions and all buffer overflows of the protocol. Through an analytical comparison of the two relief strategies we show that LRA is an absolutely no-risk improvement of SRA, i.e. using LRA instead of SRA is at no cost whatsoever, neither in space nor in time. We complement the analytical results with an empirical evaluation of the performance of LRA and SRA, which in fact reveals that LRA can in many cases yield important extra savings over SRA in both space and (especially) time.
- We propose an enhancement of partial-order reduction methods for LTL model-checking. More precisely, we present an approach which integrates the concepts underlying LRA with those underlying partial-order reduction methods to enable further savings in both space and time for the verification of linear-time temporal properties of general, finite-state concurrent systems (i.e. any system whose behavior can be defined as a finite transition system). This approach is further fine-tuned for the CFSM model, by harmonizing its formulation with the formulation of LRA. LRA thereby emerges as an effective, *uniform* relief strategy for the verification of both logical and functional correctness properties of protocols defined in the CFSM model. Empirical results are provided which attest that our approach to LTL model-checking is indeed a notable enhancement of the partial-order reduction approach.

1.3 Organization of the thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces the CFSM model and explains the technique of reachability analysis on the basis of this model. The principle limitations of reachability analysis are thereby addressed, viz. undecidability and state explosion. Chapter 3 provides a extensive survey of techniques proposed in the literature to relieve the state explosion problem, with an emphasis on state exploration based relief strategies. Chapter 4 generalizes fair reachability analysis from cyclic to multi-cyclic protocols in the CFSM model, and advocates the inherent infeasibility of this technique beyond the class of multi-cyclic protocols. Chapter 5 details the technique of leaping reachability analysis as an improvement of simultaneous reachability analysis for verifying logical correctness properties of protocols specified in the CFSM model. Chapter 6 reports on the results of a corresponding empirical comparison between these two relief strategies. Chapter 7 unfolds the proposed enhancement of partial-order reduction methods for

LTL model-checking. Finally, Chapter 8 provides the concluding remarks. The contributions of this thesis are reviewed and several directions for further research are given.

Chapter 2

Protocol design and analysis: the CFSM model

In this chapter we present a simple model for the specification and verification of communication protocols: the *communicating finite state machine* (CFSM) model. The CFSM model lends itself to *reachability analysis*, a state exploration technique which has been advocated for verifying general properties of protocols such as absence of deadlocks, non-executable transitions, unspecified receptions and unbounded channel growth [ZW+80, BZ83]. We address two principal limitations of reachability analysis, viz. *undecidability* and *state explosion*.

2.1 Modeling protocols as networks of CFSMs

A communication protocol consisting of interacting processes can be modeled as a network of communicating finite state machines (CFSMs). Each CFSM is an abstraction of one of the processes in the protocol and communicates with the other CFSMs by sending and receiving messages over error-free simplex channels, represented by FIFO queues.

Notation 2.1

Given a set A, $|A|$ denotes its cardinality and A^* denotes the set of strings of elements in A, including the empty string ε. Juxtaposition is used to denote concatenation of strings. For a string $Y \in A^*$, $|Y|$ denotes its length and *front*(Y) denotes the first element of Y; *front*(Y) is undefined if $|Y| = 0$. Also, $\langle a_i \rangle_{i \in A}$ denotes an $|A|$-tuple $(a_{i_1}, a_{i_2}, \ldots, a_{i_{|A|}})$. □

Definition 2.2

Let $I = \{1, 2, \ldots, n\}$ be a finite index set with $n \geq 2$. A *protocol* Π is a pair (P, L), where

- $P = \{P_i \mid i \in I\}$ is a set of *n processes*,
- $L \subseteq I \times I$ is an irreflexive incidence relation identifying a nonempty set of error-free *simplex channels* $\{C_{ij} \mid (i, j) \in L\}$.

Each *process* $P_i \in P$ is a quadruple $(S_i, s_i^0, M_i, \Delta_i)$, where

- S_i is a finite, nonempty set of *process states*,
- $s_i^0 \in S_i$ is the *initial state* of P_i,
- $M_i = \bigcup_{j \in I}(M_{ij} \cup M_{ji})$, with M_{ij} a finite set of *messages* that P_i can send to process P_j,
- $\Delta_i = \bigcup_{j \in I} \Delta_{ij}$ is a finite set of *transitions*, with $\Delta_{ij} \subseteq S_i \times \tilde{M}_{ij} \times S_i$ and $\tilde{M}_{ij} = \{-x \mid x \in M_{ij}\} \cup \{+y \mid y \in M_{ji}\}$,

such that $\forall j, k, l \in I$: (i) $S_i \cap S_j = \emptyset$ if $i \neq j$, (ii) $M_{ij} \cap M_{kl} = \emptyset$ if $(i, j) \neq (k, l)$ and (iii) $M_{ij} = \emptyset$ if $(i, j) \notin L$.

Each error-free *simplex channel* C_{ij} $((i, j) \in L)$ is a perfect FIFO queue linking process P_i to process P_j. The *content* c_{ij} of C_{ij} is a string of messages from M_{ij}, i.e. $c_{ij} \in M_{ij}^*$. □

A protocol can be viewed as a finite directed graph in which the nodes correspond to processes and the edges correspond to simplex channels. This graph implicitly defines the *communication topology* of the protocol. Similarly, a process can be viewed as finite directed graph in which the nodes correspond to process states and the edges correspond to transitions. Each transition represents the occurrence of an *event*, being either the transmission or the reception of a message by a process. Notice that processes need not be deterministic (i.e. any two transitions of the same process can be such that they differ only in their third component).

Definition 2.3

The *topology graph* of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$, denoted by TG_Π, is the directed graph with vertex set I such that there is an edge from $i \in I$ to $j \in I$ iff $(i, j) \in L$. The *process graph* of a process $P_i = (S_i, s_i^0, M_i, \Delta_i)$ is the labeled directed graph with vertex set S_i such that there is an edge labeled μ from $s \in S_i$ to $s' \in S_i$ iff $(s, \mu, s') \in \Delta_i$. □

Definition 2.4

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol and $t = (s, \mu, s') \in \Delta_i$ a transition (at s), for some $i \in I$. t is a *send transition* iff $\mu = -x$, with $x \in \bigcup_{j \in I} M_{ij}$. t is a *receive transition* iff $\mu = +y$, with $y \in \bigcup_{j \in I} M_{ji}$. □

The states of a protocol as a whole, or *global* states, are defined as the composite of individual process (or local) states and channel contents. A global state assigns to each process a process state of this process, and to each simplex channel a sequence of messages in transit over this channel. A special global state is designated as the initial global state.

Definition 2.5

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol. A *global state* G of Π is a pair (S, C), where

- $S = \langle s_i^G \rangle_{i \in I}$ with $s_i^G \in S_i$ the process (or local) state of process P_i in G,
- $C = \langle c_{ij}^G \rangle_{(i,j) \in L}$ with $c_{ij}^G \in M_{ij}^*$ the content of simplex channel C_{ij} in G.

The *initial global state*, denoted by G^0, is the global state $(\langle s_i^{G^0} \rangle_{i \in I}, \langle c_{ij}^{G^0} \rangle_{(i,j) \in L})$ with $s_i^{G^0} = s_i^0$ and $c_{ij}^{G^0} = \varepsilon$, for all $i \in I$ and $(i, j) \in L$. □

Processes can execute transitions at global states. A receive transition defined at (a process state of) a global state can be executed when the message to be received resides at the front of the respective queue. Although a send transition could similarly be considered executable if the respective queue is not full, in the traditional CFSM model queues are not a priori bounded[1]. A send transition defined at a global state is then always executable.

Definition 2.6

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ and $t = (s, \mu, s') \in \Delta_{ij}$ a transition, for some $i, j \in I$. When $s = s_i^G$, t is said to be *defined at* G. t is *executable at* G iff t is defined at G and

- t is a send transition, or
- t is a receive transition with $\mu = +y$ and $front(c_{ji}^G) = y$.

The set of send and receive transitions from Δ_{ij} that are executable at G are denoted by $X_{ij}^-(G)$ and $X_{ij}^+(G)$, respectively, and $X_{ij}(G) = X_{ij}^-(G) \cup X_{ij}^+(G)$. □

Notation 2.7

$$X_i^-(G) = \bigcup_{j \in I} X_{ij}^-(G) \qquad X_i^+(G) = \bigcup_{j \in I} X_{ij}^+(G) \qquad X_i(G) = X_i^-(G) \cup X_i^+(G)$$

$$X^-(G) = \bigcup_{i \in I} X_i^-(G) \qquad X^+(G) = \bigcup_{i \in I} X_i^+(G) \qquad X(G) = X^-(G) \cup X^+(G)$$ □

2.2 Reachability analysis

A protocol defined in the CFSM model is a closed system: starting at the initial global state it can evolve and change its global state by executing transitions. A natural way to analyze the possible behavior of a protocol is then to consider its set of *reachable global states* and the transitions that occur between them. More specifically, the complete interaction domain of the protocol can be

[1] In [BZ83] the use of unbounded channels is justified conceptually as follows: "The queues modeling the simplex channels have unbounded capacity to represent protocols allowing an arbitrary number of messages in transit. In a physical implementation all channels must be bounded, but the bound may be too large to be of practical use. Moreover, since protocols are supposed to operate over different channels with different capacities, a channel of unbounded capacity is the proper abstraction."

examined by exploring all possible ways in which the initial global state and the subsequent global states can be perturbed. This approach is traditionally called *reachability analysis* or *perturbation analysis*, and was first recognized by West and Zafiropulo [WZ78, Wes78] as a simple and easy-to-automate technique for the verification of communication protocols.

Definition 2.8

Let G and H be global states of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$. $G \longrightarrow H$ iff $\exists t \in X_{ij}(G)$, for some $i, j \in I$, such that one of the following two conditions holds:

- $t = (s_i^G, -x, s_i^H)$ and $c_{ij}^H = c_{ij}^G x$, while all other elements of H are the same as those of G;
- $t = (s_i^G, +y, s_i^H)$ and $c_{ji}^G = y c_{ji}^H$, while all other elements of H are the same as those of G.

H is the *successor* of G *by* t, also denoted as $G \xrightarrow{t} H$. □

The relation $\longrightarrow$ is a binary relation over the global states of a protocol. Informally, $G \longrightarrow H$ represents a perturbation of a global state G resulting in another global state H via the execution of a single transition of one of the processes, while the other processes and the unaffected simplex channels remain unchanged. This notion is naturally extended to sequences of transitions.

Definition 2.9

Let G and H be global states of a protocol Π, and denote by $\longrightarrow^*$ the reflexive and transitive closure of $\longrightarrow$. H is *reachable from* G iff $G \longrightarrow^* H$. When $G = G^0$, H is said to be *reachable*. The set of all reachable global states of Π is denoted by $\mathbf{R}_\Pi$. For a sequence of transitions $\sigma = t_1 t_2 \ldots t_m$, $G \xrightarrow{\sigma}{}^* H$ denotes the existence of global states $Q^0, \ldots, Q^m$ such that $G = Q^0 \xrightarrow{t_1} Q^1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} Q^m = H$. □

Definition 2.9 formalizes the conventional reachability analysis, yielding the set of all reachable global states of a protocol and the transitions between them. This is referred to as the *reachable global state space* of the protocol, which can be viewed as a (possibly infinite) directed graph with nodes and edges corresponding to reachable global states and global state transitions, respectively.

Definition 2.10

The *reachability graph* of a protocol Π is the labeled directed graph with vertex set $\mathbf{R}_\Pi$ such that there is an edge labeled t from $G \in \mathbf{R}_\Pi$ to $H \in \mathbf{R}_\Pi$ iff $G \xrightarrow{t} H$. □

In practice, the reachable global state space of a protocol is computed by performing a systematic search of all the global states that are reachable from the initial global state. Figure 2.1 gives a standard algorithm for such a search [Hol91]. This algorithm recursively explores all successor states of all global states encountered during the search, starting from the initial global

state, by executing all executable transitions at these states. It uses a work set W of global states to be analyzed, and a set A of global states already analyzed. As pointed out in [Hol91], the order of retrieval of states from set W is consequential: a *depth-first search* is performed if W is a stack, and a *breadth-first* search is performed if W is a queue. A brief discussion on the particular benefits of each search strategy, as related to protocol verification, is deferred until Section 2.4. Clearly, the algorithm terminates only if the number of reachable global states is finite. Upon termination, set A should contain exactly the reachable global states of the protocol. It is not difficult to prove that this is indeed the case [AHU74].

```
/* A is the set of global states that have been analyzed.        */
/* W is the set of global states that still need to be analyzed. */

/* Initialize: */
A = ∅
W = {G^0}

/* Loop: */
while W ≠ ∅ do {
    remove an element G from W
    add G to A
    for all t in X(G) do {
        derive the successor H of G by t  /* execution of transition t */
        if H is NOT already in A or W then add H to W
    }
}
```

Figure 2.1 Standard perturbation algorithm.

2.3 Example: a simple network access protocol

As an example of a communication protocol, consider a system in which a client process seeks access to a network through communication with a server process [ZW+80]. This simple network access protocol can be specified in the CFSM model as follows:

- $P = \{P_1, P_2\}$, where $P_i = (S_i, s_i^0, M_i, \Delta_i)$ $(i = 1, 2)$ with
  - $S_1 = \{10, 11, 12\}$, $S_2 = \{20, 21, 22\}$;
  - $s_1^0 = 10$, $s_2^0 = 20$;
  - $M_1 = M_2 = M_{12} \cup M_{21} = \{\text{AReq, ATer}\} \cup \{\text{APer, ARej}\}$;
  - $\Delta_1 = \Delta_{12} = \{t_1^1, t_1^2, t_1^3, t_1^4\}$, $\Delta_2 = \Delta_{21} = \{t_2^1, t_2^2, t_2^3, t_2^4\}$, where

$t_1^1 = \{10, -\text{AReq}, 11\}$ $t_2^1 = \{20, +\text{AReq}, 21\}$ (Access *Req*uest)

$t_1^2 = \{11, +\text{ARej}, 10\}$ $t_2^2 = \{21, -\text{ARej}, 20\}$ (Access *Rej*ected)

$t_1^3 = \{11, +\text{APer}, 12\}$ $t_2^3 = \{21, -\text{APer}, 22\}$ (Access *Per*mitted)

$t_1^4 = \{12, -\text{ATer}, 10\}$ $t_2^4 = \{22, +\text{ATer}, 20\}$ (Access *Ter*minated)

- $L = \{\{1, 2\}, \{2, 1\}\}$ (i.e. there are two error-free simplex channels C_{12} and C_{21}).

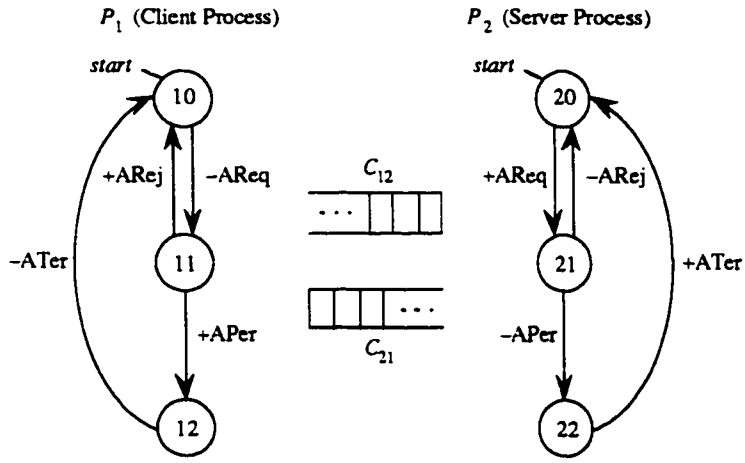


Figure 2.2 A network access protocol.

Process P_1 models the client process and process P_2 models the server process. The respective process graphs are shown in Figure 2.2. Initially, each process is at its initial state (process states 10 and 20, respectively) and both simplex channels are empty. The only event that can occur in this initial global state is the transmission of an Access Request message by P_1 (transition t_1^1). This event causes P_1 to move from state 10 to state 11 and the message "AReq" to be placed in channel C_{12}. At this point P_1 cannot progress since only receive transitions are specified at state 10 while channel C_{21} is still empty. With "AReq" in C_{12}, however, process P_2 can receive this message by executing transition t_2^1. In doing so it moves from state 20 to state 21, and channel C_{12} becomes empty again. Note that there is no assumption on the time a message spends in a channel, i.e. the delay between the transmission and the reception of a message is variable and unspecified. P_2 can now continue by putting either "ARej" or "APer" in C_{21} (transition t_2^2 or t_2^3), corresponding to rejecting or permitting network access, respectively. In either case, P_2 reaches a state at which it cannot execute any transition until P_1 places a new message in channel C_{12}. A further step-wise analysis of the joint behavior of the two processes readily yields the complete reachable global state space of the protocol, as shown by the reachability graph in Figure 2.3. It contains 8 global states

and 10 transitions. An inspection of this graph quickly reveals that the contents of both simplex channels remain finite, C_{12} holding at most two messages and C_{21} at most one message. Also, the protocol seems to manifest a "healthy" cyclic behavior since it can eventually return to its initial global state.

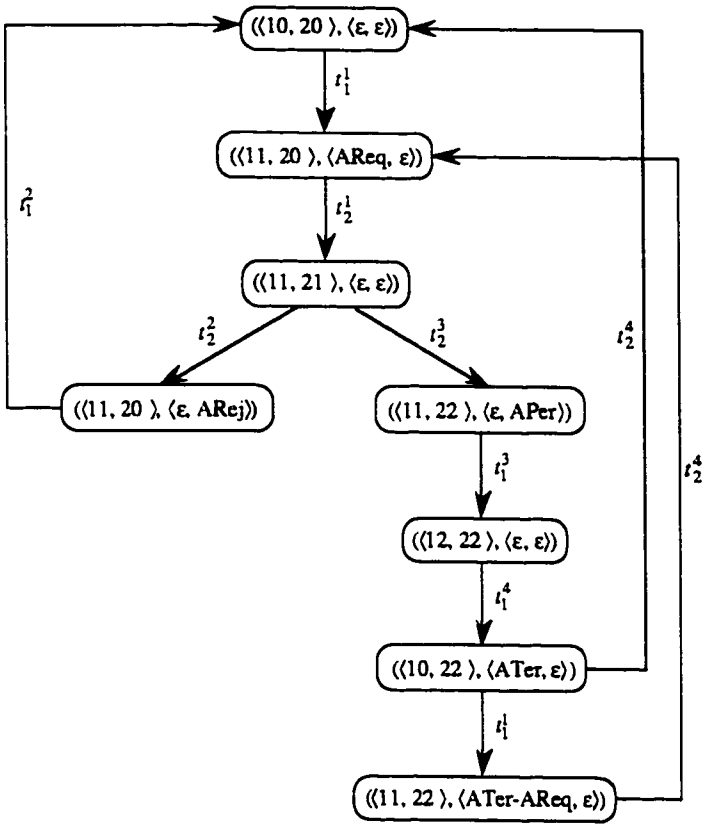


Figure 2.3 The reachability graph of the network access protocol in Figure 2.2.

2.4 Protocol verification by reachability analysis

For the network access protocol above, a simple "walkthrough" by simulating the processes provides considerable insight of the behavior of this protocol. Inspecting the reachability graph is easy because it is small. One must realize, however, that the 8 reachable global states emerge from a potentially much larger number. Let us assume that simplex channel C_{12} holds at most two messages from the set {AReq, ATer} and that simplex channel C_{21} holds at most one message from the set {ARej, APer}. With processes P_1 and P_2 each having 3 process states, there are in fact $3 \times 3 \times (2^0 + 2^1 + 2^2) \times (2^0 + 2^1) = 189$ syntactically distinct global states. Although 181 of these states play no role in the behavior of the protocol (as they are not reachable from the initial global state), this calculation demonstrates the potential exponential growth of states that may result

from reachability analysis, even when assuming a very limited number of messages in the channels at any time. As a consequence, analyzing the behavior of a protocol is generally much more complex. It requires precise definitions of error conditions or correctness conditions in order to determine the existence of design errors by examining the reachability graph constructed during reachability analysis.

The correctness conditions commonly considered for protocols specified in the CFSM model are defined in terms of erroneous transitions and (reachable) global states. It is usually desirable that a protocol be free from *non-executable transitions*, i.e. transitions that cannot be executed at any reachable global state. A non-executable transition compares to "dead code" in a computer program and indicates a design error. Presumably, the protocol designer has included the transition believing it would be used. The design needs to be reconsidered: the transition is either removed, with no effect on the behavior of the protocol, or the transition is made executable by adding or modifying other transitions.

Definition 2.11

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol and $t \in \bigcup_{i \in I} \Delta_i$ a transition. t is *non-executable* iff $\nexists G \in \mathbf{R}_\Pi$: $t \in X(G)$. □

A protocol should generally also be free from *deadlocks*. A deadlock is a reachable global state in which all channels are empty and none of the processes is ready to send a message. This is a special case of a *non-progress state*, in which the channels need not be empty but no process is able to execute a transition. A non-progress state normally testifies that the protocol has come to an "unexpected halt". The unexpected nature of halting is what indicates a design error. A deadlock state can also be seen as a special case of a *stable state*. Stable states are reachable global states with all channels empty and irrespective of the ability of processes to execute transitions. They can be useful for detecting loss of synchronization [Wes78].

Definition 2.12

Let $\Pi = (P, L)$ be a protocol. A global state $G \in \mathbf{R}_\Pi$ is a *progress state* iff $X(G) \neq \emptyset$; otherwise, G is a *non-progress state*. G is a *stable state* iff $\forall (i, j) \in L$: $c_{ij}^G = \varepsilon$. G is a *deadlock state* iff it is a non-progress state and a stable state. *Π progresses indefinitely* iff $\forall G \in \mathbf{R}_\Pi$: G is a progress state. □

Finally, a protocol should generally not exhibit any *unspecified reception states*. An unspecified reception state refers to a reachable global state in which a message at the front of some queue cannot be received due to a missing (i.e. unspecified) receive transition. The design error here is a case of "underspecification", in contrast to a non-executable transition which signifies "overspecification".

Definition 2.13

Let $\Pi = (P, L)$ be a protocol. A global state $G \in \mathbf{R}_\Pi$ is an *unspecified reception state* iff $\exists (j, i) \in L$: $front(c_{ji}^G) = y$ and $(s_i^G, +y, s) \notin \Delta_i$. The pair (s_i^G, y) is called an *unspecified reception* (ur-pair for short) for process P_i (in G). □

Note that a non-progress state is either a deadlock state or an unspecified reception state. In the latter case one also speaks of a *blocking* unspecified reception state. The absence of both deadlocks and unspecified receptions thus implies the absence of non-progress states.

The above correctness conditions have been used to verify whether the communication among the processes in a protocol progresses indefinitely [Gou84], and whether the protocol itself is complete and logically consistent [ZW+80]. This explains why they are also referred to as *progress properties* or *logical correctness properties*. A protocol is often said to be logically correct when it is free from deadlocks, unspecified receptions and non-executable transitions. Verifying the logical correctness of a protocol by reachability analysis amounts to constructing the entire reachable global state space of the protocol while checking each reachable global state against the respective correctness conditions. The verification algorithm is thus the same as the standard perturbation algorithm in Figure 2.1, apart from a few straightforward additions to report any violations of logical correctness properties. As mentioned in Section 2.2, this algorithm allows a breadth-first search as well as a depth-first search of the reachable global state space. The former has the advantage of exposing the shortest path to an "error state" first. The latter has the advantage that it does not require extra information to be stored in order to reconstruct a path to such a state: a path is implicitly defined by the content of the stack. Also, as the depth of a search tree is usually much smaller than its breadth, a depth-first search often requires considerably less memory to maintain the set W of states to be analyzed [Hol91]. Saving memory in case of a depth-first search is further possible by storing only the global states in the "current" search path. The set A of analyzed states is then not needed. Of course, this is generally at the expense of running time for it can no longer be determined whether a newly generated global state has already been encountered in a previous search path. Redundant explorations of global states are thus likely to be carried out.

Example 2.14

Consider again the network access protocol in Section 2.3. It follows from the reachability graph in Figure 2.3 that this protocol is logically correct. Indeed, there are no deadlock states since all the reachable global states have an "outgoing" transition. Furthermore, all transitions are executable since each transition appears at least once in the reachability graph. Finally, there are no unspecified reception states since all the reachable global states which still contain a message at the front of an incoming queue have an "outgoing" receive transition for that message. □

Example 2.15

Consider the protocol depicted by Figure 2.4. Figure 2.5 illustrates part of its reachability graph which consists of a total of 25 global states and 35 transitions. One can see that the protocol contains at least one deadlock state and three unspecified reception states (one being a non-progress state as well). By completing the reachability graph, one can further see that the three transitions (22, $+a$, 23), (23, $-d$, 22) and (11, $+d$, 10) are non-executable transitions. □

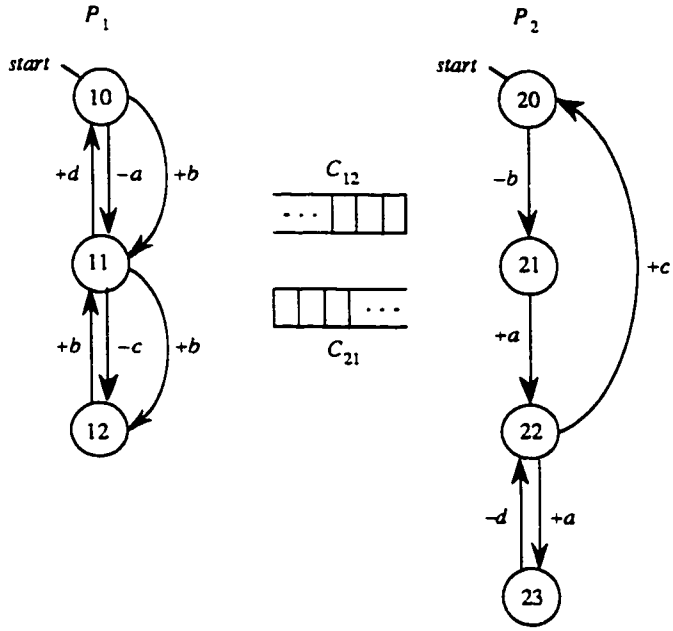


Figure 2.4 A protocol with design errors.

In principle, proving a protocol correct requires more than just verifying the absence of deadlocks, unspecified receptions and non-executable transitions. A protocol may be logically correct and yet not perform its intended functions. The traditional focus of reachability analysis is nevertheless on general properties that are of interest to all protocols independent of their intended functionality [BZ83, ZW+80], such as the ones defined. The differentiation between general correctness requirements and functional, protocol-specific requirements appears natural and has long been accepted. This is witnessed for example by the following citation, hinting at an analogy between the syntactic correctness of a program and the logical correctness of a protocol [Rud88]:

> "Just as successfully passing through a syntax-checking precompiler is no guarantee that a program written in a high-level language will perform its intended function, validation of a protocol does not guarantee that the protocol will perform its intended function."

Here, "validation" actually means "verification" for the objective is to *prove* a protocol logically

correct. Violations of logical correctness properties are thus interpreted as "syntactic" errors in the protocol, while violations of functional requirements can be seen as "semantic" errors.

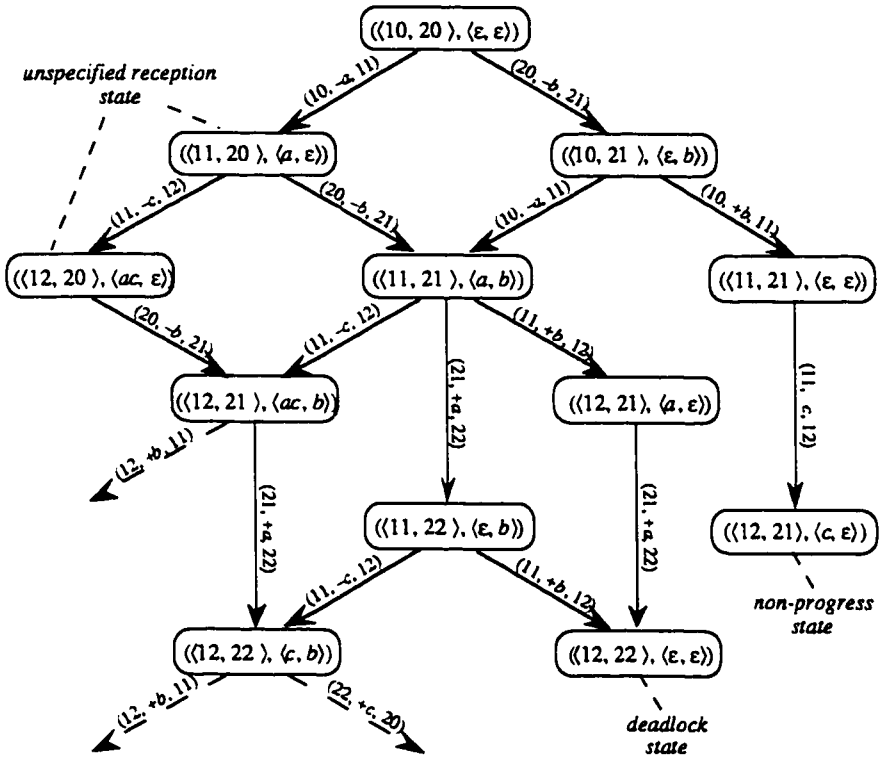


Figure 2.5 The (partial) reachability graph of the protocol in Figure 2.4.

Logical correctness properties also classify as so-called *safety properties*. Safety properties are intuitively characterized as stating that "bad things" never happen, as opposed to *liveness properties* which state that "good things" do eventually happen [Lam77]. The violation of a logical correctness property (e.g. the occurrence of a deadlock or unspecified reception) then corresponds to a "bad thing" that is irremediable and takes place fundamentally after a finite sequence of execution steps of a protocol. Liveness properties are typically used to claim that a protocol does something useful, like performing its intended functions (e.g. faithful message transfer). Most properties of this kind can be violated by infinite execution sequences only [Sis85, AS87]. We will return in detail to the analysis of general safety and liveness properties in Chapter 7.

2.5 Challenges in reachability analysis

Reachability analysis has become an established technique for protocol verification, and several notable "success stories" about applying this technique to complex, industrial-size protocols have

been reported over the past 15 years (e.g., see [Rud92]). Nevertheless, there are two fundamental problems with reachability analysis which render its use impractical, still, for most real protocols, namely *undecidability* and *state explosion.*

2.5.1 Undecidability

In [BZ81], Brand and Zafiropulo showed that most properties of interest, such as logical correctness properties, are in general undecidable for protocols specified in the CFSM model. Undecidability stems from the potential unboundedness of the channels in the model. Even a protocol consisting of two processes communicating over two channels with unbounded capacity appears as powerful as a Turing machine. In essence, the halting problem reduces to the problem of detecting (logical) design errors in a protocol by using the unbounded channels to simulate the tape of a Turing machine [BZ81].

Despite this negative result, decidability of the verification problem is known for some classes of protocols. Most eminently, reachability analysis decides the verification problem for all those protocols whose channels are bounded.

Definition 2.16
Let $\Pi = (P, L)$ be a protocol and C_{ij} a simplex channel, with $(i, j) \in L$. C_{ij} is *bounded* iff $\exists K \geq 0$ $\forall G \in \mathbf{R}_{\Pi}: |c_{ij}^{G}| \leq K$. □

Definition 2.17
A protocol Π is *bounded* iff $\mathbf{R}_{\Pi}$ is finite. □

Clearly, from these two definitions a protocol is bounded iff all its simplex channels are bounded. Although boundedness of channels is also undecidable in general, most common protocols do not make use of the full generality of the CFSM model. Brand and Zafiropulo observe [BZ83, p. 329]:

> "[...] many practical protocols do have all their channels bounded. Protocols with unbounded channels usually use them in a simple manner, which makes them worth considering."

Other classes of protocols for which the verification problem is decidable by reachability analysis similarly originate from restricting the allowable sequences of messages that can be in transit at any time. These classes are identified by, for instance, a language-like requirement on the contents of each channel [Pac87, Fin88, Oku88], or by the boundedness of specific channels in combination with a restricted communication topology [BZ83, GH85, PP90, CR93, LM94].

Lastly, for unbounded protocols one can elude undecidability to some degree by *prescribing* bounds on the channel capacities. A protocol may then not be fully analyzable but it can be verified correct by approximation [BZ83]. Precisely, by assigning to each simplex channel of a protocol an

explicit bound on the maximum number of messages it may hold, and thereby preventing the execution of send transitions involving full channels (cf. Definition 2.6), the reachability graph of the protocol is guaranteed to be finite. Channels with prescribed capacity bounds will be referred to as *prebounded* channels. For protocols with prebounded channels an additional logical correctness property is of interest, namely freedom of buffer (or channel) overflows [WZ+80].

Definition 2.18

Let $\Pi = (P, L)$ be a protocol and denote by B_{ij} the bound on a simplex channel C_{ij}. A global state $G \in \mathbf{R}_\Pi$ is a *buffer overflow state* iff $\exists (i, j) \in L$: $| c_{ij}^G | = B_{ij}$ and $(s_i^G, -x, s) \in \Delta_{ij}$, for some $x \in M_{ij}$. The pair (s_i^G, x) is called a *buffer overflow* (bo-pair for short) for process P_i in G. □

Buffer overflow states are thus reachable global states in which some process is ready to send a message onto a full channel. Detecting the underlying bo-pairs is beneficial for several reasons. A bo-pair may indicate that the prescribed bound on a channel is not sufficient and "restricts" the behavior of a protocol in an adverse way or, conversely, it may reveal that the protocol is unbounded which is likely to be a design error as well. Also, a bo-pair may indicate that a channel will overflow in practice in case of a conforming protocol implementation where processes do not know the status of their outgoing buffers. This would typically result in oblivious message loss.

2.5.2 State explosion

When a protocol is bounded it is theoretically possible to explore and analyze its entire reachable global state space. In practice, however, even the state space of a bounded protocol may be restrictively large for exhaustive search. As illustrated at the beginning of Section 2.4, simple combinatorics affirm that the number of reachable global states may grow exponentially, a phenomenon known as the *state explosion problem*. This problem comes into effect when the size of the state space surpasses the amount of memory available for the search: exhaustive state exploration becomes impractical.

Fortunately, the state explosion problem is not always unavoidable. It is well-known that many protocols manifest a large number of reachable global states and transitions that are redundant for verification purposes. Indeed, for nearly two decades much effort has been spent on devising techniques that exploit this redundancy and thereby relieve the state explosion problem. Such *relief strategies* enable the verification of properties of protocols by examining only part of their state spaces. The next chapter provides an overview of existing relief strategies, in particular of those pertaining to the CFSM model.

Chapter 3

Relief strategies to tackle state explosion

The principle objective of most improved verification techniques is to relieve the intricate problem of *state explosion*, without compromising too much the ability to verify several different types of correctness properties. This chapter gives an overview of the various *relief strategies* developed to this extent. In line with the previous chapter, we focus primarily on relief strategies that are based on state exploration. We point out that two excellent surveys of existing relief strategies already appeared in the late 80's [LCL87, Yua88]. Our overview adopts the structure of [Yua88] and includes also relief strategies that have been proposed more recently.

3.1 Improved state exploration techniques

One of the more prominent causes of the state explosion problem is the *modeling of concurrency by interleaving* or, precisely, the exploration of all possible interleavings of concurrent transitions. For instance, in reachability analysis the execution of k concurrent transitions is examined by exploring all $k!$ orderings of these transitions. Yet, many properties of interest are insensitive to the order of concurrent transitions. Protocols therefore often manifest a large number of reachable global states and transitions that are redundant for verification purposes. It is this observation which has led to the development of several improved state exploration techniques – techniques that reduce the complexity of conventional reachability analysis by examining just part of the state space of a protocol, a part that is provably sufficient to verify a given property.

3.1.1 Fair reachability analysis

Fair reachability analysis has emerged to date as an improved state exploration technique for the verification of cyclic protocols, in which two or more processes form a unidirectional ring. The technique has been proposed incrementally by several researchers [RW82, GH85, LM94, LM96].

Canonical sequences Fair reachability analysis was first proposed by Rubin & West as a nameless relief strategy for the verification of *two*-process protocols (i.e. networks of two processes communicating over two error-free simplex channels) [RW82]. They recognized that much of the redundancy in conventional perturbation analysis can be eliminated by imposing an order among transitions when both processes can execute a transition at a global state. It was shown that transitions can then always be executed pair-wise, one of each process, resulting in so-called *canonical sequences*. Instead of executing just a single transition of one process, "matching" transitions of both processes are coupled to generate the next global state. Exploring only canonical sequences can substantially reduce the number of global states and transitions analyzed, as only the states with an equal number of messages in both channels are generated. Rubin & West argued that this improvement in efficiency does not compromise the detection of deadlocks and unspecified receptions in two-process protocols.

Fair progress state exploration Gouda & Han built on the technique above, and actually introduced the name fair reachability analysis [GH85][2]. This name stems from the recognition that two processes progressing through a canonical sequence factually progress with the same (relative) speed. Put differently, a canonical sequence does not allow one of the processes to execute more transitions than the other at any given time, and is hence fair with respect to their progress speeds. Gouda & Han supported the claims in [RW82], viz. for a two-process protocol the fair reachability graph obtained by "fair progress state exploration" is usually much smaller than the corresponding reachability graph and, when finite, this graph can be used to decide the absence of deadlocks and unspecified receptions. They also learned that boundedness detection, which had not yet been addressed, is unsolvable by fair reachability analysis itself. Consequently, algorithms were presented which *augment* the fair reachability graph of a two-process protocol in order to decide whether the protocol is bounded [GH85]. The algorithms assume a *finite* fair reachability graph, which can then be employed also to find the smallest possible capacities of the two channels in the protocol. These contributions elevated fair reachability analysis as a rather effective relief strategy, suitable for detecting the most common logical errors in two-process protocols.

Generalized fair reachability analysis Recently, Liu & Miller generalized the technique of fair reachability analysis to n-process *cyclic* protocols, where $n \geq 2$ processes are joined by n simplex channels in a closed loop [LM94, LM96]. All channels are oriented in the same direction, hence forming a unidirectional ring. A two-process protocol is thus a special instance of a cyclic

[2] The name fair reachability analysis was in fact introduced earlier by Yu & Gouda [YG82], who presented an algorithm for deadlock detection which is polynomial in both time and space. However, the protocols considered are assumed to have two processes that can only send and receive one type of message, while no process state may involve both send and receive transitions. These assumptions are too restrictive in practice.

protocol. The key aspect of the generalization consists in preserving the *equal channel length property*: each global state generated by fair reachability analysis is a reachable global state in which all channels hold the same number of messages. As pointed out above, this property was already identified implicitly in [RW82, GH85]. Indeed, for two-process protocols the equal channel length property coincides with the perception of equal progress speed.

Liu & Miller showed that the decidability results established for two-process protocols are valid for all cyclic protocols in general. That is, the detection of deadlocks and unspecified receptions, and boundedness detection are decidable for any cyclic protocol whose fair reachability graph is finite. In addition, they determined the decidability of detecting non-executable transitions for these protocols, which had not been established before for two-process protocols. The fair reachability graph of a cyclic protocol was found to be finite if (at least) one of the channels in the protocol is bounded. This also follows the same result achieved earlier for two-process protocols [GCL85]. A necessary and sufficient condition for finiteness was identified as well, namely the absence of indefinite simultaneous growth of all channels in a cyclic protocol. This condition thus completely characterizes the subclass of cyclic protocols for which the detection of deadlocks, non-executable transitions, unspecified receptions and unboundedness is decidable [LM94, LM96].

Besides being an effective and efficient relief strategy for cyclic protocols, an important extra asset of fair reachability analysis is that it can solve the verification problem for various *unbounded* protocols. The conventional reachability analysis fails to deal with such protocols due to the infinity of their state spaces, induced by an unbounded accumulation of messages in (one of) the channels (cf. Section 2.5.1). Yet, the number of global states explored by fair reachability analysis may very well be finite. The technique of fair reachability analysis is presented in further detail in Chapter 4, along with one of our own contributions: a generalization of fair reachability analysis to so-called multi-cyclic protocols.

3.1.2 "Reduced" reachability analysis

For two-process protocols, two other improved state exploration techniques have appeared which are quite similar in nature to fair reachability analysis. Zhao & von Bochmann [ZB86] proposed a "reduced" reachability analysis on the basis of a different representation of the CFSM model, in which protocols are modeled by process equations. State exploration is then performed through algebraic transformation rules. In order to reduce the space and time requirements, the proposed method employs both conventional and fair progress schemes in the generation of global states. It was first shown to detect all non-progress states, i.e. all deadlocks and blocking unspecified receptions, and subsequently extended to enable the detection of all unspecified receptions [ZB86].

Cacciari & Rafiq [CR93] presented a reduced reachability analysis for two-process protocols that resembles Zhao & von Bochmann's technique. They again revert to the plain CFSM model, however, and also incorporate internal transitions in the model[3]. Cacciari & Rafiq claimed that their method improves the one in [ZB86] in the sense that it allows more properties to be verified without increasing the order of magnitude of the number of generated states. The reachability graph constructed by the proposed method spans all reachable global states in which either both channels hold the same number of messages (cf. the equal channel length property), or one of the channels is empty while the other holds one message. This reduced reachability graph thus includes all deadlock states and has further been shown appropriate for verifying the absence of unspecified receptions and certain livelocks (precisely, blocking cycles). Not all unspecified receptions are necessarily detected though, but it is guaranteed that at least one unspecified reception manifests itself in the reduced reachability graph if there exists one in the protocol.

3.1.3 Maximal progress state exploration

Like fair progress state exploration, maximal progress state exploration is a technique that attempts to eliminate redundancy in conventional reachability analysis by not examining all relative progress speeds of processes [GY84]. Its applicability is limited to two-process protocols. Rather than forcing the two processes to progress at equal speed, this technique forces one of the processes to make *maximal* progress. This means that transitions of one process, say P_1, are executed as much as possible, while the other process P_2 remains inactive. Process P_2 comes into play only when P_1 can no longer progress, i.e. when all transitions at its current process state are receive transitions that cannot be executed. In this case, state exploration continues by executing transitions of P_2 until one of the receive transitions of P_1 becomes executable. Process P_1 then resumes progress and the procedure repeats itself as long as new global states can be generated.

Gouda & Yu proved that all non-progress states are detected by performing maximal progress state exploration for either process. In addition, they proved that all buffer overflows (in case of channels with finite capacity) can be identified by performing maximal progress state exploration for both processes, once for P_1 and once for P_2. Detecting buffer overflows is thus divided into two independent subtasks, each of which generally requires less space and time than the combined task. The overall time requirements may then be reduced by executing the two subtasks in parallel, at the expense of an extra processor, whereas the space requirements may be reduced by executing the two subtasks in sequence on a single processor [GY84].

---

[3] Although considered in [CR93], internal transitions do usually not have a significant effect on state exploration based verification techniques. They are therefore largely ignored by researchers in the field.

3.1.4 Reduced implementation sequences

The relief strategy proposed by Itoh & Ichikawa [II83] (see also [KI+85]) is applicable to protocols with any number of processes (at least two, of course). Constraints are imposed, however, on the structures of these processes: all processes must synchronize on their initial process states after a finite number of execution steps and no process is allowed to have a cyclic execution that does not pass through its initial state. Although these constraints ensure that the global state space of a protocol is finite, eluding any such embedded cycle in the process graph of a process is surely restrictive in practice. Even a simple data transfer protocol usually exhibits at least one embedded cycle (e.g. the retransmission part of the sender in an alternating bit protocol).

Itoh & Ichikawa's technique entails the *simultaneous* (or parallel) execution of transitions of different processes in a global state to derive the next global state. Intuitively, the aim is to abstain as much as possible from *any* execution order among concurrent transitions. Special attention is thereby required for transitions that are not executable at the current global state, say G, but that may still become executable later at a global state reachable from G. Such transitions are called *potentially admissible events*. For each global state encountered with potentially admissible events, additional simultaneous progress schemes are considered by inhibiting the execution of transitions of the processes in which these events arise. As shown in [II83], this procedure may result in the analysis of just a small part of the global state space of a protocol. Indeed, the proposed technique examines only the so-called *reduced implementation sequences* of a protocol, which constitute a subset of all the possible protocol executions. A reduced implementation sequence is an execution from the initial global state of the protocol to either a non-progress state or a global state in which all processes have returned to their intial process states (the channels need not be empty). The set of reduced implementation sequences is used to verify the protocol against a given requirement specification, viz. a set of prescribed protocol executions. Strictly speaking, the intent of Itoh & Ichikawa is thus not to verify logical correctness properties, but rather "operational" requirements of a protocol [II83]. Regarding the former, their technique does not lend itself for detecting logical errors other than non-progress states.

3.1.5 Simultaneous reachability analysis

Simultaneous reachability analysis is a state exploration technique which generalizes the ideas behind the above work by Itoh & Ichikawa. It was proposed by Özdemir & Ural [ÖU94, ÖU95, Özd95] as a relief strategy for verifying logical correctness properties of protocols with an arbitrary number of processes, arbitrary communication topology and arbitrary process structures. That is, in contrast to the technique in [II83], restrictions no longer apply to any of the protocol attributes.

Simultaneous reachability analysis was shown to detect all non-progress states and non-executable transitions of a protocol. Furthermore, augmentations were devised to enable the detection of all unspecified receptions and all overflowed channels. An overflowed channel refers to a simplex channel for which there exists a bo-pair (cf. Definition 2.18). Note that the detection of all bo-pairs implies the detection of all overflowed channels, but not vice versa. We will return in more detail to simultaneous reachability analysis in Chapter 5, where we propose an incremental improvement of this relief strategy.

3.1.6 Partial-order reduction methods

Partial-order reduction methods [God90, Val90, HGP92, KP92a, Val92, Val93, GW93, GW94, HP95, Pel96] are a collection of cognate techniques to alleviate the state explosion problem in verifying finite-state concurrent systems (including communication protocols). Although these techniques have been proposed predominantly for systems modeled as Petri Nets (see e.g. [Pet81]), and for systems defined with CSP/CCS-style semantics [Hoa85, Mil89], they apply in principle to all models that express concurrency by interleaving [HP95, God96]. Partial-order reduction methods are effective and generally efficient for verifying local and termination properties (e.g. freedom of non-progress states and non-executable transitions [God90, Val90, HGP92, KP92a, GW93] and, moreover, for verifying linear-time temporal logic (LTL) properties [Val92, Val93, GW94, HP95, Pel96]. The latter is known as *LTL model-checking*, and captures arbitrary (temporal) safety and liveness properties of concurrent systems [Lam77, Pnu77, Lam80, Lam83, WVS83, LP85, VW86, AS87, Wol89, MP92].

Like the improved state exploration techniques discussed above (and specific to the CFSM model), partial-order reduction methods exploit the fact that in many cases the properties verified are insensitive to the order of concurrent transitions. They aim at exploring just one fixed order among concurrent transitions at global states, by executing at each global state encountered during state exploration only a discriminating *subset* of the transitions executable at that state, rather than all of them. This then yields a reduced state space which is guaranteed to preserve the property under consideration. Partial-order reduction methods are also discussed in more detail later in the thesis. Chapter 7 presents an approach of our own to LTL model-checking, which we will propose as an enhancement of the partial-order approach.

3.2 Closed covers

The "closed-cover" technique by Mohamed Gouda [Gou84] is somewhat of an intruder compared to most other relief strategies in the sense that its sole objective is to prove the absence of protocol

design errors rather than showing their existence. We quote [Yua88, p. 167]:

> "this technique is a theoretical school of thought which believes that proving protocols free from errors is much more significant than detecting errors, and detecting no errors is not sufficient enough to show protocols are free from errors [Gou84]."

A *closed cover* is basically a set of global states of a protocol containing only the initial global state and global states that are someway "closed" with respect to reachability (we refer to [Gou84] for a precise definition). It was proven that the existence of a closed cover is sufficient (and in many cases necessary) to guarantee indefinite progress for a two-process protocol. The technique thus consists in finding a closed cover which, unfortunately, must be "guessed" and is hence a difficult task. Another drawback of the closed-cover technique is its inability to verify the possibility of non-progress, but then again this complies with the aim of proving the absence instead of the presence of errors. Two clear advantages are that the size of a closed cover is usually smaller than the size of the state space of a protocol and that progress of unbounded protocols may be verified as well (cf. fair reachability analysis). Gouda claims that his technique can be extended in a straightforward manner to verify progress for protocols with more than two processes. He also signifies an analogy between the closed-cover technique and the assertion techniques to verify safety properties of sequential programs. In this analogy, "closedness" of a global state corresponds to the requirement that each assertion before a block of statements in a sequential program must be sufficient to ensure the assertion after the block.

3.3 Acyclic expansions

In [BZ83, KWN88], acyclic expansion techniques were proposed which attempt to overcome the state explosion problem by avoiding altogether the construction of a *global* reachability graph. Instead of exploring the global states of a protocol as a whole, these techniques consider processes separately by expanding their process graphs into *local trees*. Global information is then added to the trees such that each local tree represents all possible (global) executions of the corresponding process. Design errors such as deadlocks, unspecified receptions and buffer overflows can be detected during the construction of the local trees.

In comparison with conventional reachability analysis, the acyclic expansion techniques reduce the number of analyzed states from $O(m_1 \times m_2 \times \ldots \times m_n)$ to $O(m_1 + m_2 + \ldots + m_n)$, where n is the number of processes and m_i is the number of process states of process P_i. Thus, when a protocol is rather complex, involving a large number of process states per process, these techniques clearly outperform reachability analysis. An additional advantage is the modularity: processes can easily be modified without affecting the complete analysis, as all local trees are maintained individually. A considerable drawback of the approach is the algorithmic complexity. Functions are used to acquire

and add global information to the local trees. Their implementation requires complex tree searching procedures, which make the overall verification algorithm much more complicated than the standard perturbation algorithm.

3.4 Divide-and-conquer strategies

The following techniques follow the well-established divide-and-conquer paradigm to problem solving. A protocol is decomposed or partitioned into components which are subsequently verified separately to ensure the correctness of the protocol itself. The complexity of verification is thus relieved since the components are usually smaller in the number of states and transitions than the original protocol.

3.4.1 Duologue-matrix analysis

Duologue-matrix analysis is one of the first automated protocol validation techniques [Zaf78]. The technique was proposed for two-process protocols only, where each process must further obey the condition that any cycle in its process graph passes through the initial (or some quiescent) process state. Recall that this restriction was also in effect for the technique in [II83] based on reduced implementation sequences, which can then be seen as an extension from the two-process model here to multiple processes.

Duologue-matrix analysis starts off by decomposing each process into paths, or *unilogues*, that begin and end in the initial process state. The sets of unilogues of both processes are coupled to form *duologues* (i.e. sequences of two-process interactions) and the duologues are represented in a *duologue matrix*. This matrix basically resembles the Cartesian product of the sets of unilogues. Subsequently, each duologue is classified as being either well-behaved, non-occurrable or erroneous, and assigned the value +1, 0 or −1, respectively. A duologue is well-behaved if it always returns to the initial global state of the protocol and if all messages sent along one unilogue are received along the other. Clearly, this prohibits the occurrence of deadlocks and many unspecified receptions. A duologue is non-occurrable if it can never be executed or if its execution always results in the execution of another duologue. These duologues have thus little effect on the actual protocol behavior. Lastly, a duologue is erroneous if it is neither well-behaved nor non-occurrable. Each duologue in the duologue matrix is then replaced by its corresponding value, yielding a *validation matrix* that can be used to detect design errors. In particular, a protocol contains an error if its validation matrix contains any −1 elements, and the positions of these elements identify the erroneous duologues of the protocol. Furthermore, if one of the rows or columns of the validation matrix contains more than one +1 element, then the behavior of one

process is ambiguous with respect to the other. It turns out that the validation matrix of a "perfect" protocol is an identity matrix. Duologue-matrix analysis has proven to be useful for the detection of design errors in real protocols, such as the X.21 recommendation of CCITT [WZ78].

3.4.2 Decomposition methods

Vuong & Cowan observed that large, well-designed protocols represented by finite directed graphs often exhibit some basic structures which allow them to be decomposed into several smaller component graphs [VC82]. These components can be verified separately to yield a verdict about the original protocol. Three basic structures are identified (viz. the nested, sequential and parallel structures), each of which enables a specific decomposition scheme. The decompositions turn out rather simple and powerful for protocols which indeed demonstrate these structures, but are suited for two-process protocols only. Similar methods were proposed in [CM83, CM86, CGL85], where also protocols with irregular structures are partitioned into components (or multi-phases [CGL85]).

3.4.3 Protocol projections

Another decomposition approach is based on the idea of protocol projections and aims at protocols with several distinguishable functions [LS84]. A protocol undergoes a functional decomposition by means of projections. The extracted functions are verified individually on the basis of so-called *image protocols*. An image protocol is constructed for each function by aggregating groups of states, messages and events of the corresponding entities in the original protocol. Image protocols are typically smaller and easier to analyze.

3.5 Partial state exploration

Partial state exploration techniques are motivated by the perception that analyzing just that part of the state space with the largest probability of occurring may be "good enough" to judge a protocol correct. Such techniques may provide a good alternative if the amount of available memory is insufficient for exhaustive analysis [Hol91], or even for improved state exploration techniques (cf. Section 3.1). The inherent drawback of partial state exploration is, of course, that it cannot be used to verify the *absence* of errors (or conversely, the error coverage attained cannot be measured).

3.5.1 Random-walk state exploration

Colin West [Wes86] proposed a variation of conventional reachability analysis, in which a new global state is derived at random by selecting arbitrarily one transition to be executed at the current

state. Execution sequences generated in this manner can thus be seen as *random walks* through the state space of a protocol. Hence the name random-walk state exploration. Each time a random walk leads to an error, one tries to correct the error and continues with another walk. State exploration through random walks has therefore no well-defined condition for termination. West motivates this technique by the belief that the analysis of just a few execution sequences leading to a design error is sufficient to identify the cause of the error and thus to fix it. Its use appeared very effective indeed: the coverage and number of errors detected increase asymptotically with the number of random walks [Wes86]. Moreover, even *unbounded* protocols can be explored by such "random simulation" since it works largely independent of the size and complexity of a protocol.

Gerard Holzmann also used a random-walk approach in a controlled partial search technique called *supertrace* [Hol88, Hol90, Hol91]. The supertrace technique allows larger protocols to be analyzed by reducing the amount of RAM needed to store each global state. It uses *bit-state hashing*: memory is arranged as a bit array and each generated state is "hashed" (i.e. a hash value is computed from the state) into an index of this array. By ignoring hash conflicts (which arise when two different global states yield the same hash value), state generation automatically becomes partial when the number of global states exceeds the size of the bit array, and may already be partial before this threshold. Obviously, the random nature of the supertrace technique lies in the unpredictable occurrence of hash conflicts throughout the search. Holzmann claimed that this technique is superior to other partial state exploration techniques [Hol90, Hol91].

3.5.2 Probabilistic verification

Instead of performing exhaustive state exploration, a probabilistic verification approach examines the "most probable" execution sequences of a protocol. It operates under the assumption that the probability of encountering a state or transition with low probability of occurrence is very small in a real execution sequence of a protocol. Protocols with errors in sequences that are not likely to be executed are then considered to be acceptable. A concrete probabilistic verification technique is given in [MS87]. As pointed out in [Rud88, Hol90], the main difficulty with such techniques is estimating the probabilities of occurrence of states and transitions.

3.5.3 Heuristic verification

Heuristic-based verification techniques have been proposed in [Hol87, LCL87]. Heuristics are used to guide the generation of global states in order to detect design errors quickly and more efficiently without incurring too much overhead. According to [LCL87], heuristic information can be applied mainly at three points during state exploration: (1) in deciding which global state to

perturb next, (2) in deciding which transition to execute next, and (3) in deciding which global states to discard. For instance, for a faster detection of deadlocks and unspecified receptions it appears beneficial to prioritize the execution of receive transitions over send transitions, while for buffer overflows it is the other way around. Similarly, respective preference should be given to the perturbation of the global states with the largest number of executable receive transitions, or states with the "longest" buffers. In order to decide which states to discard one can use occurrence probabilities as in probabilistic verification methods. Other typical heuristic suggestions are to minimize a protocol design before verification in terms of the size of its state machines (i.e. the number of process states and transitions) and the amount of concurrency (i.e. "tightly-coupled" systems), and to limit the size of buffers (cf. Section 2.5.1).

3.6 Memory management techniques

The supertrace or bit-state hashing technique of Holzmann [Hol88, Hol90, Hol91], described above, is in fact a pure memory management technique. It does not actually reduce the number of global states generated during state exploration, as do the improved state exploration techniques in Section 3.1, but it merely decreases the memory used to store each individual state. The supertrace technique can therefore be used in conjunction with any state exploration technique (i.e. improved or not).

Another memory management technique that can be combined with state exploration techniques is *state space caching* [Hol85, Hol87, JJ91, GHP92]. It operates in the context of a depth-first search (DFS) strategy. State space caching amounts to storing all the global states in the currently explored execution path of a protocol, i.e. all the states in the current DFS stack, plus as many other global states as possible given the remaining amount of available memory. A limited *cache* is thus created consisting of selected states that have already been generated. Initially, every global state generated is stored in the cache. When the cache fills up, old states that are no longer in the DFS stack are removed from the cache to accommodate new ones. This technique never attempts to store more states than possible in the cache. Hence, if the size of the cache is larger than the length of the longest execution path of the protocol (i.e. the maximal size of the DFS stack during state exploration), the state space of the protocol will be fully explored even when the state space itself does not fit in memory (the depth of the state space is usually much smaller than its breadth [Hol91]). If the size of the cache is too small, certain execution paths will be truncated.

The problem of using state space caching is that one can no longer determine whether a newly generated global state has already been encountered in a previously explored execution path (see the discussion in Section 2.4 prior to Example 2.14). Surely, exploring a state each time it is encountered is wasteful. State space caching may thus incur many redundant explorations of global

states, yielding a potentially dramatic run-time increase. Indeed, during conventional reachability analysis, almost every global state of a protocol is typically encountered many times, primarily because all explorations of interleavings of concurrent transitions lead to the same state. The use of state space caching in conjunction with conventional reachability analysis will thus likely cause run-time explosion. Yet, using state space caching in conjunction with relief strategies like fair reachability analysis, simultaneous reachability analysis or partial-order reduction methods (or their enhancements proposed later in this thesis) can be very beneficial. Such improved state exploration techniques reduce not only the number of global states, but also the number of transitions explored. They often avoid most of the nonessential explorations of interleavings of concurrent transitions, and many global states will therefore be encountered only once during state exploration. States that are encountered only once do not need to be stored in memory. Indeed, the mere reason for storing states in memory is to avoid multiple explorations of the same state: when an already generated state is generated again later during state exploration, it is not necessary to regenerate all its successors. Even though it is impossible to anticipate which global states are generated only once (this can be determined only after completing state exploration), if most global states are generated just once, the probability that some state will be regenerated is small. Hence, the risk of redundant work when not storing an already generated state becomes small as well. This enables the combined use of state space caching and improved state exploration techniques without incurring too many redundant explorations of global states. The memory requirements can then strongly decrease without a serious increase of the run-time requirements.

A novel use of bit-state hashing and state space caching for verifying concurrent systems and protocols was recently reported in [MK96]. A method was proposed that uses bit-state hashing in a pre-processing step before verification to compute and store the so-called revisiting degree of each state of a system (similar but not identical to the indegree of a node in a directed graph). Both the pre-processing step and the actual verification of the system are performed by a DFS of the system's state space. During the first DFS, when a state is generated and its current revisiting degree is zero, the revisiting degree is set to 1 and the revisiting degrees of all successor states are calculated recursively. When a state is generated and its current revisiting degree is greater than zero (i.e. the state is regenerated), the revisiting degree is incremented by 1 and backtracking takes place. (All this applies in fact only to states that do not close a cycle when generated, since such states are already on the DFS stack and can thus easily be identified with no additional space needed [MK96]). During the second DFS, i.e. for the purpose of verification, when a state is generated its revisiting degree (stored in the hash table) is decremented by one. If this yields a revisiting degree of zero, the state is removed from memory. As a result, state space caching can be employed during the DFS without incurring *any* redundant explorations of global states. The memory requirements can still strongly decrease and at the *predetermined* expense of only one

extra DFS of the state space of the verified system, namely for computing the revisiting degrees of the states. Like bit-state hashing and "classic" state space caching, the method proposed in [MK96] can be applied to any (improved) state exploration technique. In particular, one can carry out the computation of revisiting degrees of states and the subsequent verification effort on the reduced state space spawned by relief strategies such as fair reachability analysis, simultaneous reachability analysis or partial-order reduction methods (or their enhancements proposed later in this thesis) instead of the full state space of the system spawned by conventional reachability analysis.

Chapter 4

Fair reachability analysis for multi-cyclic protocols

One of the first techniques developed to relieve the state explosion problem in protocol verification is *fair reachability analysis* (FRA). As discussed in the previous chapter, FRA is an improved state exploration technique which was initially proposed for verifying logical correctness properties of two-process protocols [RW82, GH85], and subsequently extended to deal with cyclic protocols [LM94, LM96]. In this chapter we further generalize FRA to *multi-cyclic protocols*. A multi-cyclic protocol consists of a collection of unidirectional rings, or component cyclic protocols, which are interconnected such that no two rings share more than one process. We show that FRA is effective in deciding deadlock-freedom for the class of multi-cyclic protocols whose fair reachability graphs are finite. Throughout the presentation we relate our findings to the work on cyclic protocols in [LM94a, LM96]. Furthermore, we also advocate that FRA is inherently infeasible beyond multi-cyclic protocols, i.e. for verifying protocols with more complex communication topologies. These contributions have appeared in [SU95a, SU96a, LM+96]. The chapter ends with a detailed analysis of a problem solved for cyclic protocols, but remaining for multi-cyclic protocols: deciding logical correctness properties other than deadlock-freedom by finite extension of the fair reachability graph [GH85, LM94b, LM96].

4.1 Preliminaries

Before turning to the presentation of FRA for multi-cyclic protocols, let us add some more basic notions that will be used in the sequel of this thesis.

4.1.1 Equivalent transition sequences

Following the earlier work on FRA and most other improved state exploration techniques, we consider an equivalence relation over sequences of transitions that reveals much of the redundancy

in conventional reachability analysis incurred by the modeling of concurrency by interleaving (see Section 3.1.1).

Notation 4.1

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol and $\sigma \in (\bigcup_{i \in I} \Delta_i)^*$ a (finite) sequence of transitions. We will use *pref*(σ) and *suf*(σ) to denote the set of prefixes and suffixes of σ, respectively, and $\sigma|_T$ to denote the *projection* of σ on a set of transitions T (obtained by removing from σ those transitions that are not in T). A process P_i is said to be *active* on σ if $\sigma|_{\Delta_i} \neq \varepsilon$. The set of all processes active on σ is denoted by *act*(σ). We adopt this notation also for single transitions and sets of transitions, viz. $act(t) = \{i\}$ for a transition $t \in \Delta_i$, and $act(T) = \{i \mid T \cap \Delta_i \neq \emptyset\}$ for a set of transitions T [ÖU95].□

Definition 4.2

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol. Two transition sequences $\sigma, \omega \in (\bigcup_{i \in I} \Delta_i)^*$ are defined to be equivalent, denoted by $\sigma \equiv \omega$, iff $\forall i \in I$: $\sigma|_{\Delta_i} = \omega|_{\Delta_i}$. □

Sequences of transitions may thus be grouped into equivalence classes. Transition sequences that are equivalent have the same length. Moreover, any two equivalent transition sequences that start from a common global state lead to a common global state. This characteristic, stated by Proposition 4.3, is pivotal in recognizing and exploiting the excess in conventional reachability analysis due to the interleaving of concurrent transitions. It testifies that it is in principle sufficient to explore just one transition sequence per equivalence class in order to determine all the reachable global states of a protocol, and hence to verify many of its properties (logical correctness properties in particular).

Proposition 4.3

Let $G \xrightarrow{\sigma}{}^* H$ and $G \xrightarrow{\omega}{}^* H'$. If $\sigma \equiv \omega$, then $H = H'$.

Proof: Straightforward by definition of the equivalence relation $\equiv$, and the fact that both σ and ω can be executed from G. □

Henceforth, we use $\sigma \,{}_G\!\equiv_H \omega$ to denote that $G \xrightarrow{\sigma}{}^* H$, $G \xrightarrow{\omega}{}^* H$ and $\sigma \equiv \omega$ [ÖU95].

Corollary 4.4

Let $G \xrightarrow{\sigma}{}^* H$ and $G \xrightarrow{\omega}{}^* H'$. If $act(\sigma) \cap act(\omega) = \emptyset$, then $\exists Q$: $\sigma\omega \,{}_G\!\equiv_Q \omega\sigma$. □

4.1.2 Potentially executable transitions

First defined by Itoh & Ichikawa [II83] and later adopted by Özdemir & Ural [ÖU94, ÖU95] (see Section 3.1.1), we conveniently regard the notion of potentially executable transitions to further

classify transitions that are defined but not executable at a given global state. This notion also plays an important role in the attempt to reduce the cost of modeling concurrency by interleaving.

Definition 4.5

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ and $t \in \Delta_{ij}$ a transition defined at G, for some $i, j \in I$. t is *potentially executable at* G iff

- t is a send transition and $|c_{ij}^G| = B_{ij}$, or
- t is a receive transition and $c_{ji}^G = \varepsilon$.

The set of send and receive transitions from Δ_{ij} that are potentially executable at G are denoted by $P_{ij}^-(G)$ and $P_{ij}^+(G)$, respectively, and $P_{ij}(G) = P_{ij}^-(G) \cup P_{ij}^+(G)$. □

Intuitively, a potentially executable transition at a global state G is a transition of a process P_i that is not executable at G but may still become executable at a global state reachable from G *without any progress of* P_i. Notice that the potential executability of send transitions is immaterial for protocols whose channels are not prebounded (cf. Section 2.5.1), i.e. $P_{ij}^-(G)$ is always empty if B_{ij} is unspecified.

In the formulation of FRA for cyclic protocols [RW82, LM94, LM96] discrete attention is given to those potentially executable transitions at a global state G that actually become executable at an immediate successor of G. In [LM94, LM96] such transitions are said to be enabled at G.

Definition 4.6

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ and $t \in \Delta_{ij}$ a transition defined at G, for some $i, j \in I$. t is *enabled at* G (*by* a transition t') iff

- $t \in P_{ij}^-(G)$ and $\exists t' \in \Delta_{ji}: t' \in X_{ji}^+(G)$, or
- $t = (s_i^G, +y, s) \in P_{ij}^+(G)$ and $\exists t' \in \Delta_{ji}: t' = (s_j^G, -y, s') \in X_{ji}^-(G)$.

The sets of send and receive transitions from Δ_{ij} that are enabled at G are denoted by $E_{ij}^-(G)$ and $E_{ij}^+(G)$, respectively, and $E_{ij}(G) = E_{ij}^-(G) \cup E_{ij}^+(G)$. □

It follows from Definition 4.6 that $E_{ij}(G) \subseteq P_{ij}(G)$ and that $t \in E_{ij}(G)$ implies the existence of a global state H such that $G \longrightarrow H$ and $t \in X_{ij}(H)$.

Notation 4.7

$$P_i^-(G) = \bigcup_{j \in I} P_{ij}^-(G) \qquad P_i^+(G) = \bigcup_{j \in I} P_{ij}^+(G) \qquad P_i(G) = P_i^-(G) \cup P_i^+(G)$$

$$P^-(G) = \bigcup_{i \in I} P_i^-(G) \qquad P^+(G) = \bigcup_{i \in I} P_i^+(G) \qquad P(G) = P^-(G) \cup P^+(G)$$

$E_i^-(G) = \bigcup_{j \in J} E_{ij}^-(G)$ $\quad E_i^+(G) = \bigcup_{j \in J} E_{ij}^+(G)$ $\quad E_i(G) = E_i^-(G) \cup E_i^+(G)$

$E^-(G) = \bigcup_{i \in I} E_i^-(G)$ $\quad E^+(G) = \bigcup_{i \in I} E_i^+(G)$ $\quad E(G) = E^-(G) \cup E^+(G)$ □

4.2 Generalizing FRA to multi-cyclic protocols

This section generalizes the notion of fair reachability from cyclic to multi-cyclic protocols and studies properties of their fair reachability graphs.

4.2.1 Multi-cyclic protocols

Notation 4.8

Let $k \in \{1, 2, \ldots, j\}$, then $\vec{k}$ and $\overleftarrow{k}$ are defined as follows:

$\vec{k} = 1$ if $k = j$

$= k + 1$ otherwise

$\overleftarrow{k} = j$ if $k = 1$

$= k - 1$ otherwise □

Definition 4.9

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol. A *ring* r in Π is a set $\{(i_1, i_2), (i_2, i_3), \ldots, (i_{j-1}, i_j), (i_j, i_1)\}$, $2 \le j \le |I|$, such that $\forall k, l \in \{1, 2, \ldots, j\}$: $k \neq l \Rightarrow i_k \neq i_l$ and $\forall k \in \{1, 2, \ldots, j\}$: $(i_k, i_{\vec{k}}) \in L$. The set of all rings in Π is denoted by $\mathfrak{R}_\Pi$. □

A ring in a protocol represents a simple cycle in the topology graph of the protocol. A multi-cyclic protocol is now defined as follows.

Definition 4.10

A *multi-cyclic protocol* is a protocol Π satisfying the following conditions:

i) TG_Π is strongly connected,

ii) $\forall r, r' \in \mathfrak{R}_\Pi$: $r \neq r' \Rightarrow r \cap r' = \emptyset$, and

iii) $\forall r \in \mathfrak{R}_\Pi \; \forall (i, j), (k, l) \in r$: $B_{ij} = B_{kl}$. □

It is again understood that the third condition in Definition 4.10 applies only to protocols with prebounded channels. The bounds on all the simplex channels in a given ring must then be equal, which is not a real constraint because they can always be set accordingly. Conditions (i) and (ii) on a multi-cyclic protocol confine its communication topology such that no two rings share a simplex

channel and each simplex channel belongs to *exactly* one ring. Although restrictive, this way of joining rings does not limit the end-to-end (as opposed to point-to-point) connectivity between processes since the topology graph of a multi-cyclic protocol is strongly connected. In fact, the class of multi-cyclic protocols has an interestingly wide applicability in practical protocol modeling. It includes not only protocols with a multi-ring topology (in particular the cyclic protocols studied in [LM94, LM96][4]), but also protocols with other common network topologies such as a daisy-chain [SU96a], a star or a tree, as illustrated in Figure 4.1. Furthermore, configurations of these elementary network topologies can be composed as long as each channel in the resulting protocol belongs to exactly one ring.

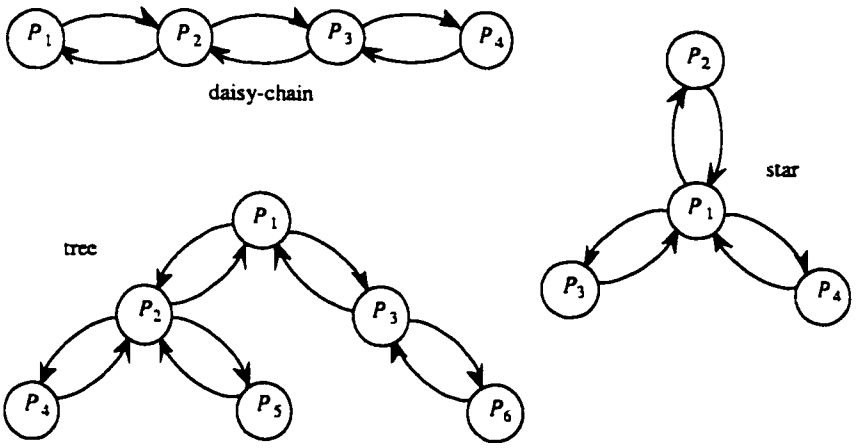


Figure 4.1 Basic multi-cyclic protocols.

4.2.2 Generalizing the fair reachability relation

Fair transition-tuples

Exploring the global state space of a multi-cyclic protocol by FRA is based on the execution of so-called fair transition-tuples, which are effectively computed from the transitions executable and enabled at global states. Two types of fair transition-tuples are distinguished, viz. ring-tuples and channel-pairs. Before presenting the definitions, we introduce some more notational conventions.

Notation 4.11

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a multi-cyclic protocol. For any ring $r = \{(i_1, i_2), (i_2, i_3), \ldots, (i_j, i_1)\} \in \mathfrak{R}_\Pi$, a simplex channel $C_{i_k i_{\vec{k}}}$ $(k \in \{1, 2, \ldots, j\})$ is denoted by $C_{k\vec{k}}(r)$. This notation is also adopted

[4] Cyclic protocols with prebounded channels are not at all considered in [LM94, LM96].

for process states and (sets of) transitions. Specifically, with $r \in \mathfrak{R}_\Pi$, $i_k \in I$ and $k \in \{1, 2, \ldots, |r|\}$, we have $P_{i_k} = P_k(r) = (S_k(r), M_k(r), s_k^0(r), \Delta_k(r))$ and $C_{k\vec{k}}(r)$ denotes the outgoing channel of process P_{i_k} associated with ring r. In addition, for a global state G

$s_k(r, G) \in S_k(r)$ is the process state of P_{i_k} in G,

$c_{k\vec{k}}(r, G) \in M_{k\vec{k}}(r)^* \subseteq M_k(r)^*$ is the content of $C_{k\vec{k}}(r)$ in G,

$t_k(r) \in X^-_{k\vec{k}}(r,G) \subseteq \Delta_{k\vec{k}}(r, G) \subseteq \Delta_k(r)$ is a send transition of P_{i_k} executable at G over $C_{k\vec{k}}(r)$,

$t_k(r) \in X^+_{\bar{k}k}(r,G) \subseteq \Delta_{\bar{k}k}(r, G) \subseteq \Delta_k(r)$ is a receive transition of P_{i_k} executable at G over $C_{\bar{k}k}(r)$.

The same conventions apply to the sets of potentially executable and enabled transitions at G. Note that when process P_{i_k} participates in more than one ring, it is named differently for each ring. □

Definition 4.12

Let G be a global state of a multi-cyclic protocol Π. A *ring-tuple* in G is a tuple $\langle t_k(r)\rangle_{k \in \{1,\ldots,|r|\}}$, for some $r \in \mathfrak{R}_\Pi$, satisfying one of the following two conditions:

- $\forall k \in \{1,\ldots,|r|\}: t_k(r) \in X^-_{k\vec{k}}(r,G)$;
- $\forall k \in \{1,\ldots,|r|\}: t_k(r) \in X^+_{\bar{k}k}(r,G)$.

□

Definition 4.13

Let G be a global state of a multi-cyclic protocol Π. A *channel-pair* in G is an ordered pair $\langle t_1, t_2\rangle$ satisfying one of the following three conditions:

- $t_1 \in X^-_{ij}(G)$ and $t_2 \in X^+_{ji}(G)$;
- $t_1 = (s_i^G, -x, s) \in X^-_{ij}(G)$ and $t_2 = (s_j^G, +x, s') \in E^+_{ji}(G)$;
- $t_1 \in X^+_{ji}(G)$ and $t_2 \in E^-_{ij}(G)$.

□

As the names suggest, each ring-tuple is associated with a ring and each channel-pair with a simplex channel. A ring-tuple in a global state G entails for each process involved in a given ring $r \in \mathfrak{R}_\Pi$ a send transition executable at G over its outgoing channel in r, or for each process a receive transition executable at G over its incoming channel in r. A channel-pair in G contains a send and receive transition from two adjacent processes such that either both are executable at G, or only one of the transitions is executable while the other is enabled at G. In the latter case, note the message correspondence required among the two transitions when the receive transition is enabled at G, in accordance with Definition 4.6.

Definition 4.14

Let G be a global state of a multi-cyclic protocol. A *fair transition-tuple* in G is a ring-tuple in G or a channel-pair in G. The set of all fair transition-tuples in G is denoted by $F(G)$. □

In [LM94, LM96], the fair transition-tuples in a global state of a cyclic protocol are called *fair progress vectors*. Ring-tuples are referred to as *concurrency vectors* and channel-pairs as *send-receive pairs* or *interaction-pairs*. Note that concurrency vectors always include one transition from *each* process in a cyclic protocol because it consists of only one ring.

The fair reachability relation

Definition 4.15

Let G be a global state of a multi-cyclic protocol and $\tau = \langle t_1, t_2, \ldots, t_m \rangle \in F(G)$. A *linearization* of τ is a sequence

$t_1 t_2$	if $m = 2$ and $t_2 \in E(G)$ (i.e. τ is a channel-pair with an enabled transition)
$t_{\pi(1)} t_{\pi(2)} \cdots t_{\pi(m)}$	otherwise, with π any permutation on $\{1, 2, \ldots, m\}$

The set of all linearizations of τ is denoted by $lin(\tau)$. □

Remark that we do not define $t_2 t_1$ as a linearization of a channel-pair $\langle t_1, t_2 \rangle$ if t_2 is an enabled transition, since t_2 can then only be executed *after* the execution of t_1. Apart from this special case, the order of executing the transitions in a fair transition-tuple is arbitrary.

Proposition 4.16

Let $\tau \in F(G)$ and $\gamma \in lin(\tau)$ with $G \xrightarrow{\gamma}{}^{*} H$, then $\forall \gamma' \in lin(\tau)$: $\gamma \, {}_G\!\equiv_H \gamma'$.

Proof: It is obvious that each linearization γ of $\tau \in F(G)$ leads to some global state H, which justifies the supposition. Yet, since τ has at most one transition per process all its linearizations are equivalent (Definition 4.2) and lead to the same global state (Proposition 4.3). □

Consider a fair transition-tuple $\tau = \langle t_1, t_2, \ldots, t_m \rangle \in F(G)$. In conventional reachability analysis all linearizations of τ are fully explored as transitions are executed one at a time. This accounts for the generation of up to $2^m - 1$ distinct global states. Instead, the generation of all but one of these global states (i.e. the common "sink" state) can be omitted at G by executing the transitions in τ together in a single step. The reachability relation underlying FRA is formulated accordingly.

Definition 4.17

Let G and H be global states of a multi-cyclic protocol. $G \xrightarrow[f]{} H$ iff $\exists \tau \in F(G)$ with $\gamma \in lin(\tau)$ such that $G \xrightarrow{\gamma}{}^{*} H$. This is also denoted by $G \xrightarrow[f]{\tau} H$. □

From Proposition 4.16, the execution of a fair transition-tuple is irrespective of the linearization considered and always yields a unique global state. The relation $\xrightarrow[f]{}$ is thus well-defined.

Definition 4.18

Let G and H be global states of a multi-cyclic protocol Π, and denote by $\xrightarrow{}_f^*$ the reflexive and transitive closure of $\xrightarrow{}_f$. H is *fair reachable from* G iff $G \xrightarrow{}_f^* H$. When $G = G^0$, H is said to be *fair reachable*. The set of all fair reachable global states of Π is denoted by $\mathbf{F}_\Pi$. For a sequence of fair transition-tuples $\upsilon = \tau_1\tau_2\ldots\tau_m$, $G \xrightarrow{\upsilon}_f^* H$ denotes the existence of global states $Q^0,\ldots, Q^m$ such that $G = Q^0 \xrightarrow{\tau_1}_f Q^1 \xrightarrow{\tau_2}_f \ldots \xrightarrow{\tau_m}_f Q^m = H$. □

```
/* A is the set of global states that have been analyzed.          */
/* W is the set of global states that still need to be analyzed. */

/* Initialize: */
A = ∅
W = {G^0}

/* Loop: */
while W ≠ ∅ do {
    remove an element G from W
    add G to A
    for all τ in F(G) do {
        /* execution of fair transition-tuple τ */
        derive H such that G -τ->_f H
        if H is NOT already in A or W then add H to W
    }
}
```

Figure 4.2 State exploration by FRA.

Figure 4.2 gives an algorithm for exploring the fair reachable global state space of a (multi-cyclic) protocol, akin to the standard perturbation algorithm discussed in Chapter 2. As indicated by the box, the algorithm above differs only in the derivation of successive global states: fair transition-tuples are executed instead of single transitions. The fair reachable global state space of a protocol can be viewed as a labeled directed graph, the *fair reachability graph*, in which the nodes correspond to fair reachable global states and the edges resemble the relation $\xrightarrow{\tau}_f$. As an immediate result, each fair reachable global state is a reachable global state.

Proposition 4.19

$\mathbf{F}_\Pi \subseteq \mathbf{R}_\Pi$

Proof: By definition of $\xrightarrow{}_f^*$. □

Example 4.20

Consider the protocol in Figure 4.3, where $a \in M_{12}$, $b \in M_{23}$, $c \in M_{31}$, $d \in M_{34}$ and $e \in M_{43}$. This multi-cyclic protocol comprises two rings $r_1 = \{(1, 2), (2, 3), (3, 1)\}$ and $r_2 = \{(3, 4), (4, 3)\}$. The initial global state has one fair transition-tuple: a ring-tuple $\langle(10, -a, 11), (20, -b, 21), (30, -c, 31)\rangle$ associated with ring r_1. Its execution yields the global state $(\langle 11, 21, 31, 40\rangle, \langle a, b, c, \varepsilon, \varepsilon\rangle)$ in which we have the channel-pair $\langle(31, -d, 32), (40, +d, 41)\rangle$. The complete fair reachability graph of the protocol, included in Figure 4.3 (the process states in the transitions of fair transition-tuples are omitted), contains four fair reachable global states and four global state transitions. In contrast, the (conventional) reachability graph contains 152 global states and 374 global state transitions. □

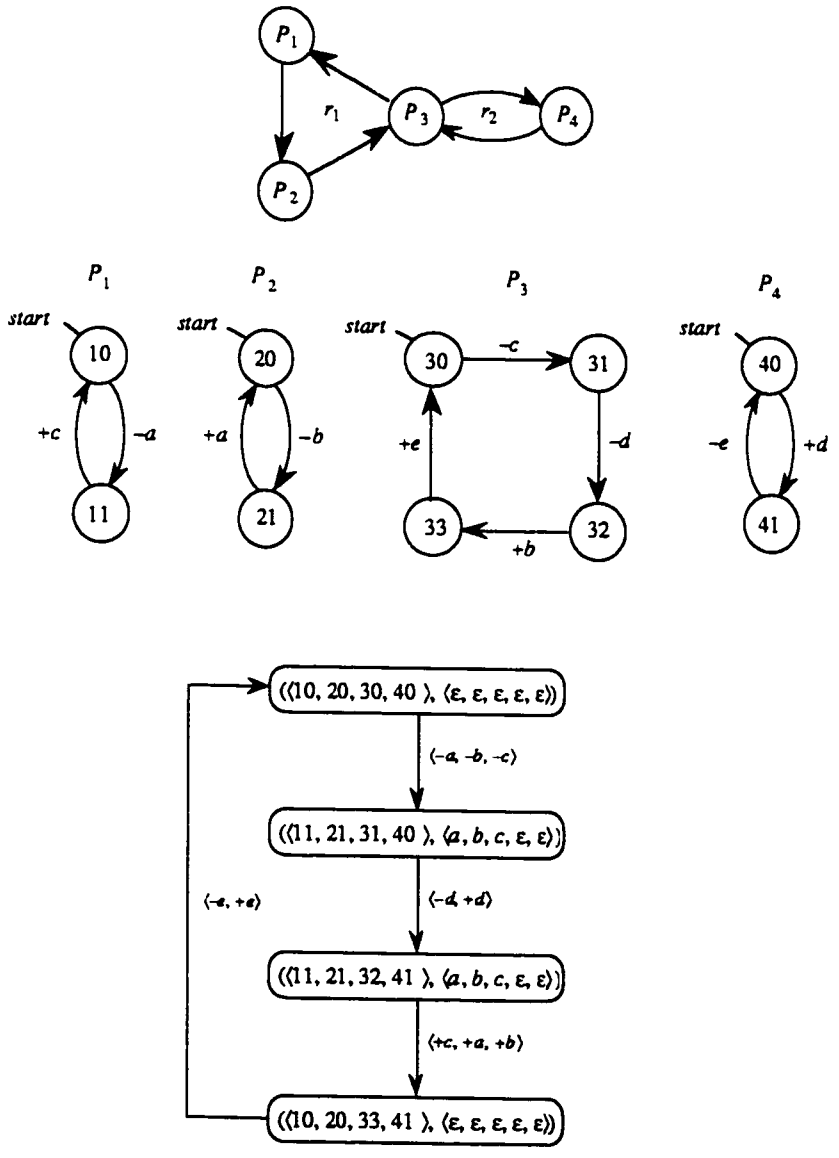


Figure 4.3 FRA applied to a concrete multi-cyclic protocol.

An important aspect of FRA in the context of cyclic protocols is the *equal channel length property* [LM94, LM96]: each fair reachable global state is a reachable global state in which all channels hold the same number of messages. This property is the key to deadlock detection. Contrary to cyclic protocols, however, multi-cyclic protocols involve processes with more than one incoming and one outgoing channel. Consequently, a relation between the set of fair reachable global states and the set of reachable global states with the equal channel length property is no longer adequate. We establish an alternate relation instead, identifying the fair reachable global states of a multi-cyclic protocol as reachable global states in which per ring all channels are of equal length. This *ring-wise* equal channel length property generalizes the equal channel length property for cyclic protocols and ultimately provides the basis for proving that FRA is effective in deciding deadlock-freedom for multi-cyclic protocols with finite fair reachability graphs.

Proposition 4.21

Let G be a global state of a multi-cyclic protocol Π. If $G \in \mathbf{F}_{\Pi}$, then G satisfies the *ring-wise equal channel length property* Γ:

$$\forall r \in \mathfrak{R}_{\Pi}\, \forall k \in \{1,\ldots,|r|\}: |c_{\overleftarrow{kk}}(r, G)| = |c_{k\vec{k}}(r, G)|$$

Proof: Let $G^0 \xrightarrow{\upsilon}_f{}^* G$. The proof is by induction on $|\upsilon|$, the number of fair transition-tuples in υ. If $|\upsilon| = 0$, then $G = G^0$ and Γ holds trivially since all channels in G^0 are empty. As induction hypothesis, suppose the claim holds for $|\upsilon| = m$. Also, let $G \xrightarrow{\tau}_f H$ for some $\tau \in F(G)$, then $|\upsilon\tau| = m+1$. We show that the execution of τ in G preserves Γ.

If τ is a ring-tuple, then the execution of τ either increases the length of each channel in r by one when all transitions in τ are send transitions, or decreases the length of each channel in r by one when all transitions in τ are receive transitions. Since all other channels remain unaffected, clearly, in both cases Γ still holds in H.

If τ is a channel-pair involving a simplex channel C_{ij}, then the execution of τ preserves the length of C_{ij}, since the message transmission of process P_i is compensated by the message reception of process P_j. Since no other channel is affected, H satisfies Γ. □

The next proposition provides more insight regarding the non-existence of fair transition-tuples in a global state G. There must be at least one process without executable and enabled transitions at G if $F(G) = \varnothing$.

Proposition 4.22

i) $F(G) = \varnothing \Rightarrow \forall i \in I: E_i(G) = \varnothing$;

ii) $F(G) = \varnothing \Rightarrow \exists i \in I: X_i(G) = \varnothing$.

Proof:

i) Directly from definitions 4.6, 4.12 and 4.13;

ii) If $F(G) = \varnothing$, then $\forall r \in \Re_\Pi \exists k \in \{1,\ldots,|r|)\}$: $X^{-}_{k\vec{k}}(r,G) \cup X^{+}_{k\vec{k}}(r,G) = \varnothing$, because otherwise there exists a ring-tuple associated with r or a channel-pair associated with a channel in r. Using the pigeon-hole principle, we must then have at least one process without executable transitions at G. □

From Proposition 4.22 we can further derive that a global state without fair transition-tuples implies the existence of a deadlock or unspecified reception in a protocol.

Proposition 4.23

Let Π be a multi-cyclic protocol and $G \in \mathbf{F}_\Pi$. If $F(G) = \varnothing$, then Π contains a deadlock state or an unspecified reception state.

Proof: From Proposition 4.22.(ii), we have $X_i(G) = \varnothing$ for some $i \in I$. Clearly, if $\forall i \in I$: $X_i(G) = \varnothing$ then G is a non-progress state, and hence a deadlock state or an unspecified reception state. On the other hand, if $\exists i \in I$: $X_i(G) \neq \varnothing$ it follows that $\exists i, j \in I$: $X_{ij}(G) \neq \varnothing \wedge X_j(G) = \varnothing$, since otherwise $F(G) \neq \varnothing$. Let $t \in X_{ij}(G)$. If t is a receive transition, then $|c_{ji}| \neq \varepsilon$ and with $G \in \mathbf{F}_\Pi$,by the ring-wise equal channel length property we have $|c_{kj}| \neq \varepsilon$, for some $(k, j) \in L$. G must thus be an unspecified reception state since $X_j(G) = \varnothing$. Alternatively, if t is a send transition then we either have G as an unspecified reception state, in case $|c_{ij}| \neq \varepsilon$, or we have the successor of G by t as an unspecified reception state, in case $|c_{ij}| = \varepsilon$ (since $E_j(G) = \varnothing$ by Proposition 4.22.(i)). □

4.3 Deciding deadlock-freedom for multi-cyclic protocols

In this section we show that FRA, as presented in Section 4.2, provides the capability of deciding deadlock-freedom for multi-cyclic protocols with finite fair reachability graphs. This result is established by completely characterizing the fair reachable global state space of such a protocol.

4.3.1 A complete characterization of the fair reachable global state space

Proposition 4.21 proved that all fair reachable global states of a multi-cyclic protocol are reachable global states satisfying the ring-wise equal channel length property. In order to show the converse, we adopt the notion of partial fair execution sequence defined in [LM94, LM96].

Remark 4.24

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol and $G^0 \xrightarrow{\sigma}{}^{*} G$. For ease of reference, the correspondence

between σ and its projections $\sigma|_{\Delta_i}$ on (the sets of transitions of) all processes P_i (see Notation 4.1) will be indicated by $\sigma \triangleq \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. σ is also called an *execution sequence* for G, and σ_i the *local execution sequence* of P_i with respect to (wrt for short) σ.

Let $G^0 \xrightarrow{\upsilon}_f{}^* G$. The projection '$|_{\Delta_i}$' is extended to sequences of fair transition-tuples in a straightforward manner, by removing from each fair transition-tuple those transitions that are not in Δ_i. Recall that a fair transition-tuple includes at most one transition per process, and hence the projection of a fair transition-tuple on a process is unique. Thus, υ also breaks down into a set of local execution sequences, i.e. $\upsilon \triangleq \{\upsilon_1, \upsilon_2, \ldots, \upsilon_n\}$. υ is called a *fair execution sequence* for G. The equivalence relation over execution sequences in Definition 4.2 then applies equally well to fair execution sequences. Furthermore, by the well-definedness of $\xrightarrow{\ }_f$, for any fair execution sequence one can always find an execution sequence with the same set of corresponding local execution sequences. □

Definition 4.25 defines a partial fair execution sequence for a reachable global state G of a multi-cyclic protocol as a "maximal fair prefix" of an execution sequence for G. A maximal fair prefix of an execution sequence $\sigma \triangleq \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ refers to a fair execution sequence with initial sequences (prefixes) of $\sigma_1, \sigma_2, \ldots, \sigma_n$ as corresponding local execution sequences, being maximal in the sense that it cannot be extended to a longer fair execution sequence by using the transitions in the remaining suffixes of $\sigma_1, \sigma_2, \ldots, \sigma_n$.

Definition 4.25

Let G and H be global states of a multi-cyclic protocol $\Pi = (\{P_i \mid i \in I\}, L)$ such that $G^0 \xrightarrow{\sigma}{}^* H$, $G^0 \xrightarrow{\upsilon}_f{}^* G \xrightarrow{\omega}{}^* H$ and $\forall i \in I$: $\sigma_i = \upsilon_i\omega_i$ (i.e. $\upsilon_i \in pref(\sigma_i)$ and $\omega_i \in suf(\sigma_i)$). υ is a *partial fair execution sequence* for H wrt σ iff $\nexists G'$: $G \xrightarrow{\ }_f G' \rightarrow^* H$ via the transitions in ω. G is called a *fair precursor* of H wrt σ. □

Proposition 4.26

Let $G^0 \xrightarrow{\sigma}{}^* G$. A partial fair execution sequence for G wrt σ is unique up to $\equiv$.

Proof: Suppose there exist υ and υ' such that $G^0 \xrightarrow{\upsilon}_f{}^* H$, $G^0 \xrightarrow{\upsilon'}_f{}^* H'$, $\upsilon \not\equiv \upsilon'$ and both υ and υ' are partial fair execution sequences for G wrt $\sigma \triangleq \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. By definition 4.25, for each $i \in I$ we have υ_i, $\upsilon'_i \in pref(\sigma_i)$, and hence $\upsilon_i \in pref(\upsilon'_i)$ or $\upsilon'_i \in pref(\upsilon_i)$. Furthermore, $\exists i \in I$: $\upsilon_i \neq \upsilon'_i$ since $\upsilon \not\equiv \upsilon'$, meaning that one of υ_i and υ'_i is in fact a *proper* prefix of the other, i.e. $|\upsilon_i| < |\upsilon'_i|$ or $|\upsilon_i| > |\upsilon'_i|$. The first case implies that $H \rightarrow^* H'$ via the transitions remaining in υ' after the execution of υ, and the second case implies that $H' \rightarrow^* H$ via the transitions remaining in υ after the execution of υ'. Clearly, the two cases are symmetric and we consider therefore only $|\upsilon_i| < |\upsilon'_i|$.

Once H is reached via υ several transitions remain in υ'. No fair transition-tuple can be formed

in H from these transitions, because otherwise υ is not a partial fair execution sequence for G wrt σ. That is, $H \xrightarrow{\upsilon}{}^{*} H'$ but not $H \xrightarrow[f]{\upsilon}{}^{*} H'$ via the transitions remaining in υ' after the execution of υ. However, this contradicts the supposition that υ' is a fair execution sequence for H'. As a result, if υ and υ' are both partial fair execution sequences for G wrt σ, then $\upsilon \equiv \upsilon'$. □

The uniqueness up to $\equiv$ of the partial fair execution sequence for G wrt σ immediately implies the uniqueness of the fair precursor of G wrt σ. Henceforth, they are denoted by $pfs(G, \sigma)$ and $fp(G, \sigma)$, respectively. Since $pfs(G, \sigma)$ is actually "built" from the set of local execution sequences $\{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ corresponding to σ, uniqueness of $pfs(G, \sigma)$ and $fp(G, \sigma)$ extends to all execution sequences that are equivalent to σ.

Corollary 4.27

Let $G^0 \xrightarrow{\sigma}{}^{*} G$. If $\sigma \equiv \omega$, then $pfs(G, \sigma) \equiv pfs(G, \omega)$ and $fp(G, \sigma) = fp(G, \omega)$. □

Example 4.28

Consider again the protocol described in Figure 4.3. It is not difficult to check that the global state $G = (\langle 10, 21, 32, 40\rangle, \langle a, b, \varepsilon, d, \varepsilon\rangle)$ is reachable via all (equivalent) execution sequences σ for G with the following local execution sequences:

$\sigma_1 = (10, -a, 11)\ (11, +c, 10)$

$\sigma_2 = (20, -b, 21)$

$\sigma_3 = (30, -c, 31)\ (31, -d, 32)$

$\sigma_4 = \varepsilon$

We also have $pfs(G, \sigma) \triangleq \{(10, -a, 11), \sigma_2, (30, -c, 31), \sigma_4\}$ and $fp(G, \sigma) = (\langle 11, 21, 31, 40\rangle, \langle a, b, c, \varepsilon, \varepsilon\rangle) \neq G$. G is thus not fair reachable. □

The following lemmas prove some properties of $fp(G, \sigma)$ and provide the basis for establishing the converse of Proposition 4.21, i.e. each reachable global state of a multi-cyclic protocol with the ring-wise equal channel length property is a fair reachable global state.

Lemma 4.29

For a multi-cyclic protocol, let $G^0 \xrightarrow{\sigma}{}^{*} G$ and $fp(G, \sigma) \xrightarrow{\omega}{}^{*} G$ with $\omega_i \in suf(\sigma_i)$ for all $i \in I$, and denote by t_i^{fp} the first transition of ω_i when $|\omega_i| > 0$. If $t_i^{fp} \in \Delta_{ij}$, for some $j \in I$, then either $|\omega_j| = 0$ or ($|\omega_j| > 0$ and $t_j^{fp} \notin \Delta_{ji}$).

Proof: Denote $fp(G, \sigma)$ by G^{fp} and observe that any t_i^{fp} that is not executable at G^{fp} must be

potentially executable at G^{fp}, since otherwise it cannot be the case that $G^{fp} \xrightarrow{\omega}{}^{*} G$. Thus, there are two cases to consider, in which we prove that $t_j^{fp} \notin \Delta_{ji}$ if $|\omega_j| > 0$:

i) $t_i^{fp} \in P_{ij}(G^{fp})$

Let $|\omega_j| > 0$, then t_j^{fp} exists. If $t_i^{fp} \in P_{ij}^{+}(G^{fp})$, then $c_{ji}^{fp} = \varepsilon$ and hence $t_j^{fp} \notin X_{ji}^{+}(G^{fp})$, because either $(i, j) \notin L$ or $c_{ij}^{fp} = \varepsilon$ from the fact that G^{fp} satisfies the ring-wise equal channel length property Γ. Also, $t_j^{fp} \notin X_{ji}^{-}(G^{fp}) \cup P_{ji}(G^{fp})$ since otherwise $\langle t_j^{fp}, t_i^{fp} \rangle$ is a channel-pair in G^{fp} (i.e. $t_j^{fp} \in X_{ji}^{-}(G^{fp})$ involves the matching message and t_i^{fp} is thus enabled at G^{fp} by t_j^{fp}), or ω is not an execution sequence from G^{fp} to G (i.e. t_i^{fp} and t_j^{fp} are both potentially executable and processes P_i and P_j are thus waiting for each other to proceed, or t_j^{fp} involves a "wrong" message and hence process P_i cannot proceed).

Similarly, if $t_i^{fp} \in P_{ij}^{-}(G^{fp})$ then $|c_{ij}^{fp}| = B_{ij}$ and hence $t_j^{fp} \notin X_{ji}^{-}(G^{fp})$, since either $(j, i) \notin L$ or $|c_{ji}^{fp}| = B_{ji}$ from the fact that G^{fp} satisfies Γ and that all the channels in a ring have equal bounds (condition (iii) of Definition 4.10). Also, $t_j^{fp} \notin X_{ji}^{+}(G^{fp}) \cup P_{ji}(G^{fp})$ since otherwise $\langle t_j^{fp}, t_i^{fp} \rangle$ is a channel-pair in G^{fp} (i.e. t_i^{fp} is enabled by $t_j^{fp} \in X_{ji}^{+}(G^{fp})$), or ω is not an execution sequence from G^{fp} to G (i.e. t_i^{fp} and t_j^{fp} are both potentially executable).

ii) $t_i^{fp} \in X_{ij}(G^{fp})$

Again, let $|\omega_j| > 0$ then t_j^{fp} exists and $t_j^{fp} \notin X_{ji}(G^{fp})$, because otherwise t_i^{fp} and t_j^{fp} form a ring-tuple or a channel-pair in G^{fp}. Also, $t_j^{fp} \notin P_{ji}(G^{fp})$ by arguments akin to case (i). □

Lemma 4.30

For a multi-cyclic protocol, let $G^0 \xrightarrow{\sigma}{}^{*} G$ and $fp(G, \sigma) \xrightarrow{\omega}{}^{*} G$ with $\omega_i \in suf(\sigma_i)$ for all $i \in I$, and denote by t_i^{fp} the first transition of ω_i when $|\omega_i| > 0$. If $fp(G, \sigma) \neq G$, then

i) $\exists i \in I$: $|\omega_i| > 0$;

ii) $\exists i \in I$: $|\omega_i| = 0$;

iii) $\exists i, j \in I$: $t_i^{fp} \in \Delta_{ij} \wedge |\omega_j| = 0$.

Proof:

i) Directly from the fact that $fp(G, \sigma) \neq G$;

ii) Suppose that $\forall i \in I$: $|\omega_i| > 0$, then upon reaching $fp(G, \sigma)$ at least the transition t_i^{fp} in each ω_i remains to be executed before reaching G. From Lemma 4.29, we have $\forall r \in \mathfrak{R}_{\Pi}$, $|r| = 2$: $t_1^{fp}(r) \notin \Delta_{12}(r, fp(G, \sigma)) \vee t_2^{fp}(r) \notin \Delta_{21}(r, fp(G, \sigma))$. Moreover, by repeatedly applying this lemma it also follows that $\forall r \in \mathfrak{R}_{\Pi}$, $|r| > 2$, $\exists k \in \{1, \ldots, |r|)\}$: $t_k^{fp}(r) \notin \Delta_{k\vec{k}}(r, fp(G, \sigma)) \cup \Delta_{k\overleftarrow{k}}(r, fp(G, \sigma))$, since otherwise there exists a ring-tuple or a channel-pair in $fp(G, \sigma)$ associated with (a channel in) r (cf. the proof of Proposition 4.22), or ω is not an execution

sequence from $fp(G, \sigma)$ to G. However, by the pigeon-hole principle, it must then be the case that at least one of the ω_i's is empty in $fp(G, \sigma)$. Contradiction;

iii) Similar to the proof of (ii), using the results of (i) and (ii). □

Proposition 4.31

Let G be a global state of a multi-cyclic protocol Π. If G satisfies the ring-wise equal channel length property Γ, then $G \in \mathbf{F}_{\Pi}$.

Proof: Let $G^0 \xrightarrow{\sigma}{}^* G$ and recall that $fp(G, \sigma)$ is fair reachable and satisfies Γ. Trivially, G is fair reachable if $fp(G, \sigma) = G$. We prove that $fp(G, \sigma) \neq G$ is impossible, by contradiction.

As in the previous lemmas, let $fp(G, \sigma) \xrightarrow{\omega}{}^* G$ with $\omega_i \in suf(\sigma_i)$ for all $i \in I$, and denote by t_i^{fp} the first transition of ω_i when $|\omega_i| > 0$. Since $fp(G, \sigma) \neq G$, by Lemma 4.30.(iii) there exist $i, j \in I$ such that $t_i^{fp} \in \Delta_{ij}$ and $|\omega_j| = 0$. There are two cases to consider:

i) t_i^{fp} is a send transition
Let $r \in \mathfrak{R}_{\Pi}$ such that $(i, j), (j, k) \in r$. The execution of t_i^{fp} increases the length of channel C_{ij}. However, since $|\omega_j| = 0$ the length of C_{ij} is not decreased and the length of channel C_{jk} is not increased along ω. Hence, $|c_{ij}^G| > |c_{jk}^G|$;

ii) t_i^{fp} is a receive transition
Let $r \in \mathfrak{R}_{\Pi}$ such that $(k, j), (j, i) \in r$. The execution of t_i^{fp} decreases the length of channel C_{ji}. However, $|\omega_j| = 0$ the length of C_{ji} is not increased and the length of channel C_{kj} is not decreased along ω. Hence, $|c_{ji}^G| < |c_{kj}^G|$.

Thus, in both cases there exists a ring whose channels are not all of equal length in G, which contradicts the fact that G satisfies Γ. □

Theorem 4.32

For a multi-cyclic protocol Π, $\mathbf{F}_{\Pi}$ is exactly the set of reachable global states with the ring-wise equal channel length property Γ.

Proof: Directly from propositions 4.21 and 4.31. □

Theorem 4.32 thus gives a complete characterization of the fair reachable global state space of a multi-cyclic protocol. An important implication of this characterization is that the generalized fair reachability relation is consistent with the notion of fair execution sequence in the following sense: if a global state is fair reachable, then it is fair reachable via all its execution sequences. This is stated in the next proposition.

Proposition 4.33

For a multi-cyclic protocol Π, let $G^0 \xrightarrow{\sigma}{}^* G$. $G \in \mathbf{F}_\Pi$ iff $pfs(G, \sigma) \triangleq \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$.

Proof: The "if" part follows directly from Definition 4.25. The "only-if" part is analogous to the proof of Proposition 4.31. Since G is fair reachable it satisfies the ring-wise equal channel length property Γ. Hence, it must be the case that $pfs(G, \sigma) \triangleq \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ and $fp(G, \sigma) = G$. □

Corollary 4.34

For a multi-cyclic protocol Π, let $G, H \in \mathbf{F}_\Pi$. If $G \rightarrow^* H$, then $G \xrightarrow{}_f{}^* H$. □

4.3.2 Deadlock detection by FRA

For multi-cyclic protocols with finite fair reachability graphs, the decidability of deadlock-freedom by FRA follows directly from Theorem 4.32. Each stable state and hence each deadlock state of a multi-cyclic protocol trivially satisfies the ring-wise equal channel length property Γ and is thus fair reachable. In order to detect all deadlock states it suffices then to explore the fair reachable global state space of the protocol. Clearly, any algorithm performing this task can terminate only if the number of fair reachable global states is finite (see Figure 4.2).

Corollary 4.35

Each stable state of a multi-cyclic protocol is fair reachable. □

Corollary 4.36

Deadlock-freedom for a multi-cyclic protocol Π is decidable if $\mathbf{F}_\Pi$ is finite. □

Naturally, $\mathbf{F}_\Pi$ is finite for bounded multi-cyclic protocols since $\mathbf{F}_\Pi \subseteq \mathbf{R}_\Pi$. However, since the boundedness of channels is undecidable [BZ83] (cf. Chapter 2) it is not surprising that finiteness of $\mathbf{F}_\Pi$ is also undecidable in general.

Theorem 4.37

It is undecidable whether the fair reachability graph of a multi-cyclic protocol is finite.

Proof: The proof is analogous to the one given in [LM94a] for cyclic protocols. That is, since boundedness detection is decidable for two-process protocols with finite fair reachability graphs [GH85], the adverse assumption of Theorem 4.37 would directlly imply a decision procedure for boundedness detection of two-process protocols in general [LM94a], in contradiction with the result established in [BZ83]. Finiteness of the fair reachability graph of a two-process protocol is thus undecidable. It is evident that this result then holds for multi-cyclic protocols in general. □

As pointed out in Chapter 2, it is at least of theoretical interest to identify classes of protocols for which verification problems such as deadlock-freedom are decidable [BZ83, Pac87, Fin88, Oku88, PP90, CR93, LM94, LM96]. Regarding FRA this raises the question of whether conditions exist that are necessary and sufficient for multi-cyclic protocols to have a finite fair reachability graph. We continue this section by studying the effect of bounded channels on the decidability of deadlock-freedom by FRA. Results established earlier in this respect for cyclic protocols [GH85, LM94a, LM96] are generalized to multi-cyclic protocols.

Definition 4.38

Let Π be a multi-cyclic protocol. A *ring* $r \in \mathfrak{R}_\Pi$ is *bounded* iff $\exists (i, j) \in r$: C_{ij} is bounded. □

Proposition 4.39

For a multi-cyclic protocol Π, $\mathbf{F}_\Pi$ is finite if $\forall r \in \mathfrak{R}_\Pi$: r is bounded.

Proof: Suppose that $\mathbf{F}_\Pi$ is infinite. Certainly, $F(G)$ is a finite set for any fair reachable global state G since each process has a finite number of transitions. Together with the infiniteness of $\mathbf{F}_\Pi$ this implies the existence of an *infinite* sequence of fair transition-tuples $\upsilon = Q^0 \xrightarrow{\tau_1}_f Q^1 \xrightarrow{\tau_2}_f \ldots$, such that $Q^0 = G^0$ and $\forall k, l \in \{1, 2, \ldots\}$: $Q^k \neq Q^l$ if $k \neq l$. Since each process has only finitely many states and messages, there must thus be a channel whose length increases unboundedly along υ. Let this unbounded channel be C_{ij}, with $(i, j) \in r$ for some $r \in \mathfrak{R}_\Pi$. Since the ring-wise equal channel length property is preserved along υ, it follows that r is unbounded. Contradiction. □

Proposition 4.39 implies that FRA decides deadlock-freedom for a multi-cyclic protocol if each ring in the protocol contains *at least* one bounded channel. As this condition does not preclude the existence of unbounded channels, interestingly, FRA is effective not only for bounded multi-cyclic protocols but also for certain unbounded multi-cyclic protocols. Still, the absence of unbounded rings does not give a precise characterization of these protocols since it is not a necessary condition for the finiteness of $\mathbf{F}_\Pi$. One can readily find a multi-cyclic protocol with an unbounded ring and a finite fair reachability graph. For instance, the two-process protocol in Figure 4.4 has a finite fair reachability graph although the ring $\{(1, 2), (2, 1)\}$ is unbounded. From both fair reachable global states $(\langle 11, 22\rangle, \langle \varepsilon, \varepsilon\rangle)$ and $(\langle 12, 21\rangle, \langle \varepsilon, \varepsilon\rangle)$ there exists an infinite execution sequence causing unboundedness of channel C_{21} and C_{12}, respectively. Note that there is only one deadlock state, namely the fair reachable global state $(\langle 11, 21\rangle, \langle \varepsilon, \varepsilon\rangle)$.

For a cyclic protocol, the absence of *simultaneous unboundedness* proved to be both necessary and sufficient for its fair reachability graph to be finite [LM94a, LM96]. This condition is weaker than the one in Proposition 4.39 applied to cyclic protocols, allowing in principle *all* channels to be

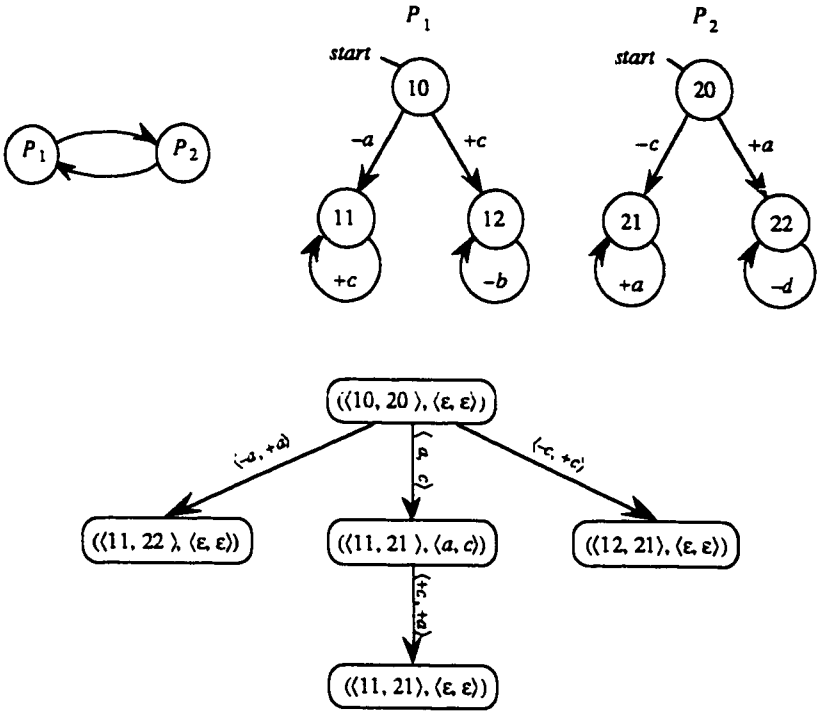


Figure 4.4 A multi-cyclic protocol with a finite fair reachability graph and an unbounded ring.

unbounded. In the rest of this section, we generalize the notion of simultaneous unboundedness to multi-cyclic protocols and show that it is no longer a necessary condition (but still sufficient) for finiteness of the fair reachability graph of a multi-cyclic protocol. For ease of comprehension, we define *weak boundedness* as the complementary notion of simultaneous unboundedness.

Definition 4.40

Let $\Pi = (P, L)$ be a multi-cyclic protocol. A ring $r \in \mathfrak{R}_\Pi$ is *weakly bounded* iff $\exists K \geq 0$ such that $\forall G \in \mathbf{R}_\Pi \exists (i, j) \in r: |c_{ij}^G| \leq K$. □

Intuitively, a ring is weakly bounded if at all times (i.e. in all reachable global states) one of its channels does not hold more than a fixed, bounded number of messages. Note that this indeed allows all channels to be unbounded, and that a ring is weakly bounded if it is bounded.

Proposition 4.41

For a multi-cyclic protocol Π, $\mathbf{F}_\Pi$ is finite if $\forall r \in \mathfrak{R}_\Pi$: r is weakly bounded.

Proof: Suppose that $\mathbf{F}_\Pi$ is infinite. Akin to the proof of Proposition 4.39, it follows that there is an infinite fair execution sequence along which the lengths of all channels in some ring $r \in \mathfrak{R}_\Pi$ increase unboundedly. Hence, r is not weakly bounded. Contradiction. □

Unfortunately, unlike for cyclic protocols, the class of multi-cyclic protocols with all rings weakly bounded does not match the class of multi-cyclic protocols with finite fair reachability graphs. This is witnessed by the protocol given in Figure 4.5, which has the same topology graph as the one in Figure 4.3 but the processes are somewhat modified. It is not difficult to see that ring $r_2 = \{(3, 4), (4, 3)\}$ is not weakly bounded while the fair reachability graph is finite.

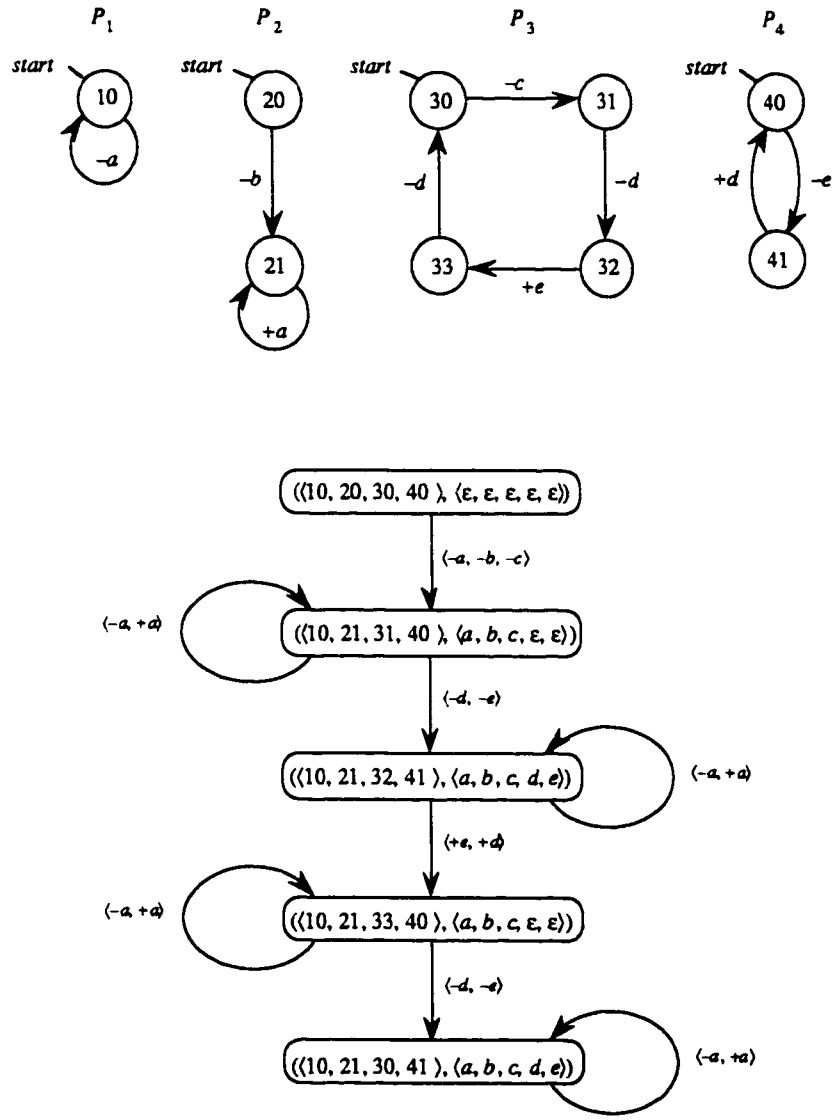


Figure 4.5 A multi-cyclic protocol with a finite fair reachability graph and a ring that is not weakly bounded.

4.4 FRA beyond multi-cyclic protocols

While striving towards a further generalization of FRA to protocols with arbitrary communication topologies, it is easily understood that the ring-wise equal channel length property satisfied by the

fair reachable global states of multi-cyclic protocols is generally too tight an invariant. For instance, for a protocol with two non-disjoint rings r_1 and r_2 (i.e. r_1 and r_2 share at least one channel and the protocol is thus not multi-cyclic) it is clear that the execution of a ring-tuple associated with r_1 at a global state where all channels in r_1 and r_2 have the same length leads to a global state in which the channels in r_2 are no longer of equal length. In this section we present an extended fair reachability relation that induces a more general invariant among the corresponding fair reachable global states of *any* protocol, irrespective of the communication topology: for each process, the total length of the contents of its incoming channels equals that of its outgoing channels. The ring-wise equal channel length property established for multi-cyclic protocols is then a special case of this so-called *I/O equilibrium property*. However, for a distinctive class of protocols we also show that not all reachable global states with the I/O equilibrium property, including deadlock states, are necessarily fair reachable. It is the apprehension of this conflicting result that leaves us to conjecture that FRA is inherently infeasible beyond multi-cyclic protocols.

4.4.1 Further generalizing the fair reachability relation

In order to deal with protocols of arbitrary topologies, we certainly need to relax the unidirectional ring concept of Definition 4.9 since there are in principle no restrictions on the link structures of these protocols. The more flexible concept of *pseudo ring* is introduced instead, which appears fundamental both in extending the fair reachability relation and in recognizing the inherently limited applicability of FRA. A pseudo ring represents a closed loop of simplex channels, like a ring, but it abstracts from the direction of the channels.

Definition 4.42

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol. A *pseudo ring* in Π is a set $\{(i_1, i_2), (i_2, i_3), \ldots, (i_{j-1}, i_j), (i_j, i_1)\}$, $2 \le j \le |I|$, such that $\forall k, l \in \{1, 2, \ldots, j\}: k \neq l \Rightarrow i_k \neq i_l$ and

$$\forall k \in \{1, 2, \ldots, j\}: (i_k, i_{\vec{k}}) \in L \vee (i_{\vec{k}}, i_k) \in L \qquad \text{if } j > 2$$

$$(i_1, i_2) \in L \wedge (i_2, i_1) \in L \qquad \text{if } j = 2$$

The set of all pseudo rings in Π is denoted by $\wp_\Pi$. □

A pseudo ring can thus be viewed as an *undirected* cycle in the undirected version of the topology graph of a protocol. Clearly, each ring is a pseudo ring but not vice versa. Examples of protocols containing pseudo rings that are not rings are depicted in Figure 4.6. The leftmost protocol consists of one pseudo ring $\{(1, 2), (2, 3), (3, 1)\}$ and no ring. The protocol in the center has three pseudo rings, viz. $\{(1, 2), (2, 3), (3, 1)\}$, $\{(1, 4), (4, 3), (3, 1)\}$ and $\{(1, 2), (2, 3), (3, 4), (4, 1)\}$. The

latter is not a ring. Finally, the protocol on the right contains as many as 11 pseudo rings, six of which are not rings.

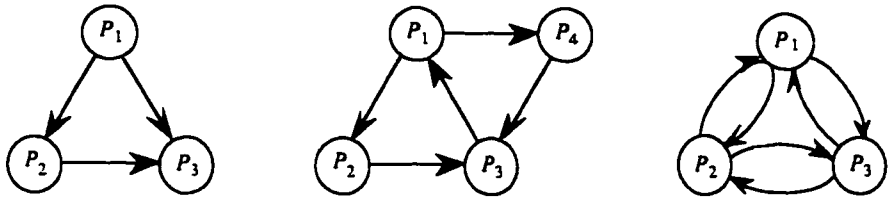


Figure 4.6 Protocols with pseudo rings that are not rings.

Based on pseudo-rings instead of rings, the next two definitions generalize Definition 4.12 and Definition 4.14, respectively.

Definition 4.43

Let G be a global state of a protocol Π. A *pseudo-ring-tuple* in G is a tuple $\langle t_k(r)\rangle_{k \in \{1,\ldots,|r|\}}$, for some $r \in \wp_\Pi$, satisfying one of the following two conditions:

- $\forall k \in \{1,\ldots,|r|\}: t_k(r) \in X_{k\vec{k}}(r, G)$;
- $\forall k \in \{1,\ldots,|r|\}: t_k(r) \in X_{k\overleftarrow{k}}(r, G)$.

□

Every pseudo-ring-tuple pertains to some pseudo ring $r \in \wp_\Pi$, containing one executable transition of each process involved in r. As opposed to ring-tuples, pseudo-ring-tuples may have a mixture of send and receive transitions provided that for each process the respective transition acts on the process' clock-wise neighbor in r, or for each process the transition acts on the counter-clock-wise neighbor in r. Channel-pairs carry over naturally without modification, i.e. they are defined as in Definition 4.13.

Definition 4.44

Let G be a global state of a protocol Π. A *generalized fair transition-tuple* in G is a pseudo-ring-tuple in G or a channel-pair in G. The set of all generalized fair transition-tuples in G is denoted by $F^*(G)$.

□

A linearization of a generalized fair transition-tuple is defined exactly as in Definition 4.11, i.e. the order of execution of the transitions in an pseudo-ring-tuple is again arbitrary. Since any two such linearizations are equivalent and lead to the same global state (cf. Proposition 4.16), the following fair reachability relation is also well-defined.

Definition 4.45

Let G and H be global states of a protocol Π. $G \xrightarrow[f^*]{} H$ iff $\exists \tau \in F^*(G)$ with $\gamma \in lin(\tau)$ such that $G \xrightarrow{\gamma}{}^* H$. This is also denoted by $G \xrightarrow[f^*]{\tau} H$. □

Definition 4.46

Let G and H be global states of a protocol Π, and denote by $\xrightarrow[f^*]{}{}^*$ the reflexive and transitive closure of $\xrightarrow[f^*]{}$. H is *(generalized) fair reachable from* G iff $G \xrightarrow[f^*]{}{}^* H$. When $G = G^0$, H is said to be *(generalized) fair reachable*. The set of all generalized fair reachable global states of Π is denoted by $\mathbf{F}^*_\Pi$. For a sequence of generalized fair transition-tuples $\upsilon = \tau_1\tau_2\ldots\tau_m$, $G \xrightarrow[f^*]{\upsilon}{}^* H$ denotes the existence of global states $Q^0,\ldots, Q^m$ such that $G = Q^0 \xrightarrow[f^*]{\tau_1} Q^1 \xrightarrow[f^*]{\tau_2} \ldots \xrightarrow[f^*]{\tau_m} Q^m = H$. □

Proposition 4.47

$\mathbf{F}_\Pi \subseteq \mathbf{F}^*_\Pi \subseteq \mathbf{R}_\Pi$

Proof: For any global state G we have $F(G) \subseteq F^*(G)$, in particular because each ring-tuple is a pseudo-ring-tuple. This implies $\mathbf{F}_\Pi \subseteq \mathbf{F}^*_\Pi$. The inclusion $\mathbf{F}^*_\Pi \subseteq \mathbf{R}_\Pi$ holds by definition of $\xrightarrow[f^*]{}{}^*$. □

Clearly, for multi-cyclic protocols we have $\mathbf{F}_\Pi = \mathbf{F}^*_\Pi$ since every pseudo ring is then a ring (and vice versa). The relation $\xrightarrow[f^*]{}{}^*$ yields the anticipated result that all generalized fair reachable global states are reachable global states in which for each process the total length of its incoming channels is equal to that of its outgoing channels.

Proposition 4.48

Let G be a global state of a protocol Π. If $G \in \mathbf{F}^*_\Pi$, then G satisfies the *I/O equilibrium property* Ψ:

$$\forall i \in I: \sum_{j,(j,i)\in L} |c^G_{ji}| = \sum_{j,(i,j)\in L} |c^G_{ij}|$$

Proof: The proof is similar to the inductive proof of Proposition 4.21. Let $G^0 \xrightarrow[f^*]{\upsilon}{}^* G$. If $|\upsilon| = 0$, then $G = G^0$ and Ψ holds trivially since all channels in G^0 are empty. As induction hypothesis suppose the claim holds for $|\upsilon| = m$, and let $G \xrightarrow[f^*]{\tau} H$ for some $\tau \in F^*(G)$. We show that the execution of τ in G preserves Ψ. Since the execution of a channel-pair does not alter the length of the channel involved (Proposition 4.21), we only need to consider the case where τ is a pseudo-ring-tuple, i.e. $\tau \in \prod_{k\in\{1,\ldots,|r|\}} X_{k\vec{k}}(r,G)$ or $\tau \in \prod_{k\in\{1,\ldots,|r|\}} X_{k\overleftarrow{k}}(r,G)$ for some $r \in \wp_\Pi$. Both these alternatives are symmetric so we regard only one. Let $\tau = \langle t_k(r)\rangle_{k\in\{1,\ldots,|r|\}} \in \prod_{k\in\{1,\ldots,|r|\}} X_{k\vec{k}}(r,G)$, then for each k there are four cases:

i) $t_{\overleftarrow{k}}(r) \in X_{\overleftarrow{k}k}(r,G)$ and $t_k(r) \in X_{k\vec{k}}(r,G)$
The execution of $t_{\overleftarrow{k}}(r)$ and $t_k(r)$ at G increases both $|c_{\overleftarrow{k}k}(r,G)|$ and $|c_{k\vec{k}}(r,G)|$, while all other channels incident to process $P_k(r)$ remain unchanged;

ii) $t_k^{\leftarrow}(r) \in X_{\overleftarrow{k}k}^{-}(r,G)$ and $t_k(r) \in X_{k\vec{k}}^{+}(r,G)$
The execution of $t_k^{\leftarrow}(r)$ and $t_k(r)$ at G increases $|c_{\overleftarrow{k}k}(r, G)|$ and decreases $|c_{\vec{k}k}(r, G)|$, while all other channels incident to process $P_k(r)$ remain unchanged;

iii) $t_k^{\rightarrow}(r) \in X_{\overrightarrow{k}k}^{+}(r,G)$ and $t_k(r) \in X_{k\vec{k}}^{-}(r,G)$
The execution of $t_k^{\rightarrow}(r)$ and $t_k(r)$ at G decreases $|c_{k\overleftarrow{k}}(r, G)|$ and increases $|c_{k\vec{k}}(r, G)|$, while all other channels incident to $P_k(r)$ remain unchanged;

iv) $t_k^{\leftarrow}(r) \in X_{\overleftarrow{k}k}^{+}(r,G)$ and $t_k(r) \in X_{k\vec{k}}^{+}(r,G)$
The execution of $t_k^{\leftarrow}(r)$ and $t_k(r)$ at G decreases both $|c_{k\overleftarrow{k}}(r, G)|$ and $|c_{\vec{k}k}(r, G)|$, while all other channels incident to process $P_k(r)$ remain unchanged.

It is not difficult to see that in each of these cases the total length of the incoming channels of process $P_k(r)$ in H still equals that of its outgoing channels. Since this holds for all k, i.e. for all processes involved in r, and since no channel incident to processes not involved in r are affected by τ, H satisfies Ψ. □

Despite this result and unlike the case for multi-cyclic protocols, FRA based on $\xrightarrow{f^*}{}^*$ is inadequate for detecting deadlocks of protocols in general. This is evidenced in the next section, where we advocate that FRA is feasible only for a characteristic class of protocols. Each reference to the notion of fair reachability, either direct or indirect, is henceforth meant in the context of $\xrightarrow{f^*}{}^*$.

4.4.2 Fair-formed protocols

The notion of pseudo ring plays an important role in identifying the boundary between protocols that are amenable and those that are not amenable to FRA. To ensure complete deadlock detection by FRA each pseudo ring in a protocol must be a ring and, in case of a protocol with prebounded channels, each ring must be such that all its channels have equal bounds (cf. Definition 4.10).

Definition 4.49
A protocol Π is *fair-formed* iff $\forall r \in \wp_\Pi: r \in \Re_\Pi \wedge \forall (i, j), (k, l) \in r: B_{ij} = B_{kl}$. □

To show that there exist indeed protocols that are not fair-formed in which not all deadlock states are fair reachable, consider the protocols in Figure 4.7. The protocol on the left is not fair-formed because the pseudo ring $\{(1, 2), (2, 3), (3, 1)\}$ is not a ring, while the protocol on the right is not fair-formed because $B_{12} \neq B_{21}$. It is easy to see that both protocols have one deadlock state (i.e. $(\langle 12, 22, 32\rangle, \langle \varepsilon, \varepsilon, \varepsilon\rangle)$ and $(\langle 14, 24\rangle, \langle \varepsilon, \varepsilon\rangle)$), but these states are not fair reachable. These two examples indicate a general problem with protocols that are not fair-formed. In a fair reachable

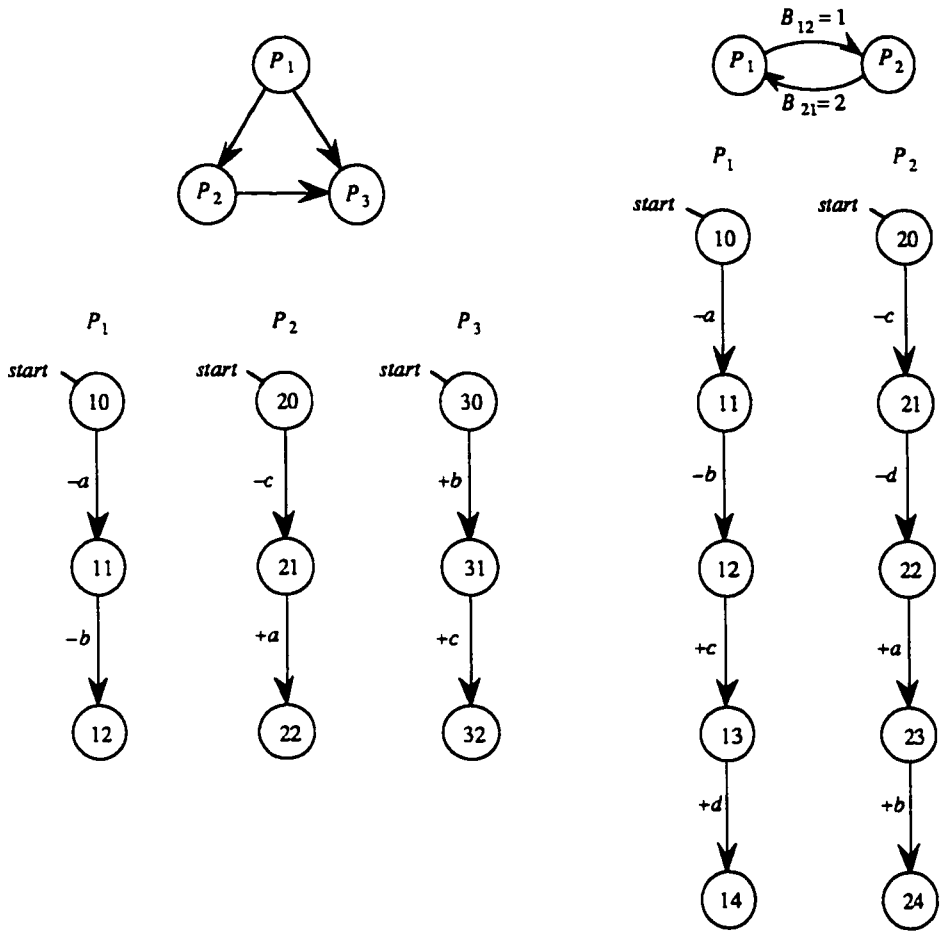


Figure 4.7 Undetected deadlocks in protocols that are not fair-formed.

global state of any such protocol all processes involved in some pseudo ring may have transitions left whose execution provides the only means to reach a particular deadlock state. However, a pseudo-ring-tuple cannot be constructed because some of the initial transitions are not yet executable but rather potentially executable. For instance, the first protocol in Figure 4.7 cannot make any "fair progress" at all because transition (30, $+b$, 31) is potentially executable at the initial global state. Similarly, the second protocol in Figure 4.7 cannot continue to make "fair progress" once it has reached the fair reachable global state ($\langle 11, 21\rangle, \langle a, c\rangle$) because transition (11, $-b$, 12) is then potentially executable. Unfortunately, there is no resolute way to anticipate if situations like this will occur. That is, static information alone is fundamentally insufficient to predict accurately the dynamic evolution of a protocol behavior.

Fair-formedness appears thus necessary in general to detect all deadlocks by FRA. Envisioning the I/O equilibrium property Ψ as the *most* general global invariant conceivable among the fair reachable global states of a protocol then justifies the following conjecture.

Conjecture 4.50

FRA is infeasible for deciding deadlock-freedom for protocols that are not fair-formed. □

The conjecture is advocated by the principle nature of FRA, forcing at each step *at least* two processes to make progress in order to preserve a global channel invariant. We advocate that the relation $\xrightarrow{f^*}$ is most general in this respect. Apart from channel-pairs, tuples of transitions which are to be considered for fair progress must at least be such that the processes involved are cyclically linked irrespective of the orientation of the links, because otherwise any intended invariant over fair reachable global states is easily violated. This requirement is indeed satisfied in the most flexible way by pseudo-ring-tuples, which allow a mixture of send and receive transitions based on the abstraction of the direction of channels in a pseudo ring. As a philosophical argument, one should observe that FRA never degrades to conventional reachability analysis except in the trivial case of a completely inactive protocol which has no executable transitions at all. Presumably, the price paid for this aptitude is the infeasibility of FRA beyond fair-formed protocols. In chapter 5 we will present a relief strategy that is indeed not as powerful as FRA for deadlock detection, but it does not put topological constraints on the protocols either.

We complete the present discussion by studying the types of link structures subsumed by fair-formedness. Certainly, the results obtained thus far indicate that the class of fair-formed protocols includes all multi-cyclic protocols. It turns out that the multi-cyclic protocols are in fact precisely those fair-formed protocols whose topology graphs are strongly connected (see Figure 4.8). We call a protocol with a strongly connected topology graph a strongly connected protocol for short.

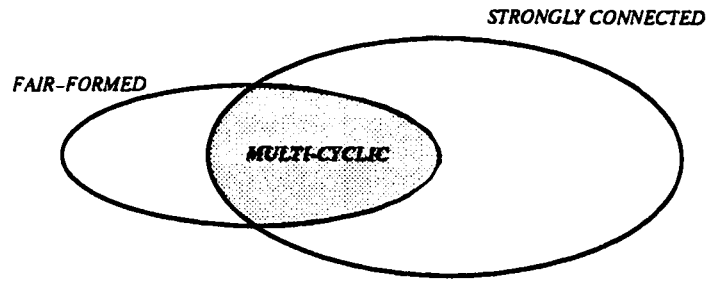


Figure 4.8 A structural classification of protocols with respect to FRA.

Proposition 4.51

A strongly connected protocol is fair-formed iff it is multi-cyclic.

Proof: Note that the second requirement for fair-formedness coincides with condition (iii) for multi-cyclic protocols in Definition 4.10. Hence, for the "if" part it remains to be shown that every

pseudo ring in a multi-cyclic protocol $\Pi = (P, L)$ is a ring. Suppose $\exists r \in \wp_\Pi: r \notin \Re_\Pi$, then $|r| \geq 3$ and $\exists k \in \{1, 2, \ldots, |r|\}: (k, \vec{k}), (k, \overleftarrow{k}) \in r$. That is, process $P_k(r)$ has no incoming channels pertaining to r. Let $P_k(r) = P_i$, $P_{\vec{k}}(r) = P_j$ and $P_{\overleftarrow{k}}(r) = P_l$, where $(i, j), (i, l) \in L$. Since Π is strongly connected there is a path p in TG_Π from vertex j to vertex i, and $r_1 = \{(i, j)\} \cup \{(v, v') \mid (v, v')$ is an edge in $p\} \in \Re_\Pi$. Similarly, there exists a path p' in TG_Π from vertex l to vertex j, and $r_2 = \{(i, l)\} \cup \{(v, v') \mid (v, v')$ is an edge in p' or $p\} \in \Re_\Pi$. Clearly, $r_1 \cap r_2 = \{(v, v') \mid (v, v')$ is an edge in $p\} \neq \varnothing$ violating condition (ii) of Definition 4.10, i.e. Π is not multi-cyclic. Contradiction.

For the "only-if" part it suffices to prove that all rings in Π are disjoint. Again by contradiction, suppose that $\exists r, r' \in \Re_\Pi: r \neq r' \wedge r \cap r' \neq \varnothing$. There are two cases to consider:

i) the incidences in $r \cap r'$ form a path in TG_Π of length $|r \cap r'|$
It follows readily that $(r \cup r') - (r \cap r')$ is a pseudo ring, but not a ring in Π. This contradicts the fact that Π is fair-formed;

ii) the incidences in $r \cap r'$ do *not* form a path in TG_Π of length $|r \cap r'|$
In this case $\exists (i, j), (k, l) \in r \cap r', i \neq k, l$: and $j \neq k, l$, such that $r \cap r'$ contains no incidence belonging to a path in TG_Π from vertex j to vertex k. Since $r, r' \in \Re_\Pi$, there must thus be two paths p and p' in TG_Π from vertex j to vertex k which have no edge in common. As a result, $\{(v, v') \mid (v, v')$ is an edge in p or $p'\}$ is a pseudo ring, but not a ring in Π. Contradiction. □

It is immediate from the proof of Proposition 4.51 that the presence of non-disjoint rings prevents *any* protocol from being fair-formed.

Corollary 4.52

A protocol Π is not fair-formed if $\exists r, r' \in \Re_\Pi: r \neq r' \wedge r \cap r' \neq \varnothing$. □

The practical significance of this result is that it can provide an efficient way to find out that a given protocol is not fair-formed and thus not suited for FRA. Of course, deciding fair-formedness in general is linear in the number of pseudo rings in the protocol, which in turn is exponential in the number of processes in the protocol in the worst case.

Examples of fair-formed protocols that are not strongly connected are illustrated in Figure 4.9. The protocols described by Figure 4.9.(a) have no pseudo rings and qualify trivially as fair-formed protocols. Note that any fair transition-tuple in a global state of such a protocol must be a channel-pair and, consequently, each fair reachable global state is a reachable global state with all channels empty. Figures 4.9.(b) and 4.9.(c) describe fair-formed protocols resulting from the composition of multi-cyclic protocols and the fair-formed protocols of Figure 4.9.(a). These types of protocols are of little practical use due to the limited end-to-end connectivity.

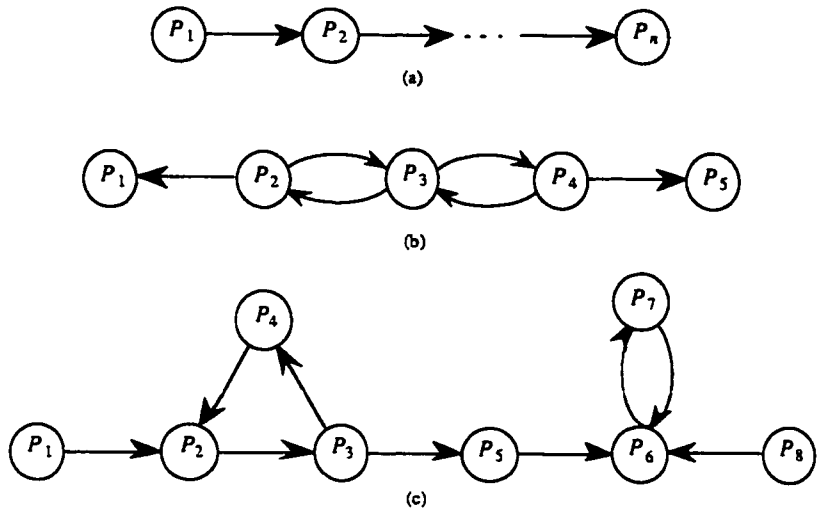


Figure 4.9 Fair-formed protocols that are not strongly connected.

4.5 FRA beyond deadlock detection

Now that we have advocated the infeasibility of FRA beyond the class of multi-cyclic (or actually fair-formed) protocols, let us return the focus to multi-cyclic protocols. In section 4.3 we showed that all the deadlock states of a multi-cyclic protocol can be detected by FRA, provided of course that the resulting fair reachability graph of the protocol is finite. A problem that remains is whether FRA can serve also as a relief strategy for the detection of other logical design errors, such as non-executable transitions and unspecified receptions. It appears that the fair reachability graph of a multi-cyclic protocol is by itself not sufficient for this purpose, primarily because it does not guarantee the exposure of all reachable process states (i.e. those process states that occur in some reachable global state) of the different processes in the protocol. This was already observed for two-process protocols by Gouda & Han [GH85]. Yet, they showed that for a two-process protocol with a finite fair reachability graph, all the reachable process states can in fact be uncovered through a finite extension of this graph. In [LM94b, LM96], Liu & Miller extended that result to general cyclic protocols. Based on the idea presented in [GH85], they devised an extension procedure to solve the following two reachability problems for the class of cyclic protocols with a finite fair reachability graph:

P-I Given a process state s of a process P_i, determine whether s is reachable, and

P-II Given a process state s of a process P_i and a message m from some process P_j to P_i, determine whether (s, m) is reachable.

Precisely, problem **P-I** amounts to checking whether there exists a reachable global state G such that $s_i^G = s$, and problem **P-II** amounts to checking whether there exists a reachable global state G such that $s_i^G = s$ and $front(c_{ji}^G) = m$. The relevance of these two reachability problems in the context of cyclic protocols is that for every such protocol Π with a finite fair reachability graph it holds that:

i) the detection of unboundedness is decidable for Π if **P-I** is decidable for Π,

ii) the detection of unspecified receptions is decidable for Π if **P-II** is decidable for Π, and

iii) the detection of non-executable transitions is decidable for Π if **P-I** and **P-II** are decidable for Π.

The validity of (ii) and (iii) follows directly from the definitions of an unspecified reception and a non-executable transition in Chapter 2. Note that they hold for all protocols, although (iii) actually relies on the implicit assumption that the simplex channels in a protocol are not prebounded (see Section 2.5.1 and footnote 4). Indeed, the executability of any send transition $(s, -m, s')$ is then independent of the channel contents (a channel is never full), and hence it is executable if process state s is reachable. The validity of (i) stems from the property that a cyclic protocol Π with a finite fair reachability graph is unbounded if and only if at least one of its processes has a reachable sending cycle, which was proven in [LM94b, LM96]. A reachable sending cycle of a process P_i is defined as a cycle of all send transitions in the process graph of P_i, such that one of the process states in this cycle is reachable. Hence, deciding unboundedness for Π reduces to deciding the existence of a reachable sending cycle for Π, which in turn reduces to deciding **P-I** for Π.

For cyclic protocols, FRA has thus already proved useful as a relief strategy beyond deadlock detection. That is, given a cyclic protocol with a finite fair reachability graph (and no prebounded channels), one can "cleverly" extend this graph using the procedure proposed in [LM94b, LM96] to uncover all reachable process states s and all reachable process state/message pairs (s, m), and hence to detect unboundedness and all non-executable transitions and all unspecified receptions. The ensuing question is then whether the same can be accomplished for multi-cyclic protocols in general. In particular, can an extension procedure be devised for the class of multi-cyclic protocols that have a finite fair reachability graph? Unfortunately, we have not succeeded to get to such a result. Nevertheless, in order to disclose our efforts, it is illustrated in this section that the extension procedure proposed in [LM94b, LM96] seems unsuited for generalization to multi-cyclic protocols, due to the possible interaction dependencies that may arise among processes in different rings in these protocols. (In contrast to a cyclic protocol, a multi-cyclic protocol can have "connector" processes with multiple input and output channels that link processes in different rings). We will unfold our findings by discussing an attempt to adapt the extension procedure for cyclic protocols

to the subclass of multi-cyclic protocols whose topology is a daisy-chain (see Figure 4.1).

Before we address in further detail the work in [LM94b, LM96], let us forthwith establish that for multi-cyclic protocols the decidability of the above reachability problem **P-I** does not imply the decidability of boundedness detection, unlike for cyclic protocols. The class of cyclic protocols that are bounded equals the class of cyclic protocols with a finite fair reachability graph that have no reachable sending cycle [GH85, LM94b, LM96], but the same does not hold for multi-cyclic protocols in general. This is formulated as a proposition below.

Proposition 4.53

For the class of multi-cyclic protocols with a finite fair reachability graph, the absence of reachable sending cycles is not a sufficient condition for boundedness.

Proof: The multi-cyclic protocol in Figure 4.10, with $a \in M_{12}$, $b \in M_{21}$, $c \in M_{23}$ and $d \in M_{32}$, proves the proposition. Indeed, one can readily check that the fair reachability graph of this protocol is finite. It consists of 7 fair reachable global states and 7 global state transitions (i.e. fair transition-tuples). Clearly, there is no reachable sending cycle, but yet the protocol is unbounded as message c can be sent infinitely many times without being received (i.e. channel C_{23} is unbounded). □

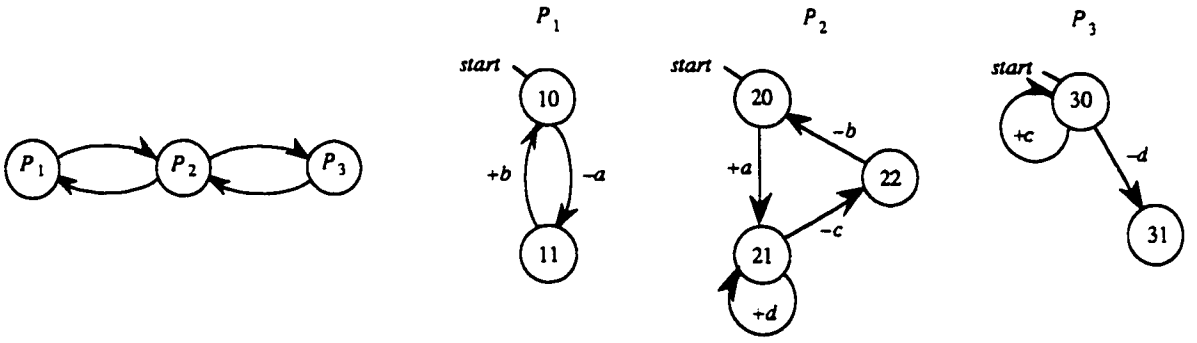


Figure 4.10 An unbounded multi-cyclic protocol with a finite fair reachability graph and no reachable sending cycle.

Thus, even if one were to devise a procedure that decides **P-I** for a multi-cyclic protocol by finite extension of its fair reachability graph, this would not enable boundedness detection for multi-cyclic protocols. Solving **P-I** and **P-II** is of course still effective for detecting non-executable transitions and unspecified receptions.

In what follows, we adopt the presentation in [LM94b, LM96] to capture the reasoning behind the extension procedure proposed for cyclic protocols, and to interpret this reasoning for daisy-chain protocols. For ease of comprehension, and conforming to Definition 4.10 of a multi-cyclic protocol (whose simplex channels are not prebounded), we define cyclic and daisy-chain protocols

explicitly as follows. A cyclic protocol is a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ with $L = \{(i, \vec{i}) \mid i \in I\}$ (cf. Notation 4.8), and a daisy-chain protocol is a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ with $L = \{(i, i+1) \mid i \in I \setminus \{n\}\} \cup \{(i, i-1) \mid i \in I \setminus \{1\}\}$ (see Figure 4.11). In the remainder of this section, when we refer to a cyclic, a daisy-chain or a general multi-cyclic protocol, it is implicitly assumed that the fair reachability graph of the protocol is finite, as in [LM94b, LM96].

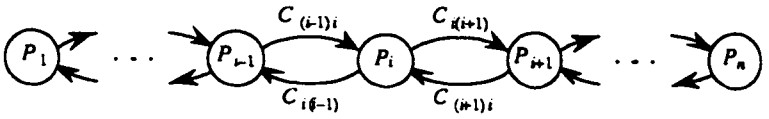


Figure 4.11 Structure of a daisy-chain protocol.

4.5.1 FRA plus finite extension: the conceptual idea

As discussed, the purpose of extending the fair reachability graph of a multi-cyclic protocol Π is to detect all reachable process states and process state/message pairs that are not already detected within $\mathbf{F}_\Pi$. There is an intuitive argument that shows that a finite extension of $\mathbf{F}_\Pi$ is indeed sufficient to accomplish this [LM94b, LM96]. Suppose s is a reachable process state, then there is at least one reachable global state $G \in \mathbf{R}_\Pi$ that contains s (i.e. $s = s_j^G$ for some $j \in I$). Let $\sigma \triangleq \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ be an execution sequence from the initial global state G^0 to G. A partial fair execution sequence $\upsilon \triangleq \{\upsilon_1, \upsilon_2, \ldots, \upsilon_n\}$ for G wrt σ can then be derived that leads to the corresponding fair precursor $fp(G, \sigma)$ (see Definition 4.25). Clearly, s is detected within $\mathbf{F}_\Pi$ if $fp(G, \sigma)$ contains s, since $fp(G, \sigma) \in \mathbf{F}_\Pi$. Yet, if $fp(G, \sigma)$ does not contain s, then there remains a non-empty but finite execution sequence from $fp(G, \sigma)$ to G, namely the sequence $\omega \triangleq \{\omega_1, \omega_2, \ldots, \omega_n\}$ such that $\forall i \in I$: $\sigma_i = \upsilon_i \omega_i$. We thus have $G^0 \xrightarrow{\sigma}{}^* G$ as well as $G^0 \xrightarrow[f]{\upsilon}{}^* fp(G, \sigma) \xrightarrow{\omega}{}^* G$, and process state s is therefore detectable by finite extension from $\mathbf{F}_\Pi$. A similar argument applies for a reachable process state/message pair (s, m). As in [LM94b, LM96], we will henceforth focus on the reachability of process states only, for the sake of simplicity and in view of the fact that **P-I** and **P-II** are cognate problems.

Although intuitive, Liu & Miller pointed out that the above argument is not immediately practical because it does not provide an upper bound on how far to extend $\mathbf{F}_\Pi$ when the execution sequence σ leading to s is unknown, which is generally the case [LM94b, LM96]. In other words, it shows only the existence of a finite extension from $\mathbf{F}_\Pi$ leading to s, but it does not suggest an algorithm to actually perform the extension. For cyclic protocols, the argument can yet be developed further to yield a more practical condition which is necessary for the existence of reachable process states that cannot be detected within $\mathbf{F}_\Pi$.

Persistent proper incompatible transition vectors

For a cyclic protocol Π, let s be a reachable process state of a process P_j that is not fair reachable, i.e. none of the reachable global states containing s is in $\mathbf{F}_\Pi$. As before, let G be any reachable global state with $s_j^G = s$, $G^0 \xrightarrow{\sigma}{}^* G$ and $G^0 \xrightarrow{\upsilon_f}{}^* fp(G, \sigma) \xrightarrow{\omega}{}^* G$. Since s is not fair reachable, it must be the case that $s \neq s_j^{fp}$ (s_j^{fp} denotes the process state of P_j in $fp(G, \sigma)$) and thus $|\omega_j| > 0$. Therefore, one can find an *interval* $[i...j]$ (an ordered set) in $fp(G, \sigma)$ of at most n–1 *consecutive* process indices $i, \vec{i}, \ldots, \overleftarrow{j}, j$ such that $|\omega_{\overleftarrow{i}}| = 0$ and $|\omega_l| > 0$ for all $l \in [i...j]$ (by Lemma 4.30.(ii)). The execution of the transitions in ω_j can then depend only on the execution of the transitions in $\omega_i, \omega_{\vec{i}}, \ldots, \omega_{\overleftarrow{j}}$. Starting with $fp(G, \sigma)$, a specific subset of the global states fair reachable from $fp(G, \sigma)$ can be conceived in the following way: in each such global state H, execute only those fair transition-tuples $\tau = \langle t_1, t_2, \ldots, t_m \rangle \in F(H)$ for which it holds true that t_l ($l \in \{1, \ldots, m\}$) is the transition of ω_l at H if $l \in [i...j]$ and there are still transitions of ω_l left to be executed at H. Note that $\omega_i, \omega_{\vec{i}}, \ldots, \omega_{\overleftarrow{j}}$ may indeed become empty during this (conceptual) construction. However, since process state s of P_j is not fair reachable, ω_j does not become empty and it can hence be assumed without loss of generality that none of the ω_l's become empty (in the case that one of them were to become empty during the construction, one would simply continue to reason with a smaller interval ending with j). Denote by $\mathbf{F}^{\min}_{[i...j]}$ the subset of global states resulting from the construction at which the sum of the remaining transitions in $\omega_i, \omega_{\vec{i}}, \ldots, \omega_j$ is minimal. It follows that $\emptyset \subset \mathbf{F}^{\min}_{[i...j]} \subseteq \mathbf{F}_\Pi$ and that $\mathbf{F}^{\min}_{[i...j]}$ is closed under the above construction: if $H \in \mathbf{F}^{\min}_{[i...j]}$ and H' is fair reachable from H by this construction, then $H' \in \mathbf{F}^{\min}_{[i...j]}$. Fundamentally, this means that H' is fair reachable from H *without progress in* the interval $[i...j]$, an observation which brings about the notion of so-called (persistent) proper incompatible transition vectors. Given a global state $H \in \mathbf{F}^{\min}_{[i...j]}$, let $\tau_{[i...j]} = \langle t_i, t_{\vec{i}}, \ldots, t_j \rangle$ be the transition vector (or tuple in our terminology) such that t_l is the transition of ω_l at H for each $l \in [i...j]$. The following four properties can then be shown to hold [LM94b, LM96]:

i) all transitions in $\tau_{[i...j]}$ are executable at H (i.e. $\forall l \in [i...j]$: $t_l \in X_l(H)$);

ii) $t_i \in X_{i\overleftarrow{i}}(H)$ or $t_j \in X_{j\vec{j}}(H)$ (note that $X_{i\overleftarrow{i}}(H) = X^+_{i\overleftarrow{i}}(H)$ and $X_{j\vec{j}}(H) = X^-_{j\vec{j}}(H)$ for cyclic protocols);

iii) no transition in $\tau_{[i...j]}$ occurs in any fair transition-tuple (i.e. channel-pair or ring-tuple) in H;

iv) if H' is fair reachable from H without progress in $[i...j]$, then $\tau_{[i...j]}$ still satisfies properties (i), (ii) and (iii) in H'.

Accordingly, for *any* global state G and contiguous interval $[i...k]$, a vector $\tau_{[i...k]} = \langle t_i, t_{\vec{i}}, \ldots, t_k \rangle$ of transitions defined at G is said in general to be a *proper incompatible transition vector* (pitv) in G

iff properties (i), (ii) and (iii) hold wrt G and $[i...k]$. It is said to be a *persistent* pitv (ppitv) in G iff properties (i), (ii), (iii) and (iv) hold wrt G and $[i...k]$. Denote by $U^p_{[i...k]}(G)$ the set of ppitv's in G wrt $[i...k]$, and remark in particular that $U^p_{[i...j]}(H) \neq \emptyset$ for every $H \in \mathbf{F}^{\min}_{[i...j]}$ [LM94b, LM96].

Example 4.54

Consider the 3-process cyclic protocol with corresponding fair reachability graph in Figure 4.12, where $a \in M_{12}$, b, c, d, e, $f \in M_{23}$ and g, $h \in M_{31}$. It is not difficult to check that transition vector $\langle(10, -a, 11)\rangle$ is a pitv in the initial global state, but not a ppitv since it is no longer a pitv in global state $(\langle 10, 21, 31\rangle, \langle \varepsilon, \varepsilon, \varepsilon\rangle)$, which is fair reachable from the initial global state without progress in interval [1]. Transition vector $\langle(23, -e, 24), (33, -h, 34)\rangle$ is indeed a ppitv in the fair reachable global state $(\langle 11, 23, 33\rangle, \langle a, d, g\rangle)$. □

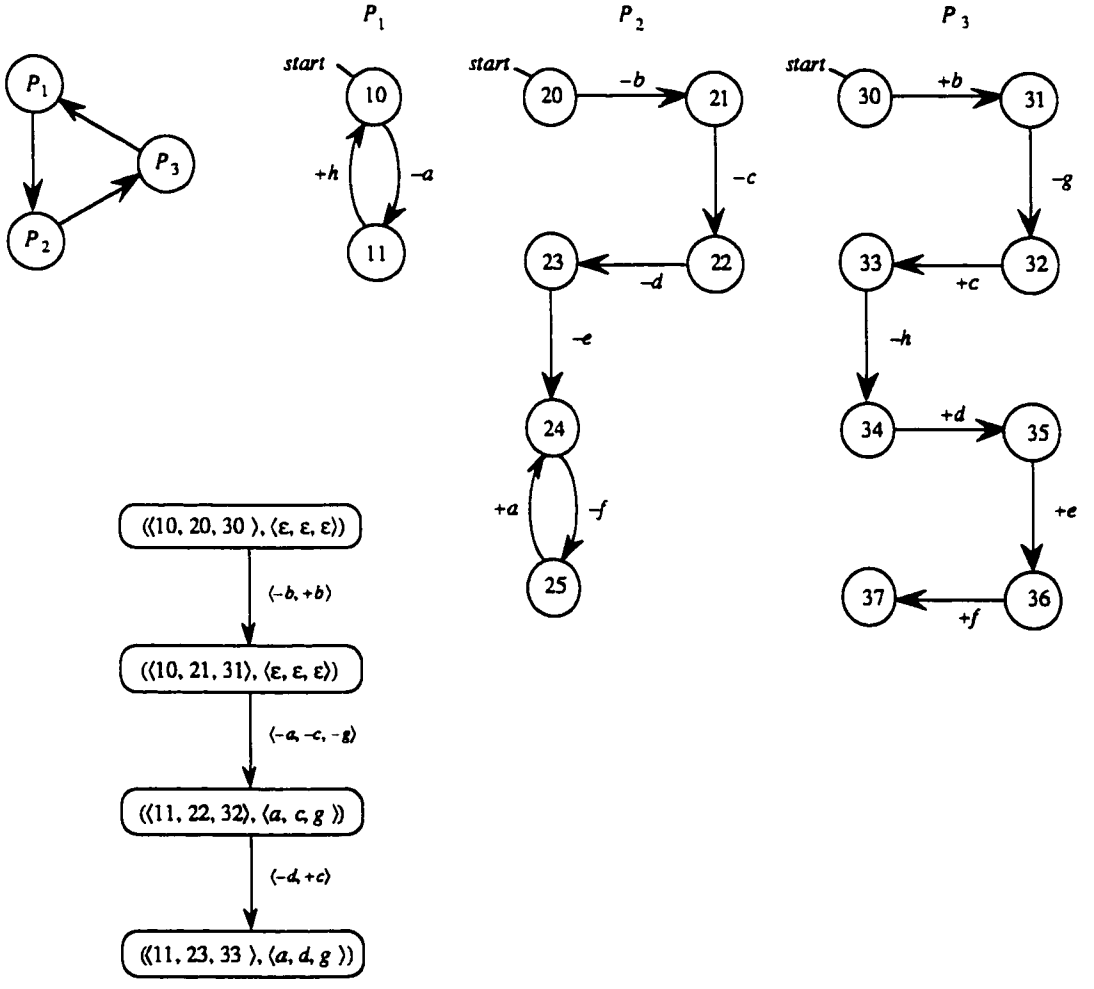


Figure 4.12 The cyclic protocol of Example 4.54.

From this intricate argumentation it can be concluded that a process state s of a process P_j in a cyclic protocol is reachable but not fair reachable only if there exists a fair reachable global state G from which s is reachable and there is an interval $[i...k]$ with $j \in [i...k]$ such that $U^p_{[i...k]}(G) \neq \emptyset$ [LM94b, LM96]. For instance, the process states 24, 25, 34, 35, 36 and 37 of the protocol in Figure 4.12 are reachable but not fair reachable, and indeed they can all be reached from the fair reachable global state $G = (\langle 11, 23, 33\rangle, \langle a, d, g\rangle)$ with $U^p_{[2,3]}(G) = \{\langle (23, -e, 24), (33, -h, 34)\rangle\} \neq \emptyset$. For a cyclic protocol Π, the existence of a fair reachable global state with a ppitv thus arises as a necessary condition for the existence of a reachable process state that cannot be detected within $\mathbf{F}_\Pi$.

Let us now translate the discussion so far in quest of a similar result for daisy-chain protocols. Given a daisy-chain protocol Π, consider the same scenario as above with s a reachable process state of a process P_j that is not fair reachable, and G a reachable global state such that $s^G_j = s$, $G^0 \xrightarrow{\sigma}{}^* G$ and $G^0 \xrightarrow{\upsilon}{}^*_f fp(G, \sigma) \xrightarrow{\omega}{}^* G$. Again, $s \neq s^{fp}_j$ and thus $|\omega_j| > 0$ because s is not fair reachable. In this case, using Lemma 4.30.(ii), one can find an interval $[i...j...k]$ in $fp(G, \sigma)$ of at most $n-1$ consecutive process indices $i, i+1, \ldots, j, j+1, \ldots, k$ ($i \leq j \leq k$) such that $|\omega_l| > 0$ for each $l \in [i...j...k]$, and one of the following holds true:

- $i = 1 \wedge k < n \wedge |\omega_{k+1}| = 0$,
- $k = n \wedge i > 1 \wedge |\omega_{i-1}| = 0$, or
- $i > 1 \wedge k < n \wedge |\omega_{i-1}| = |\omega_{k+1}| = 0$.

Thus, the execution of the transitions in ω_j can depend only on the execution of the transitions in $\omega_i, \omega_{i+1}, \ldots, \omega_{j-1}, \omega_{j+1}, \ldots, \omega_k$. In the exact same way as for cyclic protocols, one can conceptualize the set $\mathbf{F}^{\min}_{[i...j...k]}$ of global states H fair reachable from $fp(G, \sigma)$ such that the sum of the remaining transitions in $\omega_i, \ldots, \omega_j, \ldots, \omega_k$ is minimal at H. It is again not hard to see that $\emptyset \subset \mathbf{F}^{\min}_{[i...j...k]} \subseteq \mathbf{F}_\Pi$ and, if H' is fair reachable from $H \in \mathbf{F}^{\min}_{[i...j...k]}$ without progress in $[i...j...k]$, then $H' \in \mathbf{F}^{\min}_{[i...j...k]}$. We arrive at the following lemma.

Lemma 4.55

For a daisy-chain protocol, let $H \in \mathbf{F}^{\min}_{[i...j...k]}$ and $\tau_{[i...j...k]} = \langle t_i, \ldots, t_j, \ldots, t_k\rangle$ the transition vector such that t_l is the transition of ω_l at H for each $l \in [i...j...k]$ (with $[i...j...k]$, $\mathbf{F}^{\min}_{[i...j...k]}$ and $\omega_i, \ldots, \omega_j, \ldots, \omega_k$ as above). The following four properties hold:

i) each transition in $\tau_{[i...j...k]}$ is either executable at H, or it is potentially executable but not enabled at H (i.e. $\forall l \in [i...j...k]$: $t_l \in X_l(H) \cup P_l(H) \backslash E_l(H)$);

ii) $t_i \in X_{i(i-1)}(H)$ or $t_k \in X_{k(k+1)}(H)$;

iii) no transition in $\tau_{[i...j...k]}$ occurs in any fair transition-tuple in H;

iv) if H' is fair reachable from H without progress in $[i...j...k]$, then $\tau_{[i...j...k]}$ still satisfies properties (i), (ii) and (iii) in H'.

Proof: Property (iii) is obvious by virtue of $\mathbf{F}^{\min}_{[i...j...k]}$, i.e. the transitions in $\tau_{[i...j...k]}$ cannot be utilized within $\mathbf{F}^{\min}_{[i...j...k]}$ since otherwise there would be a global state in $\mathbf{F}^{\min}_{[i...j...k]}$ at which the sum of the remaining transitions in $\omega_i,\ldots,\omega_j,\ldots,\omega_k$ is not minimal. Property (iv) is then also immediate, provided that (i) and (ii) are satisfied. Property (i) holds by the simple observation that if there were some transition $t_l \notin X_l(H) \cup P_l(H)\backslash E_l(H)$ in $\tau_{[i...j...k]}$, then either t_l is enabled at H and thus occurs in a channel-pair in H, which violates property (iii), or t_l is not executable nor potentially executable (and hence not enabled since $E_l(H) \subseteq P_l(H)$) at H, which would imply that process P_l cannot proceed along ω_l from $fp(G, \sigma)$ to G, in contradiction with the assumption underlying $\mathbf{F}^{\min}_{[i...j...k]}$ that $fp(G, \sigma) \xrightarrow{\omega}{}^{*} G$. To show property (ii), recall that $|\omega_{i-1}| = |\omega_{k+1}| = 0$ when $i > 1$ and $k < n$. Hence, $t_i \notin P_{i(i-1)}(H)$ and $t_k \notin P_{k(k+1)}(H)$ again because otherwise process P_i or P_k cannot proceed along respectively ω_i and ω_k from $fp(G, \sigma)$ to G. As a consequence, $t_i \in X_{i(i-1)}(H)$ or $t_k \in X_{k(k+1)}(H)$ (i.e. (ii) holds), or both $t_i \in X_{i(i+1)}(H) \cup P_{i(i+1)}(H)\backslash E_{i(i+1)}(H)$ and $t_k \in X_{k(k-1)}(H) \cup P_{k(k-1)}(H)\backslash E_{k(k-1)}(H)$. It is not difficult to derive that the latter case is impossible, since otherwise $\langle t_l, t_{l+1}\rangle$ is a fair transition-tuple for some $l \in [i...j...k-1]$, violating property (iii), or once again ω is not an execution sequence from $fp(G, \sigma)$ to G (cf. the proof of Lemma 4.29). □

As one can see, properties (ii), (iii) and (iv) in Lemma 4.55 coincide with the aforementioned conditions (ii), (iii) and (iv) on ppitv's for cyclic protocols (listed just ahead of Example 4.54). Condition (i) on these ppitv's is stronger than Lemma 4.55.(i), and it appears indeed too strong in the context of daisy-chain protocols. This is witnessed by Example 4.56 below for the daisy-chain protocol in Figure 4.13, which draws a concrete scenario where a transition vector pertaining to some interval $[i...k]$ and a global state $H \in \mathbf{F}^{\min}_{[i...k]}$ contains a transition that is potentially executable rather than executable at H. For cyclic protocols, such a scenario simply proves to be impossible: all transitions in a transition vector associated with a global state $H \in \mathbf{F}^{\min}_{[i...k]}$ are guaranteed to be executable at H in case of a cyclic protocol (the related proof in [LM94b, LM96] testifies that this stems in fact from the specific "single input/output channel" characteristic of each process in a cyclic protocol). Yet, this means that replacing the current condition (i) on ppitv's by the condition stated in Lemma 4.55.(i) does not distort the notion of ppitv's for cyclic protocols. We hence redefine ppitv's as follows. For a cyclic or daisy-chain protocol $\Pi = (\{P_i \mid i \in I\}, L)$, let G be any global state and $[i...k]$ a contiguous interval of process indices in I. A vector $\tau_{[i...k]} = \langle t_i, \ldots, t_k\rangle \in \prod_{l\in[i...k]}\Delta_l$ of transitions defined at G is a ppitv in G iff it satisfies Lemma 4.55.(i), (ii), (iii) and (iv). The set of ppitv's in G wrt $[i...k]$ is denoted as before by $U^{p}_{[i...k]}(G)$. This modified definition thus

extends the notion of ppitv's from cyclic to daisy-chain protocols[5]. As a result, the necessary condition for the existence of a reachable but not fair reachable process state in a cyclic protocol remains valid in the case of a daisy-chain protocol, as stated by Proposition 4.57.

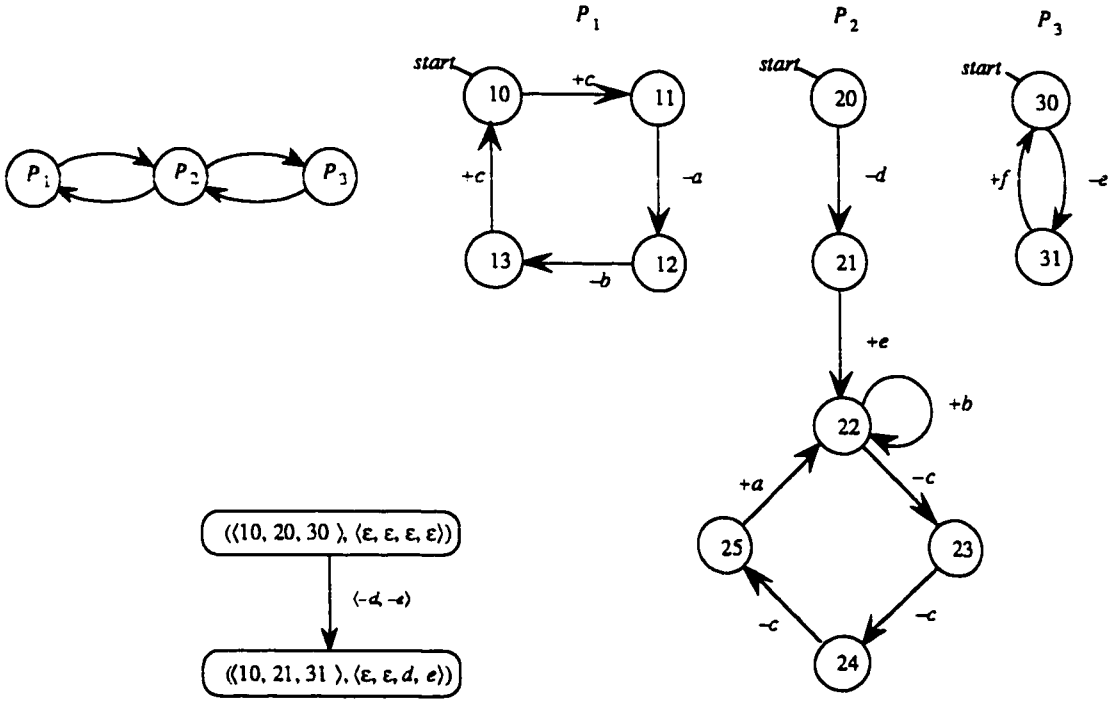


Figure 4.13 The daisy-chain protocol of Example 4.56.

Example 4.56

The fair reachability graph of the daisy-chain protocol in Figure 4.13, with $a, b \in M_{12}$, $c \in M_{21}$, $d, f \in M_{23}$ and $e \in M_{32}$, is depicted at the bottom of the figure. In the fair reachable global state $(\langle 10, 21, 31\rangle, \langle \varepsilon, \varepsilon, d, e\rangle)$ no fair transition-tuple can be formed from the transitions (10, +*c*, 11), (21, +*e*, 22) and (31, +*f*, 30) defined at this state, essentially because process P_2 wants to make fair progress with process P_3 which is permanently blocked. (Transition (31, +*f*, 30) is non-executable as it specifies the reception of a message which is incompatible with the message at the front of channel C_{23}.) Consequently, reachable process states 22, 23, 24 and 25 of P_2 as well as the reachable process states 11, 12 and 13 of P_1 are not detected by FRA. It is not difficult to see that any extension of the fair reachability graph to reveal in particular the process states of P_1 must

[5] We strongly believe that the revised definition of ppitv's applies similarly to multi-cyclic protocols in general. However, for an arbitrary multi-cyclic protocol the conception of intervals of adjacent processes that satisfy conditions akin to the intervals devised for cyclic and daisy-chain protocols is puzzling (such intervals do exist by Lemma 4.30).

capture the execution of transition (10, +c, 11), which is potentially executable at both fair reachable global states in the graph.

Indeed, following the argumentation setting up Lemma 4.55, consider the (shortest) execution sequence $\sigma \triangleq \{\sigma_1, \sigma_2, \sigma_3\} = \{(10, +c, 11), (20, -d, 21)\ (21, +e, 22)\ (22, -c, 23), (30, -e, 31)\}$ that leads P_1 from the initial global state G^0 to process state 11. Thus, $G^0 \xrightarrow{\sigma}{}^* G = (\langle 11, 23, 31\rangle, \langle \varepsilon, \varepsilon, d, \varepsilon\rangle)$ and $\upsilon \triangleq \{\upsilon_1, \upsilon_2, \upsilon_3\} = \{\varepsilon, (20, -d, 21), (30, -e, 31)\}$ is the partial fair execution sequence for G wrt σ, yielding $fp(G, \sigma) = (\langle 10, 21, 31\rangle, \langle \varepsilon, \varepsilon, d, e\rangle)$ and $G^0 \xrightarrow[f]{\upsilon}{}^* fp(G, \sigma) \xrightarrow{\omega}{}^* G$ with $\omega \triangleq \{\omega_1, \omega_2, \omega_3\} = \{(10, +c, 11), (21, +e, 22)\ (22, -c, 23), \varepsilon\}$. It follows that [1, 2] is the interval referred to in Lemma 4.55, and $\mathbf{F}^{min}_{[1,2]} = \{fp(G, \sigma)\}$. As a result, the (only) transition vector associated with ω and $\mathbf{F}^{min}_{[1,2]}$ is $\tau_{[1,2]} = \langle (10, +c, 11), (21, +e, 22)\rangle$, containing a potentially executable transition at $fp(G, \sigma) \in \mathbf{F}^{min}_{[1,2]}$. □

Proposition 4.57

If a process state s of a process P_j in a daisy-chain protocol Π is reachable but not fair reachable, then there exist a global state $G \in \mathbf{F}_\Pi$ and an interval $[i...k]$ such that s is reachable from G, $j \in [i...k]$ and $U^p_{[i...k]}(G) \neq \emptyset$.

Proof: The proposition follows directly from the definition of ppitv's in terms of the properties proven in Lemma 4.55. That is, for any execution sequence σ that leads P_j from its initial process state to s one can find a non-empty set $\mathbf{F}^{min}_{[i...j...k]} \subseteq \mathbf{F}_\Pi$, as formulated earlier. For each $H \in \mathbf{F}^{min}_{[i...j...k]}$ it holds that s is reachable from H and $U^p_{[i...j...k]}(H) \neq \emptyset$. □

The extension set within $\mathbf{F}_\Pi$

The necessary condition in Proposition 4.57 for the existence of a reachable but not fair reachable process state in a cyclic or daisy-chain protocol Π implies that the required finite extension of $\mathbf{F}_\Pi$ to solve reachability problem **P-I** (and similarly **P-II**) can be confined to the subset $\mathbf{F}^p_\Pi$ of $\mathbf{F}_\Pi$, where $\mathbf{F}^p_\Pi = \{G \in \mathbf{F}_\Pi \mid U^p_{[i...k]}(G) \neq \emptyset$ for some interval $[i...k]\}$. However, as Liu & Miller argued, there are still two concerns with this "extension set" [LM94b, LM96]. First, deciding membership of $\mathbf{F}^p_\Pi$ can be costly since checking whether $U^p_{[i...k]}(G) \neq \emptyset$ involves tracking all global states that are fair reachable from G without progress in $[i...k]$. This can be done while constructing $\mathbf{F}_\Pi$ (i.e. during FRA), but not without significant overhead. Secondly, as $\mathbf{F}^p_\Pi$ may still be quite large, extending all global states in $\mathbf{F}^p_\Pi$ can be costly as well and might even incur considerable redundant work. Liu & Miller tackled these concerns by showing that, for a cyclic protocol Π, the extension of $\mathbf{F}_\Pi$ can in fact be confined to an alternative subset of $\mathbf{F}_\Pi$ that is readily computed during the construction of $\mathbf{F}_\Pi$ and that is often much smaller than $\mathbf{F}_\Pi$ [LM94b, LM96].

Specifically, they established that it is sufficient to consider for extension only those states $G \in \mathbf{F}_\Pi$ that satisfy any of the following three conditions:

1) G is an unspecified reception state,

2) a successor of G (in $\mathbf{R}_\Pi$) is an unspecified reception state, or

3) a process state in G is in a cycle of all send transitions in the corresponding process graph.

(Note that (2) can be detected at G, by checking whether there is a send transition $(s_i^G, -m, s) \in \Delta_{ij}$ such that $c_{ij}^G = \varepsilon$ and $(s_j^G, +m, s') \notin \Delta_{ji}$.) It turns out that the same result cannot be derived for daisy-chain protocols, mainly because condition (3) emerges from the fact that the existence of a reachable sending cycle is a necessary (and sufficient) condition for a cyclic protocol (with a finite fair reachability graph) to be unbounded. It is not a necessary condition for a daisy-chain protocol to be unbounded, as shown earlier by the protocol in Figure 4.10 (see Proposition 4.53).

Despite being unable to find a more "efficient" extension set for daisy-chain protocols, it should be clear from Proposition 4.57 that there is no harm in proceeding with the extension set $\mathbf{F}_\Pi^p$, at least not from a conceptual standpoint. Hence, in our continued attempt to adapt the extension procedure for cyclic protocols to daisy-chain protocols, we will interpret this procedure as if it were defined in terms of $\mathbf{F}_\Pi^p$ instead of the actual extension set induced by the three conditions above.

4.5.2 FRA plus finite extension: the procedure (for cyclic protocols)

As just explained, in order to find all reachable process states of a cyclic or daisy-chain protocol by finite extension of its fair reachability graph, it is sufficient to extend the fair reachable global states that have one or more ppitv's. Moreover, the characteristics of ppitv's testify that for each such $G \in \mathbf{F}_\Pi^p$ one needs to consider only the *partial states* of G that are indexed by *maximal* intervals $[i...k]$ for which $U_{[i...k]}^p(G) \neq \emptyset$ [LM94b, LM96]. More precisely, when $U_{[i...k]}^p(G) \neq \emptyset$, the interval $[i...k]$ is regarded as maximal if there is no interval $[i'...k']$ such that $i' \leq i$, $k' \geq k$ and $U_{[i'...k']}^p(G) \neq \emptyset$. The partial state of G indexed by $[i...k]$, denoted by $G_{[i...k]}$, is obtained from G by keeping only the process states and *input* channel contents of G of the processes P_l with $l \in [i...k]$. Note that $G_{[1...n]} = G$. Keeping in $G_{[i...k]}$ only the input channel contents from G for the processes P_l with $l \in [i...k]$ is justified by the simple fact that any further progress of these processes is independent of the contents of their output channels when these channels are not prebounded (as assumed in [LM94b, LM96] and at the beginning of this section). Accordingly, a reachability relation among partial states of a cyclic protocol was defined in [LM94b, LM96] as the basis for the proposed extension procedure. This relation is readily extended to protocols in general by the following definition (cf. definitions 2.8 and 2.9).

Definition 4.58

Let G and H be global states of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ and $[i...k]$ an interval of process indices in I. $G_{[i...k]} \xrightarrow[[i...k]]{} H_{[i...k]}$ iff $\exists t \in X_l(G)$ with $l \in [i...k]$ such that $G \xrightarrow{t} H$. Denote by $\xrightarrow[[i...k]]{}^*$ the reflexive and transitive closure of $\xrightarrow[[i...k]]{}$. $H_{[i...k]}$ is said to be *reachable from* $G_{[i...k]}$ iff $G_{[i...k]} \xrightarrow[[i...k]]{}^* H_{[i...k]}$. □

Clearly, $G_{[i'...k']} \xrightarrow[[i'...k']]{}^* H_{[i'...k']} \Rightarrow G_{[i...k]} \xrightarrow[[i...k]]{}^* H_{[i...k]}$ if $i \le i' \le k' \le k$. In particular, $G \rightarrow^* H$ if $G_{[i...k]} \xrightarrow[[i...k]]{}^* H_{[i...k]}$ for any interval $[i...k]$. This basically explains why the extension of each $G \in \mathbf{F}^p_\Pi$ can be based on partial-state reachability, and be confined to the partial states of G indexed by maximal intervals $[i...k]$ for which $U^p_{[i...k]}(G) \neq \emptyset$. However, the reachability relation $\xrightarrow[[i...k]]{}^*$ is by itself not satisfactory for an extension procedure. The reason is that the set of partial states reachable from $G_{[i...k]}$ may very well be infinite. Furthermore, even when this set is finite (e.g. in case of a bounded protocol), its size can still be exponential in the number of processes indexed by the interval $[i...k]$. For cyclic protocols, a guaranteed *finite* extension of $G_{[i...k]}$ was secured by enforcing a channel constraint on the relation $\xrightarrow[[i...k]]{}^*$, as in Definition 4.59 [LM94b, LM96].

Definition 4.59

Let G and H be global states of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ and $[i...k]$ an interval of process indices in I. $H_{[i...k]}$ satisfies the *channel constraint* wrt $G_{[i...k]}$ iff $\forall (q, l) \in L$ with $l \in [i...k]$ it holds that $|c^H_{ql}| \le max(|c^G_{ql}|, 1)$. $G_{[i...k]} \xrightarrow[m[i...k]]{} H_{[i...k]}$ iff $G_{[i...k]} \xrightarrow[[i...k]]{} H_{[i...k]}$ and $H_{[i...k]}$ satisfies the channel constraint wrt $G_{[i...k]}$. Denote by $\xrightarrow[m[i...k]]{}^*$ the reflexive and transitive closure of $\xrightarrow[m[i...k]]{}$. $H_{[i...k]}$ is said to be *m-reachable from* $G_{[i...k]}$ iff $G_{[i...k]} \xrightarrow[m[i...k]]{}^* H_{[i...k]}$. □

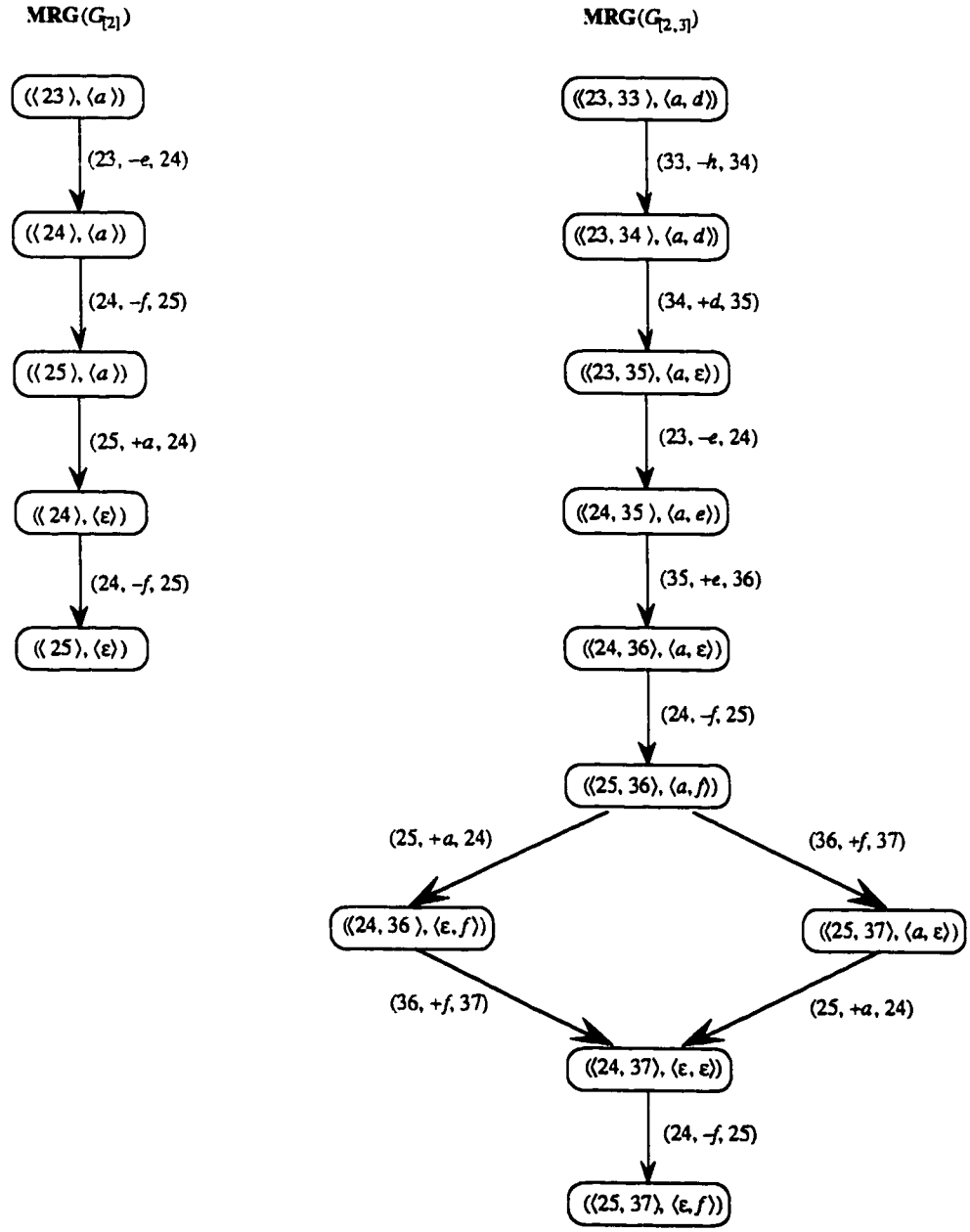
Informally, the channel constraint stipulates that no message can be added to an input channel of any process P_l indexed by $[i...k]$, unless the channel is currently empty. This certainly implies that the set of partial states m-reachable from $G_{[i...k]}$ is finite. Also, it will generally be much smaller than the set of partial states that are reachable (i.e. without the channel constraint) from $G_{[i...k]}$.

The extension procedure proposed for cyclic protocols is now described as follows: for each fair reachable global state G with $U^p_{[i...k]}(G) \neq \emptyset$ and $[i...k]$ maximal, and for each $j \in [i...k]$, explore all partial states that are m-reachable from $G_{[i...j]}$. The corresponding graph is called the m-reachability graph for $G_{[i...j]}$, denoted by $\mathbf{MRG}(G_{[i...j]})$. The extension procedure is also depicted in Figure 4.14 (cf. Figure 2.1). Its qualification to decide the reachability of process states for cyclic protocols was shown in [LM94b, LM96].

Example 4.60

For the 3-process cyclic protocol in Figure 4.12, we determined in Example 4.54 that transition vector $\langle (23, -e, 24), (33, -h, 34) \rangle$ is the only ppitv in a fair reachable global state of the protocol,

viz. in $G = (\langle 11, 23, 33\rangle, \langle a, d, g\rangle)$. Thus, $U^p_{[2,3]}(G) \neq \emptyset$, $\mathbf{F}^p_{\Pi} = \{G\}$ and interval [2, 3] is maximal. Following the extension procedure in Figure 4.14, the m-reachability graphs for partial states $G_{[2]} = (\langle s_2^G\rangle, \langle c_{12}^G\rangle) = (\langle 23\rangle, \langle a\rangle)$ and $G_{[2,3]} = (\langle s_2^G, s_3^G\rangle, \langle c_{12}^G, c_{23}^G\rangle) = (\langle 23, 33\rangle, \langle a, d\rangle)$ are constructed:



Note that, for instance, transition (23, $-e$, 24) is not executed at partial state $G_{[2,3]}$ in $\mathbf{MRG}(G_{[2,3]})$ since this would be in violation with the channel constraint. The result of constructing $\mathbf{MRG}(G_{[2]})$ and $\mathbf{MRG}(G_{[2,3]})$ is that the reachable but not fair reachable process states 24, 25, 34, 35, 36 and 37 of the protocol are all detected. □

```
for all G in F_Π with U^P_[i...k](G) ≠ ∅ and [i...k] maximal do
    for all j in [i...k] do {
        /* construct MRG(G_[i...j]) */
        A = ∅
        W = {G_[i...j]}
        while W ≠ ∅ do {
            remove an element H_[i...j] from W
            add H_[i...j] to A
            for all t in ⋃_{l∈[i...j]} X_l(G) do {
                derive H'_[i...j] such that H_[i...j] --t/π([i...j])--> H'_[i...j]
                if H'_[i...j] is NOT already in A or W then add H'_[i...j] to W
            }
        }
    }
```

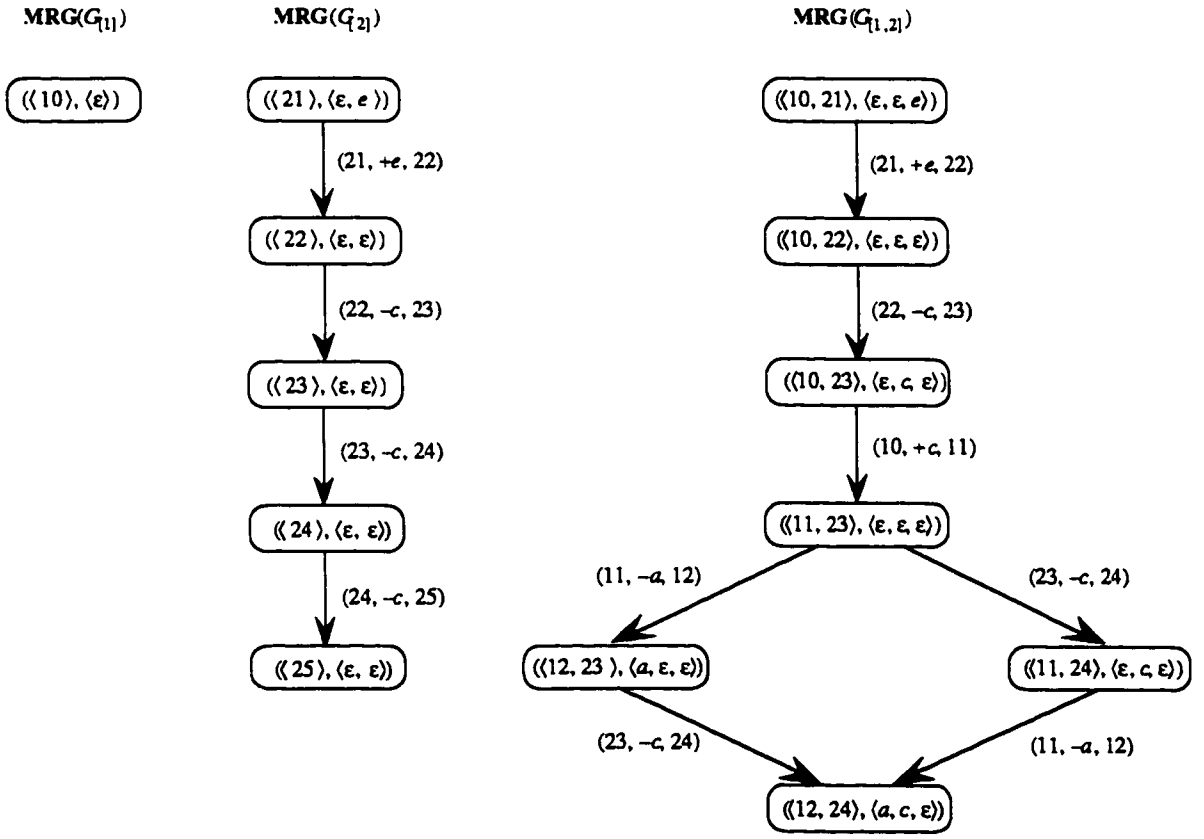
Figure 4.14 Finite extension procedure for cyclic protocols.

Let us clarify the key aspects that are combined in the extension procedure for cyclic protocols. First, there is the *m*-reachability relation among partial-states which enforces the channel constraint to guarantee termination of the procedure. Secondly, there is the construction of $\mathbf{MRG}(G_{[i...j]})$ *for each* j ranging over the given interval $[i...k]$ (the second line in Figure 4.14). The construction of $\mathbf{MRG}(G_{[i...k]})$ itself is needed to capture all interaction dependencies that may arise between the processes indexed by $[i...k]$ when they make progress from G. For instance, $\mathbf{MRG}(G_{[1, 2]})$ in Example 4.60 is required to capture the execution of transition (35, $+e$, 36) of process P_3, which can be executed only after process P_2 executes transition (23, $-e$, 24). In addition, for all $j \in [i...k]$, the construction of $\mathbf{MRG}(G_{[i...j]})$ is needed to capture all cases where process P_j is forced to execute a send transition while the corresponding output channel already holds some message(s). For these cases it is necessary to circumvent the channel constraint in Definition 4.59, which is achieved precisely by ignoring the content of the output channel of P_j in the partial state $G_{[i...j]}$ of G. For instance, if one removes transitions (35, $+e$, 36) and (36, $+f$, 37) from the protocol in Example 4.60, reachable process states 24 and 25 of the modified protocol are not detected within $\mathbf{MRG}(G_{[1, 2]})$, but only within $\mathbf{MRG}(G_{[1]})$.

Having explained the extension procedure for cyclic protocols, we are now ready to argue that this procedure is likely unsuitable for generalization to daisy-chain protocols, and thus to multi-cyclic protocols in general. Since a daisy-chain protocol (or any other multi-cyclic protocol that is not cyclic) has "connector" processes that can send messages to and receive messages from more than one process, it is evident that we need to construct additional *m*-reachability graphs if we are ever to capture all interaction dependencies arising between the processes indexed by the given interval $[i...k]$, and all pathological cases where some process is forced to execute a send transition

while the corresponding output channel is not empty (in the sense explained above). This issue can still be easily dealt with. That is, instead of constructing $\mathbf{MRG}(G_{[i\ldots j]})$ for each $j \in [i\ldots k]$, in case of a daisy-chain protocol we proceed by constructing $\mathbf{MRG}(G_{[i'\ldots k']})$ for each sub-interval $[i'\ldots k']$ of $[i\ldots k]$ (i.e. $i \leq i' \leq k' \leq k$), which amounts to a straightforward adaptation of the second line in Figure 4.14. However, even this surely required adaptation does not make the extension procedure fit for detecting all reachable process states in any daisy-chain protocol. The daisy-chain protocol in Figure 4.13 is a counter example, as illustrated below.

For the daisy-chain protocol in Figure 4.13, transition vector $\langle(10, +c, 11), (21, +e, 22)\rangle$ is the only ppitv in a fair reachable global state of the protocol, viz. in $G = (\langle 10, 21, 31\rangle, \langle \varepsilon, \varepsilon, d, e\rangle)$ (see Example 4.56). Thus, $U^p_{[1,2]}(G) \neq \emptyset$, $\mathbf{F}^p_{\Pi} = \{G\}$ and interval [1, 2] is maximal. Accordingly, the partial states $G_{[1]} = (\langle s_1^G\rangle, \langle c_{21}^G\rangle) = (\langle 10\rangle, \langle \varepsilon\rangle)$, $G_{[2]} = (\langle s_2^G\rangle, \langle c_{12}^G, c_{32}^G\rangle) = (\langle 21\rangle, \langle \varepsilon, e\rangle)$ and $G_{[1,2]} = (\langle s_1^G, s_2^G\rangle, \langle c_{12}^G, c_{21}^G, c_{32}^G\rangle) = (\langle 10, 21\rangle, \langle \varepsilon, \varepsilon, e\rangle)$ are extended, yielding the following three m-reachability graphs:



Clearly, $\mathbf{MRG}(G_{[1]})$ is void since transition (10, +c, 11) cannot be executed at $G_{[1]}$. $\mathbf{MRG}(G_{[2]})$ and $\mathbf{MRG}(G_{[1,2]})$ together reveal the reachable process states 11, 12, 22, 23, 24 and 25 (note that $\mathbf{MRG}(G_{[2]})$ is needed in particular to circumvent the channel constraint with respect to channel

C_{21}, leading to the detection of 25). However, they do not reveal reachable process state 13.

The problem with the daisy-chain protocol in Figure 4.13 originates from the interdependency between process P_1 and process P_2. As pointed out before in Example 4.56, any extension of the fair reachability graph of the protocol must capture the execution of transition (10, $+c$, 11) if the reachable process states of P_1 are to be detected. This transition is potentially executable at both fair reachable global states in the graph, and particularly so at $G = (\langle 10, 21, 31\rangle, \langle \varepsilon, \varepsilon, d, e\rangle)$ that is to be extended. The progress of P_1 from G thus depends on the progress of P_2 from G, which must first execute send transition (22, $-c$, 23). Consequently, $\mathbf{MRG}(G_{[1,2]})$ is the only m-reachability graph that may potentially uncover the reachable process states of P_1. It does indeed uncover process states 11 and 12, but not 13 because of the channel constraint $max(|c_{12}^G|, 1) = 1$ imposed on channel C_{12}. Surely, allowing C_{12} to hold two messages during the extension suffices in this case to uncover also process state 13, but merely enlarging the channel constraint in the definition of the m-reachability relation does not seem to yield a general solution for all daisy-chain protocols. Indeed, a counter argument can be sketched as follows [Liu97]. Suppose that we can use the m-reachability relation with a more flexible channel constraint to decide the reachability of any process state or process state/message pair (reachability problems **P-I** and **P-II**) for a daisy-chain protocol by finite extension of its (finite) fair reachability graph. Reconsider in particular the extension of partial state $G_{[1,2]}$ of the fair reachable global state $G = (\langle 10, 21, 31\rangle, \langle \varepsilon, \varepsilon, d, e\rangle)$ of the daisy-chain protocol above. Executing transition (21, $+e$, 22) of process P_2 results in the partial state $(\langle 10, 22\rangle, \langle \varepsilon, \varepsilon, \varepsilon\rangle)$. Since process P_3 has become permanently blocked, from this partial state and on we are essentially dealing with the two-process protocol $\Pi = (\{P_1, P_2'\}, \{(1, 2), (2, 1)\})$, where process P_2' is the same as P_2 except that transitions (20, $-d$, 22) and (21, $+e$, 22) are removed and process state 22 is the initial state of P_2'. This two-process protocol is *unbounded*. Realize that such a situation can arise for any daisy-chain protocol, i.e. the daisy-chain protocol in Figure 4.13 is just an example. Consequently, if the m-reachability relation with a more flexible channel constraint is adequate to decide **P-I** and **P-II** for any daisy-chain protocol with a finite fair reachability graph (which is our supposition), then it must be adequate also to decide these problems for any protocol of two processes communicating over two potentially unbounded channels. But this contradicts the well-known fact that logical correctness properties are in general undecidable for such protocols [BZ81] (see Section 2.5.1). One should observe that during the procedure of extending the fair reachability graph of a cyclic protocol it is impossible to encounter a situation where two not necessarily adjacent processes are *mutually* interdependent (like P_1 and P_2 above), because a cyclic protocol consists of only one ring. This ring is "broken" (in the sense of Lemma 4.30.(ii)) at the start of the extension, and remains so during the extension. A daisy-chain protocol, and any multi-cyclic protocol that is not cyclic, also exhibits at least one broken ring during the extension (the ring between P_2 and P_3 above), but it has other rings that may not be

broken. Thus, the absence/presence of multiple interconnected rings turns out to be a critical differentiation between a cyclic protocol and a daisy-chain or multi-cyclic protocol.

In conclusion, we are left with the problem of finding an alternative procedure for extending the fair reachability graph of a daisy-chain (or multi-cyclic) protocol, presumably based on a reachability relation among partial states. Such an extension procedure must of course be guaranteed to terminate (i.e. the intended extension of the fair reachability graph of a daisy-chain protocol must be finite), and to detect all reachable process states (and process state/message pairs) that are not already detected by FRA itself. In addition, it should still serve as a relief strategy that improves the conventional reachability analysis. Yet, in view of the preceding argument attesting the infeasibility of the *m*-reachability relation, one must reckon with the possibility that, unlike for cyclic protocols, a finite extension procedure may in fact not exist for daisy-chain and general multi-cyclic protocols.

4.6 Summary and remarks

In this chapter we have generalized the technique of fair reachability analysis (FRA) from cyclic to multi-cyclic protocols. A multi-cyclic protocol is made up of a collection of unidirectional rings (or component cyclic protocols), interconnected in such a way that no two rings share more than one process. The class of multi-cyclic protocols has a notably wide applicability in practical protocol modeling. In addition to protocols with a multi-ring topology, it subsumes protocols with regular network topologies such as a daisy-chain, a star or a tree, as well as many combinations of these elementary topologies.

As for cyclic protocols [RW82, GH85, LM94a, LM96], FRA is an effective and efficient relief strategy for the detection of deadlocks in multi-cyclic protocols. State exploration by FRA forces a multi-cyclic protocol to progress only through fair execution sequences. Clearly, this eliminates a large part of the redundancy in conventional reachability analysis caused by equivalent execution sequences (cf. Section 4.1.1). We proved that the global states of a multi-cyclic protocol explored by FRA, i.e. the fair reachable global states, are precisely those reachable global states in which for each ring all channels in the ring are of equal length. This so-called ring-wise equal channel length property entails in particular all deadlock states. As a result, the deadlock detection problem is decidable for each multi-cyclic protocol whose fair reachable global state space is finite. We also established two sufficient conditions relating to the boundedness aspect of channels that guarantee finiteness (propositions 4.39 and 4.41). Both these conditions allow the presence of unbounded channels, which indicates that FRA is not only significant as a relief strategy but also as a state exploration technique capable of handling various unbounded protocols.

The contributions of this chapter provide an extension of the work on FRA for cyclic protocols in [LM94, LM96] with respect to deadlock detection. The ring-wise equal channel length property

and the two boundedness conditions established for multi-cyclic protocols generalize the equal channel length property and the respective boundedness conditions given for cyclic protocols. This generalization proved necessary because multi-cyclic protocols composed of multiple rings include "connector" processes with more than one incoming and one outgoing channel. A next step along this line would be an extension of FRA to protocols with yet more complex and perhaps even arbitrary network topologies. However, we conjectured that FRA is inherently infeasible beyond multi-cyclic protocols or, more accurately, for verifying properties of protocols that are not fair formed (see Definition 4.49). This stems from the principal nature of the technique which forces progress of at least two processes at each step during state exploration while preserving a global channel invariant.

There are some open problems concerning FRA for multi-cyclic protocols. First, since FRA in its basic form is inadequate for the detection of logical errors other than deadlocks, at least a finite extension of the fair reachability graph of a multi-cyclic protocol is needed for this purpose. Such an extension has already been established for cyclic protocols. Specifically, a procedure was proposed in [LM94b, LM96] that "cleverly" extends the fair reachability graph of a cyclic protocol in order to detect unboundedness, non-executable transitions and unspecified receptions. We have explained this procedure in detail, and argued that it appears unfit for generalization to daisy-chain protocols and thus to multi-cyclic protocols in general. Hence, it remains to be seen whether FRA can be used to achieve the same logical error coverage for multi-cyclic protocols as for cyclic protocols. A second and likely related problem is that weak boundedness (see Definition 4.40) does not emerge as a necessary condition for a multi-cyclic protocol to have a finite fair reachability graph, unlike the case for cyclic protocols [LM94a, LM96]. Hence, the notion of weak boundedness does not provide a complete characterization of the class of multi-cyclic protocols for which FRA decides deadlock-freedom. It seems that the existence of such a characterization goes hand in hand with the ability to use FRA for the detection of logical errors other than deadlocks. We speculate in this respect that certain structural constraints must be imposed on the process graphs of the individual processes of a multi-cyclic protocol, but this requires further investigation.

Chapter 5

Leaping reachability analysis

Thus far we have seen that FRA is a useful relief strategy for verifying multi-cyclic protocols. In particular, the execution of multiple transitions in one atomic step (viz. through fair transition-tuples) proved to be effective in reducing the number of global states and transitions examined for the purpose of deadlock detection in multi-cyclic protocols. Itoh & Ichikawa [II83] also employed this idea of executing multiple transitions concurrently for the verification of protocols with two or more processes in an arbitrary communication topology, but with restricted process structures (see Chapter 3). The idea of executing multiple transitions concurrently was ultimately generalized by Özdemir & Ural [ÖU94, ÖU95, Özd95], who proposed *simultaneous reachability analysis* (SRA) as a relief strategy for the verification of logical correctness properties of protocols with no topological or structural constraints at all. In essence, this generality is the result of allowing processes in a protocol to progress concurrently in a more flexible way than FRA.

In this chapter we propose an incremental improvement of SRA, called *leaping reachability analysis* (LRA), which maintains the power of SRA to detect all non-progress states, all non-executable transitions, all unspecified receptions and all buffer overflows in a protocol, while further reducing the size of the global state space that needs to be analyzed. This contribution was published incrementally in [SU96b, SU98a]. We start by formalizing LRA and then provide an analytical comparison between both relief strategies. An empirical comparison follows in Chapter 6.

5.1 Leap sets and proper leap sets

At the heart of LRA lies the concurrent execution of transitions at global states. Clearly, transitions that are to be executed concurrently must pertain to different processes. This foremost requirement is captured by the notion of leap sets in Definition 5.1.

Definition 5.1

Let G be a global state of a protocol Π. A *leap set* in G is a non-empty set T of transitions

executable at G (i.e. $\emptyset \subset T \subseteq X(G)$) such that for all $t, t' \in T$, $act(t) \neq act(t')$ if $t \neq t'$. The set of all leap sets in G is denoted by $leap(G)$. □

It follows that there are $|T|!$ possible interleaving orders of the transitions in a leap set $T \in leap(G)$, all of which are equivalent up to $\equiv$ and lead to the same global state when executed from G (cf. Definition 4.15 and Proposition 4.16).

Definition 5.2

Let G be a global state of a protocol Π and $T = \{t_1, t_2, \ldots, t_k\} \in leap(G)$. A *linearization* of T is a sequence $t_{\pi(1)}t_{\pi(2)}\ldots t_{\pi(k)}$, with π any permutation on $\{1, 2, \ldots, k\}$. The set of all linearizations of T is denoted by $lin(T)$, and this notation is adopted also for sequences of leap sets: $lin(T_1T_2\ldots T_m) = \{\gamma_1\gamma_2\ldots\gamma_m \mid \gamma_i \in lin(T_i)\}$. □

Proposition 5.3

Let $T \in leap(G)$ and $\gamma \in lin(T)$ with $G \xrightarrow{\gamma,*} H$, then $\forall \gamma' \in lin(T)$: $\gamma \, {}_G\!\equiv_H \gamma'$.

Proof: Directly from the fact that all transitions in T are executable at G and no two transitions in T belong to the same process. □

Like the fair transition-tuples in FRA, leap sets provide a means to reduce the number of global states and transitions explored: all transitions in a leap set can in principle be executed concurrently in one atomic step. The reduction obtained increases with the size of a leap set and it is therefore tempting to consider only the maximal sets in $leap(G)$. However, this is inadequate for verification purposes. To see this, suppose that some process P_i has a transition t that is potentially executable at G (see Section 4.1.2). Also assume that P_i has an executable transition t' at G that is included in some leap set T in G. When all transitions in T are executed at once, t may become forever disabled by the execution of t', even though its possible execution at a global state H reachable from G could ultimately lead to a logical error. Figure 5.1 illustrates this scenario for a simple protocol. At the initial global state, the transitions $(10, -a, 11)$ and $(20, -b, 21)$ are executable and form the only maximal leap set. Transition $(20, +a, 22)$ is potentially executable and becomes executable at global state $(\langle 11, 20\rangle, \langle a, \varepsilon\rangle)$, after the sole transmission of message a. Its execution leads to the deadlock state $(\langle 11, 22\rangle, \langle \varepsilon, \varepsilon\rangle)$. Yet, when the two send transitions are executed concurrently, the reception of message a is no longer possible and this deadlock state is not detected.

In general, analyzing the effect of a potentially executable transition requires the corresponding process to refrain from progress for as long as the transition continues to be potentially executable [II83, ÖU94, ÖU95]. For instance, for the protocol above one must include a progress scheme which makes process P_2 wait until process P_1 has sent message a. Based on this observation, we

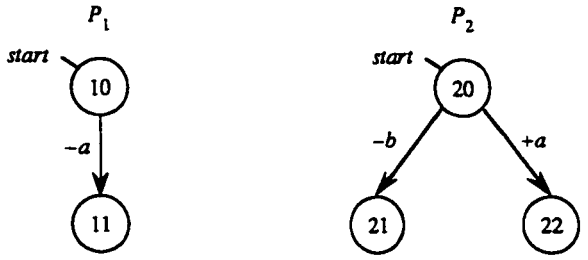


Figure 5.1 The role of potentially executable transitions.

employ a selective subset of *leap*(G) whose elements are called *proper leap sets*. Each proper leap set in G contains one executable transition from *exactly* those processes with executable transitions but no potentially executable transitions at G, provided that some such process(es) exist(s). This ensures that the possible effect of potentially executable transitions at G is not ignored: processes with such transitions are forced to "wait" at their process states in G by excluding their transitions. The other processes are still forced to proceed concurrently in order to achieve state reduction. In the special case where each process with executable transitions at G also has at least one potentially executable transition at G, there is little choice but to consider all executable transitions at G individually. Note that for each such transition t the singleton set $\{t\}$ is an element of *leap*(G).

Definition 5.4

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$. Define $wait(G) = \{i \in I \mid X_i(G) \neq \emptyset \Rightarrow P_i(G) \neq \emptyset\}$, and the set *pleap*($G$) of proper leap sets in G as follows:

$$pleap(G) = \{\, T \mid T \in leap(G) \wedge act(T) = \{i \in I \mid i \notin wait(G)\}\,\}$$
$$\text{if } wait(G) \subset I$$

$$pleap(G) = \{\, \{t\} \mid t \in X(G)\,\}$$
$$\text{otherwise.}$$

□

Note that trivially $i \in wait(G)$ if no transition of process P_i is executable at G. Some characteristic properties of *pleap*(G) are given in Proposition 5.5.

Proposition 5.5

i) $t \in X_i(G) \wedge i \notin wait(G) \Rightarrow \exists T \in pleap(G)\colon t \in T$;

ii) $T \in pleap(G) \wedge i \in act(T) \wedge t \in X_i(G) \Rightarrow (T \backslash X_i(G)) \cup \{t\} \in pleap(G)$;

iii) $T \in leap(G) \wedge i \in act(T) \cap wait(G) \Rightarrow T \notin pleap(G) \vee pleap(G) = \bigcup_{t \in X(G)} \{\{t\}\}$.

Proof: Straightforward from Definition 5.4 □

The definition of *pleap*(G) is constructive and easily translated into an optimal algorithm. It first calculates the complement $I \backslash wait(G)$ of *wait*(G) by inspecting the transition relations of all the processes. Clearly, this requires no overhead as the transition relations must be inspected equally for conventional reachability analysis. The algorithm then returns all (singleton sets of) transitions executable at G if $I \backslash wait(G) = \varnothing$, or else the cross-product $\prod_{i \in I \backslash wait(G)} X_i(G)$. The overhead incurred by this step is $O(|\prod_{i \in I \backslash wait(G)} X_i(G)|)$.

Example 5.6

Consider the protocol $(\{P_1, P_2, P_3, P_4\}, \{(1, 2), (2, 3), (3, 4), (4, 1), (4, 3)\})$, with the process graphs of processes P_1, P_2, P_3 and P_4 as in Figure 5.2 (by convention, $m_{ij} \in M_{ij}$). Let

$$t_1^1 = (10, -m_{12}, 11) \quad t_2^1 = (20, -m_{23}, 21) \quad t_3^1 = (30, -m_{34}, 31) \quad t_4^1 = (40, -m_{43}, 41)$$

$$t_1^2 = (10, +m_{41}, 12) \quad t_2^2 = (20, +m_{12}, 22) \quad t_3^2 = (31, +m_{43}, 30) \quad t_4^2 = (41, +m_{34}, 40)$$

then $X(G^0) = \{t_1^1, t_2^1, t_3^1, t_4^1\}$ and $P(G^0) = P^+(G^0) = \{t_1^2, t_2^2\}$. Hence, $leap(G^0) = 2^{X(G^0)} \backslash \{\varnothing\}$ and $wait(G^0) = \{1, 2\}$, and thus $pleap(G^0) = \{\{t_3^1, t_4^1\}\}$. □

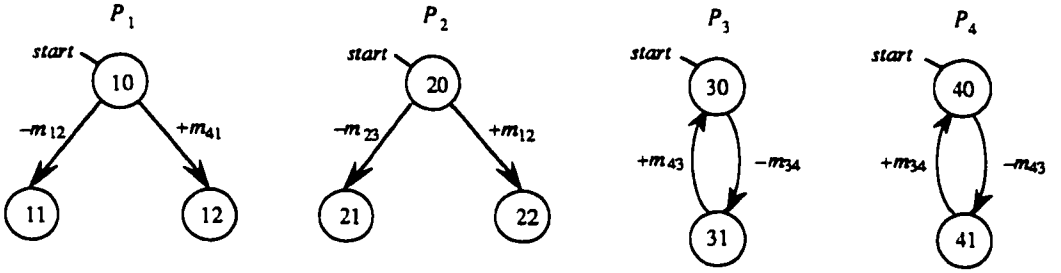


Figure 5.2 A sample protocol.

5.2 Verifying indefinite progress

Following the discussion above, we formalize the execution of proper leap sets in global states and show that this is sufficient to detect all non-progress states of a protocol.

5.2.1 ℓ-reachability

Definition 5.7

Let G and H be global states of a protocol Π. $G \xrightarrow{}_{\ell} H$ iff $\exists T \in pleap(G)$ with $\gamma \in lin(T)$ such that $G \xrightarrow{\gamma}{}^{*} H$. This is also denoted by $G \xrightarrow{T}_{\ell} H$. □

Well-definedness of $\xrightarrow{}_{\ell}$ follows from Proposition 5.3: executing a (proper) leap set in a global state G always yields a unique state H.

Definition 5.8

Let G and H be global states of a Π, and denote by $\xrightarrow{}_{\ell}^{*}$ the reflexive and transitive closure of $\xrightarrow{}_{\ell}$. H is *ℓ-reachable from* G iff $G \xrightarrow{}_{\ell}^{*} H$. When $G = G^0$, H is said to be *ℓ-reachable*. The set of all ℓ-reachable global states of Π is denoted by $\mathbf{L}_{\Pi}$. For a sequence of proper leap sets $\Omega = T_1 T_2 \ldots T_m$, $G \xrightarrow{\Omega}_{\ell}^{*} H$ denotes the existence of global states $Q^0, Q^1, \ldots, Q^m$ such that $G = Q^0 \xrightarrow{T_1}_{\ell} Q^1 \xrightarrow{T_2}_{\ell} \ldots \xrightarrow{T_m}_{\ell} Q^m = H$. □

```
/* A is the set of global states that have been analyzed.        */
/* W is the set of global states that still need to be analyzed. */

/* Initialize: */
A = ∅
W = {G^0}

/* Loop: */
while W ≠ ∅ do {
    remove an element G from W
    add G to A
    for all T in pleap(G) do {
        /* execution of proper leap set T */
        derive H such that G -T->_ℓ H
        if H is NOT already in A or W then add H to W
    }
}
```

Figure 5.3 State exploration by LRA.

An algorithm for exploring the ℓ-reachable global state space, or *ℓ-reachability graph*, of a protocol is shown in Figure 5.3. The box indicates the modification with respect to the standard perturbation algorithm in Figure 2.1. Clearly, every ℓ-reachable global state is also reachable.

Proposition 5.9

$\mathbf{L}_{\Pi} \subseteq \mathbf{R}_{\Pi}$

Proof: By definition of $\xrightarrow{}_{\ell}^{*}$. □

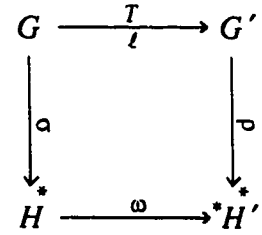
5.2.2 Detecting non-progress states

The next lemma provides the basis for proving that the set $\mathbf{L}_{\Pi}$ of ℓ-reachable global states includes all non-progress states and hence all deadlock states of a protocol.

Lemma 5.10

Let $G \xrightarrow{\sigma}{}^{*} H$ and $\sigma \neq \varepsilon$, then there exist a proper leap set $T \in pleap(G)$ with $\gamma \in lin(T)$, transition sequences ω, ρ and a global state H' such that $\sigma\omega \equiv_{H'} \gamma\rho$, $act(\omega) \cap act(\sigma) = \emptyset$ and $|\rho| \leq |\sigma|$.

Proof: The proof essentially consists in showing that the diagram below holds true for some proper leap set $T \in pleap(G)$, transition sequences ω and ρ and global states G' and H'. We show in particular that T, ω and ρ can always be fixed such that ω contains exactly those transitions in T that do not appear in σ, and ρ contains all transitions of σ except those that are in T.



Since $\sigma \neq \varepsilon$, it follows that $X(G) \neq \emptyset$ and hence $pleap(G) \neq \emptyset$. Let $first(\sigma) = \{t \mid t \in \Delta_i \wedge \mu t \in pref(\sigma) \wedge i \notin act(\mu)\}$, i.e. $first(\sigma)$ contains for each process active on σ the first transition of that process in σ. Clearly, $first(\sigma) \neq \emptyset$ since $\sigma \neq \varepsilon$. Now choose $T \in pleap(G)$ such that $T \backslash first(\sigma)$ is minimal ($\emptyset \subseteq T \backslash first(\sigma) \subseteq T$). We show that $act(T \backslash first(\sigma)) \cap act(\sigma) = \emptyset$, by contradiction. Suppose $\exists i \in I$: $i \in act(T \backslash first(\sigma)) \cap act(\sigma)$ and let t, $t' \in \Delta_i$ such that $t \in T \backslash first(\sigma)$ and $t' \in first(\sigma)$, then $t' \in X_i(G) \cup P_i(G)$. By Proposition 5.5.(ii), if $t' \in X_i(G)$ then $(T \backslash \{t\}) \cup \{t'\} \in pleap(G)$. However, $|((T \backslash \{t\}) \cup \{t'\}) \backslash first(\sigma)| < |T \backslash first(\sigma)|$ which contradicts the minimality of $T \backslash first(\sigma)$. On the other hand, if $t' \in P_i(G)$ then $i \in act(T) \cap wait(G)$ and thus $pleap(G) = \bigcup_{t \in X(G)} \{\{t\}\}$, by Proposition 5.5.(iii), which in turn implies that $T = T \backslash first(\sigma) = \{t\}$. In this case let t'' be the very first transition of σ, then $t'' \in X(G)$, $\{t''\} \in pleap(G)$ and moreover $\{t''\} \subseteq first(\sigma)$. Again, this contradicts the minimality of $T \backslash first(\sigma)$, since $\{t''\} \backslash first(\sigma) = \emptyset \subset T \backslash first(\sigma)$. Hence, the claim $act(T \backslash first(\sigma)) \cap act(\sigma) = \emptyset$ holds.

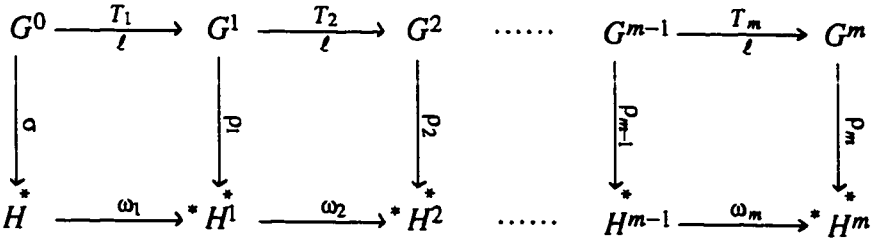
Let $\omega \in lin(T \backslash first(\sigma))$ if $T \backslash first(\sigma) \neq \emptyset$, or else $\omega = \varepsilon$. Surely, $act(\omega) \cap act(\sigma) = \emptyset$ in either case and thus there exists a global state H' such that $G \xrightarrow{\sigma\omega}{}^{*} H'$ (by Corollary 4.4). Now, let $\gamma \in lin(T)$ and ρ such that $\sigma \equiv \gamma'\rho$, with $\gamma' \in lin(T \cap first(\sigma))$. As $\gamma \equiv \gamma'\omega$ and $act(\omega) \cap act(\rho) = \emptyset$, it follows that also $G \xrightarrow{\gamma\rho}{}^{*} H'$. Thus, $\sigma\omega \equiv_{H'} \gamma\rho$, $act(\omega) \cap act(\sigma) = \emptyset$ and $|\rho| \leq |\sigma|$, which proves the lemma. □

When H is a non-progress state, ω is empty and the conclusion of Lemma 5.10 reduces to $\sigma \equiv_H \gamma\rho$ and thus $|\rho| < |\sigma|$. By repeated application of this lemma one can then prove that H is ℓ-reachable.

Theorem 5.11

Every non-progress state is ℓ-reachable.

Proof: Let $H \in \mathbf{R}_{\Pi}$ be a non-progress state and $G^0 \xrightarrow{\sigma}{}^* H$. H is trivially ℓ-reachable if $|\sigma| = 0$ since then $H = G^0$. Suppose that $|\sigma| > 0$. By Lemma 5.10 there is a proper leap set $T_1 \in pleap(G^0)$ with $\gamma_1 \in lin(T_1)$, transition sequences ω_1 and ρ_1, and a global state H^1 such that $\sigma\omega_1 \; {}_{G^0}\!\equiv_{H^1} \gamma_1\rho_1$ and $|\rho_1| \leq |\sigma|$. Let $G^0 \xrightarrow[\ell]{T_1} G^1 \xrightarrow{\rho_1}{}^* H^1$, then $|\rho_1| > 0$ equally implies the existence of $T_2 \in pleap(G^1)$ with $\gamma_2 \in lin(T_2)$, ω_2, ρ_2, and H^2 such that $\rho_1\omega_2 \; {}_{G^1}\!\equiv_{H^2} \gamma_2\rho_2$ and $|\rho_2| \leq |\rho_1|$. Applying this argument repeatedly results in the following diagram:



Thus $\rho_{j-1}\omega_j \; {}_{G^{j-1}}\!\equiv_{H^j} \gamma_j\rho_j$ and $|\rho_j| \leq |\rho_{j-1}|$, with $\gamma_j \in lin(T_j)$, $T_j \in pleap(G^{j-1})$ and $|\rho_{j-1}| > 0$ $(2 \leq j \leq m)$. As a result, $\sigma\omega_1\omega_2 \ldots \omega_m \; {}_{G^0}\!\equiv_{H^m} \gamma_1\gamma_2 \ldots \gamma_m\rho_m$.

Since H is a non-progress state we have $\omega_1\omega_2 \ldots \omega_m = \varepsilon$ and $H^m = H$, implying that $|\rho_j| < |\rho_{j-1}|$, for all $2 \leq j \leq m$, and $|\rho_1| < |\sigma|$. We may then assume that $|\rho_m| = 0$ for σ is finite and Lemma 5.10 applies as long as $|\rho_j| > 0$. Thus, $\sigma \; {}_{G^0}\!\equiv_H \gamma_1\gamma_2 \ldots \gamma_m$ and $G^0 \xrightarrow[\ell]{T_1T_2 \ldots T_m}{}^* H$, i.e. H is ℓ-reachable. □

Corollary 5.12

Every deadlock state is ℓ-reachable. □

Corollary 5.13

Indefinite progress and deadlock-freedom for a protocol Π are decidable if $\mathbf{L}_{\Pi}$ is finite. □

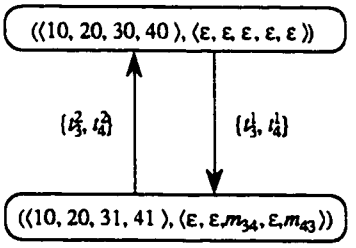


Figure 5.4 The ℓ-reachability graph of the protocol of Example 5.6.

Example 5.14

The ℓ-reachability graph of the protocol of Example 5.6 (see Figure 5.2) is shown in Figure 5.4. For the initial global state we have $pleap(G^0) = \{ \{ t_3^1, t_4^1 \} \}$. Executing proper leap set $\{ t_3^1, t_4^1 \}$ in G^0 results in the global state (⟨10, 20, 31, 41⟩, ⟨ε, ε, m_{34}, ε, m_{43}⟩). This state has also one proper leap set, namely $\{ t_3^2, t_4^2 \}$, and its execution leads back to the initial global state. State exploration

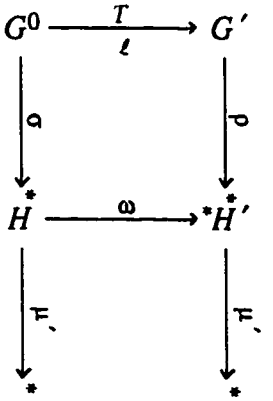
based on $\xrightarrow{}_{\ell}^{*}$ thus terminates, and since both states are progress states the protocol is (correctly) found to progress indefinitely. Of course, the same result is obtained by conventional reachability analysis but this requires exploring as many as 40 global states and 100 global state transitions. □

While undecidability prevails in general, it is again interesting to note that LRA, like FRA (cf. Section 4.3.2), is effective beyond the class of bounded protocols. For instance, consider a small modification of the protocol of Example 5.6, replacing transition (10, $-m_{12}$, 11) of process P_1 by the "cyclic" transition (10, $-m_{12}$, 10). It is easy to see that the resulting protocol is unbounded, whereas its ℓ-reachability graph is the same as in Figure 5.4 and thus finite. Another interesting result is the fact that the relation $\xrightarrow{}_{\ell}^{*}$ preserves the *possibility* of indefinite progress of a protocol, in the sense stated by Proposition 5.15.

Proposition 5.15

There exists an infinite sequence of reachable global states in the (conventional) reachability graph of a protocol Π iff there exists an infinite sequence of ℓ-reachable global states in the ℓ-reachability graph of Π.

Proof: The "if" part is immediate by definition of the ℓ-reachability relation $\xrightarrow{}_{\ell}$. For the "only-if" part, assume the existence of an infinite sequence of reachable global states in the reachability graph of Π, with μ the corresponding infinite transition sequence from the initial global state G^0. Let $\mu = \sigma\mu'$ such that $act(\sigma) = act(\mu)$, i.e. $\sigma \in pref(\mu)$ contains at least one transition from each process that is active on μ. We have $\sigma \neq \varepsilon$, $G^0 \xrightarrow{\sigma}^{*} H$ and μ' is an infinite transition sequence from H, for some reachable global state H. By Lemma 5.10, there exist $T \in pleap(G^0)$ with $\gamma \in lin(T)$, transition sequences ω and ρ, and a global state H' such that $\sigma\omega \equiv_{G^0,H'} \gamma\rho$, $act(\omega) \cap act(\sigma) = \emptyset$ and $|\rho| \leq |\sigma|$. By the choice of σ, $act(\omega) \cap act(\mu) = \emptyset$ and thus $act(\omega) \cap act(\mu') = \emptyset$. It follows that μ' is also an infinite transition sequence from H':



Since μ is infinite we can apply the same reasoning infinitely often, continuing with the sequence $\rho\mu'$ from G', to yield an infinite sequence of ℓ-reachable global states. □

5.3 Verifying freedom of non-executable transitions

State exploration by LRA based on the relation $\xrightarrow{\ell}{}^*$ is inadequate for verifying the absence of non-executable transitions. This is witnessed by the protocol of Example 5.6: transition $(20, +m_{12}, 22)$ of process P_2 is surely executable, but not at an ℓ-reachable global state (see Figure 5.4). Thus, a decision procedure based on $\xrightarrow{\ell}{}^*$ would mistakenly report this transition as being non-executable. The problem lies in the conception of delaying the processes with potentially executable transitions at global states, which results in a phenomenon referred to as the *ignoring problem* [Val90]: some processes may be delayed indefinitely from a certain global state, causing the behavior of these processes to be ignored. For instance, the behavior of processes P_1 and P_2 of the protocol of Example 5.6 is completely ignored because in each ℓ-reachable global state they have potentially executable transitions, while P_3 and P_4 always have exclusively executable transitions.

In general, due to the ignoring problem the relation $\xrightarrow{\ell}{}^*$ may not expose all reachable process states (i.e. process states appearing in some reachable global state). This clearly hinders not only the detection of non-executable transitions, but that of unspecified receptions and buffer overflows as well. For the verification of logical correctness properties other than indefinite progress and deadlock-freedom we must therefore augment the state exploration scheme defined in the previous section. Preferably, it should be adapted to the property one wants to verify. This section presents a simple extension of the ℓ-reachability relation $\xrightarrow{\ell}{}^*$ for detecting the non-executable transitions of a protocol. Unspecified receptions and buffer overflows are dealt with in Section 5.4.

5.3.1 ℓ^*-reachability

An effective solution to the ignoring problem is obtained by executing a minimal number of extra leap sets in a given global state G, in addition to the proper leap sets, such that each executable transition at G is executed via at least one leap set. Precisely, we extend $pleap(G)$ to form the set $xpleap(G)$ by adding for one arbitrary $T \in pleap(G)$ all the leap sets $T \cup \{t\}$, where t is a transition executable at G but not included in any proper leap set in $pleap(G)$. This extension ensures that processes with executable transitions are not expelled from progress.

Definition 5.16

Let G be a global state of a protocol Π and $T \in pleap(G)$ an arbitrary proper leap set in G. Define

$$xpleap(G) = pleap(G) \cup \{T \cup \{t\} \mid t \in X(G) \wedge act(t) \in wait(G)\}$$
$$\text{if } wait(G) \subset I$$

$$xpleap(G) = pleap(G)$$
$$\text{otherwise.}$$

□

Note that $xpleap(G) = pleap(G)$ also when $wait(G) = \emptyset$. Further note that if $pleap(G)$ consists of more than one proper leap set without capturing all executable transitions at G, then $xpleap(G)$ is not unique since the proper leap set to be used for extension can be selected non-deterministically. Yet, we stress that this nondeterminism does not affect in any way the theoretical results in this section. For the mere sake of convenience, we use as a heuristic to always choose the first element of $pleap(G)$ when viewed as an ordered set. This allows us to refer to $xpleap(G)$ as a unique set and an algorithm for its construction then follows directly from Definition 5.16. Clearly, the fact that we need to fix only one proper leap set for extension (as opposed to all of them) is important for efficiency considerations, particularly when $|pleap(G)|$ is large.

Insightful properties of the set $xpleap(G)$ are stated in Proposition 5.17, analogous to those of $pleap(G)$ in Proposition 5.5. Observe especially that $\bigcup_{T \in xpleap(G)} T = X(G)$, i.e. for each executable transition t at G there is a leap set in $xpleap(G)$ containing t (Proposition 5.17.(i)), which does not necessarily hold for $pleap(G)$ (cf. Proposition 5.5.(i)).

Proposition 5.17

i) $t \in X(G) \Rightarrow \exists T \in xpleap(G): t \in T$;

ii) $T \in xpleap(G) \wedge i \in act(T) \cap wait(G) \wedge t \in X_i(G) \Rightarrow (T \setminus X_i(G)) \cup \{t\} \in xpleap(G)$;

iii) $T \in xpleap(G) \wedge i \in act(T) \cap wait(G) \Rightarrow T \setminus X_i(G) \in pleap(G) \vee xpleap(G) = \bigcup_{t \in X(G)} \{\{t\}\}$.

Proof: Straightforward from Definition 5.16. □

Example 5.18

Consider once again the protocol of Example 5.6: $X(G^0) = \{t_1^1, t_2^1, t_3^1, t_4^1\}$, $wait(G^0) = \{1, 2\}$ and $pleap(G^0) = \{\{t_3^1, t_4^1\}\}$. That is, all four processes have executable transitions at G^0, but process P_1 and process P_2 also have potentially executable transitions at G^0. With neither transition t_1^1 nor transition t_2^1 occurring in a proper leap set in G^0, by the first clause of Definition 5.16 we have $xpleap(G^0) = \{\{t_3^1, t_4^1\}, \{t_3^1, t_4^1, t_1^1\}, \{t_3^1, t_4^1, t_2^1\}\}$. □

As before, we define a reachability relation among global states which governs the execution of all elements of $xpleap(G)$.

Definition 5.19

Let G and H be global states of a protocol Π. $G \xrightarrow[\ell^*]{} H$ iff $\exists T \in xpleap(G)$ with $\gamma \in lin(T)$ such that $G \xrightarrow{\gamma}{}^* H$. This is also denoted by $G \xrightarrow[\ell^*]{T} H$. □

Definition 5.20

Let G and H be global states of a Π, and denote by $\xrightarrow[\ell^*]{}{}^*$ the reflexive and transitive closure of $\xrightarrow[\ell^*]{}$.

H is ℓ^**-reachable from G* iff $G \xrightarrow{}_{\ell^*}{}^{*} H$. If $G = G^0$, then *H* is said to be ℓ^**-reachable*. The set of all ℓ^*-reachable global states of Π is denoted by $\mathbf{L}^*_\Pi$. For a sequence of leap sets $\Omega = T_1T_2\ldots T_m$, $G \xrightarrow{\Omega}_{\ell^*}{}^{*} H$ denotes the existence of global states $Q^0, Q^1,\ldots, Q^m$ such that $G = Q^0 \xrightarrow{T_1}_{\ell^*} Q^1 \xrightarrow{T_2}_{\ell^*} \ldots \xrightarrow{T_m}_{\ell^*} Q^m = H$. □

Surely, an algorithm for constructing the ℓ^*-reachability graph of a protocol (representing its ℓ^*-reachable global state space) is akin to the one in Figure 5.3. We arrive at some anticipated results. By definition of the relation $\xrightarrow{}_{\ell^*}{}^{*}$ as an extension of $\xrightarrow{}_{\ell}{}^{*}$ it follows that the ℓ-reachability graph is a subgraph of the ℓ^*-reachability graph. Consequently, all ℓ-reachable global states and thus all non-progress states of a protocol (by Theorem 5.11) are ℓ^*-reachable.

Proposition 5.21

$\mathbf{L}_\Pi \subseteq \mathbf{L}^*_\Pi \subseteq \mathbf{R}_\Pi$

Proof: By definition of *xpleap*(*G*) we have *pleap*(*G*) $\subseteq$ *xpleap*(*G*), for any global state *G*. This implies $\mathbf{L}_\Pi \subseteq \mathbf{L}^*_\Pi$. The inclusion $\mathbf{L}^*_\Pi \subseteq \mathbf{R}_\Pi$ holds by definition of $\xrightarrow{}_{\ell^*}{}^{*}$. □

Corollary 5.22

Every non-progress state is ℓ^*-reachable. □

Corollary 5.23

Every deadlock state is ℓ^*-reachable. □

5.3.2 Detecting non-executable transitions

The relation $\xrightarrow{}_{\ell^*}{}^{*}$ is intended for detecting non-executable transitions. We prove that exploring the ℓ^*-reachability graph is indeed sufficient for this purpose: for each executable transition of a protocol Π, there is at least one global state in $\mathbf{L}^*_\Pi$ at which the transition is executable. As a result, LRA based on $\xrightarrow{}_{\ell^*}{}^{*}$ can verify the absence of non-executable transitions (and non-progress states) for any protocol with a finite ℓ^*-reachability graph.

Lemma 5.24

Let $G \xrightarrow{\sigma}{}^{*} H$ and $\sigma \neq \varepsilon$, then there exist $T \in$ *xpleap*(*G*) with $\gamma \in$ *lin*(*T*), transition sequences ω, ρ and a global state *H'* such that $\sigma\omega \underset{G}{\equiv}_{H'} \gamma\rho$, *act*($\omega$) $\cap$ *act*(σ) = $\emptyset$ and $|\rho| < |\sigma|$.

Proof: By Lemma 5.10, we have a *proper* leap set $T' \in$ *pleap*(*G*) $\subseteq$ *xpleap*(*G*) with $\gamma' \in$ *lin*(*T'*), transition sequences ω' and ρ', and a global state *H'* such that $\sigma\omega' \underset{G}{\equiv}_{H'} \gamma'\rho'$, *act*($\omega'$) $\cap$ *act*(σ) = $\emptyset$ and $|\rho'| \leq |\sigma|$. We know from the proof of Lemma 5.10 that *T'* and ω' are such that $T' \backslash$ *first*(σ) is

minimal and $\omega' \in lin(T' \backslash first(\sigma))$, where $first(\sigma) = \{t \mid t \in \Delta_i \wedge \mu t \in pref(\sigma) \wedge i \notin act(\mu)\}$. Clearly, if $T' \backslash first(\sigma) \subset T'$ then $|\omega'| < |\gamma|$ and hence $|\rho'| < |\sigma|$. Thus, in this case the lemma holds, i.e. choose $T = T'$, $\gamma = \gamma'$, $\omega = \omega'$ and $\rho = \rho'$.

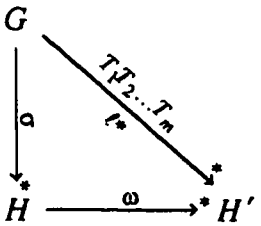
Alternatively, if $T' \backslash first(\sigma) = T'$ then $\omega' \equiv \gamma' \neq \varepsilon$ and $act(\sigma) \cap act(T') = \emptyset$. We must have $wait(G) \subset I$ because otherwise $pleap(G) = \bigcup_{t \in X(G)} \{\{t\}\}$ and $|\{t'\} \backslash first(\sigma)| = 0 < |T' \backslash first(\sigma)|$, with t' the first transition of σ, which contradicts the minimality of $T' \backslash first(\sigma)$. Definition 5.4 then states that $act(T) = \{i \in I \mid i \notin wait(G)\}$, for all $T \in pleap(G)$. Since $act(\sigma) \cap act(T') = \emptyset$ it follows that $act(\sigma) \subseteq wait(G)$ and $\forall T \in pleap(G)$: $act(\sigma) \cap act(T) = \emptyset$. In particular, the first transition t' of σ is executable at G but not in any proper leap set in G. According to Definition 5.16, in this case $pleap(G) \subset xpleap(G)$ and some proper leap set in G, say $T'' \in pleap(G)$, is selected to form $xpleap(G)$. Now choose $T = T'' \cup \{t'\}$, then $T \in xpleap(G)$, $T \backslash first(\sigma) = T'' \backslash first(\sigma) = T''$ and $act(T) \cap act(\sigma) = act(t')$. Let $\gamma \in lin(T)$, $\omega \in lin(T'')$ and $\sigma = t'\rho$, then $\sigma\omega \ {}_G\!\equiv_{H'} \gamma\rho$, for some global state H', $act(\omega) \cap act(\sigma) = \emptyset$ and $|\rho| < |\sigma|$, i.e. again the lemma holds. □

Notice that Lemma 5.24 differs from Lemma 5.10 in the strict inequality $|\rho| < |\sigma|$, which now holds irrespective of whether H is a non-progress state. This enables the following generalization of Lemma 5.24 to sequences of leap sets.

Lemma 5.25

Let $G \xrightarrow{\sigma}{}^* H$ and $\sigma \neq \varepsilon$, then there exist a sequence of leap sets Ω with $\eta \in lin(\Omega)$, a transition sequence ω and a global state H' such that $G \xrightarrow{\Omega}{}^* H'$ and $\sigma\omega \ {}_G\!\equiv_{H'} \eta$.

Proof: The proof essentially consists in showing that the following diagram holds true for some sequence of leap sets $\Omega = T_1 T_2 \ldots T_m$, transition sequence ω and a global state H':



According to Lemma 5.24, there exist $T_1 \in xpleap(G)$ with $\gamma_1 \in lin(T_1)$, ω_1, ρ_1 and H^1 such that $\sigma\omega_1 \ {}_G\!\equiv_{H^1} \gamma_1\rho_1$ and $|\rho_1| < |\sigma|$. Let $G \xrightarrow{T_1} G^1 \xrightarrow{\rho_1}{}^* H^1$, then $|\rho_1| > 0$ equally implies the existence of $T_2 \in xpleap(G^1)$ with $\gamma_2 \in lin(T_2)$, ω_2, ρ_2 and H^2 such that $\rho_1\omega_2 \ {}_{G^1}\!\equiv_{H^2} \gamma_2\rho_2$ and $|\rho_2| < |\rho_1|$. As in the proof of Theorem 5.11, applying this argument m times yields $\sigma\omega_1\omega_2 \ldots \omega_m \ {}_G\!\equiv_{H^m} \gamma_1\gamma_2 \ldots \gamma_m\rho_m$. Since σ is finite and Lemma 5.24 can be applied as long as $|\rho_j| > 0$ $(1 \leq j \leq m)$, we may assume that $|\rho_m| = 0$, i.e. $\sigma\omega_1\omega_2 \ldots \omega_m \ {}_G\!\equiv_{H^m} \gamma_1\gamma_2 \ldots \gamma_m$. Let $\Omega = T_1 T_2 \ldots T_m$, $\eta = \gamma_1\gamma_2 \ldots \gamma_m \in lin(\Omega)$, $\omega = \omega_1\omega_2 \ldots \omega_m$ and $H' = H^m$, then $G \xrightarrow{\Omega}{}^* H'$ and $\sigma\omega \ {}_G\!\equiv_{H'} \eta$. □

Theorem 5.26

A transition t is executable iff t is executable at an ℓ^*-reachable global state.

Proof: The "if" part holds directly since $\mathbf{L}^*_{\Pi} \subseteq \mathbf{R}_{\Pi}$. For the "only-if" part, when t is executable there must be a reachable global state H such that $G^0 \xrightarrow{\mu}{}^* H$, for some transition sequence μ. By Lemma 5.25, there exist Ω with $\eta \in lin(\Omega)$, ω and H' such that $G^0 \xrightarrow{\Omega}{}^*_{\ell^*} H'$ and $\mu t \omega \,{}_{G^0}\!\equiv_{H'} \eta$. That is, t appears in η and is thus executable at an ℓ^*-reachable global state. □

Corollary 5.27

Freedom of non-executable transitions for a protocol Π is decidable if $\mathbf{L}^*_{\Pi}$ is finite. □

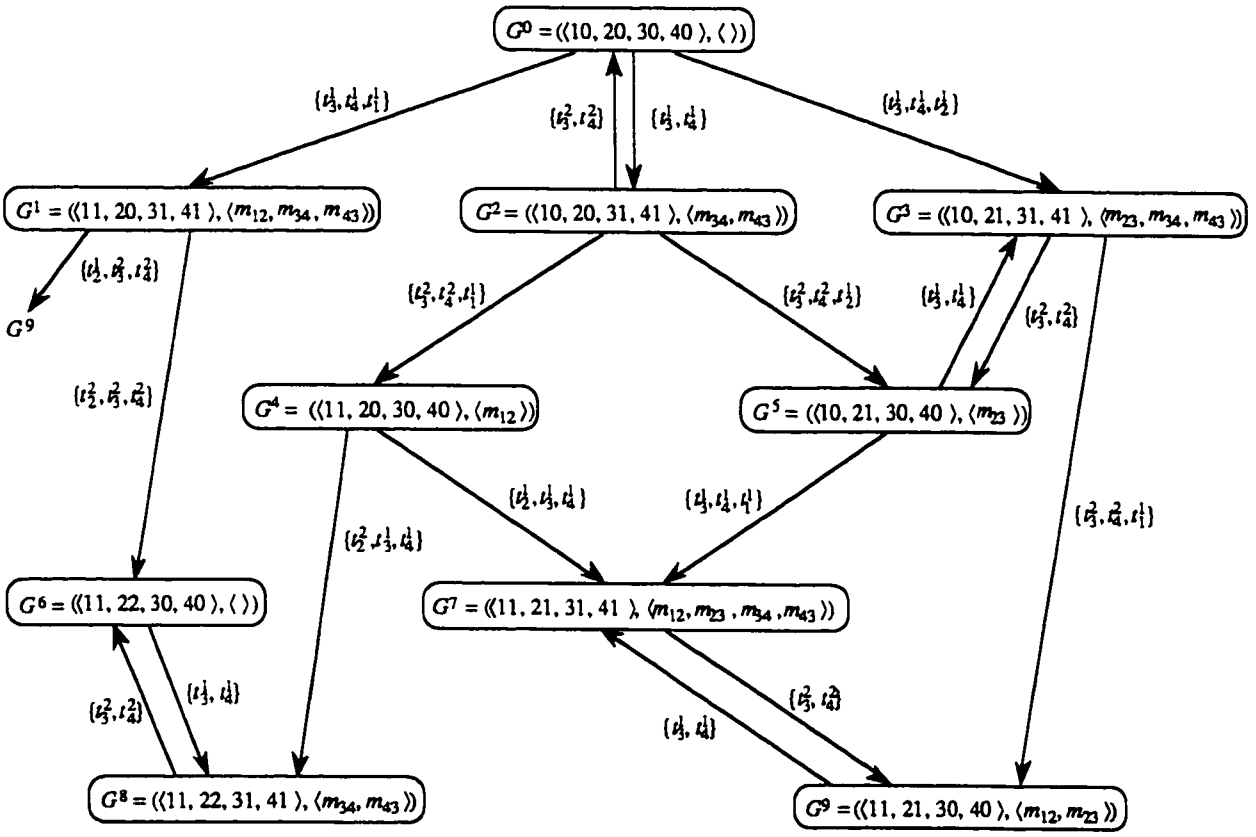


Figure 5.5 The ℓ^*-reachability graph of the protocol of Example 5.6.

Example 5.28

Figure 5.5 shows the ℓ^*-reachability graph of the protocol of Example 5.6, consisting of 10 ℓ^*-reachable global states and 18 global state transitions (empty channels are not indicated). For each ℓ^*-reachable global state G^k, Table 5.1 lists the data used to calculate the set $xpleap(G^k)$ of leap sets executed at G^k. The elements of $xpleap(G^k)$ are precisely the labels of the outgoing edges of

the node labeled G^k. Again the protocol is found to progress indefinitely (all the nodes in the ℓ^*-reachability graph have outgoing edges) and, moreover, state exploration by LRA based on $\xrightarrow{\ell^*}$ correctly reveals one non-executable transition, namely $t_1^2 = (10, +m_{41}, 12)$ (t_1^2 does not occur in the ℓ^*-reachability graph). Recall that a notably larger number of global states and transitions are explored in conventional reachability analysis (40 and 100, respectively). □

Table 5.1 Data for Figure 5.5.

Global state G^k	$X(G^k)$	$P(G^k)$	$wait(G^k)$	$pleap(G^k)$
G^0	$\{t_1^1, t_2^1, t_3^1, t_4^1\}$	$\{t_1^2, t_2^2\}$	$\{1, 2\}$	$\{\{t_3^1, t_4^1\}\}$
G^1	$\{t_2^1, t_2^2, t_3^2, t_4^2\}$	$\emptyset$	$\{1\}$	$\{\{t_2^1, t_3^2, t_4^2\}, \{t_2^2, t_3^2, t_4^2\}\}$
G^2	$\{t_1^1, t_2^1, t_3^2, t_4^2\}$	$\{t_1^2, t_2^2\}$	$\{1, 2\}$	$\{\{t_3^2, t_4^2\}\}$
G^3	$\{t_1^1, t_3^2, t_4^2\}$	$\emptyset$	$\{1, 2\}$	$\{\{t_3^2, t_4^2\}\}$
G^4	$\{t_2^1, t_2^2, t_3^1, t_4^1\}$	$\emptyset$	$\{1\}$	$\{\{t_2^1, t_3^1, t_4^1\}, \{t_2^2, t_3^1, t_4^1\}\}$
G^5	$\{t_1^1, t_3^1, t_4^1\}$	$\{t_1^2\}$	$\{1, 2\}$	$\{\{t_3^1, t_4^1\}\}$
G^6	$\{t_3^1, t_4^1\}$	$\emptyset$	$\{1, 2\}$	$\{\{t_3^1, t_4^1\}\}$
G^7	$\{t_3^2, t_4^2\}$	$\emptyset$	$\{1, 2\}$	$\{\{t_3^2, t_4^2\}\}$
G^8	$\{t_3^2, t_4^2\}$	$\emptyset$	$\{1, 2\}$	$\{\{t_3^2, t_4^2\}\}$
G^9	$\{t_3^1, t_4^1\}$	$\emptyset$	$\{1, 2\}$	$\{\{t_3^1, t_4^1\}\}$

As pointed out earlier, the ℓ-reachable global state space $\mathbf{L}_\Pi$ may be finite even for unbounded protocols. The same is true for $\mathbf{L}_\Pi^*$ as witnessed by the simple protocol with two processes P_1 and P_2, where P_1 consists of the single cyclic transition $(10, -m_{12}, 10)$ and P_2 of the single cyclic transition $(20, +m_{12}, 20)$. Assuming that the simplex channel from P_1 to P_2 is not prebounded, this protocol has an infinite number of reachable global states but only two ℓ^*-reachable global states. Yet, the class of unbounded protocols with finite $\mathbf{L}_\Pi^*$ is *properly* included in the class of unbounded protocols with finite $\mathbf{L}_\Pi$. This can be seen from the protocol of Example 5.6 with the cyclic transition $(10, -m_{12}, 10)$ replacing transition $(10, -m_{12}, 11)$ of process P_1. Recall from the discussion following Example 5.14 that the ℓ-reachability graph of this protocol is finite (see Figure 5.4). One may check that its ℓ^*-reachability graph is infinite.

5.4 Verifying freedom of unspecified receptions & buffer overflows

Despite its qualification to solve the ignoring problem and thereby the problem of detecting non-executable transitions, the ℓ^*-reachability relation is still unsuited for the detection of unspecified

receptions (ur-pairs) and buffer overflows (bo-pairs). The protocol of Example 5.6 serves again as an example. Two ur-pairs (30, m_{43}) and (40, m_{34}) occur in, for instance, the reachable global states ($\langle 10, 20, 30, 41\rangle$, $\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon, m_{43}\rangle$) and ($\langle 10, 20, 31, 40\rangle$, $\langle \varepsilon, \varepsilon, m_{34}, \varepsilon, \varepsilon\rangle$), but not in any ℓ^*-reachable global state (see Figure 5.5). Also, under the assumption that all simplex channels in the protocol are one-slot buffers, one may check that two bo-pairs (30, m_{34}) and (40, m_{43}) exist which do not emerge in the ℓ^*-reachability graph. In both cases the riddle relates to process P_3 and process P_4, which are always forced to progress in parallel as they are not indexed in any of the wait-sets of the ℓ^*-reachable global states (i.e. they always have an executable transition and no potentially executable transitions in these states).

In order to identify the crux of the matter, let us look at two scenarios. Regarding unspecified receptions, suppose we want to find all ur-pairs for a process P_i and specifically those that involve a message over simplex channel C_{ji}. Let G be the current global state and let y be the first message in C_{ji} if C_{ji} is not empty in G. Clearly, we can determine whether (s_i^G, y) is a ur-pair for P_i in G (i.e. a reception of y either is or is not defined at s_i^G) but, regardless, further progress of P_i is required to enable the detection of possible other ur-pairs for P_i with respect to C_{ji} in global states reachable from G. Put differently, in this case process P_i need not be forced to wait in G unless it has potentially executable transitions at G, as was the case before. The complication arises when channel C_{ji} is empty in G. It may then be the case that process P_j can send a message y to P_i, yielding a global state H with $s_i^H = s_i^G$ and $c_{ji}^H = y$, while (s_i^G, y) is a ur-pair for P_i in H (i.e. no reception of y is defined at s_i^G). Yet, with the ℓ^*-reachability relation P_i and P_j are forced to progress concurrently when $i, j \notin wait(G)$. That is, process P_i need not remain at s_i^G and hence the ur-pair (s_i^G, x) may not be detected.

Regarding buffer overflows, suppose similarly that the objective is to find all bo-pairs which involve a message over a (prebounded) channel C_{ij}. For a global state G, let $|c_{ij}^G| = B_{ij} - 1$ (i.e. C_{ij} can hold exactly one more message) and $t_1 = (s_i^G, -x_1, s_i^H) \in X_{ij}(G)$ (i.e. $G \xrightarrow{t_1} H$). In addition, let $t_2 = (s_i^H, -x_2, s) \in P_{ij}(H)$ and $t_3 = (s_j^G, +y, s') \in X_{ji}(G)$. Since t_2 is potentially executable at H, $(s_i^H, -x_2)$ is a bo-pair for P_i in H. If transitions t_1 and t_3 are executed concurrently at G, yielding a global state H', then $|c_{ij}^{H'}| = B_{ij} - 1$ (C_{ij} still has one slot available) and hence t_2 is executable at H'. In effect, we then leap over global state H and may thereby miss bo-pair $(s_i^H, -x_2)$. To detect it, process P_j must remain at s_j^G in G in particular because it has an executable receive transition pertaining to channel C_{ij}.

We conclude from the above scenarios that, in general, special attention must be given to a process when it has an incoming empty channel or an executable receive transition at the current global state. More precisely, in order to detect the ur-pairs (with respect to a selected subset J of channels) in a protocol, the processes *with an empty incoming channel* (in J) in a global state G should be treated the same way as processes with potentially executable transitions at G, i.e. they

should be forced to wait in G. Likewise, bo-pairs (with respect to a subset K of channels) can be detected by forcing the processes *with an executable receive transition* at G (from a channel in K) to wait in G.

Definition 5.29

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol and $J, K \subseteq L$. A ur-pair (s, y) for process P_i with $y \in M_{ji}$ is said to be a ur-pair *with respect to (wrt)* J iff $(j, i) \in J$. A bo-pair (s, x) for process P_i with $x \in M_{ij}$ is said to be a bo-pair wrt K iff $(i, j) \in K$. □

Thus, when the aim is to identify all ur-pairs (bo-pairs) for a certain process P_i, the index set J (K) should include every incoming (outgoing) channel of P_i.

5.4.1 $\ell(J, K)^*$-reachability

In this section the ℓ^*-reachability relation is parameterized with the channel-index sets J and K. In analogy with definitions 5.4 and 5.16, we define the sets $pleap(G, J, K)$ and $xpleap(G, J, K)$ on the basis of a wait-set $\text{wait}(G, J, K)$ which identifies not only the processes without executable transitions or with potentially executable transitions at G (as before), but also the processes with an incoming channel indexed in J that is empty in G and those with an executable receive transition at G from a channel indexed in K.

Definition 5.30

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ and $J, K \subseteq L$. Define $wait(G, J, K) = \{i \in I \mid X_i(G) \neq \emptyset \Rightarrow (P_i(G) \neq \emptyset \vee \exists (j, i) \in J\colon\ c_{ji}^G = \varepsilon \vee \exists (j, i) \in K\colon\ X_{ij}^+(G) \neq \emptyset)\}$ and

$$pleap(G, J, K) = \{\ T \mid T \in leap(G) \wedge act(T) = \{i \in I \mid i \notin wait(G, J, K)\}\ \}$$
$$\text{if } wait(G, J, K) \subset I$$

$$pleap(G, J, K) = \{\ \{t\} \mid t \in X(G)\ \}$$
$$\text{otherwise.}$$

□

Definition 5.31

Let G be a global state of a protocol Π and T an arbitrary element of $pleap(G, J, K)$. Define

$$xpleap(G, J, K) = pleap(G, J, K) \cup \{T \cup \{t\} \mid t \in X(G) \wedge act(t) \in wait(G, J, K)\}$$
$$\text{if } wait(G, J, K) \subset I$$

$$xpleap(G, J, K) = pleap(G, J, K)$$
$$\text{otherwise.}$$

□

Example 5.32

For the protocol of Example 5.6, let $G = (\langle 10, 21, 31, 41\rangle, \langle \varepsilon, m_{23}, m_{34}, \varepsilon, m_{43}\rangle)$ be a (reachable) global state, then $X(G) = \{ t_1^1, t_3^2, t_4^2 \}$ and $wait(G, \{(2, 3), (4, 3)\}, \{(3, 4)\}) = \{1, 2, 4\}$ since process P_1 has a potentially executable transition at G (viz. t_1^2), P_2 has no executable transitions at G, and P_4 has an executable receive transition at G involving channel C_{34}. Even though both incoming channels C_{23} and C_{43} of process P_3 are indexed, P_3 is not in the wait-set as both these channels are non-empty in G. Thus, we have $pleap(G, \{(2, 3), (4, 3)\}, \{(3, 4)\}) = \{ \{ t_3^2 \} \}$ and $xpleap(G, \{(2, 3), (4, 3)\}, \{(3, 4)\}) = \{ \{ t_3^2 \}, \{ t_3^2, t_1^1 \}, \{ t_3^2, t_4^2 \} \}$ □

Again, without affecting the upcoming results we regard $xpleap(G, J, K)$ as a unique set by selecting for extension always the first element of $pleap(G, J, K)$ when viewed as an ordered set. Note that $wait(G, \emptyset, \emptyset) = wait(G)$ and hence $pleap(G, \emptyset, \emptyset) = pleap(G)$. Further note that $J \subseteq J'$ and $K \subseteq K'$ implies $wait(G, J, K) \subseteq wait(G, J', K')$ but not $pleap(G, J, K) \subseteq pleap(G, J', K')$ nor $|pleap(G, J, K)| < |pleap(G, J', K')|$. Thus, in particular $pleap(G)$ and therefore $xpleap(G)$ are not necessarily subsets of respectively $pleap(G, J, K)$ and $xpleap(G, J, K)$. On the other hand, it is not hard to see that the properties of $pleap(G)$ and $xpleap(G)$ in propositions 5.5 and 5.17 hold similarly for $pleap(G, J, K)$ and $xpleap(G, J, K)$. Only those of $xpleap(G, J, K)$ are formulated for later reference.

Proposition 5.33

i) $t \in X(G) \Rightarrow \exists T \in xpleap(G, J, K): t \in T$;

ii) $T \in xpleap(G, J, K) \wedge i \in act(T) \cap wait(G, J, K) \wedge t \in X_i(G) \Rightarrow$
$(T \backslash X_i(G)) \cup \{t\} \in xpleap(G, J, K)$;

iii) $T \in xpleap(G, J, K) \wedge i \in act(T) \cap wait(G, J, K) \Rightarrow$
$T \backslash X_i(G) \in pleap(G, J, K) \vee xpleap(G, J, K) = \bigcup_{t \in X(G)} \{\{t\}\}$.

Proof: Straightforward from definitions 5.30 and 5.31. □

Naturally, the set $xpleap(G, J, K)$ induces a parameterized reachability relation, referred to as $\ell(J, K)^*$-reachability.

Definition 5.34

Let G and H be global states of a protocol Π. $G \xrightarrow[\ell(J,K)^*]{} H$ iff $\exists T \in xpleap(G, J, K)$ with $\gamma \in lin(T)$ such that $G \xrightarrow{\gamma}{}^* H$. This is also denoted by $G \xrightarrow[\ell(J,K)^*]{T} H$. □

Definition 5.35

Let G and H be global states of a Π, and denote by $\xrightarrow[\ell(J,K)^*]{}{}^*$ the reflexive and transitive closure of

$\xrightarrow{\ell(J,K)^*}$. H is $\ell(J,K)^*$-*reachable from* G iff $G \xrightarrow{\ell(J,K)^*}{}^* H$. If $G = G^0$, then H is said to be $\ell(J,K)^*$-*reachable*. The set of $\ell(J,K)^*$-reachable global states of Π is denoted by $\mathbf{L}(J,K)^*_\Pi$. For a sequence of leap sets $\Omega = T_1T_2\ldots T_m$, $G \xrightarrow{\Omega}_{\ell(J,K)^*}{}^* H$ denotes the existence of global states $Q^0, Q^1, \ldots, Q^m$ such that $G = Q^0 \xrightarrow{T_1}_{\ell(J,K)^*} Q^1 \xrightarrow{T_2}_{\ell(J,K)^*} \cdots \xrightarrow{T_m}_{\ell(J,K)^*} Q^m = H$. □

Proposition 5.36
$\mathbf{L}(J,K)^*_\Pi \subseteq \mathbf{R}_\Pi$

Proof: By definition of $\xrightarrow{\ell(J,K)^*}$. □

It is clear that $\mathbf{L}(\emptyset,\emptyset)^*_\Pi = \mathbf{L}^*_\Pi$ and thus $\mathbf{L}(\emptyset,\emptyset)^*_\Pi$ reveals all non-progress states and all non-executable transitions of a protocol (by Corollary 5.22 and Theorem 5.26). As we will show, the same holds true for $\mathbf{L}(J,K)^*_\Pi$ in general, but note that this cannot be established directly since $\mathbf{L}^*_\Pi$ need not be included in $\mathbf{L}(J,K)^*_\Pi$ (i.e. *xpleap*(G) is generally not a subset of *xpleap*(G, J, K)).

5.4.2 Detecting ur-pairs and bo-pairs

We continue by proving that state exploration based on $\xrightarrow{\ell(J,K)^*}{}^*$ preserves all non-progress states, all (non-)executable transitions, all unspecified receptions wrt J and all buffer overflows wrt K.

Lemma 5.37
Let $G \xrightarrow{\sigma}{}^* H$ and $\sigma \neq \varepsilon$, then there exist a leap set $T \in$ *xpleap*(G, J, K) with $\gamma \in$ *lin*(T), transition sequences ω, ρ and a global state H' such that

i) $\sigma\omega \underset{G}{\equiv}_{H'} \gamma\rho$, *act*$(\omega) \cap$ *act*$(\sigma) = \emptyset$ and $|\rho| < |\sigma|$;

ii) (s, y) is a ur-pair wrt J in $H \Rightarrow (s, y)$ is a ur-pair (wrt J) in G or in H';

iii) (s, x) is a bo-pair wrt K in $H \Rightarrow (s, x)$ is a bo-pair (wrt K) in G or in H'.

Proof:

i) Akin to the proof of Lemma 5.24 (and Lemma 5.10), mutatis mutandis. That is, substitute *pleap*(G, J, K) for *pleap*(G) and *xpleap*(G, J, K) for *xpleap*(G). Observe from this proof that $T \in$ *xpleap*(G, J, K) remains such that $T \backslash$ *first*(σ) is minimal and $\omega \in$ *lin*$(T \backslash$ *first*$(\sigma))$, with *first*$(\sigma) = \{t \mid t \in \Delta_i \wedge \mu t \in$ *pref*$(\sigma) \wedge i \notin$ *act*$(\mu)\}$.

ii) By definition, if (s, y) is a ur-pair wrt J in H then $s = s_i^H$, $y \in M_{ji}$ and *front*$(c_{ji}^H) = y$, for some $(j, i) \in J$. If $i \notin$ *act*(ω), then since $H \xrightarrow{\omega}{}^* H'$ we have $s_i^{H'} = s_i^H$ and thus (s_i^H, y) is a ur-pair in H'. Alternatively, if $i \in$ *act*(ω) then $i \notin$ *act*(σ) and thus $s_i^H = s_i^G$. Three cases must then be considered:

- $c_{ji}^G = \varepsilon$
  Since $i \in act(\omega)$ and $(j, i) \in J$ we have $i \in act(T) \cap wait(G, J, K)$ and consequently, by Proposition 5.33.(iii), $T \backslash X_i(G) \in pleap(G, J, K)$ or $xpleap(G, J, K) = \bigcup_{t \in X(G)} \{\{t\}\}$. Either case contradicts the minimality of $T \backslash first(\sigma)$;
- $front(c_{ji}^G) = y$
  Since (s_i^H, y) is a ur-pair in H and $s_i^H = s_i^G$, (s_i^H, y) is also a ur-pair in G;
- $front(c_{ji}^G) = z$, with $z \neq y$
  Since $i \notin act(\sigma)$ we have $front(c_{ji}^H) = z$, which contradicts the fact that $front(c_{ji}^H) = y$.

In conclusion, (s, y) is a ur-pair wrt J in G or in H'.

iii) By definition, if (s, x) is a bo-pair wrt K in H then $s = s_i^H$, $x \in M_{ij}$ and $|c_{ij}^H| = B_{ij}$, for some $(i, j) \in K$. Three cases are considered:

- $i \notin act(\sigma)$
  In this case, $s_i^H = s_i^G$ and $|c_{ij}^G| = B_{ij}$ (no message is sent to C_{ij} or received from C_{ij} along σ, since otherwise $i \in act(\sigma)$ or $|c_{ij}^H| < B_{ij}$). Thus, (s_i^H, x) is a bo-pair in G;
- $i \in act(\sigma), j \notin act(\omega)$
  Since $act(\omega) \cap act(\sigma) = \emptyset$ we have $i \notin act(\omega)$. Clearly, $H \xrightarrow{\omega}{}^* H'$ and $i, j \notin act(\omega)$ imply that $s_i^H = s_i^{H'}$ and $|c_{ij}^H| = B_{ij}$, i.e. (s, x) is a bo-pair in H';
- $i \in act(\sigma), j \in act(\omega)$
  Again, $i \in act(\sigma)$ implies $i \notin act(\omega)$. Since $\omega \in lin(T \backslash first(\sigma))$ and $T \backslash first(\sigma) \in leap(G)$, there is exactly one transition t from process P_j in ω and $t \in X_j(G)$. Clearly, if t does not entail a reception from channel C_{ij}, then (s_i^H, x) is a bo-pair in H'. On the other hand, if t does entail a reception from C_{ij} we have $j \in act(T) \cap wait(G, J, K)$ since $j \in act(\omega)$ and $(i, j) \in K$. Again, $T \backslash X_j(G) = (T \backslash \{t\}) \in pleap(G, J, K)$ or $xpleap(G, J, K) = \bigcup_{t \in X(G)} \{\{t\}\}$, by Proposition 5.33.(iii), and either case contradicts the minimality of $T \backslash first(\sigma)$.

In conclusion, (s, x) is a bo-pair wrt K in G or in H'. □

Lemma 5.38

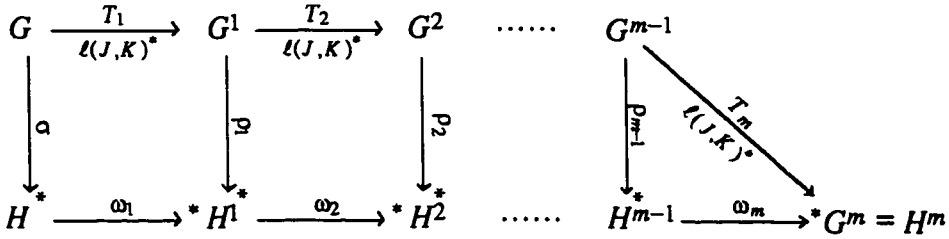
Let $G \xrightarrow{\sigma}{}^* H$ and $\sigma \neq \varepsilon$, then there exist a sequence of leap sets Ω with $\eta \in lin(\Omega)$, a transition sequence ω and a global state H' such that $G \xrightarrow[\ell(J,K)^*]{\Omega}{}^* H'$ and

i) $\sigma\omega \underset{G}{\equiv}_{H'} \eta$;

ii) (s, y) is a ur-pair wrt J in $H \Rightarrow \exists \Omega' \in pref(\Omega)$: $G \xrightarrow[\ell(J,K)^*]{\Omega'}{}^* G'$ and (s, y) is a ur-pair in G';

iii) (s, x) is a bo-pair wrt K in $H \Rightarrow \exists \Omega' \in pref(\Omega)$: $G \xrightarrow[\ell(J,K)^*]{\Omega'}{}^* G'$ and (s, x) is a bo-pair in G'.

Proof: Repeated application of Lemma 5.37.(i) yields the following diagram (cf. Lemma 5.25):



i) Let $\Omega = T_1T_2...T_m$ with $\eta \in lin(\Omega)$, $\omega = \omega_1\omega_2...\omega_m$ and $H' = H^m$, then $G \xrightarrow[\ell(J,K)^*]{\Omega}{}^* H'$ and $\sigma\omega \; {}_G{\equiv}_{H'} \eta$;

ii) Assume that (s, y) is a ur-pair wrt J in H, then by Lemma 5.37.(ii), (s, y) is a ur-pair wrt J in G or in H^1. The claim trivially holds if (s, y) is a ur-pair wrt J in G (let $\Omega' = \varepsilon$). On the other hand, if (s, y) is a ur-pair wrt J in H^1, then again by Lemma 5.37.(ii) ($|\rho_1| > 0$), (s, y) must be a ur-pair wrt J in G^1 or in H^2. By repeating this argument it follows easily that (s, y) must be a ur-pair wrt J in (at least) one of the $\ell(J, K)^*$-reachable global states $G^1, G^2,..., G^m$, say in G^j, in particular because $G^m = H^m$. The claim then holds by choosing $\Omega' = T_1T_2...T_j$;

iii) Analogous to the proof of part (ii), using Lemma 5.37.(iii). □

Note the resemblance between Lemma 5.38.(i) and Lemma 5.25. As a consequence, the results established for $\mathbf{L}^*_\Pi$ also hold for $\mathbf{L}(J,K)^*_\Pi$, viz. exploring the $\ell(J, K)^*$-reachability graph of a protocol suffices to detect all non-progress states and all non-executable transitions (independent of J and K). Lemma 5.38.(ii) and (iii) yield the result anticipated for unspecified receptions and buffer overflows.

Theorem 5.39

Every non-progress state is $\ell(J, K)^*$-reachable, for all $J, K \subseteq L$.

Proof: Let H be a non-progress state with $G^0 \xrightarrow{\sigma}{}^* H$. H is trivially $\ell(J, K)^*$-reachable if $|\sigma| = 0$. If $|\sigma| > 0$, then by Lemma 5.38.(i) and the fact that H is a non-progress state (i.e. $\omega = \varepsilon$), there is a sequence of leap sets Ω with $\eta \in lin(\Omega)$ such that $G^0 \xrightarrow[\ell(J,K)^*]{\Omega}{}^* H$ and $\sigma \; {}_{G^0}{\equiv}_H \eta$. Again, H is $\ell(J, K)^*$-reachable. □

Theorem 5.40

A transition t is executable iff t is executable at an $\ell(J, K)^*$-reachable global state, for all $J, K \subseteq L$.

Proof: The "if" part holds directly since $\mathbf{L}(J,K)^*_\Pi \subseteq \mathbf{R}_\Pi$. For the "only-if" part, when t is executable there exists a global state H such that $G^0 \xrightarrow{\mu}{}^* H$, for some transition sequence μ. By Lemma 5.38.(i), there exist Ω with $\eta \in lin(\Omega)$, ω and H' such that $G^0 \xrightarrow[\ell(J,K)^*]{\Omega}{}^* H'$ and $\mu t\omega \; {}_{G^0}{\equiv}_{H'} \eta$. Hence, since t appears in η it must be executable at an $\ell(J, K)^*$-reachable global state. □

Theorem 5.41

If (s, y) is a ur-pair wrt J, then (s, y) is a ur-pair in an $\ell(J, K)^*$-reachable global state, for all $K \subseteq L$.

Proof: By definition, (s, y) is a ur-pair wrt J in some reachable global state H. Let $G^0 \xrightarrow{\sigma}{}^* H$ for some transition sequence σ. The theorem holds trivially if $|\sigma| = 0$. For the case $|\sigma| > 0$, the proof is immediate from Lemma 5.38.(ii). □

Theorem 5.42

If (s, x) is a bo-pair wrt K, then (s, x) is a bo-pair in an $\ell(J, K)^*$-reachable global state, for all $J \subseteq L$.

Proof: By definition, (s, x) is a bo-pair wrt K in some reachable global state H. Let $G^0 \xrightarrow{\sigma}{}^* H$ for some transition sequence σ. The theorem holds trivially if $|\sigma| = 0$. For the case $|\sigma| > 0$, the proof is immediate from Lemma 5.38.(iii). □

The detection of all unspecified receptions in a protocol is thus guaranteed by choosing $J = L$ and K arbitrary, and vice versa for buffer overflows. Of course, for protocols whose channels are not prebounded the detection of buffer overflows is immaterial and K should be set to empty.

Corollary 5.43

For a protocol Π, indefinite progress, freedom of non-executable transitions, and freedom of unspecified receptions (wrt J) and buffer overflows (wrt K) are decidable if $\mathbf{L}(J,K)^*_\Pi$ is finite. □

It is obvious from the presentation that the task of detecting unspecified receptions and buffer overflows in a protocol can be performed via multiple independent subtasks, simply by partitioning the index set L of all channels. Surely, this is an important feature of the parameterized reachability relation $\xrightarrow{\ell(J,K)^*}{}^*$ since in many cases each subtask utilizes a smaller number of $\ell(J, K)^*$-reachable global states and transitions than the corresponding full task. The memory needed for verification may then be reduced further by executing the subtasks in sequence on a single processor, whereas the time consumption may decrease by executing them in parallel at the expense of using multiple processors.

Example 5.44

Figure 5.6 shows in part the $\ell(J, K)^*$-reachability graph of the protocol of Example 5.6 for $J = L = \{(1, 2), (2, 3), (3, 4), (4, 1), (4, 3)\}$ and $K = \emptyset$ (empty channels in the states are omitted and dashed arrows indicate incomplete paths). For each $\ell(L, \emptyset)^*$-reachable global state G^k in the figure, Table 5.2 lists the data used to calculate the set $xpleap(G^k, L, \emptyset)$ of leap sets executed at G^k. The elements of $xpleap(G^k, L, \emptyset)$ are precisely the labels of the outgoing edges of the node labeled G^k. The complete $\ell(L, \emptyset)^*$-reachability graph of the protocol of Example 5.6 consists of 29 nodes and

69 edges versus respectively 40 and 100 for the conventional reachability graph. State exploration by LRA based on $\xrightarrow{\ell(J,K)^*}$ correctly determines that the protocol has no non-progress states and one non-executable transition, viz. t_1^2 = (10, $+m_{41}$, 12). It also reveals all unspecified receptions, viz. (21, m_{12}), (30, m_{23}), (30, m_{43}), (31, m_{23}) and (40, m_{34}).

To illustrate the extra space reduction that can be obtained by dividing the task of detecting all unspecified receptions into several independent subtasks, consider the partitioning of $J = L$ into the disjoint subsets J_1 = {(4, 1), (1, 2)} (i.e. the incoming channels of processes P_1 and P_2), J_2 =

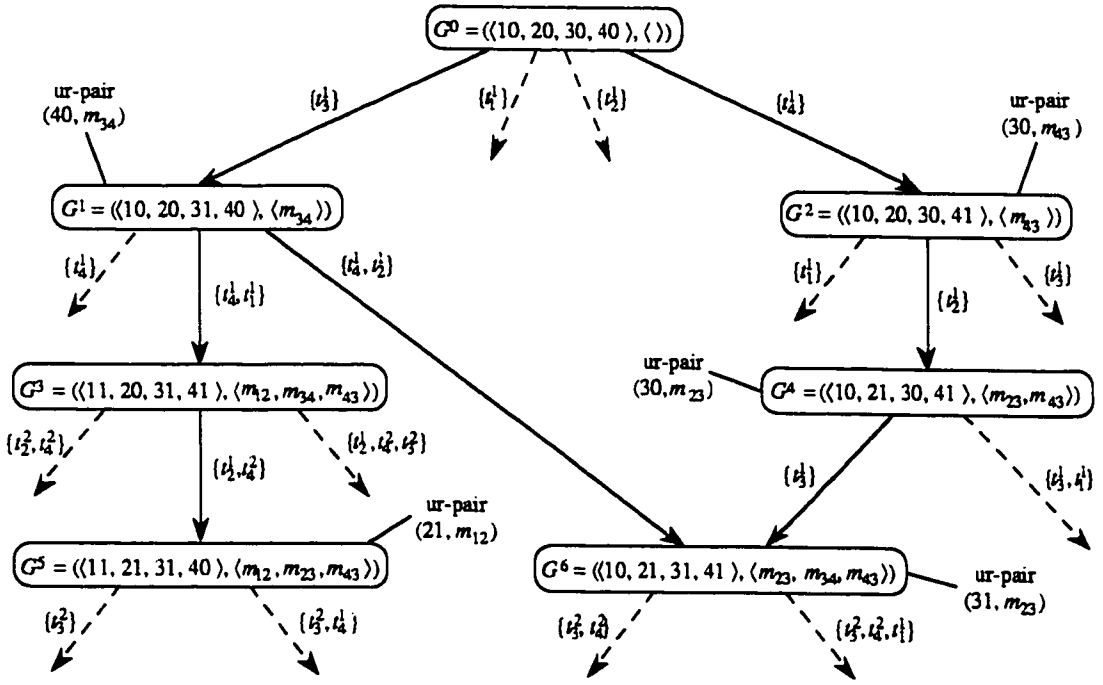


Figure 5.6 The partial $\ell(L, \emptyset)^*$-reachability graph of the protocol of Example 5.6.

Table 5.2 Data for Figure 5.6.

Global state G^k	$X(G^k)$	$P(G^k)$	$wait(G^k, L, \emptyset)$	$pleap(G^k, L, \emptyset)$
G^0	$\{t_1^1, t_2^1, t_3^1, t_4^1\}$	$\{t_1^2, t_2^2\}$	I	$\{\{t_1^1\}, \{t_2^1\}, \{t_3^1\}, \{t_4^1\}\}$
G^1	$\{t_1^1, t_2^1, t_4^1\}$	$\{t_1^2, t_2^2, t_3^2\}$	{1, 2, 3}	$\{\{t_4^1\}\}$
G^2	$\{t_1^1, t_2^1, t_3^1\}$	$\{t_1^2, t_2^2, t_4^2\}$	I	$\{\{t_1^1\}, \{t_2^1\}, \{t_3^1\}\}$
G^3	$\{t_2^1, t_2^2, t_3^2, t_4^2\}$	$\emptyset$	{1, 3}	$\{\{t_2^1, t_4^2\}, \{t_2^2, t_4^2\}\}$
G^4	$\{t_1^1, t_3^1\}$	$\{t_1^2, t_4^2\}$	{1, 2, 4}	$\{\{t_3^1\}\}$
G^5	$\{t_3^2, t_4^1\}$	$\emptyset$	{1, 2, 4}	$\{\{t_3^2\}\}$
G^6	$\{t_1^1, t_3^2, t_4^2\}$	$\{t_1^2\}$	{1, 2}	$\{\{t_3^2, t_4^2\}\}$

{(2, 3), (4, 3)} (i.e. the incoming channels of process P_3) and J_3 = {(3, 4)} (i.e. the incoming channel of process P_4). We obtain the following results:

- the $\ell(J_1, \emptyset)^*$-reachability graph consists of 10 nodes and 18 edges (this graph is in fact the same as the ℓ^*-reachability graph in Figure 5.5) – three ur-pairs are detected, including all those for processes P_1 and P_2 (if any): (21, m_{12}), (30, m_{23}), (31, m_{23});
- the $\ell(J_2, \emptyset)^*$-reachability graph consists of 22 nodes and 51 edges – four ur-pairs are detected, including all those for process P_3: (21, m_{12}), (30, m_{23}), (31, m_{23}) *and* (30, m_{43});
- the $\ell(J_3, \emptyset)^*$-reachability graph consists of 15 nodes and 32 edges – four ur-pairs are detected, including all those for process P_4: (21, m_{12}), (30, m_{23}), (31, m_{23}) *and* (40, m_{34}).

All five unspecified receptions are thus detected and since the $\ell(J_2, \emptyset)^*$-reachability graph is the largest one constructed, it represents the total space required for this analysis. Only 22 instead of 29 nodes need be stored, i.e. additional space reduction is achieved at the expense of additional running time (when using only one processor). □

Example 5.45

Consider the protocol of Example 5.6 with a capacity bound of one message for each channel. Its $\ell(J, K)^*$-reachability graph is shown in part in Figure 5.7 for $J = \emptyset$ and $K = L$ (empty channels in the states are omitted and dashed arrows indicate incomplete paths). For every $\ell(\emptyset, L)^*$-reachable global state G^k in the figure, Table 5.3 provides the data used to calculate $xpleap(G^k, \emptyset, L)$, the elements of which match the labels of the outgoing edges of the node labeled G^k. The complete

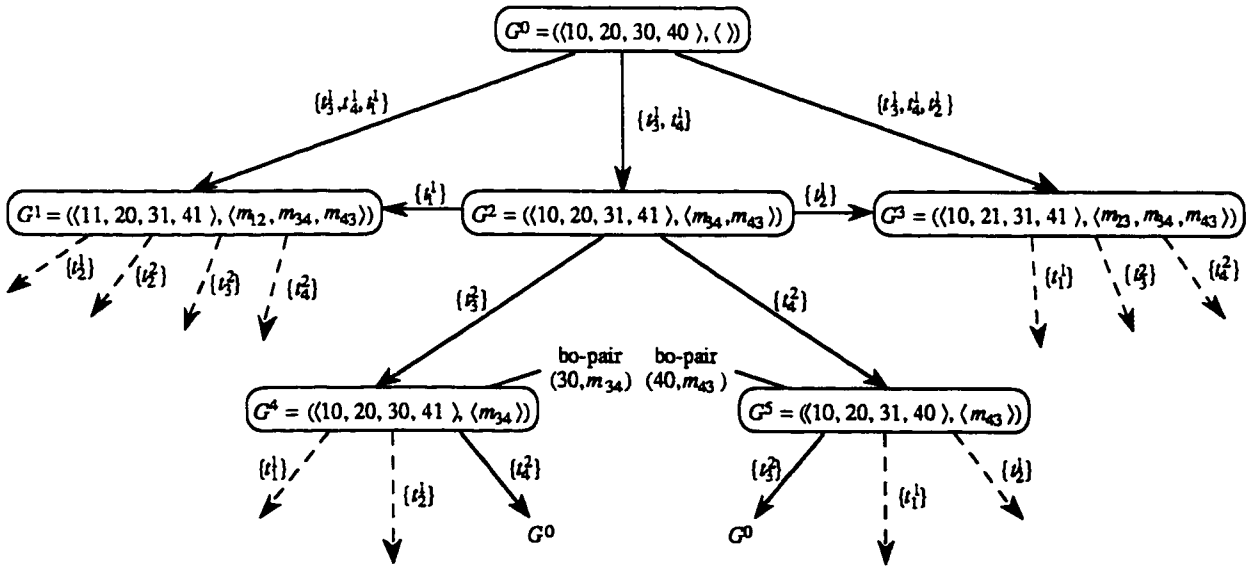


Figure 5.7 The partial $\ell(\emptyset, L)^*$-reachability graph of the protocol of Example 5.6 (with $B_{ij} = 1$ for all $(i, j) \in L$).

Table 5.3 Data for Figure 5.7.

Global state G^k	$X(G^k)$	$P(G^k)$	$wait(G^k, \emptyset, L)$	$pleap(G^k, \emptyset, L)$
G^0	$\{t_1^1, t_2^1, t_3^1, t_4^1\}$	$\{t_1^2, t_2^2\}$	$\{1, 2\}$	$\{\{t_3^1, t_4^1\}\}$
G^1	$\{t_2^1, t_2^2, t_3^2, t_4^2\}$	$\emptyset$	1	$\{\{t_2^1\}, \{t_2^2\}, \{t_3^2\}, \{t_4^2\}\}$
G^2	$\{t_1^1, t_2^1, t_3^2, t_4^2\}$	$\{t_1^2, t_2^2\}$	1	$\{\{t_1^1\}, \{t_2^1\}, \{t_3^2\}, \{t_4^2\}\}$
G^3	$\{t_1^1, t_3^2, t_4^2\}$	$\{t_1^2\}$	1	$\{\{t_1^1\}, \{t_3^2\}, \{t_4^2\}\}$
G^4	$\{t_1^1, t_2^1, t_4^2\}$	$\{t_1^2, t_2^2, t_3^1\}$	1	$\{\{t_1^1\}, \{t_2^1\}, \{t_4^2\}\}$
G^5	$\{t_1^1, t_2^1, t_3^2\}$	$\{t_1^2, t_2^2, t_4^1\}$	1	$\{\{t_1^1\}, \{t_2^1\}, \{t_3^2\}\}$

$\ell(\emptyset, L)^*$-reachability graph of the protocol of Example 5.6 consists of 20 nodes and 45 edges versus respectively 30 and 70 for the conventional reachability graph (which is different from the previous examples due to the imposed channel bounds). Additional reductions are obtained when $K = L$ is partitioned into disjoint subsets, similar as in Example 5.44. Once again, the protocol is found to have no non-progress states and one non-executable transition. Only three of five ur-pairs are detected in this case, which is not unexpected since $J = \emptyset$. Lastly, two bo-pairs $(30, m_{34})$ and $(40, m_{43})$ are detected as well, indicating that channels C_{34} and C_{43} require a bound larger than one (a bound of two messages turns out to be sufficient here). □

5.5 More reduction with a depth-first search

Thus far in this chapter no specific search technique has been assumed in the formulation of LRA. In particular, the $\ell(J, K)^*$-reachable (or ℓ-reachable) global state space of a protocol can be searched in depth-first order as well as in breadth-first order (given of course that the search space is finite). As discussed in Chapter 2, a depth-first search uses a stack to trace the global states encountered but not yet expanded during the search, while a breadth-first search uses a queue for this purpose. In this section we explicitly assume a depth-first search (DFS) to accommodate a further reduction of the number of global states and transitions explored for detecting non-executable transitions, unspecified receptions and buffer overflows.

The use of a DFS can be turned into an opportunity to refine the $\ell(J, K)^*$-reachability relation. Essentially, it warrants a more accurate solution to the ignoring problem than the one given earlier in terms of the set $xpleap(G, J, K)$. Recall from the beginning of Section 5.3 that the ignoring problem occurs when the progress of some processes is indefinitely deferred along a *cycle* in the ℓ-reachability graph. This cannot happen in the $\ell(J, K)^*$-reachability graph since it is guaranteed that $\bigcup_{T \in xpleap(G,J,K)} T = X(G)$ for every $\ell(J, K)^*$-reachable global state G (cf. Proposition 5.33.(i)), and

thus in particular for *all* such states that constitute a cycle in the $\ell(J,K)^*$-reachability graph. Simple graph-based reasoning shows, however, that it is sufficient to have in each cycle just *one* global state with this property. Avoiding the ignoring problem then becomes a matter of detecting cycles during state exploration, for which a DFS lends itself preeminently: a cycle is detected precisely when a global state reached from the current global state already resides on the DFS stack [CLR90, Pel96]. This is translated into an extra condition for extending sets of proper leap sets, viz. for a global state G the set $pleap(G,J,K)$ is extended only if (1) $wait(G,J,K) \subset I$ (as in Definition 5.31) *and* (2) at least one element of $pleap(G,J,K)$ leads from G to a global state on the DFS stack.

Definition 5.46

Let G be a global state of a protocol Π to be expanded during the DFS, and let T be an arbitrary element of $pleap(G,J,K)$. Define

$$xpleap\text{-}2(G,J,K) = pleap(G,J,K) \cup \{T \cup \{t\} \mid t \in X(G) \wedge act(t) \in wait(G,J,K)\}$$
$$\text{if } wait(G,J,K) \subset I \text{ and } \exists T' \in pleap(G,J,K)\colon G \xrightarrow[\ell(J,K)^*]{T'} H \wedge H \text{ is on the DFS stack}$$
$$xpleap\text{-}2(G,J,K) = pleap(G,J,K)$$
$$\text{otherwise.}$$

□

The reachability relation resulting from the execution of the leap sets in $xpleap\text{-}2(G,J,K)$ in global states is referred to accordingly as $\ell 2(J,K)^*$-reachability and denoted by $\xrightarrow[\ell 2(J,K)^*]{}{}^*$. The set of $\ell 2(J,K)^*$-reachable global states of a protocol Π (i.e. $\ell 2(J,K)^*$-reachable from its initial global state) is denoted by $\mathbf{L2}(J,K)^*_\Pi$. When $J = K = \emptyset$, we may drop these index sets from the notations, as in Section 5.4. Under the assumption that it takes constant time to check whether a global state is on the DFS stack (which can indeed be implemented efficiently by a simple hash-table look-up), the extra cost incurred for computing $xpleap\text{-}2(G,J,K)$ instead of $xpleap(G,J,K)$ is $O(|pleap(G,J,K)|)$ in the worst case. On the other hand, it should be clear from the definitions that $xpleap\text{-}2(G,J,K) \subseteq xpleap(G,J,K)$, for any global state G. The $\ell 2(J,K)^*$-reachability graph of a given protocol is thus a subgraph of its $\ell(J,K)^*$-reachability graph. Furthermore, by construction, for every cycle in the $\ell 2(J,K)^*$-reachability graph there exists *at least one* global state G in that cycle for which $\bigcup_{T \in xpleap\text{-}2(G,J,K)} T = X(G)$.

Proposition 5.47

$\mathbf{L2}(J,K)^*_\Pi \subseteq \mathbf{L}(J,K)^*_\Pi$

Proof: Since $xpleap\text{-}2(G,J,K) \subseteq xpleap(G,J,K)$ for any global state G, it is immediate that every $\ell 2(J,K)^*$-reachable global state is $\ell(J,K)^*$-reachable. □

The next two lemmas come in place of Lemma 5.37 and Lemma 5.38 in Section 5.4.2, proving that the $\ell 2(J, K)^*$-reachability graph of a protocol equally reveals all non-progress states, all non-executable transitions, all unspecified receptions wrt J and all buffer overflows wrt K.

Lemma 5.48

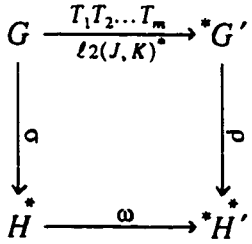
Let G be a global state that is removed from the DFS stack during the construction of the $\ell 2(J, K)^*$-reachability graph, with $G \xrightarrow{\sigma}{}^* H$ and $\sigma \neq \varepsilon$, then there exist a sequence $G \xrightarrow[\ell 2(J,K)^*]{T_1} G^1 \xrightarrow[\ell 2(J,K)^*]{T_2} \cdots \xrightarrow[\ell 2(J,K)^*]{T_m} G^m$ with $\gamma_i \in lin(T_i)$, transition sequences ω, ρ and a global state H' such that

i) $\sigma\omega \;{}_G{\equiv}_{H'}\; \gamma_1\gamma_2\cdots\gamma_m\rho$, $act(\omega) \cap act(\sigma) = \emptyset$ and $|\rho| < |\sigma|$;

ii) (s, y) is a ur-pair wrt J in $H \Rightarrow (s, y)$ is a ur-pair (wrt J) in G or in H';

iii) (s, x) is a bo-pair wrt K in $H \Rightarrow (s, x)$ is a bo-pair (wrt K) in G or in H'.

Proof: We prove only part (i). Part (ii) and (iii) are derived from part (i) in exactly the same way as Lemma 5.37.(ii) and (iii) are derived from Lemma 5.37.(i), particularly because the property of $xpleap(G, J, K)$ stated by Proposition 5.33.(iii) also holds for $xpleap\text{-}2(G, J, K)$ (cf. the proof of Lemma 5.37). Part (i) is visualized as follows:



Analogous to the proof of Lemma 5.10, it can be shown that there exist $T' \in pleap(G, J, K) \subseteq xpleap\text{-}2(G, J, K)$ with $\gamma' \in lin(T')$, transition sequences ω' and ρ', and a global state H' such that $\sigma\omega' \;{}_G{\equiv}_{H'}\; \gamma'\rho'$, $act(\omega') \cap act(\sigma) = \emptyset$ and $|\rho'| \leq |\sigma|$. In particular, T' and ω' are such that $T'\backslash first(\sigma)$ is minimal and $\omega' \in lin(T'\backslash first(\sigma))$, where $first(\sigma) = \{t \mid t \in \Delta_i \wedge \mu t \in pref(\sigma) \wedge i \notin act(\mu)\}$. Clearly, if $T'\backslash first(\sigma) \subset T'$ then $|\omega'| < |\gamma'|$ and thus $|\rho'| < |\sigma|$. Consequently, in this case the lemma holds by letting $m = 1$, $T_1 = T'$, $\gamma_1 = \gamma'$, $\omega = \omega'$ and $\rho = \rho'$.

Alternatively, if $T'\backslash first(\sigma) = T'$ then $\omega' \equiv \gamma' \neq \varepsilon$ and $act(\sigma) \cap act(T') = \emptyset$. We must have $wait(G, J, K) \subset I$ because otherwise $pleap(G, J, K) = \bigcup_{i \in X(G)} \{\{t\}\}$ and thus $\{t'\} \in pleap(G, J, K)$, with t' the first transition of σ, but $|\{t'\}\backslash first(\sigma)| = 0 < |T'\backslash first(\sigma)|$ contradicting the minimality of $T'\backslash first(\sigma)$. Definition 5.30 then states that $\forall T \in pleap(G, J, K)$: $act(T) = \{i \in I \mid i \notin wait(G, J, K)\}$, and since $act(\sigma) \cap act(T') = \emptyset$ it follows further that $\forall T \in pleap(G, J, K)$: $act(\sigma) \cap act(T) = \emptyset$ and $act(\sigma) \subseteq wait(G, J, K)$. In particular the first transition t' of σ is therefore not in any element of $pleap(G, J, K)$. The proof now continues by induction on the order in which $\ell 2(J, K)^*$-reachable global states are removed from the DFS stack. Two cases can arise when removing G from the stack:

- some $T \in pleap(G, J, K)$ leads to a global state that is already on the DFS stack

  Remark that this case covers in particular the induction basis where G is the first state removed from the DFS stack: each set in $pleap(G, J, K)$ executed in G leads back to a global state on the DFS stack since any other global state would have been removed before G (a characteristic of a depth-first search). According to Definition 5.46, some element of $pleap(G, J, K)$, say T'', is extended to form $xpleap\text{-}2(G, J, K)$. Now choose $T_1 = T'' \cup \{t'\}$ (t' is the first transition of σ), then $T_1 \in xpleap\text{-}2(G, J, K)$, $T_1 \backslash first(\sigma) = T'' \backslash first(\sigma) = T''$ and $act(T_1) \cap act(\sigma) = act(t')$. The lemma holds again by letting $m = 1$, $\gamma_1 \in lin(T_1)$, $\omega \in lin(T'')$ and $\sigma = t'\rho$, viz. $\sigma\omega \ {}_G{\equiv}_{H'} \ \gamma_1\rho$ for some global state H', $act(\omega) \cap act(\sigma) = \emptyset$ and $|\rho| < |\sigma|$.

- no $T \in pleap(G, J, K)$ leads to a global state that is already on the DFS stack

  Let $T_1 = T'$ and $G \xrightarrow[\ell 2(J,K)^*]{T_1} G^1$. Since $T_1 \in pleap(G, J, K)$, G^1 is not on the DFS stack and when added it will be removed before G itself is removed (a characteristic of a depth-first search). We also know that $act(\sigma) \cap act(T_1) = \emptyset$ and hence σ can still be executed from G^1. But this means that the induction hypothesis can be applied to G^1, viz. there exist a sequence $G^1 \xrightarrow[\ell 2(J,K)^*]{T_2} \cdots \xrightarrow[\ell 2(J,K)^*]{T_m} G^m$ with $\gamma_i \in lin(T_i)$, transition sequences ω' and ρ', and a global state H' such that $\sigma\omega' \ {}_{G^1}{\equiv}_{H'} \ \gamma_2 \ldots \gamma_m\rho'$, $act(\omega') \cap act(\sigma) = \emptyset$ and $|\rho'| < |\sigma|$. It follows that $\gamma_1\sigma\omega' \ {}_G{\equiv}_{H'} \ \sigma\gamma_1\omega' \ {}_G{\equiv}_{H'} \ \gamma_1\gamma_2 \ldots \gamma_m\rho'$ and $act(\gamma_1\omega') \cap act(\sigma) = \emptyset$. The lemma thus holds with $\omega = \gamma_1\omega'$ and $\rho = \rho'$. □

Lemma 5.48 differs from Lemma 5.37 mainly in stipulating the existence of *a sequence* of leap sets that satisfies the three stated properties, instead of a single leap set. Notice indeed that the sequence sought in Lemma 5.48 is guaranteed to be of length one when applying the lemma to the $\ell(J, K)^*$-reachability graph of a protocol.

Lemma 5.49

Let G be a global state that is removed from the DFS stack during the construction of the $\ell 2(J, K)^*$-reachability graph, with $G \xrightarrow{\sigma}{}^* H$ and $\sigma \neq \varepsilon$, then there exist a sequence of leap sets Ω with $\eta \in lin(\Omega)$, a transition sequence ω and a global state H' such that $G \xrightarrow[\ell 2(J,K)^*]{\Omega}{}^* H'$ and

i) $\sigma\omega \ {}_G{\equiv}_{H'} \ \eta$;

ii) (s, y) is a ur-pair wrt J in $H \Rightarrow \exists \Omega' \in pref(\Omega)$: $G \xrightarrow[\ell 2(J,K)^*]{\Omega'}{}^* G'$ and (s, y) is a ur-pair in G';

iii) (s, x) is a bo-pair wrt K in $H \Rightarrow \exists \Omega' \in pref(\Omega)$: $G \xrightarrow[\ell 2(J,K)^*]{\Omega'}{}^* G'$ and (s, x) is a bo-pair in G'.

Proof: By repeated application of Lemma 5.48, akin to the proof of Lemma 5.38. □

Lemma 5.49 is in effect identical to Lemma 5.38. Hence, the results stated in theorems 5.39-5.42

also hold for the $\ell 2(J, K)^*$-reachability relation (see the proofs of these theorems). We summarize with a corollary.

Corollary 5.50

For a protocol Π, indefinite progress, freedom of non-executable transitions, and freedom of unspecified receptions wrt J and buffer overflows wrt K are decidable if $\mathbf{L2}(J,K)^*_\Pi$ is finite. □

Example 5.51

The $\ell 2^*$-reachability graph (i.e. $J = K = \emptyset$) of the protocol of Example 5.6 is shown in Figure 5.8, as part of the ℓ^*-reachability graph already given in Figure 5.5. The dashed nodes and edges indicate the ℓ^*-reachable global states and transitions that are no longer explored, i.e. the $\ell 2^*$-reachability graph consists of only 9 nodes and 13 edges versus 10 nodes and 18 edges for the ℓ^*-reachability graph. Observe that this difference is due in particular to the global states G^0 and G^5, where $xpleap\text{-}2(G^k) \subset xpleap(G^k)$ since the proper leap sets in these states do not close a cycle on the "current" DFS stack. □

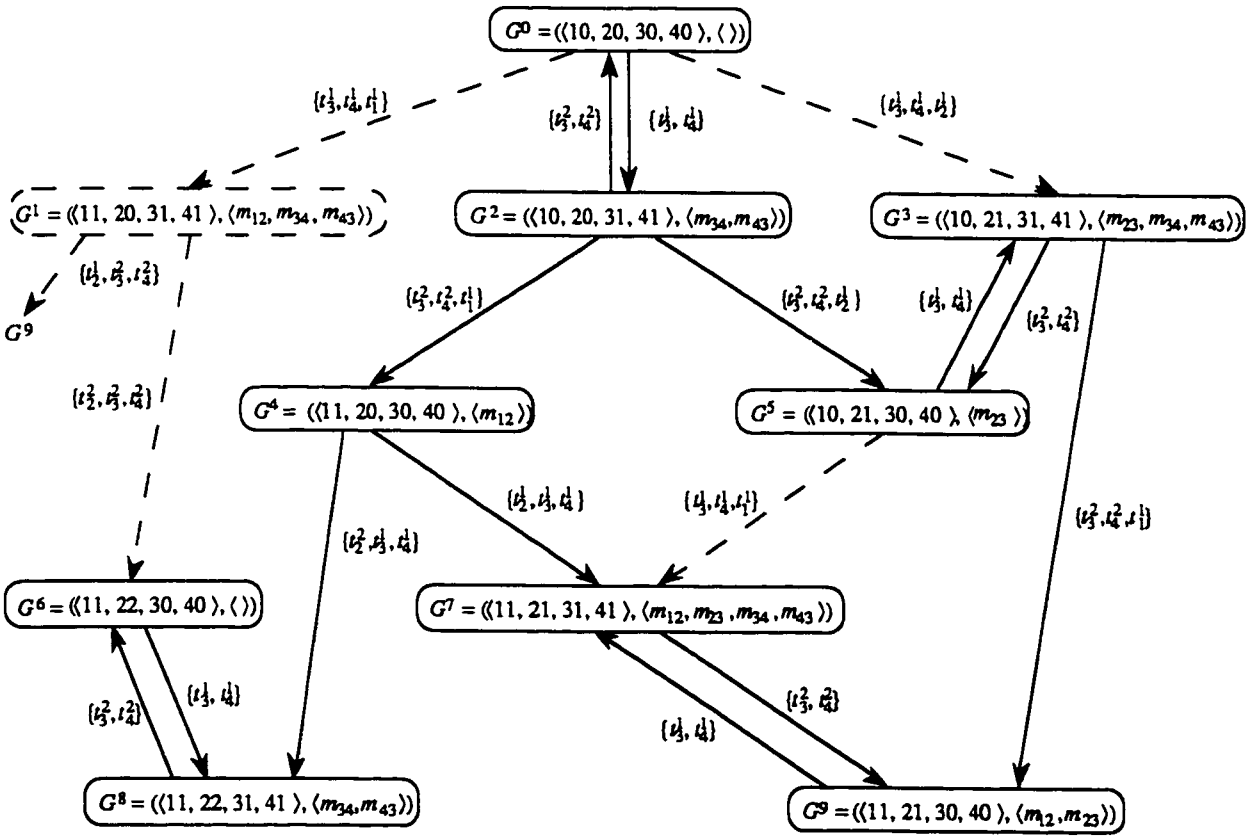


Figure 5.8 The $\ell 2(\emptyset, \emptyset)^*$-reachability graph of the protocol of Example 5.6.

5.6 Related work: LRA versus simultaneous reachability analysis

LRA borrows ideas from earlier work by Itoh & Ichikawa [II83] and by Özdemir & Ural [ÖU94, ÖU95, Özd95]. Their respective relief strategies also entail the concurrent execution of transitions at global states and tackle the issue of potentially executable transitions (cf. Section 4.1.2 and Section 5.1). Itoh & Ichikawa proposed a technique to explore only the *reduced implementation sequences* of a protocol, which constitute a subset of all the possible protocol executions (see Chapter 3). This technique is limited to the detection of non-progress states, however, and it imposes constraints on the structures of the processes in a protocol. All the processes are required to synchronize on their initial (process) states after a finite number of execution steps and no process is allowed to have a cyclic execution that does not pass through its initial state [II83]. The latter amounts to eluding any embedded cycles in the process graph of a process, which is clearly restrictive in practice for even a simple data transfer protocol usually exhibits such cycles (e.g. the retransmission part of the sender in an alternating bit protocol).

Özdemir and Ural generalized the idea of executing sets of concurrent transitions as a relief strategy for detecting all four types of logical errors, and without confining any of the protocol attributes [ÖU94, ÖU95, Özd95]. Their so-called *simultaneous reachability analysis* (SRA) thus applies to protocols in the CFSM model with an arbitrary number of processes, an arbitrary communication topology and arbitrary process structures. SRA is certainly the technique closest to LRA and, in fact, LRA has largely emerged as an incremental improvement of SRA [SU95b]. Similar to LRA, SRA governs the execution of leap sets in global states, called simultaneously executable sets in [ÖU95, Özd95], in order to detect non-progress states, non-executable transitions, unspecified receptions and buffer overflows in a protocol. This section gives an analytical comparison between the two techniques. An empirical comparison is included in Chapter 6. For ease of presentation we take LRA as defined in Section 5.1 through Section 5.4. It should be clear that the results established are then valid also for the refined, "depth-first search" version of LRA discussed in Section 5.5.

5.6.1 Detecting non-progress states and non-executable transitions

Unlike LRA, SRA does not support the option to carry out the detection of non-progress states separate from the detection of non-executable transitions. Where LRA employs the set $pleap(G)$ in a global state G for detecting non-progress states alone (see Section 5.1), and the extended set $xpleap(G)$ for detecting non-progress states and non-executable transitions (see Section 5.2), SRA always employs the set $sses(G) \subseteq leap(G)$ of so-called *selected simultaneously executable sets* for the detection of these two types of logical errors [ÖU95, Özd95].

Definition 5.52

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$. The set $sses(G)$ of *selected simultaneously executable sets* in G is defined as follows:

$$sses(G) = \{T \mid T \in leap(G) \wedge act(T) \supseteq \{i \in I \mid X_i(G) \neq \emptyset \wedge P_i(G) = \emptyset\}\ \}$$ □

Informally, every selected simultaneously executable set in a global state G obeys the following two rules: (1) it *must* contain an executable transition from each process with executable and no potentially executable transitions at G, and (2) it *may* contain an executable transition from any process with both executable and potentially executable transitions at G. The leap sets in $pleap(G)$ and $xpleap(G)$ also adhere to the first rule, but not to the second rule. Indeed, by definitions 5.4 and 5.16, every leap set in $pleap(G)$ or in $xpleap(G)$ contains *at most one* transition from among the processes with potentially executable transitions at G, whereas a leap set in $sses(G)$ can have multiple transitions from such processes. The difference between the three sets is further illustrated in Example 5.53, and expressed formally by Proposition 5.54.

Example 5.53

For the protocol of Example 5.6, $X(G^0) = \{t_1^1, t_2^1, t_3^1, t_4^1\}$ and $P(G^0) = \{t_1^2, t_2^2\}$. We already derived $pleap(G^0) = \{\ \{t_3^1, t_4^1\}\ \}$ and $xpleap(G^0) = \{\ \{t_3^1, t_4^1\}, \{t_3^1, t_4^1, t_1^1\}, \{t_3^1, t_4^1, t_2^1\}\ \}$, and now $sses(G^0) = \{\ \{t_3^1, t_4^1\}, \{t_3^1, t_4^1, t_1^1\}, \{t_3^1, t_4^1, t_2^1\}, \{t_3^1, t_4^1, t_1^1, t_2^1\}\ \}$. The additional selected simultaneously executable set $\{t_3^1, t_4^1, t_1^1, t_2^1\}$ contains more than one transition from among the processes with potentially executable transitions at G^0. □

Proposition 5.54

For a set of sets S, let $min(S) = \{s \in S \mid \nexists s' \in S: s' \subset s\}$, then

$$pleap(G) = min(sses(G)) \subseteq xpleap(G) \subseteq sses(G)$$

Proof: It is not difficult to see that $sses(G)$ can be defined equivalently in terms of the "wait-set" $wait(G)$ (see Definition 5.4) as follows:

$$sses(G) = \{\ T \mid T \in leap(G) \wedge act(T) \supseteq \{i \in I \mid i \notin wait(G)\}\ \}$$

if $wait(G) \subset I$

$$sses(G) = leap(G)$$

otherwise.

The proposition is then immediate by the definitions of $pleap(G)$ and $xpleap(G)$ (Definition 5.4 and Definition 5.16, respectively). □

As a direct consequence of Proposition 5.54, for any protocol both the ℓ-reachability graph and the ℓ^*-reachability graph are subgraphs of the corresponding "simultaneous reachability graph" resulting from SRA. LRA thus explores at most as many global states and transitions as SRA for detecting non-progress states and non-executable transitions. In general LRA can be expected to perform significantly better than SRA. This is evident especially for protocols with state spaces that manifest a wide distribution of potentially executable transitions. Consider for instance a global state G where k_1 is the number of processes with executable transitions and without potentially executable transitions at G, and where k_2 is the number of processes with both executable and potentially executable transitions at G ($0 \le k_1, k_2 \le n$). Assume for simplicity that all these processes have the same number m of executable transitions at G. The cardinalities of $pleap(G)$, $xpleap(G)$ and $sses(G)$ are then as follows:

$$
\begin{array}{lcll}
|pleap(G)| & = & m^{k_1} & \text{if } k_1 > 0 \text{ (i.e. } wait(G) \subset I) \\
|pleap(G)| & = & k_2 \cdot m & \text{otherwise} \\
|xpleap(G)| & = & |pleap(G)| + k_2 \cdot m & \text{if } k_1 > 0 \\
|xpleap(G)| & = & |pleap(G)| & \text{otherwise} \\
|sses(G)| & = & |pleap(G)| + |pleap(G)| \cdot \sum_{j=1}^{k_2} \binom{k_2}{j} m^j & \\
 & = & |pleap(G)| + |pleap(G)| \cdot ((m+1)^{k_2} - 1) & \text{if } k_1 > 0 \\
|sses(G)| & = & |pleap(G)| + \sum_{j=2}^{k_2} \binom{k_2}{j} m^j & \\
 & = & |pleap(G)| + ((m+1)^{k_2} - 1 - k_2 \cdot m) & \text{otherwise}
\end{array}
$$

One can see that the size of $sses(G)$ grows very rapidly for increasing k_2: $|sses(G)| - |pleap(G)|$ is exponential in k_2 whereas $|xpleap(G)| - |pleap(G)|$ is only linear in k_2. Overall, SRA may thus compute and execute a significantly larger number of leap sets during state exploration than LRA. In the next chapter we will evaluate empirically the impact of this on the number of global states and transitions explored by SRA and LRA for detecting non-progress states and non-executable transitions, and on the actual space and time consumed by both techniques.

5.6.2 Detecting unspecified receptions

The detection of unspecified receptions by SRA proceeds in two stages. A given protocol is first augmented with extra receive transitions. These transitions are guaranteed to be non-executable and hence they do not alter the behavior of the protocol. State exploration is then carried out for the augmented protocol in the exact same way as described above, namely by executing selected simultaneously executable sets in global states [ÖU95, Özd95].

Definition 5.55

Let $\Pi = (\{P_i \mid i \in I\}, L)$ be a protocol, and denote by $@_{ij}$ a new unique message from process P_i to process P_j such that $@_{ij} \notin \bigcup_{i \in I} M_i$, for each $(i, j) \in L$. The triple $(s, +@_{ij}, s)$, with $s \in S_j$, is said to be an *extraneous receive transition* for Π iff $M_{ij} \neq \emptyset$ and there exists no $(s_j, +y, s'_j) \in \Delta_j$ such that $s_j = s$ and $y \in M_{ij}$. □

```
for all (i,j) in J do
  if M_ij ≠ ∅ then
     for all s in S_j do
        if there is no receive transition at s
           that involves a message from M_ij
        then add (s, @_ij, s) to Π
```

Figure 5.9 Constructing Π'_J.

A protocol Π is *augmented with respect to* $J \subseteq L$ by adding to Π every extraneous receive transition $(s, +@_{ij}, s)$ for which $(i, j) \in J$. Denote the resulting protocol by Π'_J. As illustrated in Figure 5.9, constructing Π'_J takes $O(\sum_{(i,j) \in J} |S_j|)$ time, where $|S_j|$ is the number of process states of process P_j. It is clear that all extraneous receive transitions are non-executable, since no matching send transitions are specified. Therefore, $\mathbf{R}_\Pi = \mathbf{R}_{\Pi'_J}$ for any J. The application of SRA to Π'_J instead of Π now reveals all unspecified receptions (ur-pairs) wrt J in Π [ÖU95, Özd95]. Recall from Section 5.4 that LRA uses the set $xpleap(G, J, \emptyset)$ for this purpose. In comparing the two techniques we arrive at the following proposition.

Proposition 5.56

Let G be a global state of a protocol Π, and G' the corresponding global state of Π'_J, then

$$pleap(G, J, \emptyset) = min(sses(G')) \subseteq xpleap(G, J, \emptyset) \subseteq sses(G')$$

Proof: Similar to the proof of Proposition 5.54, we show that $sses(G')$ is defined in terms of the "wait-set" $wait(G, J, \emptyset)$ as follows:

$$sses(G') = \{\, T \mid T \in leap(G) \wedge act(T) \supseteq \{i \in I \mid i \notin wait(G, J, \emptyset)\} \,\}$$

$$\text{if } wait(G, J, \emptyset) \subset I$$

$$sses(G') = leap(G)$$

$$\text{otherwise.}$$

The inclusion $xpleap(G, J, \emptyset) \subseteq sses(G')$ is then immediate by definition of $pleap(G, J, \emptyset)$ and $xpleap(G, J, \emptyset)$ (see Definition 5.30 and Definition 5.31, respectively).

We need to prove that $\{i \in I \mid i \notin wait(G, J, \emptyset)\} = \{i \in I \mid X_i(G') \neq \emptyset \wedge P_i(G') = \emptyset\}$ (see Definition 5.52), or equivalently, that $wait(G, J, \emptyset) = \{i \in I \mid X_i(G') = \emptyset \vee P_i(G') \neq \emptyset\}$, where $wait(G, J, \emptyset) = \{i \in I \mid X_i(G) = \emptyset \vee P_i(G) \neq \emptyset \vee \exists (j, i) \in J\colon c_{ji}^G = \varepsilon\}$ (see Definition 5.30). For this it is sufficient to show that the following two claims hold true:

i) $X_i(G') = \emptyset$ iff $X_i(G) = \emptyset$;

ii) $P_i(G') \neq \emptyset$ iff $P_i(G) \neq \emptyset \vee \exists (j, i) \in J\colon c_{ji}^G = \varepsilon$.

Remark that G and G' are the same global state (viz. with the same process states and the same channel contents), except for possible extraneous receive transitions defined at process states in G' as a result of the augmentation. Since these transitions are non-executable it follows directly that $X_i(G') = X_i(G)$, which proves claim (i). Regarding claim (ii), for the "only-if" part suppose that $P_i(G') \neq \emptyset$ and let $t \in P_i(G')$. It is clear that $t \in P_i(G)$ if t is not an extraneous receive transition, while $\exists (j, i) \in J\colon c_{ji}^{G'} = c_{ji}^G = \varepsilon$ if t is an extraneous receive transition. For the "if" part, $P_i(G) \neq \emptyset$ implies that $P_i(G') \neq \emptyset$. The only case left is then that $P_i(G) = \emptyset \wedge \exists (j, i) \in J\colon c_{ji}^G = \varepsilon$. In this case Π has no receive transition at s_i^G involving a message from M_{ji} because otherwise $P_i(G) \neq \emptyset$. Hence, $(s_i^G, +@_{ji}, s_i^G)$ is an extraneous receive transition for Π and, moreover, this transition is potentially executable at G' since $c_{ji}^G = \varepsilon$. Again, $P_i(G') \neq \emptyset$. □

As before, simple combinatorics testify that the difference in size between $xpleap(G, J, \emptyset)$ and $sses(G')$ can be substantial. We conclude from Proposition 5.56 that the $\ell(J, \emptyset)^*$-reachability graph obtained by LRA for the original protocol Π is a subgraph of the simultaneous reachability graph obtained by SRA for the augmented protocol Π'_J. Taking into account also the computational overhead associated with the augmentation procedure, SRA will thus utilize more space and time for detecting ur-pairs than LRA.

Example 5.57

The extraneous receive transitions for the protocol of Example 5.6 (see Figure 5.2) are:

$(21, +@_{12}, 21)$	$(30, +@_{23}, 30)$	$(30, +@_{43}, 30)$
$(22, +@_{12}, 22)$	$(31, +@_{23}, 31)$	$(40, +@_{34}, 40)$

Augmenting this protocol wrt $J = \{(2, 3), (4, 3)\}$ is accomplished by adding the three transitions $(30, +@_{23}, 30)$, $(31, +@_{23}, 31)$ and $(30, +@_{43}, 30)$. The application of SRA to the augmented protocol then yields a simultaneous reachability graph consisting of 25 nodes and 84 edges. In contrast, the $\ell(J, \emptyset)^*$-reachability graph of the original protocol consists of 22 nodes and 52 edges (see Example 5.44). □

5.6.3 Detecting buffer overflows

Before comparing SRA and LRA for the detection of buffer overflows (bo-pairs), we should first point out that the objectives of both techniques are actually different. While LRA aims at detecting *all* bo-pairs that cause a channel overflow, SRA aims at identifying just the overflowed channels [ÖU95, Özd95]. For the latter it suffices to detect only one bo-pair (if any) per channel. Although not explicitly stated in [ÖU95, Özd95], we do recognize that SRA lends itself also for detecting all the bo-pairs in a protocol. Further left unmentioned in [ÖU95, Özd95] is the fact that this can be done for multiple channels at once, i.e. in a single verification run. It is stated instead that the detection of all overflowed channels necessitates the application of SRA once for every channel in the protocol. Nevertheless, when we take the same general objective of detecting all bo-pairs with respect to an index set K of channels (cf. Section 5.4) for both LRA and SRA, the argument in favor of LRA is once again that the $\ell(\emptyset, K)^*$-reachability graph of any given protocol is guaranteed to be a subgraph of the simultaneous reachability graph resulting from SRA for detecting these bo-pairs. In the rest of this section we substantiate also this claim.

In order to determine the possibility of an overflow for a specific channel C_{jk}, SRA employs the following subset of leap sets in a global state G [ÖU95, Özd95] (cf. Definition 5.52):

$$\{T \in leap(G) \mid act(T) \supseteq \{i \in I \mid X_i(G) \neq \emptyset \wedge P_i(G) = \emptyset \wedge (i = k \Rightarrow c_{jk}^G = \varepsilon)\}$$

Accordingly, we provide a more general definition to deal with a set K of channels instead of just a single channel.

Definition 5.58

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ and $K \subseteq L$. Define the set $sses(G, K)$ of selected simultaneously executable sets *wrt* K in G as follows:

$$sses(G,K) = \{T \in leap(G) \mid act(T) \supseteq \{i \in I \mid X_i(G) \neq \emptyset \wedge P_i(G) = \emptyset \wedge \forall (j,i) \in K\colon\ c_{ji}^G = \varepsilon\}\ \}$$ □

We establish that $xpleap(G, \emptyset, K)$ is included in $sses(G, K)$, for any K, and this proves that LRA outperforms SRA for the detection of bo-pairs.

Proposition 5.59

$xpleap(G, \emptyset, K) \subseteq sses(G, K)$

Proof: Notice that $\{i \in I \mid X_i(G) \neq \emptyset \wedge P_i(G) = \emptyset \wedge \forall (j,i) \in K\colon\ c_{ji}^G = \varepsilon\} \subseteq I \setminus wait(G, \emptyset, K) = \{i \in I \mid X_i(G) \neq \emptyset \wedge P_i(G) = \emptyset \wedge \forall (j,i) \in K\colon\ X_{ij}^+(G) = \emptyset\}$ (see Definition 5.30), in particular because $X_{ij}^+(G) = \emptyset$ if $c_{ji}^G = \varepsilon$. The set $sses(G, K)$ can hence be defined equivalently as follows:

$$sses(G, K) = \{\ T \mid T \in leap(G) \wedge act(T) \supseteq \{i \in I \mid i \notin wait(G, \emptyset, K) \wedge \forall (j, i) \in K\colon\ c_{ji}^{G} = \varepsilon\}\ \}$$
$$\text{if } wait(G, \emptyset, K) \subset I$$

$$sses(G, K) = leap(G)$$
$$\text{otherwise.}$$

The inclusion $xpleap(G, \emptyset, K) \subseteq sses(G, K)$ follows then readily by definition of $pleap(G, \emptyset, K)$ and $xpleap(G, \emptyset, K)$. □

5.7 Summary

In this chapter we have developed a relief strategy, named *leaping reachability analysis* (LRA), for the verification of logical correctness properties of protocols defined in the CFSM model. We proved that, for any given protocol in this model, LRA maintains the power of conventional reachability analysis to detect all non-progress states (including deadlocks), all non-executable transitions, all unspecified receptions and all buffer overflows. Yet, in contrast to conventional reachability analysis where transitions are executed one at a time, LRA employs the *concurrent* execution of transitions at global states. By executing concurrent transitions collectively as sets, it "leaps" through the state space of a protocol and may thereby reduce significantly the number of global states and transitions explored. The potential impact of LRA is thus a large decrease in memory and time needed for verifying logical correctness properties.

LRA has been inspired to a large extent by earlier work of Özdemir & Ural on *simultaneous reachability analysis* (SRA) [ÖU94, ÖU95, Özd95]. In fact, we propose LRA as an incremental improvement of SRA. SRA similarly employs the execution of sets of concurrent transitions to verify the same four logical correctness properties, and is arguably the first relief strategy applicable to protocols in the CFSM model without confining any of the protocol attributes (viz. the number of processes in a protocol, its communication topology and the individual process structures). Through an analytical comparison we have shown that, for any protocol, the "reduced" reachability graph resulting from LRA is a subgraph of the one resulting from SRA, for each of the four logical correctness properties. Thus, LRA never explores more global states or transitions than SRA. Moreover, LRA never incurs more run-time overhead, and even eliminates the need for a protocol augmentation in detecting unspecified receptions. Using LRA instead of SRA is therefore a *no-risk improvement*. In the next chapter we will complement the analytical results with an empirical evaluation of the performance of LRA, with respect to SRA and conventional reachability analysis.

Chapter 6

Experiments

This chapter reports on the results of an empirical evaluation of the performance of LRA with respect to both SRA [ÖU95, Özd95] and conventional reachability analysis (henceforth referred to as CRA). These results provide empirical evidence of the potential significance of LRA as a(n) (improved) relief strategy for verifying logical correctness properties of protocols.

6.1 Method of evaluation

How much can be gained by using a relief strategy like LRA (or SRA) as opposed to using CRA? It is difficult to give a general answer to this question. Indeed, one can easily construct classes of protocols for which nothing is gained whatsoever. Examples are protocols where the coupling between the processes is so tight that two transitions from distinct processes are never simultaneously executable. Such protocols are in fact purely sequential systems. In these cases, LRA (and SRA) yields no reduction at all as it becomes equivalent to CRA. On the other hand, it is also rather easy to find classes of protocols for which the reduction obtained by LRA (and SRA) can reach several orders of magnitude. One should think here of cases where the reachable global state space of a protocol grows exponentially when the number of processes in the protocol is increased, while the reduced state space resulting from LRA (and SRA) grows just polynomially. A concrete example is a protocol modeling a distributed sorting algorithm for k numbers using $k + 1$ parallel processes. One process is used to initiate the sort by sending all the numbers to the next process in sequence, and each of the other k processes carries out the pair-wise comparisons between the numbers it receives from the preceding process. The number of reachable global states explored by CRA, and the number of global states explored by LRA (and SRA) for detecting non-progress states and non-executable transitions (i.e. $\ell(\varnothing, \varnothing)^*$-reachable global states) are given in Figure 6.1 for increasing k (logarithmic scale). By a similar token, we can readily find examples of protocols for which the reachable global state space grows in size when the capacity bound of some simplex channel is

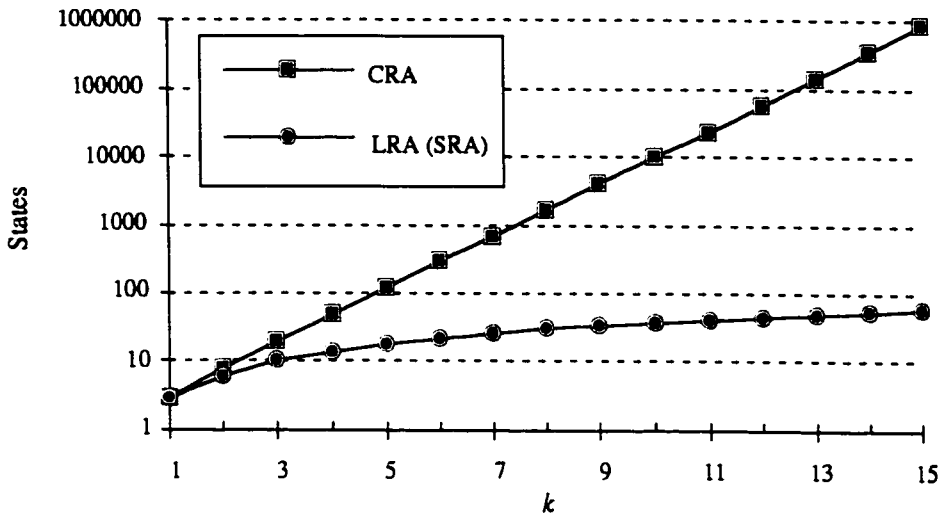


Figure 6.1 Reduction by LRA (and SRA) for distributed sorting.

increased, while the reduced state space resulting from LRA (and SRA) remains the same. This is illustrated by a simple "producer-consumer" protocol consisting of processes $P_1 = (\{10\}, 10, \{a\}, \{(10, -a, 10)\})$ and $P_2 = (\{20\}, 20, \{a\}, \{(20, +a, 20)\})$, and a simplex channel C_{12} with bound $B_{12} > 1$. The full reachable global state space obtained with CRA, and the reduced global state space obtained with LRA (and SRA) for detecting non-progress states, non-executable transitions and unspecified receptions (i.e. the $\ell(J, \varnothing)^*$-reachable global state space) for this protocol are shown in Figure 6.2. The size of the full state space is proportional to the value of B_{12}, while the reduced state space is independent of B_{12}. The full state space is in fact infinite when channel C_{12} is unbounded.

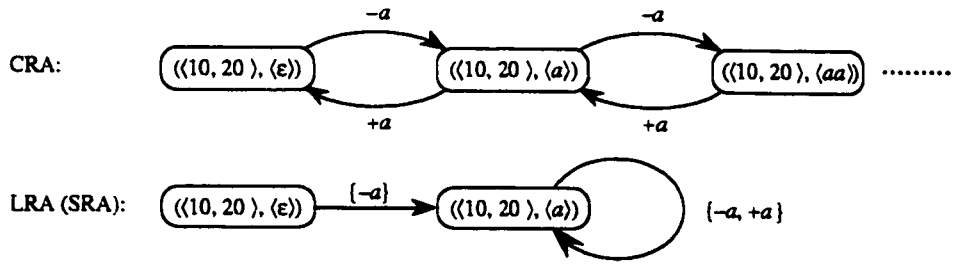


Figure 6.2 Reduction by LRA (and SRA) for the "producer-consumer" protocol.

It is thus clear that an exaggerated impression of the performance of LRA (and SRA), or any other relief strategy, could be suggested by a favorable choice of examples. In order to address this concern of potential favoritism, Özdemir & Ural [ÖU95] evaluated the performance of SRA by experimenting with a large number of protocols that were generated randomly with an automatic

protocol synthesizer. The results of their empirical study appeared to be quite objective and informative. Consequently, we have decided to conduct a similar study for LRA, taking advantage of the automatic protocol synthesizer and the existing implementations of SRA and CRA to construct and experiment with 400 such random protocols. In addition, we have experimented with a few "real" protocols. The results of these experiments are discussed in Section 6.3, following a description of the tool package we have used for our study.

6.2 The research tool package RELIEF

RELIEF is a research tool package for the automated verification of protocols specified in the CFSM model. Built entirely in UNIX C, it has been under development since 1994. The initial version of the package was created by Kadir Özdemir, in connection with his doctoral research on SRA [Özd95]. The implementation of subsequent extensions and enhancements was carried out by Tuong Nguyen [Ngu97].

The current version of RELIEF (version 3.5) has three main constituents, namely a protocol analyzer, the aforementioned protocol synthesizer and a collection of empirical-study tools. The protocol analyzer embodies the various state exploration techniques, including CRA, SRA, LRA and also FRA. These can be engaged via a text-based and menu-driven user interface in order to analyze protocols individually. The logical errors found are then reported, as are the numbers of global states and transitions explored, and the actual space and time consumed by the selected technique. The state exploration techniques can be engaged also via the empirical-study tools for the purpose of generating a comparative report of the average performance of two techniques on a multitude of protocols. The implementations of CRA, SRA, LRA and FRA all utilize the same data structures, procedures and functions to store and access global states. Global states are stored only in main memory to avoid serious run-time penalties for disk-access. Time is calculated as UNIX system time plus user time.

The impartial construction of a set of sample protocols is facilitated in RELIEF by the automatic protocol synthesizer, which constructs protocols as follows [ÖU95, Özd95] (see Figure 6.3). Using a random number generator, it first builds an incomplete specification of a protocol by specifying randomly (between preset parameter bounds) the number of processes in the protocol, and for each process the number of process states, its message set and a partial, deterministic transition function that defines only send transitions. The incomplete protocol is then subjected to CRA to augment it with receive transitions. Precisely, whenever a ur-pair is encountered during the generation of the reachable global states of the protocol, a receive transition for the ur-pair is defined with probability 0.75. This follows the idea behind so-called non-service oriented synthesis methods, whose aim is to derive logically correct protocols from incomplete protocol descriptions (as opposed to service-

oriented synthesis methods, which attempt to derive functionally correct protocols) [PS91]. The reason for omitting about 25% of missing receive transitions is to allow the synthesized protocol to exhibit each of the four types of logical errors. Indeed, if receive transitions were specified for all ur-pairs, the resulting protocol could still have deadlocks and buffer overflows, but it would be free from unspecified receptions and non-executable transitions. This implementation choice was motivated in [Özd95] by the view that the synthesizer simulates in some sense a protocol designer, and even an experienced designer cannot be expected to create error-free designs. Lastly, for the purpose of the experiments, a lower and upper bound on the number of reachable global states of a protocol are enforced during the synthesis process to rule out protocols with an insignificantly small or an impractically large state space. For further details on the automatic protocol synthesizer and the associated algorithms, one is advised to refer to [Özd95].

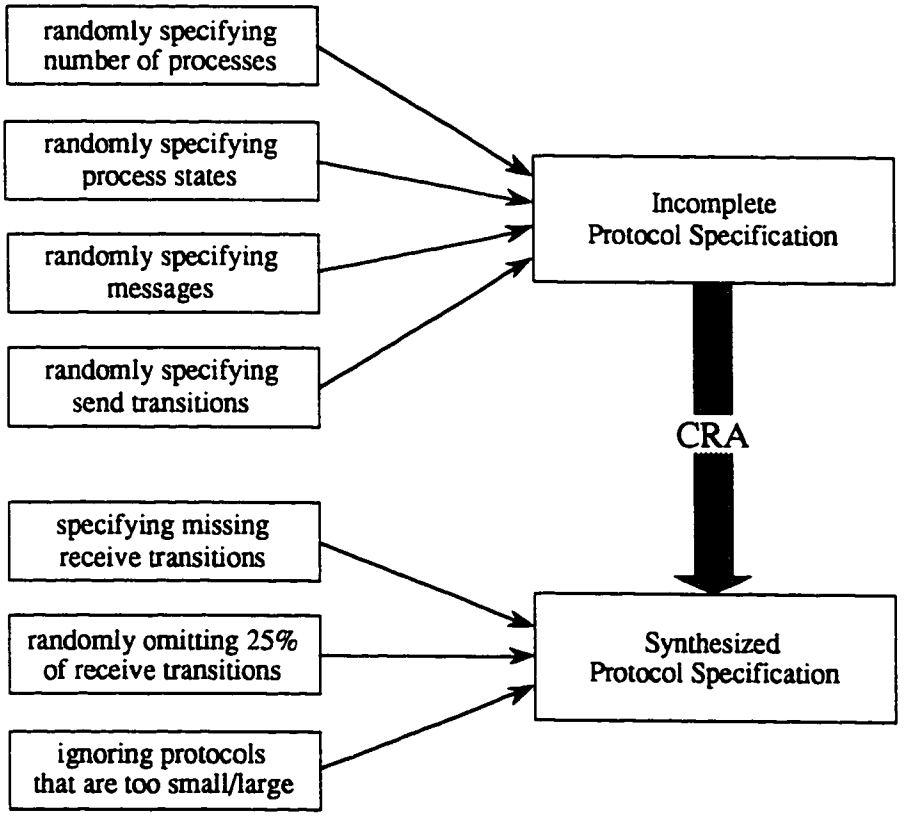


Figure 6.3 The automatic protocol synthesizer in RELIEF.

6.3 Experimental results

LRA and SRA have been tested on a large number of protocols: 400 protocols were obtained with the automatic protocol synthesizer described above, and three real protocols were taken from the

literature. From the analytical comparison in Section 5.6 we already knew that LRA would perform at least as good as SRA. The aim of our experiments was hence to determine the amount of reduction in space and time that can be expected in practice when using LRA in place of SRA (or CRA). All experiments were performed on a SPARC Classic with 48 megabytes of RAM. We first discuss the results obtained with the 400 synthesized protocols.

6.3.1 Experiments with the synthesized protocols

It is of course not feasible, nor worthwhile, to detail all the synthesized protocols used for our empirical study. Table 6.1 gives instead an overall summary of some of their principal attributes. The 400 protocols are classified by the number of processes in a protocol, ranging from two to

Table 6.1 Synopsis of the set of synthesized protocols.

Attributes		*n*-process protocols						
		2	3	4	5	6	7	8
Number of protocols		66	57	63	58	56	60	40
Number of states per process	avg	11.64	9.06	8.10	7.07	6.12	4.95	3.99
	std	17.71	16.79	12.02	9.79	5.99	3.93	2.05
Number of send transitions per process state	avg	2.18	1.46	1.06	1.01	0.81	0.76	0.64
	std	1.62	1.72	1.68	1.54	1.06	1.06	0.71
Number of receive transitions per process state	avg	0.68	0.66	0.67	0.66	0.65	0.65	0.63
	std	0.51	0.49	0.50	0.49	0.44	0.42	0.34
Number of reachable global states (× 1000)	avg	31.71	34.02	49.09	60.87	78.37	108.04	149.99
	std	39.56	37.60	41.11	40.00	37.78	31.39	13.09
Number of global state transitions (× 1000)	avg	69.21	100.77	171.29	210.95	319.40	483.39	744.27
	std	86.17	125.80	151.56	154.59	176.05	156.45	87.49
Memory (megabytes)	avg	1.67	1.84	2.67	3.31	4.26	5.94	8.28
	std	2.08	2.07	2.26	2.21	2.09	1.74	0.77
Time (seconds)	avg	5.72	9.03	16.09	22.52	36.27	62.17	160.95
	std	7.83	11.90	15.19	17.29	20.87	21.42	20.38
Percentage of deadlock states	avg	0.23	0.15	0.08	0.04	0.02	0.01	0.00
	std	0.32	0.17	0.10	0.05	0.02	0.01	0.00
Percentage of non-progress states	avg	3.46	1.90	1.04	1.12	0.34	0.13	0.04
	std	1.98	1.54	1.19	1.23	0.59	0.19	0.06
Percentage of non-executable transitions	avg	0.45	5.09	10.48	11.48	13.70	15.51	13.46
	std	1.43	5.30	7.60	8.04	8.34	8.22	6.10
Percentage of ur-pairs	avg	21.00	21.01	19.74	22.17	17.66	17.60	15.13
	std	6.49	5.51	7.70	7.17	6.71	7.48	5.12
Percentage of bo-pairs	avg	24.27	22.47	20.34	16.97	15.56	14.81	15.41
	std	5.03	6.00	6.07	5.61	5.08	5.92	5.87

eight processes. For each class of n-process protocols, the number of process states and the number of send and receive transitions per process state (these are the structural attributes of the processes) disperse quite well, and the same can be said for the number of global states and transitions of the protocols explored by CRA (cf. rows 2 through 6 resp. in Table 6.1). This is observed from the standard deviations for these attributes, which are fairly high relative to the averages. Table 6.1 also gives an indication of the space and time consumed, and the logical errors detected by CRA for the set of synthesized protocols (cf. rows 7 through 13 resp. in Table 6.1). In line with the formulation of LRA in Chapter 5, we evaluate the performance of LRA with respect to SRA and CRA separately for each of the four logical correctness properties.

Detecting non-progress states

Before contemplating the results for the detection of non-progress states, it is important to stress that SRA does not support the option to carry out this verification task separate from the detection of non-executable transitions. Recall from Section 5.6.1 that SRA uses the set $sses(G)$ of selected simultaneously executable sets in a global state G for detecting all non-progress states *and* all non-executable transitions. LRA uses the subset $xpleap(G)$ of $sses(G)$ for this purpose, but it uses a different (and smaller) subset $pleap(G) = min(sses(G)) \subseteq xpleap(G)$ (see Proposition 5.54) when the objective is to detect non-progress states alone. A comparison of the space and time required by LRA and SRA for detecting just non-progress states is thus warranted, even though SRA actually combines this task with detecting non-executable transitions.

Table 6.2 LRA compared to SRA for detecting non-progress states.

Techniques		Average reductions (%) per number of processes						
		2	3	4	5	6	7	8
SRA vs. CRA	states	49.61	57.79	64.64	68.63	76.99	82.87	88.49
	transitions	52.83	55.61	58.17	62.72	69.62	78.26	83.92
	space	49.36	57.82	64.63	68.69	77.07	82.92	88.47
	time	45.54	45.01	39.79	40.98	41.36	20.88	62.59
LRA vs. CRA	states	55.94	64.65	72.36	75.68	83.54	89.56	94.10
	transitions	65.49	74.76	81.77	85.02	90.76	94.79	97.36
	space	55.70	64.70	72.41	75.81	83.68	89.66	94.15
	time	56.15	64.48	72.36	75.78	83.52	89.62	95.07
LRA vs. SRA	states	12.56	16.25	21.83	22.47	28.47	39.05	48.74
	transitions	26.84	43.14	56.42	59.82	69.59	76.03	83.58
	space	12.52	16.31	22.00	22.74	28.83	39.46	49.26
	time	19.48	35.41	54.09	58.96	71.90	86.88	86.82

Table 6.2 shows the performance of LRA with respect to CRA and SRA for detecting non-progress states. Given are average percentages of reduction, per class of n-process protocols, of the number of global states stored and transitions explored, and of the actual amount of space and time used. As one can see, the space reductions match the respective reductions in the number of stored states. Hence, as expected, the memory allocated for data structures that accommodate the computation of proper leap sets (or selected simultaneously executable sets in case of SRA) is insignificant with respect to the overall memory requirements for state exploration.

The first two rows in Table 6.2 compare the performance of resp. SRA and LRA versus CRA. The third row compares the performance of LRA directly versus SRA, i.e. the reductions obtained with LRA are normalized with respect to those obtained with SRA. Evidently, where SRA already yields substantial space and time reductions over CRA, LRA achieves significant further reductions over SRA, especially in time. This is explained by the fact that the set $pleap(G)$ can be much smaller than $sses(G)$, according to the combinatorics in Section 5.6.1. Consequently, LRA seems in general to incur less run-time overhead (computing $pleap(G)$ is cheaper than computing $sses(G)$ as witnessed by Proposition 5.54), and to explore fewer global state transitions and thereby to generate fewer global states than SRA.

Both LRA and SRA tend to perform better as protocols contain more processes. Indeed, with an increasing number of processes, the number of concurrently executable transitions at a global state generally increases as well. More pragmatically, larger reductions can be expected for protocols with higher degrees of parallelism (i.e. for so-called "loosely-coupled" protocols). A conceivable measure for the degree of parallelism of a protocol is the average number of processes

Table 6.3 Reductions in Table 6.2 arranged by concurrency level.

Techniques		Average reductions (%) per concurrency level			
		[0, 1]	(1, 2]	(2, 3]	(3, 4]
SRA vs. CRA	states	47.63	67.22	85.66	94.91
	transitions	44.69	62.41	82.47	95.52
	space	47.45	67.26	85.68	94.86
	time	31.70	37.76	50.99	88.18
LRA vs. CRA	states	54.03	74.48	92.11	97.98
	transitions	64.76	83.62	96.30	99.22
	space	53.88	74.56	92.18	97.96
	time	53.49	74.55	92.85	98.32
LRA vs. SRA	states	12.22	22.15	44.98	60.31
	transitions	36.29	56.42	78.89	82.59
	space	12.24	22.30	45.39	60.31
	time	31.90	59.11	85.41	85.79

with executable transitions and no potentially executable transitions at a reachable global state. This measure is defined as the *concurrency level* of a protocol [ÖU95, Özd95]. Table 6.3 organizes the data in Table 6.2 by concurrency level. Observe that the performance of LRA improves steadily with the level of concurrency, compared to CRA as well as to SRA. With respect to CRA, LRA reaches reductions in space and time of up to two orders of magnitude. With respect to SRA, it still reaches reductions in time of close to one order of magnitude, and reductions in space of over 50%. Suffice it to say that LRA can outperform SRA significantly, in both space and time, for the detection of non-progress states in a protocol.

To complete the picture on detecting non-progress states, we have compared the performance of LRA also against FRA. Since the applicability of FRA is limited to multi-cyclic protocols (see Chapter 4), the comparison is based on the subset of synthesized protocols that are in fact multi-cyclic. Bear in mind that FRA is actually not suited to detect all non-progress states in a multi-cyclic protocol, but just the deadlock states (i.e. the non-progress states with all channels empty). Within the set of 400 synthesized protocols we found 88 multi-cyclic protocols, including 66 2-process protocols, 18 3-process protocols, three 4-process protocols and one 5-process protocol. Table 6.4 gives the average reductions obtained with LRA and FRA for these protocols. FRA clearly outperforms LRA in space. The extra space reductions over LRA can be very impressive. On the other hand, FRA may be slower than LRA. The time reductions in the table suggest that FRA incurs quite a bit more run-time overhead than LRA (the time reductions reported do not incorporate the time needed to determine whether a protocol is multi-cyclic, since this would be known for real protocols). At least for protocols that manifest little concurrency, the time gained by

Table 6.4 LRA compared to FRA for detecting deadlock states in multi-cyclic protocols.

Techniques		Average reductions (%) per number of processes				concurrency level	
		2	3	4	5	[0, 1]	(1, 2]
LRA vs. CRA	states	55.94	71.86	68.28	59.24	55.78	73.74
	transitions	65.49	80.89	79.69	77.74	65.57	82.67
	space	55.70	72.14	68.50	59.35	55.56	73.99
	time	53.77	71.53	67.92	61.75	53.64	73.73
FRA vs. CRA	states	77.02	92.41	94.23	97.46	77.60	93.27
	transitions	80.47	94.89	96.29	98.07	81.04	95.48
	space	76.69	92.50	94.44	97.60	77.30	93.35
	time	-80.85	49.88	62.29	86.35	-76.30	57.90
FRA vs. LRA	states	47.84	73.03	81.81	93.77	49.34	74.37
	transitions	43.41	73.26	81.73	91.33	44.93	73.92
	space	47.38	73.08	82.35	94.10	48.92	74.34
	time	-291.20	-76.04	-17.55	64.31	-280.28	-60.26

FRA due to the reduction in the number of explored transitions appears insufficient to make up the time lost for computing fair transition-tuples in global states (cf. Definition 4.14). Nevertheless, the space reductions by FRA can be so good that it is a technique to consider as the first choice when verifying a multi-cyclic protocol.

Detecting non-executable transitions

The performance of LRA and of SRA on the set of synthesized protocols for the detection of non-executable transitions (and non-progress states) are compared in Table 6.5. As before, the figures indicate average percentages of reduction, and they are arranged by the number of processes in a protocol and by the concurrency level of a protocol. Note that the reductions by SRA over CRA in the first row of Table 6.5 are the same as in Table 6.2 and Table 6.3, since SRA employs the same set $sses(G)$ as just regarded for detecting non-progress states. For LRA we have considered two

Table 6.5 LRA compared to SRA for detecting non-executable transitions.

Techniques		Average reductions (%) per number of processes							concurrency level			
		2	3	4	5	6	7	8	[0, 1]	(1, 2]	(2, 3]	(3, 4]
SRA vs. CRA	states	49.61	57.79	64.64	68.63	76.99	82.87	88.49	47.63	67.22	85.66	94.91
	transitions	52.83	55.61	58.17	62.72	69.62	78.26	83.92	44.69	62.41	82.47	95.52
	space	49.36	57.82	64.63	68.69	77.07	82.92	88.47	47.45	67.26	85.68	94.86
	time	45.54	45.01	39.79	40.98	41.36	20.88	62.59	31.70	37.76	50.99	88.18
LRA vs. CRA	states	51.85	60.17	67.28	71.31	79.34	84.98	90.56	50.04	69.77	87.81	95.63
	transitions	57.94	65.57	72.66	76.69	83.78	88.49	92.87	56.15	75.03	90.93	97.26
	space	51.60	60.19	67.27	71.38	79.43	85.03	90.55	49.87	69.81	87.84	95.58
	time	49.08	54.73	58.78	62.50	69.79	75.43	82.23	44.37	61.59	79.83	92.36
LRA2 vs. CRA	states	54.92	63.93	71.46	75.17	82.93	88.29	93.05	53.22	73.64	91.15	96.84
	transitions	63.97	73.67	80.66	84.38	90.05	93.80	96.50	63.56	82.61	95.46	98.43
	space	54.68	63.98	71.51	75.29	83.06	88.38	93.08	53.08	73.73	91.22	96.80
	time	35.28	48.14	57.90	64.87	73.94	73.46	88.86	32.72	60.16	84.06	94.00
LRA vs. SRA	states	4.45	5.64	7.47	8.54	10.21	12.32	17.98	4.60	7.78	14.99	14.15
	transitions	10.83	22.44	34.64	37.47	46.61	47.06	55.66	20.72	33.57	48.26	38.84
	space	4.42	5.62	7.46	8.59	10.29	12.35	18.04	4.61	7.79	15.08	14.01
	time	6.50	17.68	31.54	36.46	48.48	68.95	52.50	18.55	38.29	58.85	35.36
LRA2 vs. SRA	states	10.54	14.55	19.29	20.85	25.81	31.64	39.62	10.67	19.59	38.28	37.92
	transitions	23.62	40.68	53.77	58.10	67.25	71.48	78.23	34.12	53.74	74.10	64.96
	space	10.51	14.60	19.45	21.08	26.12	31.97	39.98	10.71	19.76	38.69	37.74
	time	-18.84	5.69	30.08	40.48	55.56	66.46	70.22	1.49	35.99	67.48	49.24
LRA2 vs. LRA	states	6.38	9.44	12.78	13.45	17.38	22.04	26.38	6.37	12.80	27.40	27.69
	transitions	14.34	23.53	29.26	32.99	38.66	46.13	50.91	16.90	30.36	49.94	42.70
	space	6.36	9.52	12.95	13.66	17.65	22.38	26.77	6.40	12.98	27.80	27.60
	time	-27.10	-14.56	-2.13	6.32	13.74	-8.02	37.31	-20.94	-3.72	20.97	21.47

sets of results: the second row of Table 6.5 gives the reductions obtained by executing at each generated global state G the leap sets in *xpleap*(G, J, K) with $J = K = \emptyset$ (see Definition 5.16 and Definition 5.31), and the third row gives the reductions obtained by executing the leap sets in *xpleap*-2(G, J, K) equally with $J = K = \emptyset$ (see Definition 5.46). In order to distinguish the two versions, LRA based on *xpleap*-2(G, J, K) is henceforth called LRA2. LRA2 was proposed in Section 5.5 as a refinement of LRA, based on the characteristics of a depth-first search, to enable yet more reductions in the number of stored states and explored transitions for detecting non-executable transitions, unspecified receptions and buffer overflows.

It is apparent from the first three rows of Table 6.5 that SRA, LRA and LRA2 can all produce large savings over CRA for detecting non-executable transitions. The fourth and fifth row show the reductions by LRA and LRA2 normalized with respect to SRA. LRA combines modest reductions in space over SRA with more discrete reductions in time. LRA2 yields noticeable better space reductions over SRA. It can sometimes be slower than LRA (and SRA), when the extra reduction in the number of explored transitions is not sufficient to counteract the run-time overhead for computing *xpleap*-2$(G, \emptyset, \emptyset)$ instead of *xpleap*$(G, \emptyset, \emptyset)$ (see Section 5.5), but for protocols with higher concurrency levels this circumstance disappears. The last row of Table 6.5 confirms that LRA2 is a worthy refinement of LRA (and thus of SRA) for the detection of non-executable transitions in a protocol.

Detecting unspecified receptions

As explained in Section 5.6.2, when using SRA for detecting unspecified receptions (ur-pairs) in a protocol, the protocol must first be preprocessed. In this preprocessing step, the given protocol is augmented with extraneous receive transitions (see Definition 5.55) that involve a simplex channel indexed in the selected subset J of channel indices. All ur-pairs wrt J in the original protocol are then detected by applying SRA to the augmented protocol instead [ÖU95, Özd95]. In order to detect the same ur-pairs with LRA, no protocol augmentation is required. State exploration is carried out on the original protocol by executing at each generated global state G the leap sets in *xpleap*$(G, J, \emptyset)$, or those in *xpleap*-2$(G, J, \emptyset)$ in case of the refined version LRA2 proposed in Section 5.5.

Table 6.6 compares the average reduction performance of LRA and LRA2 against SRA for detecting ur-pairs in the synthesized protocols. In every experiment, we fixed the index set J as the complete set L of channel indices for each synthesized protocol $\Pi = (P, L)$. That is, the figures in Table 6.6 indicate average reductions obtained by SRA, LRA and LRA2 for detecting *all* the ur-pairs of a protocol in a *single* verification run. One can see that LRA and LRA2 achieve significant

reductions over CRA in both space and time. SRA also yields good space reductions over CRA,but at a potential expense of time. It tends to be increasingly slower as the number of processes or the concurrency level of a protocol grows, slower even than CRA. Consequently, while LRA and LRA2 perform just moderately better than SRA in space, they can perform much better than SRA in time. The increasing time overhead incurred by SRA can be related to the protocol augmentation required before state exploration. First, there is the extra overhead for adding extraneous receive transitions to the processes in a protocol. With more processes, more such transitions may need to be added (see Figure 5.9). Second, and more importantly, during state exploration these extraneous receive transitions may manifest themselves as potentially executable transitions at various global states. This can cause a large increase of the number of selected simultaneously executable sets that are computed and executed by SRA, as testified by the combinatorics at the end of Section 5.6.1 (this number can be substantially larger than the number of leap sets executed by LRA or LRA2).

Table 6.6 LRA compared to SRA for detecting ur-pairs.

Techniques		Average reductions (%) per number of processes							concurrency level			
		2	3	4	5	6	7	8	[0, 1]	(1, 2]	(2, 3]	(3, 4]
SRA vs. CRA	states	49.54	50.37	48.69	50.87	55.43	62.94	71.44	44.98	50.95	68.12	71.73
	transitions	52.62	40.39	23.19	19.51	9.22	19.16	26.82	36.27	21.33	32.39	32.85
	space	49.30	50.47	48.75	50.94	55.60	63.00	71.39	44.80	51.04	68.17	71.69
	time	43.49	27.81	-10.17	-24.87	-76.87	-77.96	-101.01	17.21	-25.76	-56.88	-96.20
LRA vs. CRA	states	51.77	53.78	52.27	54.28	58.83	65.31	74.57	47.27	54.26	71.22	75.76
	transitions	57.81	58.04	57.00	58.79	62.95	69.15	77.67	52.96	58.70	74.80	79.48
	space	51.53	53.87	52.35	54.39	59.03	65.43	74.58	47.10	54.37	71.32	75.74
	time	47.72	44.20	35.87	36.30	34.72	41.15	59.60	39.10	36.64	53.36	55.50
LRA2 vs. CRA	states	54.98	58.62	57.55	58.44	63.54	69.93	78.41	50.52	58.90	76.03	80.47
	transitions	64.01	69.49	68.95	69.77	75.07	80.37	87.08	60.28	70.57	85.28	88.59
	space	54.74	58.75	57.73	58.63	63.84	70.17	78.52	50.38	59.09	76.25	80.56
	time	34.57	37.13	33.84	38.39	42.87	30.47	66.58	28.25	33.95	56.03	55.69
LRA vs. SRA	states	4.42	6.87	6.98	6.94	7.63	6.40	10.96	4.16	6.75	9.72	14.26
	transitions	10.95	29.61	44.02	48.80	59.19	61.84	69.49	26.19	47.50	62.73	69.44
	space	4.40	6.86	7.02	7.03	7.73	6.57	11.15	4.17	6.80	9.90	14.31
	time	7.49	22.70	41.79	48.99	63.09	66.93	79.90	26.44	49.62	70.27	77.32
LRA2 vs. SRA	states	10.78	16.62	17.27	15.41	18.20	18.86	24.40	10.07	16.21	24.81	30.92
	transitions	24.04	48.82	59.58	62.44	72.54	75.72	82.34	37.67	62.59	78.23	83.01
	space	10.73	16.72	17.52	15.67	18.56	19.38	24.92	10.11	16.44	25.38	31.33
	time	-15.78	12.91	39.95	50.66	67.70	60.93	83.37	13.33	47.48	71.97	77.42
LRA2 vs. LRA	states	6.66	10.47	11.06	9.10	11.44	13.32	15.10	6.16	10.14	16.71	19.43
	transitions	14.70	27.29	27.79	26.64	32.71	36.37	42.14	15.56	28.74	41.59	44.40
	space	6.62	10.58	11.29	9.30	11.74	13.71	15.50	6.20	10.34	17.19	19.87
	time	-25.15	-12.67	-3.17	3.28	12.48	-18.15	17.28	-17.82	-4.25	5.72	0.43

Detecting buffer overflows

The last comparison between LRA, LRA2 and SRA for the set of synthesized protocols concerns the detection of buffer overflows (bo-pairs). Recall that LRA and LRA2 detect all bo-pairs wrt a selected subset K of channel indices by executing at each generated global state G the leap sets in respectively $xpleap(G, \emptyset, K)$ and $xpleap\text{-}2(G, \emptyset, K)$. SRA uses the set $sses(G, K)$ instead, where $sses(G, K) \supseteq xpleap(G, \emptyset, K)$ (see Section 5.6.3). For each synthesized protocol $\Pi = (P, L)$, the index set K was set to L to enable the detection of all bo-pairs of the protocol in a single verification run. The results of the experiments are collected in Table 6.7. Observe that LRA and LRA2 yield little or no extra space reductions over SRA for detecting bo-pairs. However, SRA achieves respectable reductions in space over CRA only at a considerable expense of time, as it continually explores more global state transtions than CRA. LRA and LRA2 clearly demonstrate a much better time performance than SRA.

Table 6.7 LRA compared to SRA for detecting bo-pairs.

Techniques		Average reductions (%) per number of processes							concurrency level			
		2	3	4	5	6	7	8	[0, 1]	(1, 2]	(2, 3]	(3, 4]
SRA vs. CRA	states	4.58	15.81	29.17	27.79	39.57	48.55	59.70	7.38	27.83	50.39	66.72
	transitions	-40.42	-51.21	-50.62	-60.66	-67.15	-61.65	-47.42	-45.39	-58.61	-57.73	-7.29
	space	4.55	15.64	28.85	27.49	39.15	48.08	59.16	7.30	27.52	49.92	66.23
	time	-40.04	-69.35	-100.52	-143.94	-206.75	-321.49	-206.17	-58.21	-140.88	-246.64	-144.73
LRA vs. CRA	states	4.67	15.93	29.71	28.57	40.55	48.69	61.61	7.46	28.30	51.19	71.16
	transitions	3.84	17.75	32.83	31.52	44.13	51.79	64.72	7.44	31.23	54.23	73.63
	space	4.63	15.76	29.39	28.26	40.12	48.19	61.03	7.37	27.99	50.68	70.61
	time	-6.33	1.89	10.51	2.30	11.37	19.23	44.94	-6.15	6.25	26.43	53.57
LRA2 vs. CRA	states	4.83	16.02	30.04	28.91	40.90	48.92	62.04	7.58	28.50	51.60	72.22
	transitions	8.72	22.49	39.79	40.41	52.31	60.50	73.46	11.84	38.10	64.10	83.11
	space	4.79	15.85	29.72	28.60	40.47	48.43	61.46	7.50	28.19	51.10	71.68
	time	-32.19	-11.92	1.88	0.28	13.41	-5.33	49.00	-27.24	-3.67	20.60	57.20
LRA vs. SRA	states	0.09	0.14	0.76	1.08	1.62	0.27	4.74	0.09	0.65	1.61	13.34
	transitions	31.52	45.61	55.40	57.38	66.57	70.18	76.07	36.34	56.64	70.98	75.42
	space	0.08	0.14	0.76	1.06	1.59	0.21	4.58	0.08	0.65	1.52	12.97
	time	24.07	42.07	55.37	59.95	71.11	80.84	82.02	32.91	61.08	78.78	81.03
LRA2 vs. SRA	states	0.26	0.25	1.23	1.55	2.20	0.72	5.81	0.22	0.93	2.44	16.53
	transitions	35.00	48.74	60.03	62.91	71.47	75.56	82.00	39.36	60.97	77.24	84.26
	space	0.25	0.25	1.22	1.53	2.17	0.67	5.63	0.22	0.92	2.36	16.14
	time	5.61	33.91	51.07	59.12	71.77	75.01	83.34	19.58	56.96	77.09	82.51
LRA2 vs. LRA	states	0.17	0.11	0.47	0.48	0.59	0.45	1.12	0.13	0.28	0.84	3.68
	transitions	5.07	5.76	10.36	12.98	14.64	18.07	24.77	4.75	9.99	21.56	35.95
	space	0.17	0.11	0.47	0.47	0.58	0.46	1.10	0.14	0.28	0.85	3.64
	time	-24.32	-14.08	-9.64	-2.07	2.30	-30.41	7.37	-19.87	-10.58	-7.92	7.82

LRA in a nutshell

As a supplement to the detailed comparisons in the preceding tables, Figure 6.4 recapitulates the performance of LRA as a whole, with respect to CRA. The average percentages of reduction by LRA are arranged by concurrency level and by the different types of logical errors covered, viz. non-progress states (type A), non-executable transitions (type B), unspecified receptions (type C) and buffer overflows (type D). For every two adjacent bars with the same shade, the left bar depicts

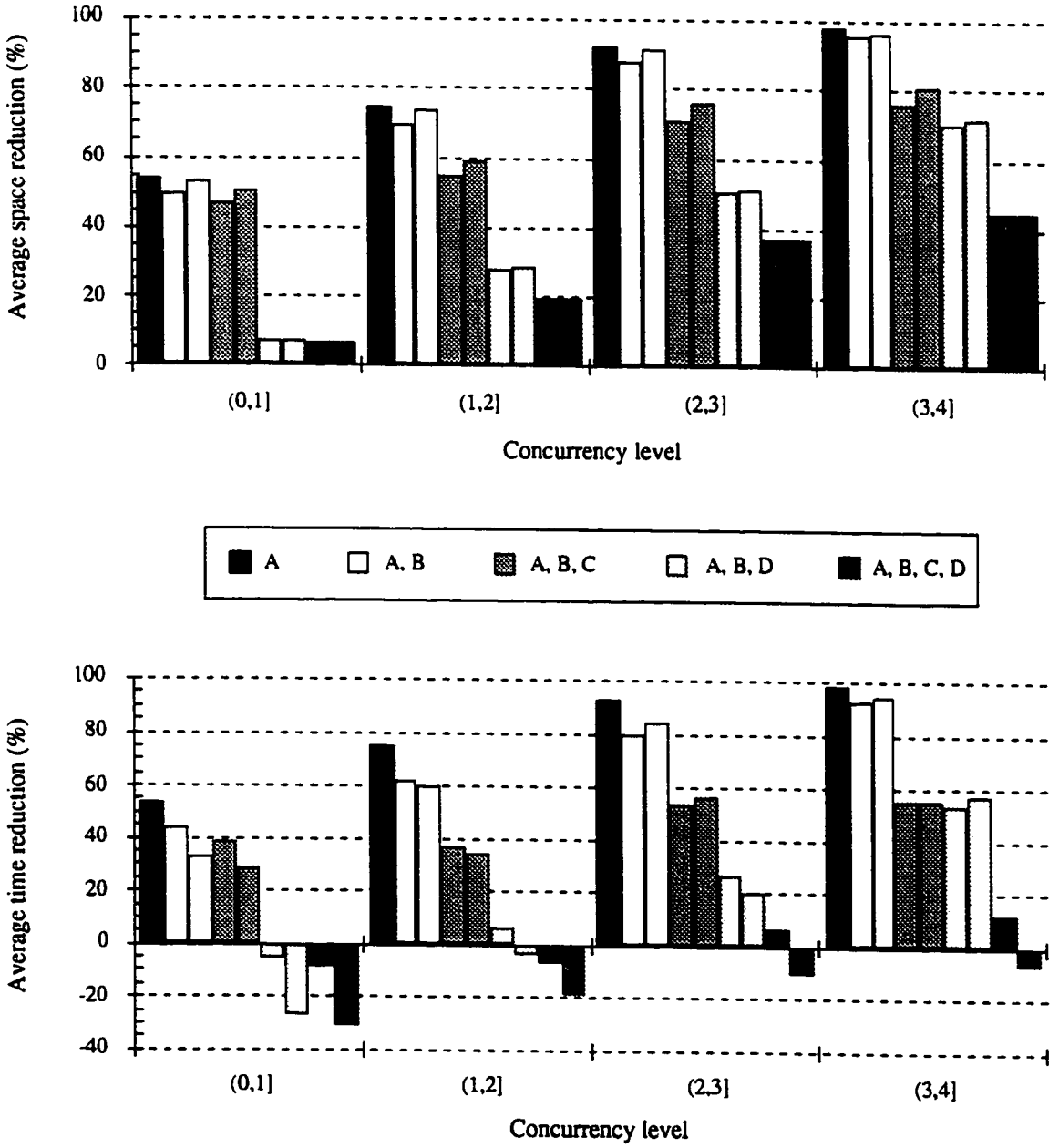


Figure 6.4 LRA versus CRA for verifying logical correctness properties.

the reduction by the basic version of LRA as defined in Section 5.1 through Section 5.4, and the right bar depicts the reduction by the refined version LRA2 of LRA as defined in Section 5.5 (this does not apply for the detection of non-progress states alone). Observe once more that, when the error coverage is fixed, the performance in space and time of either version improves steadily with increasing level of concurrency. Observe also that, merely natural, the performance drops with increasing logical error coverage and a fixed level of concurrency. This is most perceptible when the verification task comprises the detection of all buffer overflows, especially for protocols with a low concurrency level. One should recall, however, that the detection of all buffer overflows and unspecified receptions in a protocol can be divided into several independent and potentially smaller subtasks (cf. Example 5.44 and Example 5.45). Executing these subtasks in parallel on different processors will generally result in larger space and time reductions.

6.3.2 Experiments with real protocols

Table 6.8 through Table 6.10 contain the results of our experiments with three "real" protocols taken from the literature. The first protocol is the CCITT X.21 interface specification of the *call establishment/clear procedure* for connecting Data Terminal Equipment (DTE) to Data Circuit-termination Equipment (DCE) in a public data network [CCITT76]. This specification can be formalized in the CFSM model as a 2-process protocol that models the interactions between a DTE and DCE [WZ78]. The second protocol is an *alternating bit protocol* for the transmission of data between two users over two *unreliable* simplex channels. This protocol is specified in the CFSM

Table 6.8 Experimental results for the X.21 call establishment/clear protocol.

Logical error coverage	Technique	States	Transitions	Space (MB)	Time (sec)
Types A, B, C and D	CRA	29868	64903	1.42	10.58
	LRA	25649	56150	1.22	9.85
	LRA2	25202	48332	1.20	11.33
Types A, B and D	SRA	26085	76744	1.24	12.90
	LRA	25596	56019	1.22	9.73
	LRA2	25124	48094	1.20	11.25
Types A, B and C	SRA	16400	31680	0.79	6.45
	LRA	15926	29126	0.77	5.90
	LRA2	15427	26666	0.74	6.33
Types A and B	SRA	16397	31654	0.79	6.57
	LRA	15922	29110	0.77	5.83
	LRA2	15443	26680	0.74	6.22
Type A	LRA	15313	26237	0.74	5.48

Table 6.9 Experimental results for the alternating bit protocol.

Logical error coverage	Technique	States	Transitions	Space (MB)	Time (sec)
Types A, B, C and D	CRA	135352	626608	6.84	64.68
	LRA	97068	462088	4.91	58.02
	LRA2	96280	355461	4.87	66.85
Types A, B and D	SRA	106232	1366696	5.37	190.57
	LRA	86526	411242	4.41	52.37
	LRA2	86525	310607	4.41	59.73
Types A, B and C	SRA	71200	389016	3.61	67.68
	LRA	63342	277920	3.22	50.60
	LRA2	60466	251741	3.07	58.70
Types A and B	SRA	71192	333886	3.61	61.65
	LRA	63708	265772	3.24	49.62
	LRA2	60670	242855	3.08	54.38
Type A	LRA	57898	223954	2.94	42.42

Table 6.10 Experimental results for the cache coherence protocol.

Logical error coverage	Technique	States	Transitions	Space (MB)	Time (sec)
Types A, B, C and D	CRA	37037	126152	1.90	32.52
	LRA	37037	126152	1.90	32.35
	LRA2	37037	126152	1.90	32.80
Types A, B and D	SRA	24858	156145	1.28	75.83
	LRA	19781	56901	1.03	19.47
	LRA2	18797	36526	0.98	20.43
Types A, B and C	SRA	26888	271472	1.37	111.08
	LRA	26857	88666	1.37	22.72
	LRA2	26857	84610	1.37	22.68
Types A and B	SRA	7120	14211	0.36	6.97
	LRA	6356	11749	0.32	5.25
	LRA2	5572	7920	0.28	5.23
Type A	LRA	5572	7619	0.28	3.53

model as a network of four communicating processes, viz. two processes to model the users and two processes to model possible message loss from the channels between the users [Pac87]. The third protocol is a *cache coherence protocol* originally described in 255 lines of a protocol specification language called Promela [Hol91]. We have rewritten the Promela specification as a protocol in the CFSM model. It consists of six communicating processes. Both the Promela specification and its translation in the CFSM model are given in the appendix of this thesis. We

have listed for each protocol the number of global states stored and transitions explored, and the actual amount of space (in megabytes) and time (in seconds) consumed by CRA, and by SRA, LRA and LRA2 for the different levels of logical error coverage they support. As in Figure 6.4, logical errors of type A, B, C and D stand respectively for non-progress states, non-executable transitions, unspecified receptions and for buffer overflows. Note that SRA is not listed in the first row of Table 6.8 through Table 6.10 where the logical error coverage is type A, B, C and D. This is because the implementation of SRA does not support the detection of both unspecified receptions and buffer overflows together in a single verification run [Özd95]. SRA is also not listed in the last row of Table 6.8 through Table 6.10 where the logical error coverage is type A only, because the technique is not formulated for the detection of non-progress states separate from the detection of non-executable transitions (recall the discussion before Table 6.2). LRA2 is not listed in this last row either because it applies as a refinement of LRA for detecting logical errors other than non-progress states (see Section 5.5).

6.4 Conclusion

In this chapter we have discussed the results of an empirical evaluation of the performance of LRA with respect to SRA and CRA for the verification of logical correctness properties of protocols. In order to eliminate potential favoritism, the three techniques were tested on several hundred protocols that were generated randomly with an automatic protocol synthesizer. In addition, they were tested on three real protocols taken from the literature. From all the experiments performed, we observed that LRA can yield significant extra reductions over SRA in both space and time for the detection of non-progress states in a protocol. For the detection of non-executable transitions, unspecified receptions and buffer overflows, the space reductions by LRA over SRA may be less significant, but the time reductions by LRA over SRA remain very good. We therefore conclude that LRA is a useful improvement of SRA as a relief strategy for protocol verification. In particular, it is an absolutely no-risk improvement of SRA: using LRA instead of SRA is at no cost whatsoever, neither in space nor in time.

Chapter 7

Leaping reachability analysis for LTL model-checking

Throughout the preceding chapters we have concentrated on relief strategies for verifying logical correctness properties of protocols, viz. indefinite progress, freedom of non-executable transitions, freedom of unspecified receptions and freedom of buffer overflows. These are general properties that are pertinent to basically all protocols, independent of their intended functionality. They are therefore often regarded also as "syntactic" correctness requirements. The focus of this chapter is instead on proving "semantic", or functional correctness requirements of protocols and, moreover, of concurrent systems in general. Specifically, we consider the verification of properties that are expressible as formulas in *linear-time temporal logic*. Linear-time temporal logic (LTL) is a popular formalism for reasoning about the semantic correctness of concurrent systems. It is well suited for specifying temporal properties over infinite executions of a system, including arbitrary safety and liveness properties. Given a concurrent system and a temporal formula (in LTL), verifying that every execution of the system satisfies the formula is known as *(LTL) model-checking*.

For many concurrent systems, model-checking also suffers severely from the state explosion problem. An eminent and quite general approach to relieve the state explosion problem for model checking is the partial-order approach, which actually captures a group of cognate state exploration techniques called *partial-order reduction methods*. These methods have been developed in recent years by different researchers [God90, Val90, HGP92, KP92a, Val92, Val93, GW93, GW94, HP95, Pel96], largely independent of the specific model used for specifying concurrent systems. They have proved effective for verifying local and termination properties of concurrent systems, as well as for LTL model-checking. Experiments have shown that these methods can substantially reduce the space and time needed for LTL model-checking.

Notwithstanding these results, in this chapter we propose an enhancement of partial-order reduction methods, in particular of the most recent and advanced one described in [HP95, Pel96]. Following a prelude to concurrent systems, temporal logic and LTL model-checking, we describe

the partial-order approach to LTL model-checking. We then show how the concepts underlying LRA in Chapter 5 can be combined with this approach to further relieve the state explosion problem for LTL model-checking. This is done first for concurrent systems in general, and subsequently for protocols defined in the CFSM model. Lastly, we provide an empirical comparison between the partial-order reduction method in [HP95, Pel96] and its proposed enhancement on the basis of experiments performed with the research tool package RELIEF discussed in Chapter 6. The main contributions of this chapter appeared in [Sch97, SU98b]

7.1 Preliminaries

7.1.1 Representing concurrent systems

Thus far we have confined ourselves to the CFSM model for the specification and verification of communication protocols. In Chapter 2, a protocol was defined explicitly as a set of processes (or finite state machines) which communicate asynchronously by exchanging messages over FIFO queues. In this chapter we consider more generally any (concurrent) system that can be formalized as a *finite* labeled transition system (LTS). An LTS is defined in the customary way as a quadruple (Q, q^0, Σ, Δ), where

- Q is a finite set of *states*,
- $q^0 \in Q$ is the *initial state* of the LTS,
- Σ is a finite set of *labels* (the alphabet of the LTS), and
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation.

An LTS can be used most elementary to formalize the behavior of a single sequential process. It can also formalize the *joint* behavior of a finite number of interacting and concurrently executing sequential processes. Each transition of the LTS then corresponds to the execution of a specific atomic operation or statement within one (or some, in case of synchronization) of the processes, in accordance with a standard interleaving semantics of concurrency. An LTS is sufficiently abstract to model virtually any finite-state system. In particular, every (bounded) protocol specified in the CFSM model can be represented by an LTS: the behavior of such a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ is formalized by the tuple (Q, q^0, Σ, Δ), with $Q = \mathbf{R}_\Pi$, $q^0 = G^0$, $\Sigma = \bigcup_{i \in I} \Delta_i$, and $\Delta = \{(G, t, H) \mid G \in \mathbf{R}_\Pi \wedge G \xrightarrow{t} H\}$ (cf. Chapter 2). In other words, all send and receive transitions in the protocol are viewed as atomic operations that act as global state transformers. Henceforth, we designate the term *concurrent system* to mean any system that can be formalized as a finite LTS, and we assume implicitly that the system is composed of a finite number of distinguishable sequential processes

$P_1, P_2, \ldots, P_n$. The labels of the transitions of a concurrent system are referred to as *operations*. Definition 7.1 captures the basic semantics of concurrent systems.

Definition 7.1

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system. An operation $a \in \Sigma$ is *executable* at a state $q \in Q$ if $(q, a, q') \in \Delta$, for some $q' \in Q$. The set of all operations (of a sequential process P_i in S) that are executable at q is denoted by $X(q)$ ($X_i(q)$). A *computation* of S over a sequence $\sigma = a_1 a_2 \ldots$ of operations from Σ is a finite or infinite sequence $\theta = q_0 q_1 q_2 \ldots$ of states from Q, where (i) $q_0 = q^0$, i.e. the sequence starts at the initial state of S, (ii) $(q_{i-1}, a_i, q_i) \in \Delta$ for all $i \geq 1$ (and $i \leq |\sigma|$ when σ is finite), and (iii) the sequence is *maximal*, i.e. θ is either infinite or its last state $q_{|\sigma|}$ satisfies $X(q_{|\sigma|}) = \emptyset$. A state $q \in Q$ is a *reachable* state if it occurs in some computation of S. □

Like in the previous chapters, we use $q \xrightarrow{a} q'$ to denote that the operation a leads from the state q to the state q' (i.e. $(q, a, q') \in \Delta$), and $q \xrightarrow{\sigma}{}^{*} q'$ to denote that the finite sequence of operations σ leads from q to q'. Naturally, a computation of a concurrent system, or any segment thereof, may be seen either as a sequence of reachable states, or as the corresponding sequence of executed operations. For ease of presentation, we will denote a computation interchangeably by a sequence of operations or by a sequence of states. As for protocols specified in the CFSM model, the complete reachable state space of a concurrent system S, including all its computations, can be represented by a labeled directed graph. Every (finite or infinite) path through this reachability graph that starts with the node corresponding to the initial state of S resembles the effects of a computation of S. There is one such path for each possible way in which the execution of operations can be interleaved in time.

7.1.2 Expressing properties of concurrent systems in temporal logic

Among the non-classical logics used in computer science, *temporal logic* has probably been the most successful. It is an extension of traditional logic (i.e. Boolean algebra and simple predicate calculus), and was first suggested by Pnueli [Pnu77] as a tool for the specification and verification of concurrent systems (also called concurrent programs). Temporal logic provides a sound logical basis for reasoning about the time varying behavior of concurrent systems without introducing an explicit notion of time. That is, it has been designed to reason about the order in which system events occur, as opposed to the actual times at which they occur. Expressions in temporal logic typically assert properties of *sequences* of states, whereas expressions in traditional logic assert properties of individual states.

Two variants of temporal logic are commonly distinguished, based on different conceptions of the nature of time: *linear* or *branching*. In linear temporal logic, time is viewed as linear, meaning

that each instant in time has a unique possible future (i.e. next instant). The structures over which linear temporal logic is interpreted are thus linear sequences. In branching temporal logic, each time instant may split into several possible futures, for instance those resulting from nondeterminism. All these possible futures are then considered to be equally real, while in linear temporal logic only one of them is regarded as the future that will actually occur. The structures over which branching temporal logic is interpreted can be viewed as infinite trees. Both the linear and branching variants of temporal logic are well-established. Excellent surveys are given in [Lam80, Wol89, Eme90]. Lamport [Lam80] has argued that the logic of linear time is better suited for reasoning about concurrent systems. Wolper [Wol89] has found that the linear variant is natural when the properties of a system can be expressed in terms of its computations, and that the branching variant is well adapted when the properties are thought of in terms of the structure of the system. In general, it appears largely a matter of debate as to which variant is to be preferred [Eme90]. For the purpose of this chapter we consider linear-time temporal logic (LTL) only.

Formulas in LTL assert properties of *infinite* sequences of states. An LTL formula is built from Boolean propositions, the Boolean connectives '$\neg$', '$\wedge$', ..., and the *temporal operators* '$\bigcirc$' (read as "next-time"), '$\Box$' (read as "henceforth" or "always"), '$\Diamond$' (read as "eventually"), and 'U' (read as "until"). Precisely, let AP denote a finite set of atomic propositions, and $p \in AP$, then the syntax of LTL is as follows:

$$f \ ::= \ p \mid \neg f \mid f_1 \wedge f_2 \mid \bigcirc f \mid \Box f \mid \Diamond f \mid f_1 \, U \, f_2 \ .$$

The atomic propositions in the set AP are assumed to refer to the states of the concurrent system for which an LTL formula asserts a property. Indeed, since all states of a concurrent system S are in essence just different combinations of values assigned to the variables that constitute S, each state is uniquely characterized by the subset of atomic propositions which hold true in that state. Every computation of S can therefore be interpreted as a *propositional sequence* over the set 2^{AP}. Although LTL allows assertions only over infinite sequences of states, in order to take into account also terminating computations of S it is common practice to transform these finite computations into infinite ones simply by repeating the last state forever [Val92, Pel96]. Let $\theta = q_0 q_1 q_2 \ldots$ be an infinite sequence of states interpreted as a propositional sequence, and denote by θ^i the suffix of θ starting from its i-th state. For an LTL formula f, one usually writes $\theta \models f$ to mean that θ *satisfies* f (or f holds true for θ), in accordance with the following semantics:

- $\theta \models p$ iff $p \in AP$ holds true in q_0,
- $\theta \models \neg f$ iff not $\theta \models f$,
- $\theta \models f_1 \wedge f_2$ iff $\theta \models f_1$ and $\theta \models f_2$,
- $\theta \models \bigcirc f$ iff $\theta^1 \models f$,

- $\theta \models \Box f$ iff $\forall i \geq 0$: $\theta^i \models f$,
- $\theta \models \Diamond f$ iff $\exists i \geq 0$: $\theta^i \models f$,
- $\theta \models f_1 \, U \, f_2$ iff $\exists i \geq 0$: $\theta^i \models f_2$ and $\forall 0 \leq j < i$: $\theta^j \models f_1$.

Note that the "eventually" operator is the dual of the "henceforth" operator, viz. $\Diamond f = \neg\Box\neg f$. Also note that $\Diamond f$ and $\Box f$ can be expressed in terms of the "until" operator as *true* $U f$ and $f U$ *false*, respectively. Informally, where the Boolean connectives (including '$\vee$' and '$\Rightarrow$' defined in terms of '$\neg$' and '$\wedge$') have their usual interpretations, the temporal operators have the following meaning:

- $\bigcirc f$ holds in the current state if f holds in the next state,
- $\Box f$ holds in the current state if f holds in the current state and in all subsequent states (in the linear sequence on which the formula is interpreted),
- $\Diamond f$ holds in the current state if f holds in the current state or in some subsequent state, and
- $f_1 \, U \, f_2$ holds in the current state if $\Diamond f_2$ holds in the current state, and if f_1 holds in the current state and in all subsequent states preceding the state in which f_2 holds.

Unlike the "henceforth", "eventually" and "until" operators, the "next-time" operator is often omitted by researchers in reasoning about temporal properties of systems. This has been instigated mostly by Lamport, who strongly objects to the use of the "next-time" operator, claiming that it introduces a notion of time which is too discrete (namely between two immediately following time instants) to fit the level of abstraction appropriate for a specification formalism [Lam83]. Formally, he found that every *nexttime-free LTL* formula is *closed under stuttering*, meaning that the formula cannot distinguish between two *stuttering equivalent* sequences. Two sequences of states, or the corresponding propositional sequences, are stuttering equivalent if they can be made identical by replacing in both sequences every finite adjacent number of occurrences of the same state with a single occurrence [Lam83]. Since a temporal formula containing the "next-time" operator is not necessarily closed under stuttering, Lamport argued that this operator enables the expression of distinctions between systems that should be considered equivalent.

Many interesting properties of concurrent systems can be formalized in LTL, even without the "next-time" operator. For instance, that some property P is invariant throughout system execution is expressed simply as $\Box P$. In order to state that a property P always causes a property Q to hold subsequently, one writes $\Box(P \Rightarrow \Diamond Q)$. This combination of operators is often used to specify the eventual response (i.e. Q) to some given request (i.e. P). Asserting that a property P is satisfied infinitely often is done by writing $\Box\Diamond P$. This means that for each state along a computation there is a future state in which P will be true. Lastly, an expression of the form $P \Rightarrow (P \, U \, Q)$ asserts that

if P is true in the current state, then it will remain true at least until Q becomes true.

In general, (nexttime-free) LTL is sufficiently expressive to capture all *safety* and *liveness* properties of concurrent systems. These properties have been regarded by many researchers as the two most fundamental types of properties one would want to prove of a concurrent system (see e.g. [Lam77, Lam80, Lam83, WVS83, AS87, MP92]). Intuitively, a safety property asserts that something bad never happens, whereas a liveness property asserts that something good must eventually happen [Lam80]. "Something bad" thereby refers to the system entering an unacceptable state, and "something good" to the system entering a desirable state. Well-known safety properties are partial correctness (a program never halts with the wrong answer), mutual exclusion (two processes are never in their critical sections at the same time), and indefinite progress or deadlock-freedom (a system never enters a state in which no further progress is possible). The absence of unspecified receptions and buffer overflows considered earlier for protocols in the CFSM model also classify as safety properties. Note that all these examples of safety properties are (global or local) state invariances. Safety properties can also stipulate some precedence relation between states or events. For systems that implement FIFO buffers, an example is the assertion that messages are taken from a buffer in the same order as they are put into the buffer (this property was assumed to hold for protocols in the CFSM model). Familiar liveness properties are termination (a program eventually halts) and freedom from starvation or livelocks (each process makes progress infinitely often). Other kinds of liveness properties that are important for concurrent systems include such assertions as "a request for service will eventually be granted", "a process will eventually enter its critical section", or "a message will eventually reach its destination".

A formalization of safety and liveness properties appeared in [AS87], where the two kinds of properties are characterized from a language-theoretic point of view. It was also shown in [AS87] that every property which classifies neither as a safety property nor as a liveness property is in fact the conjunction of a safety and a liveness property. The safety/liveness classification is further discussed in view of a more refined hierarchy of temporal properties in [MP92], which includes a syntactic characterization of safety and liveness properties in terms of the temporal formulas that specify them. Basically, safety properties can be expressed using only the concept of "henceforth" (typically in the form $\Box P$ or $P \Rightarrow \Box Q$), whereas one needs the additional concept of "eventually" to express liveness properties (usually in the form $\Box\Diamond P$ or $\Box(P \Rightarrow \Diamond Q)$).

Another useful concept in the application of temporal logic to concurrent systems is *fairness*. Considering fairness means taking into account certain assumptions about the context in which processes of a concurrent system are executed. For instance, if concurrent processes are executed on different processors it is customary to assume that, if a process has an operation that remains executable, it will eventually execute it (this assumption is often called weak fairness). Various notions of fairness have been studied in [Fra86, MP92]. The purpose of these notions is to exclude

from analysis the computations of a concurrent system that would not be permitted by the specific type of process scheduler that is assumed. The fairness assumptions then act as filters, removing certain classes of infinite computations that conflict with the assumptions made about the process scheduler. Like safety and liveness properties, fairness assumptions can be expressed in (nexttime-free) LTL [LP85]. If a fairness assumption is formalized by an LTL formula f_1, one can use a logical implication $f_1 \Rightarrow f_2$ to assert that the property expressed by f_2 holds true under this fairness assumption.

7.1.3 Model-checking

Currently the most advocated method for verifying temporal properties of (finite-state) concurrent systems is *model-checking*. In the context of LTL, model-checking refers to a fully automatic procedure for checking that a given concurrent system satisfies, or is a model of, some property formalized as an LTL formula [LP85, VW86, Wol89]. A concurrent system is thereby defined to satisfy an LTL formula if *all* the computations of the system satisfy the formula. At first, (LTL) model-checking was proposed as an *off-line* procedure. This means that the actual algorithms for verifying the satisfiability of LTL formulas are applied to a concurrent system *after* constructing the reachable state space of the system. These algorithms do not work directly on the state space, but rather construct from it a graph which contains in each node information to derive the formulas that hold true in the state represented by the node, based on fixpoint characterizations of the temporal operators (for instance, the fixpoint characterization of $\Box P$ is $P \wedge \bigcirc\Box P$) [MW84, LP85]. It was recognized later that model-checking can also be performed *on-the-fly* [VW86], in which case the verification algorithms start to examine a given concurrent system *during* the construction of its reachable state space, not waiting for this construction to be completed. The main advantage of on-the-fly model-checking is that, if the checked formula does not hold true for the system, a counter example may be encountered before completing the construction of its state space. It is well argued in [Val93] that this advantage appears exactly when needed the most: the state spaces of incorrect systems tend to be "extra" large due typically to their faulty behavior. Another advantage of on-the-fly model-checking is that, in some cases, certain parts of the state space that are not important to the verification of the checked formula may be omitted, even when the formula happens to be satisfied by the system.

The ability to verify temporal properties on-the-fly has actually emerged from the so-called *automata-theoretic approach* to model-checking [WVS83, VW86]. This approach is based on the fact that for each LTL formula it is possible to construct a non-deterministic *Büchi automaton* [Büc62] that accepts exactly the (infinite) sequences of states satisfying the formula. Formally, a Büchi automaton is a quintuple $A = (Q, q^0, \Sigma, \Delta, F)$, where

- Q is a set of states,
- $q^0 \in Q$ is the initial state,
- Σ is an alphabet,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and
- $F \subseteq Q$ is a set of acceptance states.

Büchi automata are a theoretical means to define languages of infinite strings: a string is accepted by a Büchi automaton if the automaton enters one of its acceptance states infinitely many times while reading the string. Notice that a Büchi automaton can be seen as an LTS with various states predefined as acceptance states. One defines a computation of a Büchi automaton A over an infinite sequence of symbols $a_1a_2\ldots$ from Σ as an infinite sequence of states $q_0q_1q_2\ldots$ starting at the initial state q^0 of A, with $(q_{i-1}, a_i, q_i) \in \Delta$ for all $i \geq 1$ (cf. Definition 7.1). A is then said to *accept* the computation (or the computation is *accepting*) iff for some acceptance state $q \in F$ there are infinitely many states q_i such that $q_i = q$.

For an LTL formula f, the transitions in the corresponding Büchi automaton A_f carry predicate labels from the alphabet 2^{AP}, each of which represents a Boolean proposition [WVS83]. Recall that the atomic propositions of AP in f are supposed to refer only to the states of the concurrent system S for which the formula formalizes a property, and hence these Boolean propositions are in fact propositions on the states of S (i.e. on the values of the variables that constitute S). The Büchi automaton A_f then accepts an infinite computation θ of S (a sequence of system states) iff there exists an accepting computation of A_f (a sequence of automaton states) over θ. In other words, θ satisfies the formula f if there is a "path" p in A_f starting from the initial state of A_f, such that the label of the i-th edge in p holds true in the i-th state of θ, for all $i \geq 1$, and some acceptance state of F_f appears infinitely often in p. Further recall that the finite computations of S are taken into account by first transforming them into infinite ones, through infinite repetition of their last states. An algorithmic construction of a Büchi automaton A_f from an LTL formula f can be found in for instance [Wol89]. This construction is exponential in the length of the formula, defined as the number of symbols (propositions and connectives) it contains. However, the exponential blow-up of the number of states in A_f is usually not a concern since most formulas checked in practice are quite short, and since the construction algorithm often behaves much better than its upper bound [VW86, Wol89].

Example 7.2

Figure 7.1 depicts a Büchi automaton which accepts exactly all the infinite computations satisfying the LTL formula $\neg\Box(P \Rightarrow \Diamond Q) = \Diamond(P \wedge \Box\neg Q)$. That is, every sequence of states containing a

state in which $P \wedge \neg Q$ holds true, and from which Q never holds true in any state in the remainder of the sequence, is accepting. The initial state of the automaton is indicated by the symbol '$\vee$', and its only acceptance state by a double circle. This formula can be used, for example, to express the negation of a precedence property of a concurrent system, stipulating that it is always the case that the execution of some operation a (for instance, a send operation) is eventually followed by the execution of an operation b (for instance, the matching receive operation). The LTL formula then accepts all computations that violate this property, i.e. all computations in which an occurrence of a is never followed by an occurrence of b. The formula can indeed be described equivalently by a Büchi automaton over the alphabet of operations of the system, rather than by an automaton over state predicates as in Figure 7.1. Such an automaton over system operations is obtained from the automaton in Figure 7.1 by replacing *true* with Σ, $P \wedge \neg Q$ with a, and $\neg Q$ with $\Sigma \backslash \{b\}$. □

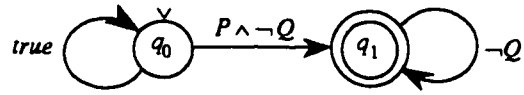


Figure 7.1 A Büchi automaton for $\neg\Box(P \Rightarrow \Diamond Q)$.

The automata-theoretic approach to LTL model-checking now proceeds as follows [WVS83, VW86]. Given a concurrent system S and an LTL formula f to be checked for S, one first builds a Büchi automaton $A_{\neg f}$ for the *negation* of f. It accepts all and only sequences of states that satisfy $\neg f$, i.e. that violate f. Secondly, one computes the so-called *synchronous product* of (the reachable state space of) S and $A_{\neg f}$, a Büchi automaton which accepts exactly those computations of S that violate f. This automaton is then checked for emptiness: either it does not accept any computation, implying that all computations of S do in fact satisfy f, or it accepts at least one computation of S which is a counter example to f.

The synchronous product of S and $A_{\neg f}$ in the above three-step procedure is actually defined as the product of the two Büchi automata A_S and $A_{\neg f}$, where A_S is obtained from S by designating all states of S as acceptance states. Precisely, if $S = (Q_S, q_S^0, \Sigma_S, \Delta_S)$ then $A_S = (Q_S, q_S^0, \Sigma_S, \Delta_S, F_S)$, with $F_S = Q_S$, and the product of A_S and $A_{\neg f} = (Q_{\neg f}, q_{\neg f}^0, \Sigma_{\neg f}, \Delta_{\neg f}, F_{\neg f})$ is the Büchi automaton $A_S \times A_{\neg f} = (Q, q^0, \Sigma, \Delta, F)$ defined by

- $Q = Q_S \times Q_{\neg f}$,
- $q^0 = (q_S^0, q_{\neg f}^0)$,
- $\Sigma = \Sigma_S \times \Sigma_{\neg f}$,
- $\Delta \subseteq Q \times \Sigma \times Q$ such that $((x, y), (a, \mathbf{P}), (x', y')) \in \Delta$ iff $(x, a, x') \in \Delta_S$, $(y, \mathbf{P}, y') \in \Delta_{\neg f}$, *and*

the Boolean proposition **P** holds true in state $x \in Q_S$,

- $F = F_S \times F_{\neg f}$.

For further referencing we call the Büchi automaton A_S the *full automaton* for S. Notice that the transitions of $A_{\neg f}$ are essentially used to test the values of the variables of the concurrent system S whenever the system is ready to execute an operation, as explained before. Operations of S that can affect the truth value of any proposition in the LTL formula f are also said to be *visible*. That is, an operation a of S is visible if there exists a transition label in $A_{\neg f}$ for which the corresponding proposition has a truth value in a system state q that is different from its truth value in a state q' with $(q, a, q') \in \Delta_S$; otherwise it is *invisible*. The set of all visible operations of S with respect to the formula f is denoted by $vis_f(S)$. As the exact set of visible operations is generally too hard to determine, in practice one would have to compute some upper approximation of $vis_f(S)$, which can be done by a syntactic analysis of (the operations of) S [Val92].

An alternative definition for the product automaton $A_S \times A_{\neg f}$ would apply if, as indicated in Example 7.2, $A_{\neg f}$ were taken as an automaton over the alphabet Σ_S of system operations. In that case, the transitions of the product automaton would also carry labels from Σ_S, and the relation $\Delta \subseteq Q \times \Sigma_S \times Q$ would be such that $((x, y), a, (x', y')) \in \Delta$ iff $(x, a, x') \in \Delta_S$ and $(y, a, y') \in \Delta_{\neg f}$. In this framework, the transitions in A_S and in $A_{\neg f}$ are thus synchronized on operations (i.e. on the transition labels), and the operations of S that can, through synchronization, alter the state of $A_{\neg f}$ are the visible transitions. In the first definition of the product automaton one can see the transitions as being synchronized on states [God96]. Both frameworks are used in like manner for model-checking.

As it appears, the product automaton $A_S \times A_{\neg f}$ can be computed without ever building the full automaton A_S. In other words, the reachable state space of S need not be constructed explicitly. The product automaton can at the same time also be checked for emptiness. This is precisely what is meant by *on-the-fly* model-checking. First, the inspection of the state space of S is guided by the checked formula, which acts as a constraint on the system's behavior through the required accordance of proposition labels (or the required synchronization of operations). In some cases the automaton $A_S \times A_{\neg f}$ may therefore be smaller than A_S itself. Second, the product automaton may be found non-empty before completing its construction. It is well-argued in [CV+92, Val93] that this advantage appears exactly when needed the most: state spaces of incorrect systems tend to be "extra" large due typically to their erroneous behavior. Deciding emptiness thereby amounts to checking whether there exists a cycle in $A_S \times A_{\neg f}$ (when viewed as a graph) that is reachable from the initial state $(q_S^0, q_{\neg f}^0)$ and that contains an acceptance state (which is hence repeated infinitely often). A particular memory-efficient algorithm for on-the-fly detection of such acceptance cycles is given in [CV+92]. It requires space linear in the number of states of $A_S \times A_{\neg f}$ and implements a so-

called nested depth-first search: a first search to find a reachable acceptance state, followed by a second search to determine whether this acceptance state can be reached from itself. Checking whether the product of two Büchi automata is empty is known to be much easier than checking whether the language generated by one of the automata is included in the other (a PSPACE-complete problem), which explains why one uses the Büchi automaton for the formula $\neg f$ instead of f [VW86, Wol89]. An overview of various algorithms for checking the emptiness of Büchi automata is given in [GH93], all of which also run in time linear to the size of these automata.

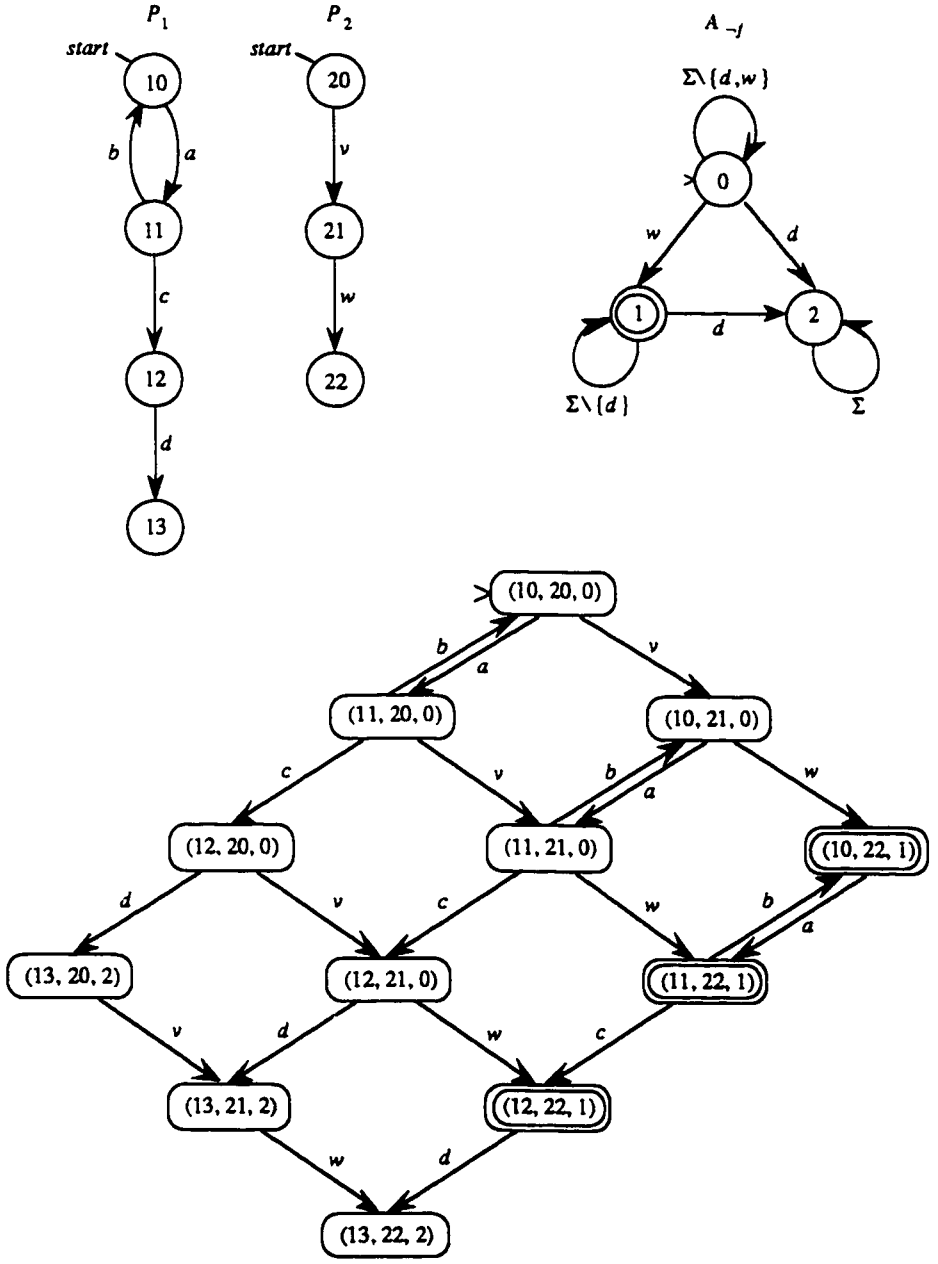


Figure 7.2 The product $A_S \times A_{\neg f}$ for the concurrent system S and an LTL formula f in Example 7.3.

Example 7.3

Consider a concurrent system S composed of two sequential processes P_1 and P_2, depicted in Figure 7.2, and assume for simplicity that these processes execute autonomously, i.e. without any interaction. Suppose we want to check the absence of computations of S in which operation w occurs and operation d does not occur. This property can be expressed by an LTL formula f of the form $\Diamond$('P_2 is at local state 22') $\Rightarrow \Diamond$('P_1 is at local state 13'). The negation of the property is described in Figure 7.2 as a Büchi automaton $A_{\neg f}$ over the alphabet Σ of operations of S. It accepts all infinite sequences of states starting with its initial state 0 in which its (only) acceptance state 1 is repeated infinitely often, and thus all computations of S along which w occurs and d does not occur. Note that the operations w and d are the visible operations of S, as they can change the state of $A_{\neg f}$. The product automaton $A_S \times A_{\neg f}$ is illustrated at the bottom of Figure 7.2, where the three double-circled states are the acceptance states (i.e. $A_{\neg f}$ must be in state 1). It is non-empty as it accepts several computations of S, namely those in which w is executed and in which a and b are executed infinitely many times thereafter. In practice, the on-the-fly construction of $A_S \times A_{\neg f}$ need not be completed, but can be stopped as soon as one of the accepting computations is found. □

7.2 The partial-order approach to LTL model-checking

Being the main practical limitation for all verification methods based on state exploration, the state explosion problem must be reckoned with also when using model-checking for verifying temporal properties of concurrent systems. Certainly, on-the-fly model-checking already has a head start over off-line model-checking, but the often excessive size of the full automaton A_S of a concurrent system S still renders the construction of the product automaton $A_S \times A_{\neg f}$ impractical for most real systems. An eminent and quite general approach to tackle the state explosion problem in concurrent system verification is the partial-order approach. This actually refers to a collection of cognate state exploration techniques, called *partial-order reduction methods*, that have been developed in recent years by different researchers [God90, Val90, HGP92, KP92a, Val92, Val93, GW93, GW94, HP95, Pel96]. Partial-order reduction methods are largely independent of the specific model used for specifying concurrent systems, and they have proved adequate not only for verifying local and termination properties [God90, Val90, HGP92, KP92a, GW93], but also for LTL model-checking [Val92, Val93, GW94, HP95, Pel96]. Experiments have indicated that these methods can substantially reduce the space and time needed for model-checking. They have also been shown to combine well with on-the-fly model-checking [Val93, GW94, Pel96] and with model-checking under certain fairness assumptions [Pel93, Pel96].

Like most improved state exploration techniques, partial-order reduction methods are inspired by the observation that many properties of interest to concurrent systems are insensitive to the

execution order of concurrent, or independent, atomic operations (see Chapter 3). Common to all partial-order reduction methods is the use of an explicit *dependency relation* among the operations of a concurrent system, which induces an equivalence relation between computations of the system (cf. Section 4.1.1). The term "partial-order reduction" then stems from the fact that equivalence classes of computations are actually partial orders of operation occurrences.

Definition 7.4

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system. A *dependency relation* for S is a reflexive and symmetric relation $D \subseteq \Sigma \times \Sigma$ such that for all $a, b \in \Sigma$, $(a, b) \notin D$ implies that the following two properties hold for each $q \in Q$:

i) if $a \in X(q)$ and $q \xrightarrow{a} q'$, then $b \in X(q)$ iff $b \in X(q')$;

ii) if $a, b \in X(q)$, then there exists a unique state q' such that $q \xrightarrow{ab}{}^* q'$ and $q \xrightarrow{ba}{}^* q'$.

Two operations $a, b \in \Sigma$ are *dependent* iff $(a, b) \in D$; otherwise, they are *independent*. □

The first requirement listed in Definition 7.4 states that independent operations can neither enable nor disable each other, and the second requirement states that executing independent operations is commutative. The definition itself may at first seem of no more than semantic use, since it is not practical to check these requirements for every pair of operations and for every state of a concurrent system. However, in practice it is possible indeed to give easily checkable syntactic conditions that are sufficient for operations to be independent [HP95, God96]. For instance, two operations from the same sequential process can generally not be independent. If the two operations are defined in sequence, executing the first one will enable the other. If they appear together is a single selection, executing either operation will disable the other. Operations from distinct sequential processes can be independent under certain conditions. Two operations from distinct sequential processes that access only local variables within each process will in general be independent. Two send or receive operations on distinct message queues are usually also independent, but two such operations on the same queue need not be. For two send operations on the same queue, the operation that is executed first may disable the second if it yields a full queue, or the execution order of the two operations may be distinguished by the order in which the messages sent appear in the destination queue, which violates the commutativity requirement. For two receive operations on the same queue, the operation that is executed first may disable the second if it yields a empty queue. For a send and receive operation on the same queue, the send operation may enable the receive operation if the queue is currently empty, or vice versa if the queue is full. Various ways to refine dependency relations, in order to increase the number of pairs of independent operations, can be found in [KP92b, Val92, GP93, God96]. A particularly evident way is to define them as being *conditional*

upon states: instead of defining a dependency relation that holds for all (reachable) states of a system, it is possible to define such relation for each state individually [KP92b]. Definition 7.4 then becomes as follows.

Definition 7.4bis

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system. A relation $D \subseteq \Sigma \times \Sigma \times Q$ is a *conditional dependency relation* for S iff for all $a, b \in \Sigma$ and $q \in Q$, $(a, b, q) \notin D$ implies that $(b, a, q) \notin D$, and that the following two properties hold in state q:

i) if $a \in X(q)$ and $q \xrightarrow{a} q'$, then $b \in X(q)$ iff $b \in X(q')$;

ii) if $a, b \in X(q)$, then there exists a unique state q' such that $q \xrightarrow{ab}{}^{*} q'$ and $q \xrightarrow{ba}{}^{*} q'$.

Two operations $a, b \in \Sigma$ are *dependent in* a state $q \in Q$ iff $(a, b, q) \in D$; otherwise, they are *independent in* q. □

For ease of presentation, we will adhere to the use of a binary, unconditional dependency relation between operations as in Definition 7.4. Nevertheless, all what follows in the rest of this chapter is valid also with a conditional dependency relation, and can readily be interpreted in that context.

Definition 7.5

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system. Two *finite* sequences of operations $\sigma, \sigma' \in \Sigma^*$ are *equivalent* with respect to (wrt) a dependency relation D for S, denoted by $\sigma \equiv_D \sigma'$, iff there exist sequences $\sigma_1, \sigma_2, \ldots, \sigma_k$ such that $\sigma_1 = \sigma$, $\sigma_k = \sigma'$, and for all $1 \leq i < k$, $\sigma_i = \mu ab \rho$ and $\sigma_{i+1} = \mu ba \rho$, for some $\mu, \rho \in \Sigma^*$ and $a, b \in \Sigma$ with a and b independent wrt D (in the states where they are permuted, in case of a conditional dependency relation D).

For $\upsilon, \upsilon' \in \Sigma^* \cup \Sigma^\omega$ (i.e. υ and υ' can be finite or infinite), define $\upsilon \preceq_D \upsilon'$ iff for all $\mu \in pref(\upsilon)$ there exist $\mu' \in pref(\upsilon')$ and $\rho \in \Sigma^*$ such that $\mu' \equiv_D \rho \wedge \mu \in pref(\rho)$, where $pref(\upsilon)$ is the set of finite prefixes of a (finite or infinite) sequence of operations υ. Two *infinite* sequences of operations σ, $\sigma' \in \Sigma^\omega$ are equivalent wrt D iff $\sigma \preceq_D \sigma'$ and $\sigma' \preceq_D \sigma$. □

Intuitively, two sequences of operations are equivalent (wrt a given dependency relation) if one sequence can be obtained from the other by repeatedly permuting adjacent independent operations [Maz86]. The extension of the equivalence relation '$\equiv_D$' to infinite sequences of operations in the definition above is adopted from [Pel96]. Equivalence classes induced by '$\equiv_D$' are also called *traces* [Maz86], and traces consisting of computations (i.e. sequences of operations that are maximal) of a concurrent system are sometimes referred to as *runs* of the system. For any finite sequence of operations σ, it follows readily from Definition 7.4 that all sequences of operations equivalent to σ lead to the same state (cf. Proposition 4.3).

Since equivalent computations of a concurrent system differ only in the order of independent, commutative operations, it appears not necessary in general to examine all computations in order to verify the system against various desirable properties. It is instead sufficient for many properties to examine just one representative computation per equivalence class of computations. Accordingly, partial-order reduction methods attempt as much as possible to fix an order among independent operations, by executing at each state encountered during state exploration only a *subset* of the operations executable at that state, rather than all of them. State exploration is thereby performed usually via a depth-first search, or some variation of it. Some conditions apply to selecting a subset of the executable operations at a given state, which must guarantee that the reduced part of the reachable state space of a concurrent system that is explored by a partial-order reduction method preserves the property being checked. Devised by different researchers, such subsets adhere to different names: *stubborn* sets [Val90, Val92, Val93], *persistent* sets [God90, GW93, GW94], *faithful decompositions* [KP92] or *ample* sets [HP95, Pel96]. Although the definitions of these sets and the associated algorithms differ, they do have much in common [God96] and are therefore referred to collectively as partial-order reduction methods.

For the course of this chapter it would be futile to try to capture all the subtleties of the various suggested partial-order reduction methods, some of which take advantage of confining themselves to a restricted class of properties. We will describe, and subsequently enhance, the partial-order reduction method based on ample sets [HP95, Pel96]. This particular method has been proposed most recently, and it is fairly generic in the sense that it can be adapted without too much difficulty to resemble the other partial-order reduction methods. Furthermore, it is advocated as the most advanced partial-order reduction method in terms of the properties that can be checked, the way fairness is dealt with, and the low overhead and high overall performance of its implementation [HP95, Pel96]. The method has been implemented as an extension to SPIN, a verification tool which is increasingly being used for teaching and for industrial applications [Hol91, SPIN95, SPIN96, SPIN97]. For convenience, the partial-order reduction method based on ample sets will be referred to as POVAS (Partial-Order Verification with Ample Sets).

POVAS is intended as a relief strategy for verifying concurrent systems against properties formalized by nexttime-free LTL formulas, i.e. for nexttime-free LTL model-checking. It comes in four different "modes", depending on whether model-checking is done off-line or on-the-fly, and with or without certain fairness assumptions. We focus primarily on the off-line and on-the-fly versions without fairness assumptions. Model-checking under fairness assumptions with POVAS is conceptually not much different, and will be addressed later in Section 7.3.

7.2.1 Off-line LTL model-checking with POVAS

POVAS implements a depth-first search (DFS) algorithm which, in contrast to a classical brute-force DFS, incorporates three conditions for selecting a *subset* of the operations that are to be executed at a given state encountered during state exploration. For the off-line version in particular, when a state q on the DFS stack is expanded and at least one operation is executable at q, a non-empty subset of $X(q)$ is used to generate successor states for q in accordance with the following definition [Pel96].

Definition 7.6

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system, D a dependency relation for S, f an LTL formula to be checked for S, and $q \in Q$ the current state to be expanded during the DFS. An *ample set* in q is a non-empty subset $A \subseteq X(q)$ of operations executable at q satisfying the following three conditions:

i) for every non-empty sequence $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots q_k \xrightarrow{a_k} q_{k+1}$ from $q_1 = q$, with $a_i \in \Sigma \backslash A$ for all $1 \leq i \leq k$, each operation a_i is independent wrt D (in q_i, in case of a conditional dependency relation D) of all operations in A;

ii) if $A \subset X(q)$, then no operation $a \in A$ with $q \xrightarrow{a} q'$ is such that q' is on the current DFS stack;

iii) if $A \subset X(q)$, then $A \cap vis_f(S) = \emptyset$.

An ample set in q is denoted by *ample*(q). □

Note that the set $X(q)$ itself is trivially an ample set in state q. In the sequel, we refer to the three conditions on ample sets in Definition 7.6 as conditions **C1**, **C2** and **C3**, respectively. The first condition **C1** is a consistency requirement. It guarantees that after state q is reached, no operation outside *ample*(q) that is dependent of an operation in *ample*(q) can be executed before an operation in *ample*(q) is executed. Equivalently, every single operation outside *ample*(q) is either independent (in q) of all operations in *ample*(q), or it is not executable at q and at every state that can be reached from q without executing an operation in *ample*(q). The execution at q of only the operations in *ample*(q) does therefore not affect "negatively" the executability of any operation outside *ample*(q), for operations outside *ample*(q) which are already executable at q remain executable, while those not executable at q can "only" become executable. Condition **C2** is enforced to avoid the ignoring problem, which may cause the execution of operations to be deferred indefinitely along a cycle. We have already addressed this problem in Chapter 5 in the context of LRA for verifying logical correctness properties of protocols in the CFSM model. It is of the same nature here: condition **C2** guarantees that a state q is fully expanded (i.e. *ample*$(q) = X(q)$) whenever one of the operations in *ample*(q) closes a cycle on the DFS stack, thereby providing an exit from this cycle if one exists

(cf. Section 5.5). Lastly, condition **C3** is enforced in view of the fact that the checked LTL formula may very well be sensitive to the order of two visible operations of the concurrent system (i.e. operations that can affect the truth value of the formula), even when these operations are mutually independent. The effect of not allowing visible operations in $ample(q)$, in case it is a proper subset of $X(q)$, is that all the possible execution orders of all visible operations will be explored. Every two visible operations are then essentially treated as being always dependent. In this regard, condition **C3** actually justifies why POVAS should be restricted to LTL formulas that are nexttime-free or, in general, to temporal properties that are stuttering closed (see Section 7.1.2). Although POVAS can in principle also handle LTL formulas that do contain a "next-time" operator, such formulas generally cause *all* the operations of a concurrent system to be visible, and hence they would all be considered as dependent. Each state encountered during state exploration by POVAS would then be fully expanded, exactly as in a classical DFS, which annihilates any benefit coming from the use of POVAS. In the remainder of this chapter, we implicitly mean nexttime-free LTL when we refer to LTL.

The algorithm presented in [HP95, Pel96] for calculating ample sets is as follows. Based on the fact that operations of a single sequential process can generally not be independent, it seeks some process in a concurrent system whose set of operations executable at the current state q satisfy conditions **C1** to **C3**. As soon as such a process P_i is found, the set $X_i(q)$ is returned as $ample(q)$. If no such process exists, the algorithm returns the entire set $X(q)$ of all operations executable at q. Condition **C2** must be checked during state exploration by inspecting the current contents of the DFS stack. However, most of the information required for checking **C1** and **C3** is gathered efficiently by a static analysis of the concurrent system *before* state exploration [HP95]. That is, during system compilation each local state of each sequential process is analyzed and annotated with one of three types of labels: *safe*, *conditionally safe* upon some condition, or *unsafe*. These labels signify whether at run time (i.e. during state exploration), when a system state q is expanded and some process is at its local state l, the set of operations of this process that are defined at l and executable at q satisfies conditions **C1** and **C3**. A local state l of a process is labeled "safe" if it is determined at compile time that the set of (executable) operations defined at l will qualify as an ample set. This would be the case if all these operations are invisible, and if they are independent of every operation belonging to another process. Recall from Section 7.1.3 that (an upper approximation of) the set $vis_f(S)$ of visible operations of a concurrent system is itself computed statically [Val92]. Analogously, l is labeled "unsafe" if it is already decided at compile time that no ample set can be formed from the operations defined at l. The local state l is labeled "conditionally safe" for some *precomputed* condition C (which is one out of a small number of conditions [Pel96]) when the operations defined at l form an ample set only if C holds during run time. For example, if only send operations are defined at l, such a condition can be that none of the

corresponding queues are filled to their capacity.

Several other algorithms have been proposed earlier to compute sets of operations that satisfy in particular condition **C1**[6]. An overview and a comparison of these algorithms can be found in [God96], all of which also infer such sets from the syntactic structure of a concurrent system, but not statically as in the algorithm described above. They further differ from the algorithm in [HP95, Pel96] in their aim to compute the smallest sets of operations satisfying **C1**. Typically, the more information (static or dynamic) used, the smaller these sets can be, but at the cost of an increased computational complexity. Moreover, exploring the smallest number of operations at each step during state explarion is only a heuristic: it does not necessarily lead to the generation, and hence the storage, of the smallest number of states. For these reasons, checking **C1** in the ample set algorithm is based on a more delicate trade-off between the storage space and the overall execution time needed for model-checking [HP95, Pel96].

To sum up, off-line model-checking with POVAS proceeds as a "selective" DFS, using the above algorithm for calculating ample sets to determine for each state encountered during the DFS the subset of successor states that need be expanded next. As a result, it explores only a reduced part of the reachable state space of a concurrent system S. Precisely, instead of building the full automaton $A_S = (Q_S, q_S^0, \Sigma_S, \Delta_S, Q_S)$ for S, it builds a *reduced* automaton A'_S for S defined by the tuple $(Q'_S, q_S^0, \Sigma_S, \Delta'_S, Q'_S)$, where $Q'_S \subseteq Q_S$ and $\Delta'_S \subseteq Q'_S \times \Sigma_S \times Q'_S$ such that $(q, a, q') \in \Delta'_S$ iff $a \in ample(q)$ (as opposed to $a \in X(q)$) and $q \xrightarrow{a} q'$. This reduced automaton A'_S preserves all non-progress states of S (states at which no operation is executable), and it reveals all reachable local (or process) states and thus all non-executable operations (operations that are not executable at any reachable state of S). Moreover, for every computation σ of S (or equivalently of A_S), there is at least one computation σ' of A'_S such that σ and σ' are stuttering equivalent [HP95, Pel96]. Hence, when a temporal property is closed under stuttering, the property holds true for (all computations of) S iff it holds true for all the computations of A'_S. Since all nexttime-free LTL formulas are closed under stuttering, algorithms for off-line LTL model-checking [LP85] can be applied directly to A'_S, rather than to the full automaton A_S, in order to verify these formulas.

7.2.2 On-the-fly LTL model-checking with POVAS

POVAS can readily be combined with on-the-fly LTL model-checking in order to gain from both. Verifying an LTL formula f for a concurrent system S then involves constructing the product of the reduced automaton A'_S for S and the Büchi automaton $A_{\neg f}$ for the negation of f, and checking its emptiness. This can again be done by seeking acceptance cycles in $A'_S \times A_{\neg f}$, similar as explained

[6] Condition **C1** is in fact the basic consistency requirement that underlies also the other partial-order reduction methods based on stubborn sets, persistent sets, or faithful decompositions.

before in Section 7.1.3. POVAS utilizes the nested DFS algorithm given in [CV+92], or actually a slight modifcation of it to ensure compatibility with the selective DFS algorithm based on ample sets [HPY96] (the need for this modification was not yet recognized in [HP95, Pel96], but the authors proposed the correction in [HPY96]).

For the on-the-fly version of POVAS, the calculation of ample sets itself also undergoes a minor change so that it applies to composite states of the product automaton $A'_S \times A_{\neg f}$ instead of single system states. When a composite state (q, r) of $A'_S \times A_{\neg f}$ on the DFS stack is expanded and at least one operation is executable at system state q, a non-empty subset $ample(q, r)$ of $X(q)$ is employed to generate successor states for (q, r), which satisfies the earlier conditions **C1** and **C3** in Definition 7.6 and the new condition **C2′** [Pel96]:

C2′ if $ample(q, r) \subset X(q)$, then no operation $a \in ample(q, r)$ with $q \xrightarrow{a} q'$ is such that the composite state (q', r) is on the current DFS stack.

Conditions **C1** and **C3** remain unaffected because the dependency relation D and the set $vis_f(S)$ of visible operations are irrespective of the state of the Büchi automaton $A_{\neg f}$. Condition **C2** entails inspecting the DFS stack and must be adapted in particular because each system state may yield several composite states that differ in the state of the Büchi automaton $A_{\neg f}$. That is, the on-the-fly construction of $A'_S \times A_{\neg f}$ operates on a different DFS stack and may postpone the closing of cycles compared to the off-line construction of A'_S [Pel96]. The new condition **C2′** appears sufficient to guarantee that the modified version [HPY96] of the nested DFS algorithm in [CV+92], with the calculation of ample sets to determine successor states, detects at least one acceptance cycle on-the-fly in $A'_S \times A_{\neg f}$ if one or more such cycles exist in the "full product" $A_S \times A_{\neg f}$. Thus, model-checking the LTL formula f on-the-fly with POVAS will yield a correct and conclusive verdict: one either finds a counter example to f (i.e. an acceptance cycle), or else f is satisfied by concurrent system S.

Example 7.7

Consider again the concurrent system S and the LTL formula f described in Example 7.3, and depicted in Figure 7.2. Recall that the two sequential processes P_1 and P_2 execute autonomously, meaning that each operation of P_1 is always independent of each operation of P_2. Also recall that the operations w and d are the only visible operations of S. The product automaton $A'_S \times A_{\neg f}$ obtained with POVAS, using the algorithm suggested in [HP95, Pel96] for calculating ample sets, is given in Figure 7.3. It is non-empty, like the "full product" $A_S \times A_{\neg f}$ shown in Figure 7.2, but it contains less states and transitions. For instance, at the initial composite state (10, 20, 0) only operation a of P_1 is executed since $X_1((10, 20)) = \{a\}$ is an ample set in this state, i.e. $\{a\}$ satisfies conditions **C1**, **C2′** and **C3**. This is not the case for $X_1((11, 20)) = \{b, c\}$ in the subsequent

composite state (11, 20, 0), which violates **C2′** since the execution of operation b leads back to the initial state (10, 20, 0) that is on the current DFS stack. However, $X_2((11, 20)) = \{v\}$ is now an ample set in (11, 20, 0). Note further that, amongst others, the state (12, 21, 0) is fully expanded as both w and d are visible operations and thus neither $\{w\}$ nor $\{d\}$ is an ample set in this state. □

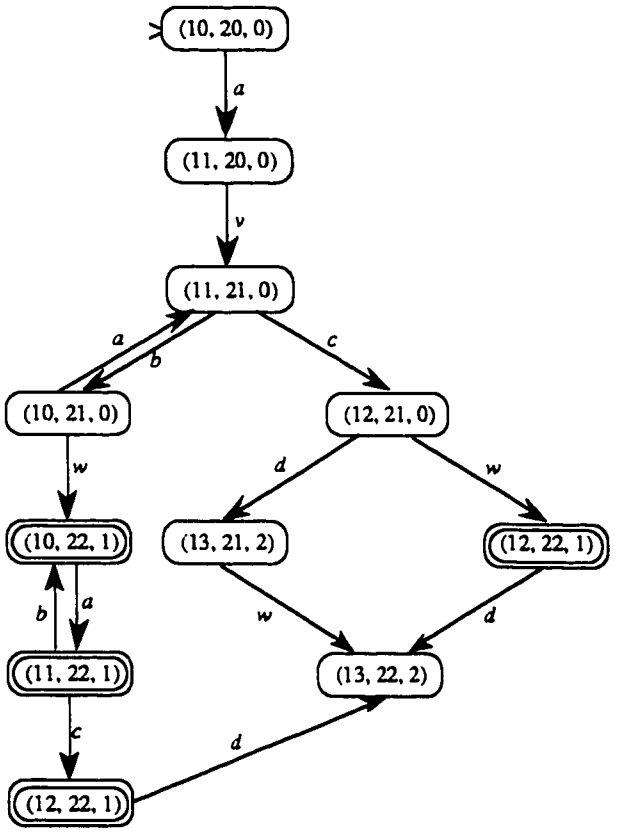


Figure 7.3 The product $A'_S \times A_{\neg f}$ for the concurrent system S and the LTL formula f in Example 7.3.

7.3 Enhancing POVAS

For many concurrent systems, the reduced automaton A'_S resulting from POVAS can be much smaller than the full automaton A_S, as witnessed by the experiments reported in [HP95, Pel96]. Nevertheless, we recognize that even A'_S may still manifest a notable amount of redundancy that can be eliminated. A simplified example explains this. Consider a concurrent system composed of n sequential processes $P_1, P_2, \ldots, P_n$, where each process P_i terminates after executing a single operation, say a_i $(1 \leq i \leq n)$. Also assume that all the operations a_i are mutually independent and invisible, i.e. the processes execute autonomously and no particular temporal property is checked. Obeying conditions **C1**, **C2** and (trivially) **C3** on ample sets, it is not difficult to construct the reduced automaton A'_S for this system. It generates only one of the $n!$ possible orderings of the

a_i's, namely $a_1a_2...a_n$. Yet, even the generation of just this one order appears redundant. Since all the operations a_i are mutually independent and invisible, one can in fact avoid an order altogether by mimicking a truly concurrent execution of these operations, i.e. by executing them collectively. A further reduction of A'_S is then obtained since the n–1 intermediate states reached after executing the operations $a_1a_2...a_{n-1}$ from the initial state are no longer generated. Remark that this follows the same idea as employed earlier in Chapter 5 in the context of LRA for the verification of logical correctness properties of protocols in the CFSM model. Indeed, using concepts analogous to those underlying LRA, we propose an enhancement of POVAS in terms of the space and time (in particular, the number of stored states and explored transitions) needed for LTL model-checking. In the remainder of this section, we set out the enhancement for concurrent systems in general (i.e. modeled as finite LTSs). The realization of POVAS and the proposed enhancement for protocols in the CFSM model is then discussed in Section 7.4, and an experimental comparison based on this realization is provided in Section 7.5.

7.3.1 Proper leap sets

The key to enhancing POVAS lies in a rather simple observation: states of a concurrent system may have multiple *disjoint* ample sets. For instance, a state q has multiple disjoint ample sets particularly when there is more than one sequential process P_i whose set $X_i(q)$ is non-empty and satisfies the conditions **C1**, **C2** and **C3** (which is the case in the above example). Two disjoint ample sets in the same state have the nice characteristic that no operation in either ample set can be dependent of an operation in the other ample set. Furthermore, two disjoint ample sets in the same state cannot contain visible operations. Proposition 7.8 proves these claims.

Proposition 7.8

Let q be a state of a concurrent system S, with $X(q) \neq \emptyset$, and let A_1 and A_2 be sets of operations of S such that $\emptyset \subset A_1, A_2 \subseteq X(q)$ and $A_1 \cap A_2 = \emptyset$. If A_1 and A_2 satisfy condition **C1**, then all operations in A_1 are independent (in q) of all operations in A_2. If A_1 and A_2 satisfy condition **C3**, then all operations in A_1 and A_2 are invisible.

Proof: For the first claim, suppose there exist operations $a_1 \in A_1$ and $a_2 \in A_2$ such that a_1 and a_2 are mutually dependent (in q, in case of a conditional dependency relation). Since A_1 and A_2 are disjoint, this yields the contradiction that A_1 and A_2 do not satisfy condition **C1**: $a_2 \notin A_1$ ($a_1 \notin A_2$) is dependent of an operation in A_1 (A_2) but can be executed at q before an operation from A_1 (A_2) is executed (cf. Definition 7.6). For the second claim, both A_1 and A_2 must be (non-empty) *proper* subsets of $X(q)$ since otherwise they cannot be disjoint. Thus, by condition **C3**, A_1 and A_2 do not contain visible operations. □

```
/* S = (Q,q^0,Σ,Δ) is a concurrent system and q ∈Q is a state of S. */
/* f is an LTL formula to be checked for S.                        */
/* AS is a set of multiple disjoint and non-empty subsets of X(q)  */
/* that satisfy conditions C1 and C3.                              */

AS = ∅
for all processes P_i do
   if X_i(q) ≠ ∅ then {
      C3 = (X_i(q) ∩ vis_f(S) = ∅)
      if C3 then {
         C1 = check_C1(X_i(q))
         /* check_C1(X_i(q)) returns true if X_i(q) satisfies C1,  */
         /* and false otherwise (see Section 7.2.1).               */
         if C1 then add the set X_i(q) to AS
      }
   }
return AS
```

Figure 7.4 Finding multiple disjoint ample sets (wrt **C1** and **C3**).

To exploit the possible existence of multiple pairwise disjoint ample sets in a system state q, or actually the existence of disjoint subsets of $X(q)$ that satisfy **C1** and **C3**, we adopt the algorithm in Figure 7.4 for calculating such sets. This algorithm differs from the one given in [HP95, Pel96] in two ways. First, it enforces only conditions **C1** and **C3** on ample sets. The reason for omitting **C2** will become clear in the next subsection. Secondly, it returns the set of *all* sets $X_i(q)$ satisfying **C1** and **C3**, or the empty set if no such $X_i(q)$ exists. Thus, we calculate only nontrivial ample sets with respect to **C1** and **C3**. Our algorithm is a straightforward adaptation of the algorithm in [HP95, Pel96] for finding just one ample set, and it does not introduce significant extra overhead. This is true especially since **C1** and **C3** are already checked for the most part statically by a prescan of the sequential processes during system compilation, as described in Section 7.2.1. Both algorithms then have a time complexity linear in the number of processes, since the algorithm in [HP95, Pel96] must also scan all processes in the worst case.

Let k be the number of ample sets, with respect to **C1** and **C3**, returned by the algorithm in Figure 7.4 (i.e. k is the cardinality of the returned set AS in Figure 7.4 and $0 \leq k \leq$ the number of sequential processes). When $k > 0$, each such ample set is a subset of $X(q)$ satisfying **C1** and **C3** and thus, by Proposition 7.8, all its operations are invisible and independent (in q) of all operations in the other ample sets. This then motivates that any collection of executable operations forming an element of the Cartesian product of the k ample sets can be executed concurrently at state q. When $k = 0$, no appropriate proper subset of $X(q)$ has been found and all operations executable at q are to be executed separately. That is, the state q is then fully expanded as is the case in [HP95, Pel96]. Formally, we employ the following definition.

Definition 7.9

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system, and for some $q \in Q$ let $ample_1(q)$, $ample_2(q)$,..., $ample_k(q)$ denote the k disjoint subsets of $X(q)$ satisfying conditions **C1** and **C3** that are returned by the algorithm in Figure 7.4. The set $pleap(q)$ of *proper leap sets* in q is defined as follows:

$$pleap(q) = \prod_{j=1}^{k} ample_j(q) \qquad \text{if } k > 0$$

$$pleap(q) = \{\ \{a\} \mid a \in X(q)\ \} \qquad \text{if } k = 0$$ □

When $k = 0$, all operations executable at q are thus considered individually by including them in $pleap(q)$ in the form of singleton sets, like in Chapter 5.

We have designated the term "proper leap set" and the corresponding set denotation $pleap(q)$ to comply with the terminology in Chapter 5. In further analogy with Chapter 5, a permutation of the operations in a proper leap set T is called a *linearization* of T, and the set of all linearizations of T is denoted by $lin(T)$. For a finite or infinite sequence of proper leap sets $\Omega = T_1T_2\ldots$ we have $lin(\Omega) = \{\gamma_1\gamma_2\ldots \mid \gamma_i \in lin(T_i)$ for all $i \geq 1\}$. Since all operations in a proper leap set are mutually independent, it follows from Definition 7.4 that all its linearizations are equivalent and lead to the same state. Hence, we write $q \xrightarrow[\ell]{T} q'$ to mean that there is a set $T \in pleap(q)$ with $\gamma \in lin(T)$ such that $q \xrightarrow{\gamma}{}^{*} q'$, and $q \xrightarrow[\ell]{\Omega}{}^{*} q'$ to mean that the sequence Ω of proper leap sets leads from q to q'.

Akin to a DFS with ample sets implemented by POVAS, one can now perform a DFS that governs the execution of proper leap sets to determine (not necessarily immediate) successor states for each state expanded during the search. The fraction of the state space of a concurrent system $S = (Q_S, q_S^0, \Sigma_S, \Delta_S)$ explored by such a reduced search can again be viewed as an automaton, defined by the tuple $A_S^\ell = (Q_S^\ell, q_S^0, 2^{\Sigma_S}, \Delta_S^\ell, Q_S^\ell)$ with $Q_S^\ell \subseteq Q_S$ and $\Delta_S^\ell \subseteq Q_S^\ell \times 2^{\Sigma_S} \times Q_S^\ell$ such that $(q, T, q') \in \Delta_S^\ell$ iff $T \in pleap(q)$ and $q \xrightarrow[\ell]{T} q'$. This so-called *leap automaton* for S is adequate for verifying indefinite progress, but it does not yet lend itself for LTL model-checking. We will come back to this shortly. Let us first show that exploring A_S^ℓ indeed reveals all non-progress states of S. Theorem 7.10 below[7] proves that for every terminating computation σ of a concurrent system S there is at least one sequence of proper leap sets in A_S^ℓ, starting from the initial state of S, whose linearizations are equivalent (wrt any dependency relation D) to σ. This directly implies that every non-progress state of S is a state in the leap automaton A_S^ℓ.

Theorem 7.10

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system and D a dependency relation for S. For every *finite* computation $\sigma \in \Sigma^*$ of S, there exists a sequence of proper leap sets Ω in A_S^ℓ from the initial state q^0 of S, with $\eta \in lin(\Omega)$, such that $\sigma \equiv_D \eta$.

---

[7] This theorem should be compared with Lemma 5.10 and Theorem 5.11 in Chapter 5.

Proof: Let $q^0 \xrightarrow{\sigma}{}^* q$, then $X(q) = \emptyset$ by Definition 7.1 (i.e. computations of S are defined to be maximal). We first prove that there exists a proper leap set T_1 in the initial state q^0 of S, with $\gamma_1 \in lin(T_1)$, such that $\gamma_1 \preceq_D \sigma$ (see Definition 7.5). If $pleap(q^0) = \{ \{a\} \mid a \in X(q^0) \}$ then T_1 exists trivially, namely T_1 is the singleton set containing the first operation of σ. If $pleap(q^0) = \prod_{j=1}^{k} ample_j(q^0)$, then by condition **C1** and the fact that σ is a terminating computation, it follows that for each process P_i with $X_i(q) = ample_j(q)$ there exists an operation from $ample_j(q)$ in σ (since otherwise $X(q) \neq \emptyset$). Let a_j denote the first operation in σ from $ample_j(q)$, for each $1 \leq j \leq k$. Clearly, $\{a_1, a_2, \ldots, a_k\} \in pleap(q^0)$. By **C1**, each a_j is independent of all operations that occur in σ before the occurrence of a_j itself, and hence the operations $a_1, a_2, \ldots, a_k$ can all be permuted to the front of σ. Consequently, in this case let $T_1 = \{a_1, a_2, \ldots, a_k\}$ with $\gamma_1 \in lin(T_1)$, then for some ρ we have $\sigma \equiv_D \gamma_1\rho$ and thus $\gamma_1 \preceq_D \sigma$. Let then $q^0 \xrightarrow[l]{T_1} q^1 \xrightarrow{\rho}{}^* q$. The proof of the theorem is now straightforward by finite repetition of the above reasoning, continuing with $q^1 \xrightarrow{\rho}{}^* q$ (cf. the proof of Theorem 5.11). □

One should realize that the detection of non-progress states does not require the specification of a temporal formula, meaning that condition **C3** on ample sets is actually void. Indeed, this condition is not used in the proof of Theorem 7.10. It is clear that enforcing just condition **C1** generally aids the calculation of larger proper leap sets, which may result in a further reduction of the number of states and transitions explored.

7.3.2 Off-line LTL model-checking with (proper) leap sets

As just mentioned, a DFS that governs the execution of proper leap sets is not yet fit for LTL model-checking. It should be no surprise that the reason for this is that we have thus far omitted condition **C2** on ample sets. Recall from Section 7.2.1 that this condition is enforced to avoid the ignoring problem, which may cause the execution of operations to be deferred indefinitely along a cycle. Condition **C2** prohibits any operation in $ample(q)$ from closing a cycle on the DFS stack in case $ample(q)$ is a proper subset of $X(q)$. However, incorporating **C2** directly in the formulation of proper leap sets does not solve the ignoring problem for a DFS that governs the execution of these sets, because it employs a different DFS stack (see also Section 7.2.2). We enforce the following condition on proper leap sets instead. Denote by $op(pleap(q))$ the set of all operations in all proper leap sets in q, i.e. $op(pleap(q)) = \{a \in T \mid T \in pleap(q)\}$. If $op(pleap(q))$ is a proper subset of $X(q)$, then for no proper leap set $T \in pleap(q)$ it should hold that the execution of T at q leads to a state that is already on the DFS stack. Otherwise, $pleap(q)$ is extended by adding to it all sets $T \cup \{a\}$, where T ranges over the sets in $pleap(q)$ that do lead to a state on the DFS stack, and a ranges over the operations in $X(q)$ that are not in $op(pleap(q))$.

Definition 7.11

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system, and let $q \in Q$ be the current state to be expanded during the DFS. The set $xpleap(q)$ is defined as follows:

$$xpleap(q) = pleap(q) \cup \{T \cup \{a\} \mid a \in X(q) \backslash op(pleap(q)) \text{ and } T \in pleap(q): q \xrightarrow{\frac{T}{t}} q' \wedge q' \text{ is on the DFS stack } \}$$ □

Observe that each operation (if any) in $X(q)\backslash op(pleap(q))$ belongs to some sequential process P_i whose set $X_i(q)$ is non-empty but does *not* satisfy **C1** or **C3**. Thus, these operations can easily be determined also with the algorithm in Figure 7.4 (only a few straightforward statements need be added to the algorithm), i.e. together with the search for multiple disjoint ample sets that satisfy **C1** and **C3**. It is further important to stress that we extend $pleap(q)$ rather than returning all singleton sets of executable operations, which would reflect the calculation of ample sets in [HP95, Pel96]. In this way the calculation of $pleap(q)$ is not wasted when it turns out that the execution of some proper leap set leads back to a state on the DFS stack. Lastly, observe that for each set $T \in xpleap(q)$ all operations in T are mutually independent and at most one of these operations is visible, by the construction and condition **C1**.

Analogous to the leap automaton A_S^{ℓ} for a concurrent system S, an *extended leap automaton* $A_S^{\ell^e}$ for S can be defined on the basis of the execution of elements from the extended set $xpleap(q)$. Since $xpleap(q) \supseteq pleap(q)$ for any state q, $A_S^{\ell^e}$ strictly extends A_S^{ℓ}. That is, each state of A_S^{ℓ} is a state of $A_S^{\ell^e}$ and each transition of A_S^{ℓ} is a transition of $A_S^{\ell^e}$. This extension solves the ignoring problem as it causes every cycle in $A_S^{\ell^e}$ (when viewed as a graph) to contain at least one state q for which $op(xpleap(q)) = X(q)$, which is not necessarily the case for $pleap(q)$ (cf. Section 5.5 in Chapter 5).

Henceforth, the term "leap set" refers to an element of $xpleap(q)$ (in state q). As before, $lin(\Omega)$ denotes the set of all linearizations of a (finite or infinite) sequence Ω of leap sets. In addition, $q \xrightarrow{\frac{T}{t^e}} q'$ denotes that (any linearization of) the leap set $T \in xpleap(q)$ leads from state q to state q', and likewise $q \xrightarrow{\frac{\Omega}{t^e}}{}^{*} q'$ in case of a sequence of leap sets Ω. Remark that all linearizations of a sequence of leap sets from the initial state of S are computations of S. Hence, like for computations of S, an infinite sequence of leap sets Ω from the initial state of S is said to satisfy an LTL formula f iff the Büchi automaton A_f accepts the computation of $A_S^{\ell^e}$ over Ω (i.e. the sequence of states corresponding to Ω, see Section 7.1.3). Recall thereby once more that finite sequences of leap sets are taken into account by repeating forever the last states generated by these sequences. Since any leap set contains at most one operation from $vis_f(S)$, all visible operations in Ω appear in the same order in each linearization of Ω, while the invisible operations correspond to stuttering steps. Therefore, either all or none of the linearizations of Ω satisfy f and, moreover, Ω itself satisfies f iff all linearizations of Ω satisfy f.

Let $D' = D \cup (vis_f(S) \times vis_f(S))$ be the dependency relation D of a concurrent system S augmented with dependencies between all the visible operations for the checked formula f. This makes *f equivalence robust* [Pel93, Pel96]: all sequences equivalent wrt D' contain the same visible operations and in the same order, and thus f has the same truth value for each of them (i.e. either all or none of these sequences satisfy f). Surely, all linearizations of a sequence of leap sets are equivalent wrt D'. The dependency relation D' is used specifically to show that a DFS governing the execution of leap sets retains the order of visible operations in the computations of a concurrent system (Lemma 7.13 and Lemma 7.14 below). This in turn is the key to proving the main result of the chapter: LTL model-checking can be conducted faithfully with the extended leap automaton $A_S^{f^*}$ for S (Theorem 7.15 below). The associated proofs are somewhat similar to the proofs of the corresponding claims in [Pel96]. For convenience, the next definition is therefore taken directly from [Pel96].

Definition 7.12

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system and D a dependency relation for S. For a (finite or infinite) sequence $\sigma \in \Sigma^* \cup \Sigma^\omega$ of operations of S, denote by $op(\sigma)$ the set of operations occurring in σ, by $\sigma(i)$ the i-th operation in σ, and by $\sigma(i{+}1\ldots)$ all but the first i operations in σ. A *selection function* for σ is a function $c : \{1,\ldots,|\sigma|\} \mapsto \{true, false\}$, mapping each operation in σ to either *true* or *false*. Denote by σ_c ($\sigma_{\overline{c}}$) the sequence remaining from σ after the removal of all operations $\sigma(i)$ with $c(i) = false$ ($c(i) = true$). Also, denote by $c\ll r$ the selection function c shifted to the left r places, i.e. $(c\ll r)(i) = c(i + r)$. Define $\sigma \preceq_D^A \sigma'$ iff there exists a selection function c for σ' such that (i) $\sigma \equiv_D \sigma'_c$, (ii) $op(\sigma'_{\overline{c}}) \subseteq A \subseteq \Sigma$, and (iii) for all $1 \le i \le |\sigma'|$, if $c(i) = false$ then each operation in $op(\sigma'(i{+}1\ldots)_{c\ll i})$ is independent wrt D of $\sigma'(i)$. □

Informally, $\sigma \preceq_D^A \sigma'$ if a sequence equivalent (wrt D) to σ can be obtained from σ' by removing from σ' some operations in A that are independent of all the non-removed operations of σ' that appear after them [Pel96]. For example, let $\Sigma = \{a, b, v, w\}$, $D = \{(a, a), (b, b), (v, v), (w, w)\}$, $\sigma = abvw(ab)^\omega$, $\sigma' = (bwav)^\omega$ and $A = \{v, w\}$, then $\sigma \preceq_D^A \sigma'$. To see this, choose a selection function c for σ' such that $c(i) = true$ if i is odd or less than 5; $c(i) = false$ otherwise. One obtains $\sigma'_c = bwav(ba)^\omega$, i.e. all occurrences of v and w in σ' except the first ones are removed. The equivalence $\sigma \equiv_D \sigma'_c$ is then immediate. Notice that every removed operation v or w is independent of all the operations occurring to its right in σ' that are not removed (these are the occurrences of a and b except the first a and b).

Lemma 7.13

For a concurrent system $S = (Q, q^0, \Sigma, \Delta)$, let $q \in Q$ be a state that is removed from the DFS stack during the construction of the extended leap automaton $A_S^{f^*}$ for S, and let $\mu a \sigma \in \Sigma^* \cup \Sigma^\omega$ be a

computation of S, with $q^0 \xrightarrow{\mu}{}^* q$. Then, there exist a sequence $q \xrightarrow[\ell^*]{T_1} q_1 \xrightarrow[\ell^*]{T_2} \ldots \xrightarrow[\ell^*]{T_{m-1}} q_{m-1} \xrightarrow[\ell^*]{T_m} q'$ ($m \geq 1$) in $A_S^{\ell^*}$, and a selection function c for $\gamma \in lin(T_1T_2\ldots T_{m-1}(T_m \backslash \{a\}))$, such that

i) $a \in T_m$,

ii) no operation in $T_1, T_2, \ldots, T_{m-1}, T_m \backslash \{a\}$ is visible,

iii) no operation in $T_1, T_2, \ldots, T_{m-1}, T_m \backslash \{a\}$ is dependent wrt D' of a,

iv) $\gamma a \equiv_{D'} a\gamma_c\gamma_{\bar{c}}$,

v) $\exists \rho$: $\gamma_c \rho \equiv_{D'} \sigma$ (i.e. $\gamma_c \preceq_{D'} \sigma$), and

vi) the operations in $\gamma_{\bar{c}}$ are independent wrt D' of the operations in ρ.

Proof: If $xpleap(q) = \{ \{b\} \mid b \in X(q) \}$, then a sequence with the required properties exists trivially, namely $q \xrightarrow[\ell^*]{\{a\}} q'$. Alternatively, if $xpleap(q) \supseteq pleap(q) = \prod_{j=1}^{k} ample_j(q)$, then for each $ample_j(q)$, no operation in $ample_j(q)$ is visible (by **C3**), and no operation outside $ample_j(q)$ that is dependent wrt D of an operation in $ample_j(q)$ can occur in $a\sigma$ before the occurrence in $a\sigma$ of some operation in $ample_j(q)$ itself (by **C1**). The holds then true also wrt $D' = D \cup (vis_f(S) \times vis_f(S))$. It follows that for each process P_i such that $X_i(q) = ample_j(q)$, either $ample_j(q)$ contains the *first* operation of P_i in $a\sigma$, or all operations in $a\sigma$ are independent wrt D' of all operations in $ample_j(q)$ (i.e. P_i has no operations occurring in $a\sigma$). Note that the latter cannot be the case if σ is finite (which was in fact the key to proving Theorem 7.10). Thus, there exists $T = \{a_1, a_2, \ldots, a_k\} \in pleap(q) \subseteq xpleap(q)$ such that a_j is the first operation from $ample_j(q)$ occurring in $a\sigma$ if there is such operation, or else a_j is any operation from $ample_j(q)$, for each $1 \leq j \leq k$, and all the a_j's are invisible and mutually independent wrt D'. The proof now continues by induction on the order in which states are removed from the DFS stack during the construction of the extended leap automaton $A_S^{\ell^*}$ for S. When removing the state q from the DFS stack, two cases can be distinguished:

- $a \in T$ or T leads to a state on the DFS stack

  Remark that this case covers in particular the induction basis where q is the first state removed from the DFS stack: each proper leap set in $pleap(q)$ executed in q leads to a state that is already on the DFS stack since any other state would have been removed before q (a characteristic of a depth-first search). Let $T_1 = T \cup \{a\}$ if $a \notin T$ (i.e. $a \in X(q) \backslash op(pleap(q))$; $T_1 = T$ otherwise. By construction, $a \in T_1$ and $T_1 \in xpleap(q)$, and all operations in $T_1 \backslash \{a\}$ are invisible and independent wrt D' of a, viz. $q \xrightarrow[\ell^*]{T_1} q'$ is a sequence satisfying properties (i), (ii) and (iii). For any sequence of operations γ, define the required selection function c such that $c(i) = true$ if $\gamma(i) \in op(\sigma)$; $c(i) = false$ otherwise. Here, $\gamma \in lin(T_1 \backslash \{a\})$ and properties (iv), (v) and (vi) follow readily, again by construction of T_1.

- $a \notin T$ and T does not lead to a state on the DFS stack

  Let $T = T_1$ and $q \xrightarrow{T_1}_{t^*} q_1$. Thus, q_1 is not on the DFS stack and when added it will be removed before q itself is removed (a characteristic of a depth-first search). But this means that the induction hypothesis can be applied to q_1, viz. there exists a sequence of leap sets $T_2 \ldots T_{m-1} T_m$ from q_1 with respective selection function c for $\gamma' \in lin(T_2 \ldots T_{m-1}(T_m \backslash \{a\}))$ (defined as above) satisfying properties (i) to (vi). Now, since $a \notin T_1$ and each operation in T_1 is invisible and independent wrt D' of all operations in $a\sigma$ before its own occurrence (if any) in σ (because $T_1 \in \prod_{j=1}^{k} ample_j(q)$), it is immediate that $T_1 T_2 \ldots T_{m-1} T_m$ is a sequence of leap sets from q which satisfies properties (i), (ii) and (iii). In order to prove property (iv), let $\tau \in lin(T_1)$ and $\gamma \in lin(T_1 T_2 \ldots T_{m-1}(T_m \backslash \{a\})) = \tau\gamma'$. We derive $\gamma a = \tau\gamma' a \equiv_{D'} \tau a \gamma'_c \gamma'_{\bar{c}} \equiv_{D'} \tau_c \tau_{\bar{c}} a \gamma'_c \gamma'_{\bar{c}}$, using the induction hypothesis and the fact that the operations in a leap set are mutually independent wrt D'. Since all the operations in T_1 are independent wrt D' of a and all the operations in $\tau_{\bar{c}}$ are independent wrt D' of all operations in σ and hence in γ'_c, it follows that $\tau_c \tau_{\bar{c}} a \gamma'_c \gamma'_{\bar{c}} \equiv_{D'} a \tau_c \gamma'_c \tau_{\bar{c}} \gamma'_{\bar{c}} = a \gamma_c \gamma_{\bar{c}}$. Properties (v) and (vi) are proved similarly via the induction hypothesis. □

Lemma 7.13 implies that for each operation a that becomes executable along some computation of a concurrent system S, the extended leap automaton $A_S^{\xi^*}$ for S also contains a sequence along which a becomes executable. Thus, aside from our objective, $A_S^{\xi^*}$ serves to detect all (non-)executable operations of S. Condition **C3** on ample sets can thereby be void, as was the case for detecting non-progress states, which similarly favors the size of the extended leap automaton.

Lemma 7.14

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system and f an LTL formula to be checked for S. For every computation $\sigma \in \Sigma^* \cup \Sigma^\omega$ of S there exists a sequence of leap sets Ω in $A_S^{\xi^*}$ from the initial state q^0 of S, with $\eta \in lin(\Omega)$, such that $\sigma \preceq_{D'}^{\Sigma \backslash vis_f(S)} \eta$.

Proof: For any (finite or infinite) computation σ of S , while reading σ, we describe a traversal of $A_S^{\xi^*}$ starting from the initial state q^0 that yields a sequence Ω of leap sets, with $\eta \in lin(\Omega)$, such that $\sigma \preceq_{D'}^{\Sigma \backslash vis_f(S)} \eta$. The following variables are used in the process:

- σ' the sequence of operations read so far from σ;
- Ω' the sequence of leap sets in $A_S^{\xi^*}$ traversed so far, with $\eta' \in lin(\Omega')$;
- ρ a linearization of Ω', projected on the set of operations that have not yet been read from σ (i.e. removed from $\eta' \in lin(\Omega')$ are the operations already read from σ);
- q the current state of $A_S^{\xi^*}$.

The variables σ', Ω' and ρ are initialized to the empty sequence, and q is initialized to q^0. Whenever the next operation $a \in \Sigma$ is read from σ, the following updates are made:

1. $\sigma' := \sigma' a$;
2. **if** $\rho = \upsilon a \upsilon'$, for some $\upsilon, \upsilon' \in \Sigma^*$ such that all operations in υ are independent wrt D' of a, **then** $\rho := \upsilon\upsilon'$;
3. **else** choose a sequence of leap sets $T_1 T_2 \ldots T_{m-1} T_m$ from the state q, leading to a state q', such that $a \in T_m$ and $\eta'_c \gamma_c a \equiv_{D'} \eta'_c a \gamma_c \preceq_{D'} \sigma$, with $\gamma \in lin(T_1 T_2 \ldots T_{m-1}(T_m \backslash \{a\}))$ and the selection function c , for any sequence of operations μ, such that $c(i)$ = true if $\mu(i) \in op(\sigma)$; $c(i) = false$ otherwise. Make the following updates:

   $\Omega' := \Omega' T_1 T_2 \ldots T_{m-1} T_m$;

   $\rho := \rho \gamma_c$;

   $q := q'$.

The following properties are now inductively proved to be invariant while reading σ:

i) $\sigma' \rho \equiv_{D'} \eta'_c$,

ii) $\eta'_c \preceq_{D'} \sigma$,

iii) if the condition of Step 2 does not hold when checking it, then all the operations occurring in ρ are independent wrt D' of a, and

iv) the choice of the sequence $T_1 T_2 \ldots T_{m-1} T_m$ required by Step 3 can always be made when taking Step 3.

Initially, the properties (i) to (iv) trivially hold since σ', Ω' and ρ are empty, and the algorithm is just before Step 1. At each step of the update procedure, $\sigma' a \preceq_{D'} \sigma$ since a is the next operation from σ read after σ', and $\sigma' \rho \preceq_{D'} \sigma$ by the induction hypotheses (i) and (ii). This together implies that ρ cannot contain operations that are dependent wrt D' of a before the occurrence of a (if any) in ρ, which proves (iii). When a does not occur in ρ and thus not in any leap set in Ω', Step 3 is taken and the existence of the sequence of leap sets $T_1 T_2 \ldots T_{m-1} T_m$ from state q required by Step 3 is guaranteed by Lemma 7.13, proving (iv). It is then easy to check that both (i) and (ii) are preserved by taking either Step 2 or Step 3 of the update procedure.

Let Ω denote the entire (finite or infinite) sequence of leap sets collected into Ω' along a traversal of $A_S^{f^*}$ upon reading σ, with $\eta \in lin(\Omega)$. From (i), for each $\sigma' \in pref(\sigma)$, $\sigma' \preceq_{D'} \eta'_c$ and hence $\sigma' \preceq_{D'} \eta_c$. Also, from (ii), for each $\gamma \in pref(\eta_c)$ we have $\gamma \preceq_{D'} \sigma$. Thus, $\sigma \equiv_{D'} \eta_c$ by Definition 7.5, and $\sigma \preceq_{D'}^{\Sigma \backslash vis_f(S)} \eta$ by definition of '$\preceq_{D'}^{\Sigma \backslash vis_f(S)}$' (in particular because all operations in $\eta_{\bar{c}}$ are invisible and independent wrt D' of the operations in σ). □

Theorem 7.15

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system and f an LTL formula to be checked for S. For every computation $\sigma \in \Sigma^* \cup \Sigma^\omega$ of S there exists a sequence of leap sets Ω in $A_S^{f^*}$ from the initial state q^0 of S, such that σ satisfies f iff Ω satisfies f.

Proof: By Lemma 7.14, there exists a sequence of leap sets Ω in the extended leap automaton $A_S^{\ell^*}$ for S, with $\eta \in lin(\Omega)$, such that $\sigma \preceq_{D'}^{\Sigma \setminus vis_f(S)} \eta$. To prove that σ satisfies f iff Ω satisfies f, it is sufficient to show that σ satisfies f iff η satisfies f. This follows from the fact that Ω captures the same visible operations and in the same order as they occur in η (any leap set contains at most one visible operation). That is, a computation of the extended leap automaton $A_S^{\ell^*}$ over Ω is stuttering equivalent wrt f to a computation of the full automaton A_S over η, and thus f cannot distinguish between these two computations (recall that f is implicitly assumed to be a nexttime-free LTL formula, i.e. f is stuttering closed). It remains to be shown that σ and η yield stuttering equivalent sequences. Since $\sigma \preceq_{D'}^{\Sigma \setminus vis_f(S)} \eta$, this is immediate by the definition of '$\preceq_{D'}^{\Sigma \setminus vis_f(S)}$' and the inclusion $vis_f(S) \times vis_f(S) \subseteq D'$. □

In conclusion, Theorem 7.15 warrants the application of off-line LTL model-checking algorithms [LP85] to the extended leap automaton of a concurrent system, as opposed to its full automaton or the reduced automaton resulting from POVAS.

7.3.3 On-the-fly LTL model-checking with (proper) leap sets

We now turn to the aptness of the extended leap automaton $A_S^{\ell^*}$ for *on-the-fly* LTL model-checking. Recalling the preliminaries in Section 7.1.3, when the Büchi automaton $A_{\neg f}$ for the negation of the checked LTL formula f is defined over state predicates, each transition of the product automaton $A_S^{\ell^*} \times A_{\neg f}$ is of the form $(q, r) \xrightarrow{(T,P)} (q', r')$, with $q \xrightarrow{T}_{\ell^*} q'$ a transition of $A_S^{\ell^*}$ and $r \xrightarrow{P} r'$ a transition of $A_{\neg f}$ such that proposition **P** is true in system state q. Alternatively, when $A_{\neg f}$ is defined over the alphabet of operations of the concurrent system S, each transition of $A_S^{\ell^*} \times A_{\neg f}$ is of the form $(q, r) \xrightarrow{T} (q', r')$, with $q \xrightarrow{T}_{\ell^*} q'$ a transition of $A_S^{\ell^*}$, $\gamma \in lin(T)$ and $r \xrightarrow{\gamma}{}^{*} r'$ a sequence in $A_{\neg f}$. Since there is at most one visible operation in any leap set, the product automaton is well-defined in either case. For the latter case in particular, at most one operation in T can actually cause a state transformation of $A_{\neg f}$, and r' follows therefore uniquely from r and (any linearization of) T. The next theorem proves that it is sufficient to check the emptiness of $A_S^{\ell^*} \times A_{\neg f}$ in order to verify S against the formula f.

Theorem 7.16

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system and f an LTL formula to be checked for S. The product automaton $A_S \times A_{\neg f}$ is empty iff the product automaton $A_S^{\ell^*} \times A_{\neg f}$ is empty.

Proof: By Theorem 7.15, for every computation of S there exists a sequence of leap sets in the extended leap automaton $A_S^{\ell^*}$ for S, such that either both sequences satisfy f or both do not satisfy

f. Thus, $A_S \times A_{\neg f}$ is non-empty iff there exists some computation σ of S satisfying $\neg f$ iff there exists some sequence of leap sets Ω in $A_S^{\ell\ell}$ satisfying $\neg f$ iff $A_S^{\ell\ell} \times A_{\neg f}$ is non-empty. □

As for the on-the-fly construction of $A_S' \times A_{\neg f}$ with POVAS (see Section 7.2.2), the on-the-fly construction of the product automaton $A_S^{\ell\ell'} \times A_{\neg f}$ can likewise postpone the closing of cycles on the DFS stack because it operates on a DFS stack (made up of composite states) that is different from the DFS stack used by the off-line construction of $A_S^{\ell\ell'}$ (made up of single states of S). Since this affects solely condition **C2** and since the definition of *pleap*(q) does not rely on **C2** (the ample sets used to construct *pleap*(q) obey only **C1** and **C3**), we need to modify only the definition of *xpleap*(q) in order to make the execution of leap sets compatible with the algorithms for on-the-fly LTL model-checking. Similar to the modified condition **C2'** on ample sets, composite states of the product automaton $A_S^{\ell\ell'} \times A_{\neg f}$ are accounted for in leap sets as follows.

Definition 7.17

Let $S = (Q, q^0, \Sigma, \Delta)$ be a concurrent system, f an LTL formula to be checked for S, and (q, r) the current composite state of $A_S^{\ell\ell'} \times A_{\neg f}$ to be expanded during the DFS. Define

$$\begin{aligned} \mathit{xpleap}(q, r) \;=\; & \mathit{pleap}(q) \cup \{T \cup \{a\} \mid a \in X(q) \setminus \mathit{op}(\mathit{pleap}(q)) \text{ and} \\ & T \in \mathit{pleap}(q) \colon q \xrightarrow[t]{T} q' \wedge (q', r) \text{ is on the DFS stack } \} \end{aligned}$$ □

The adaptation of the algorithm for on-the-fly detection of acceptance cycles in the context of POVAS [HP95, Pel96, HPY96] now simply consists in using *xpleap*(q, r) instead of *ample*(q, r) to determine the subset of (not necessarily immediate) successor states of q that need be explored next. Suffice it to say that this adaptation is indeed adequate for effectively combining the execution of leap sets with on-the-fly LTL model-checking. That is, the adapted algorithm returns true if the given concurrent system does not satisfy the checked LTL formula, and false otherwise. The correctness proof is virtually identical to the one given in [Pel96] for the on-the-fly construction algorithm of POVAS, considering also the modification suggested in [HPY96] (see Section 7.2.2) of the nested DFS algorithm in [CV+92] (see Section 7.1.3). It entails a reduction from the on-the-fly algorithm to a non-deterministic variant of the off-line algorithm, which preserves the validity of Lemma 7.13 and hence of Lemma 7.14 and Theorem 7.15. We refer the interested reader to [CV+92, Pel96, HPY96] for precise details.

It is appropriate here to point out that an alternative definition can be given for *xpleap*(q, r), and likewise for *xpleap*(q), that does not require inspection of the DFS stack. By always adding the sets $T \cup \{a\}$ to *pleap*(q) for each $a \in X(q) \setminus \mathit{op}(\mathit{pleap}(q))$ *and for each* $T \in \mathit{pleap}(q)$, i.e. even if T does not lead to a state on the DFS stack, all the previous results still hold (in particular, the sequence of leap sets sought in Lemma 7.13 is then guaranteed to be of length one). Although this

generally increases the size of the extended leap automaton $A_S^{\xi'}$ for a concurrent system S, and of the corresponding product automaton $A_S^{\xi'} \times A_{\neg f}$, it may reduce the time for constructing these automata, especially when it turns out that at many expansion steps during state exploration the number of *proper* leap sets (i.e. the leap sets in $pleap(x)$) and the current DFS stack are large. In addition, for on-the-fly cycle detection one can directly use the nested DFS algorithm of [CV+92] without any modification.

Example 7.18

Consider once more the concurrent system S and the LTL formula f described in Example 7.3, and depicted in Figure 7.2. The product automaton $A_S^{\xi'} \times A_{\neg f}$ is shown in Figure 7.5. It is again non-empty, like the "full product" $A_S \times A_{\neg f}$ in Figure 7.2, and smaller than the product $A_S' \times A_{\neg f}$ in Figure 7.3 obtained with POVAS. For instance, at the initial state (10, 20, 0) both $X_1((10, 20)) = \{a\}$ and $X_2((10, 20)) = \{v\}$ satisfy conditions **C1** and **C3**, yielding $pleap((10, 20, 0)) = \{\{a, v\}\}$. Furthermore, since this proper leap set does not lead to a composite state already on the DFS stack, $xpleap((10, 20, 0)) = pleap((10, 20, 0))$. Notice that at the composite state (10, 21, 0) the subset $X_1((10, 21)) = \{a\}$ satisfies **C1** and **C3**, but $X_2((10, 21)) = \{w\}$ does not because operation w is visible. Thus, $pleap((10, 21, 0)) = \{\{a\}\}$. This proper leap set leads back to a state on the DFS stack, however, and hence $xpleap((10, 21, 0)) = \{\{a\}, \{a, w\}\}$. For this small but illustrative example, the extra reduction achieved amounts to just one state and one transition. Later we will see that the overall gain over POVAS is generally more significant for "larger" concurrent systems. □

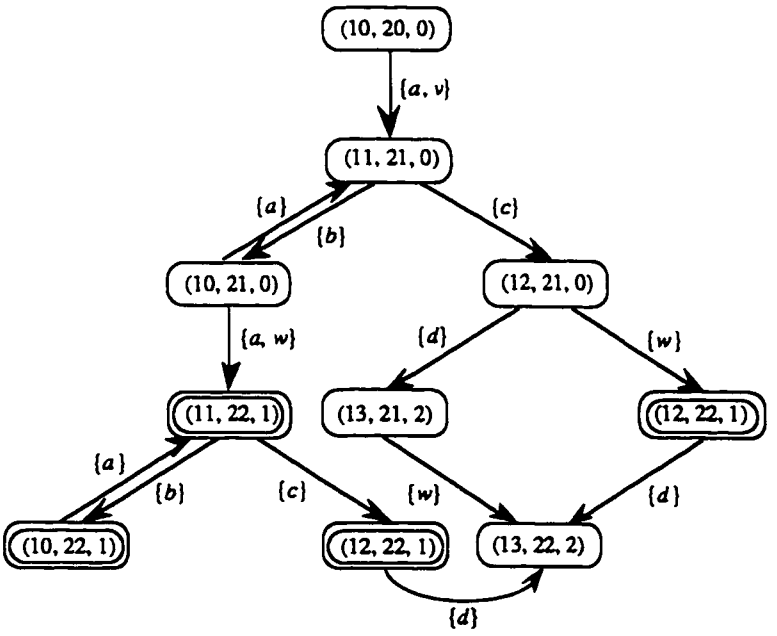


Figure 7.5 The product $A_S^{\xi'} \times A_{\neg f}$ for the concurrent system S and the LTL formula f in Example 7.3.

7.3.4 LTL model-checking under fairness assumptions

The presentation of POVAS and its proposed enhancement so far has been confined to LTL model-checking *without fairness*. As explained at the end of the preliminary subsection 7.1.2, when the interleaving semantics of a concurrent system involves fairness, all computations of the system that violate the assumed fairness assumptions are no longer considered. Since fairness assumptions can also be expressed in nexttime-free LTL [LP85], LTL model-checking under fairness assumptions can be done simply by checking formulas of the form $f_1 \Rightarrow f_2$, where f_1 formalizes a conjunction of fairness assumptions and f_2 a desirable property. Unfortunately, adding f_1 as part of the formula often introduces many additional dependencies among operations [GW94, Pel96], since f_1 causes more (usually all) operations to be visible and condition **C3** on ample sets must be applied also to f_1. A DFS based on the execution of ample sets or leap sets will then yield little or no gain at all.

In order to exploit a restricted class of fairness assumptions more efficiently and, in general, to introduce dependencies among visible operations more carefully, it was shown in [Pel93] that a temporal formula f which is not equivalence robust (see Section 7.3.2) can sometimes be made equivalence robust by rewriting f as a Boolean combination of sub-formulas f_i and treating each f_i individually when adding dependencies among visible operations. This is based on the simple fact that when two formulas f_1 and f_2 are equivalence robust, then so are $f_1 \wedge f_2$, $f_1 \vee f_2$ and $\neg f_1$. Precisely, instead of augmenting the dependency relation D of a concurrent system S with *all* pairs of visible operations, which is the effect of imposing condition **C3** on ample sets and thereby on (proper) leap sets, it appears sufficient to augment D with the pairs in $\bigcup_i (vis_{f_i}(S) \times vis_{f_i}(S))$. This union is a subset of $vis_f(S) \times vis_f(S)$ and can yield much fewer dependencies in several cases. For example, if $f = \Diamond(P \vee Q)$ then $vis_f(S)$ includes all the operations of S whose execution can change the truth value of the Boolean propositions P or Q. However, this formula is logically equivalent to $f_1 \vee f_2 = \Diamond P \vee \Diamond Q$, where $vis_{f_1}(S)$ includes the operations that can change P and $vis_{f_2}(S)$ includes those that can change Q. Thus, any two operations such that one can change only P but not Q, and the other can change Q but not P, are dependent wrt to $D \cup vis_f(S) \times vis_f(S)$ but not necessarily wrt $D \cup \bigcup_i (vis_{f_i}(S) \times vis_{f_i}(S))$. Other logical equivalences among temporal formulas that can be used profitably as rewriting rules are: $\Box(f_1 \wedge f_2) \equiv \Box f_1 \wedge \Box f_2$, $\Box\Diamond(f_1 \vee f_2) \equiv \Box\Diamond f_1 \vee \Box\Diamond f_2$, $\Diamond\Box(f_1 \wedge f_2) \equiv \Diamond\Box f_1 \wedge \Diamond\Box f_2$, as well as $(f_1 \wedge f_2)\, U\, f_3 \equiv (f_1\, U\, f_3) \wedge (f_2\, U\, f_3)$ and $f_3\, U\, (f_1 \vee f_2) \equiv (f_3\, U\, f_1) \vee (f_3\, U\, f_2)$. Although rewriting can increase the length of a formula exponentially, it is argued in [Pel93, Pel96] that the checked formulas are generally quite short and, moreover, that they need not be rewritten completely. That is, the rewriting rules can be used to separate Boolean components of a formula one at a time without explicitly generating the rewritten formula. This is done with a recursive algorithm in time linear in the length of the formula [Pel93]. Following this algorithm, the original formula can still be used for actual model-checking.

In summary, POVAS employs the dependency relation $D \cup \bigcup_i (vis_{f_i}(S) \times vis_{f_i}(S))$ instead of $D \cup vis_f(S) \times vis_f(S)$ (the latter implicitly through condition **C3**) for LTL model-checking with a certain class of fairness assumptions [Pel96], including such assumptions as weak fairness, process fairness and process justice [Fra86, MP92]. This is accomplished by dropping condition **C3** and enforcing condition **C1** with respect to $D \cup \bigcup_i (vis_{f_i}(S) \times vis_{f_i}(S))$ as opposed to just D. In effect, the fairness assumptions then act as "low-cost" filters on the computations of a concurrent system, allowing the calculation of ample sets with respect to a subset of these computations. This may decrease the size of ample sets and thus result in the exploration of a yet smaller number of states and transitions. It is evident that the same advantage applies also to the proposed enhancement of POVAS. Indeed, we can equally use $D \cup \bigcup_i (vis_{f_i}(S) \times vis_{f_i}(S))$ in place of $D \cup vis_f(S) \times vis_f(S)$, and construct leap sets as before from ample sets that respect this refined dependency relation.

A final note concerns yet another, very recent improvement of POVAS for on-the-fly LTL model-checking, where the number of visible operations with respect to a formula f may diminish *during* model-checking [KPV97]. Roughly speaking, it is shown that the set $vis_f(S)$ itself can in many cases be reduced dynamically, yielding even fewer dependencies, by exploiting information of the current state of the Büchi automaton $A_{\neg f}$. This is translated into a visibility condition that relaxes condition **C3**. We refer to [KPV97] for details. Because this improvement operates "only" at the level of (the Büchi automaton $A_{\neg f}$ for) the formula f, it can also be applied directly to our enhancement of POVAS.

7.4 LTL model-checking in the CFSM model

Having addressed the topic of LTL model-checking in general for finite-state concurrent systems formalized as LTSs, for the remainder of the chapter we turn the focus back to protocols defined as networks of CFSMs. As noted earlier, every bounded protocol Π in the CFSM model qualifies as a finite-state concurrent system: its behavior is defined by the LTS $(\mathbf{R}_\Pi, G^0, \bigcup_{i \in I} \Delta_i, \{(G, t, H) \mid G \in \mathbf{R}_\Pi \wedge G \xrightarrow{t} H\})$. POVAS and its proposed enhancement based on leap sets are thus suited for LTL model-checking in the CFSM model. We now show how to realize these two relief strategies specifically for the CFSM model, by harmonizing their formulation with the formulation of LRA in Chapter 5. This facilitates the integration of POVAS and its enhancement in the research tool package RELIEF discussed in Chapter 6, which can then be used to perform an experimental comparison of the performance of both techniques.

POVAS and its proposed enhancement make use of a dependency relation among transitions (or operations) to tackle the wasteful exploration of many equivalent interleavings of concurrent transitions. As discussed, it is thereby important that it can be easily checked in practice whether two transitions are (in)dependent. This is certainly the case for protocols in the CFSM model: a

syntactic condition that is sufficient for two transitions t and t' to be independent is that they are not from the same process and they do not involve the same simplex channel. It is not difficult to see that the dependency relation induced by this condition is a valid one according to Definition 7.4. Nevertheless, for the CFSM model we can readily establish a weaker condition by considering a conditional dependence among transitions, as in Definition 7.4bis. For each individual global state G, two transitions are independent *in* G if they are not from the same process and if neither one of them is *enabled* at G by the other. Proposition 7.19 proves that this condition is sufficient to meet the requirements listed in Definition 7.4bis. Recall from Section 4.1 that a transition t defined at G is enabled at G by a transition $t' \in X(G)$ if t is potentially executable at G and the execution of t' at G causes t to become executable immediately thereafter. A send (receive) transition is potentially executable at G if it involves a channel that is full (empty) in G.

Proposition 7.19

Let G be a global state of a protocol Π. Two transitions t and t' are independent in G if it holds that (i) $act(t) \neq act(t')$, and (ii) t is not enabled at G by t' and t' is not enabled at G by t.

Proof: Since transitions t and t' are not from the same sequential process (i.e. $act(t) \neq act(t')$), and since neither transition is enabled at G by the other transition, it follows immediately that the two requirements in Definition 7.4bis are satisfied. That is, if t (t') is executable at G, leading to some global state H, then t' (t) is executable at G iff it is executable at H, and if both t and t' are executable at G, then executing the sequence tt' from G yields the same global state as executing the sequence $t't$ from G. Transitions t and t' are thus independent in G. □

The above translated requirement on a conditional dependency relation for protocols in the CFSM model renders in turn a translation of condition **C1** on ample sets (see Definition 7.6) for these protocols, as stated by the next proposition.

Proposition 7.20

Let G be a global state of a protocol Π, and let $A \subseteq X(G)$ be a (non-empty) subset of transitions executable at G. A satisfies condition **C1** if for each $i \in act(A)$ it holds that:

i) $X_i(G) \subseteq A$, and

ii) if $A \subset X(G)$, then $P_i(G) = \emptyset$.

Proof: By Definition 7.6, condition **C1** stipulates that for each non-empty sequence $\sigma = G^1 \xrightarrow{t_1} G^2 \xrightarrow{t_2} \ldots G^m \xrightarrow{t_m} G^{m+1}$ from $G^1 = G$, with $t_j \in \bigcup_{i \in I} \Delta_i \setminus A$ for all $1 \leq j \leq m$, each transition t_j is independent in G^j of all transitions in A. This holds trivially if $A = X(G)$, satisfying properties (i) and (ii), since in that case no such non-empty sequence σ from G exists. Alternatively, if $A \subset X(G)$

then by properties (i) and (ii), $X_i(G) \subseteq A$ and $P_i(G) = \emptyset$ for all $i \in act(A)$. We prove that this implies that for all $1 \leq j \leq m$: (1) $act(t_j) \cap act(A) = \emptyset$, and (2) if $i \in act(A)$, then $P_i(G^j) = \emptyset$.

To show (1), suppose that $act(t_j) = \{i\} \subseteq act(A)$ for some j, then $X_i(G) \subseteq A$ and $P_i(G) = \emptyset$. Hence, $t_j \notin X_i(G) \cup P_i(G)$. However, since no transition in A, and thus no transition in $X_i(G)$, is executed along σ, it follows that t_j cannot become executable at G^j – a contradiction. To show (2), suppose that $i \in act(A)$, then $X_i(G) \subseteq A$ and $P_i(G) = \emptyset$. From the proof of (1), no transition of process P_i is executed along σ, implying that $P_i(G^j) = \emptyset$ for all $1 \leq j \leq m$.

In conclusion, from (1) it follows that for each transition $t \in A$ we have $act(t_j) \neq act(t)$, while from (2) it follows that for each $i \in act(A)$ we have $E_i(G^j) \subseteq P_i(G^j) = \emptyset$ (i.e. no process with transitions in A has enabled transitions at G^j). Thus, by Proposition 7.19, transition $t_j \in X(G^j)$ is independent in G^j of all transitions in A. As this holds for all $1 \leq j \leq m$, A satisfies condition **C1**.□

The first requirement in Proposition 7.20 stipulates that the subset A of $X(G)$ contains for each process either all or none of this process' executable transitions at G, which is in fact necessary for condition **C1** to hold. To see this, suppose that $X_i(G) \not\subseteq A$ for some $i \in act(A)$, then process P_i has two transitions $t, t' \in X_i(G)$ where $t \in A$ and $t' \notin A$. These transitions are dependent in G since they are from the same process. But $G \xrightarrow{t'} H$ is then a non-empty sequence from G of transitions outside A containing a transition that is dependent with a transition in A, and thus **C1** is violated. The second requirement in Proposition 7.20 prohibits every process with transitions in A from having potentially executable transitions at G if A is a proper subset of $X(G)$. To illustrate the importance of this requirement for condition **C1**, suppose that $P_i(G) \neq \emptyset$ for some $i \in act(A)$, then process P_i has two transitions $t \in X_i(G)$ and $t' \in P_i(G)$ such that $t \in A$, $t' \notin A$, and t and t' are dependent in G. If $A \subset X(G)$, then since t' is potentially executable at G it *may* be possible that t' becomes executable, and is executed, along a sequence from G of only transitions outside A. Again, in that case **C1** would be violated. This scenario can actually be drawn for the initial state G^0 of the simple protocol depicted in Figure 5.1, by letting $A = X_2(G^0) = \{(20, -b, 21)\}$. We have $X_2(G^0) \subset X(G^0)$ and $P_2(G^0) = \{(20, +a, 22)\} \neq \emptyset$, and hence $X_2(G^0)$ does not satisfy the second requirement in Proposition 7.20. Condition **C1** is violated here because $G^0 \xrightarrow{(10,-a,11)} G^1 \xrightarrow{(20,+a,22)} G^2$ is a sequence from G^0 of transitions outside $X_2(G^0)$, while transition $(20, +a, 22)$ is dependent in G^1 with $(20, -b, 21) \in X_2(G^0)$.

In Section 7.2 we described the algorithm used by POVAS for computing an ample set in a global state G. It aims at finding some process P_i whose set of executable transitions $X_i(G)$ is non-empty and satisfies the three conditions **C1**, **C2** (or **C2'** for on-the-fly model-checking) and **C3**. For the enhancement of POVAS in Section 7.3 we adopted a similar algorithm to find *all* processes P_i for which $X_i(G)$ satisfies just **C1** and **C3**. Concerning the implementation of these algorithms in the CFSM model, it is now immediate from Proposition 7.20 that one can check **C1** simply by

establishing whether process P_i has potentially executable transitions at G. Checking **C2** and **C3** is of course done as before by examining the DFS stack and the visibility of the transitions in $X_i(G)$ with respect to the given LTL formula. Precisely, when G is the current global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$ to be expanded during a DFS, and f is the LTL formula to be checked for Π, then for each $i \in I$ we have that:

- $X_i(G)$ satisfies **C1** if $P_i(G) = \emptyset$;
- $X_i(G)$ satisfies **C2** if no $t \in X_i(G)$ with $G \xrightarrow{t} H$ is such that H is on the DFS stack;
- $X_i(G)$ satisfies **C3** if $X_i(G) \cap vis_f(\Pi) = \emptyset$.

In terms of the algorithm in Figure 7.4 for finding multiple disjoint ample sets wrt **C1** and **C3**, the function call *check*_C1($X_i(G)$) is thus replaced by the simple test $P_i(G) = \emptyset$.

With the above translation of condition **C1** for protocols in the CFSM model, the formulation of LRA for verifying logical correctness properties in Chapter 5 can now be adapted easily to incorporate also the proposed enhancement of POVAS for LTL model-checking. Specifically, the leap sets to be used for LTL model-checking in the CFSM model can be constructed on the basis of wait-sets, in accordance with the following two definitions (cf. definitions 5.30 and 5.46).

Definition 7.21

Let G be a global state of a protocol $\Pi = (\{P_i \mid i \in I\}, L)$, and let $J, K \subseteq L$ and $V \subseteq \bigcup_{i \in I} \Delta_i$. Define $wait(G, J, K, V) = \{i \in I \mid X_i(G) \neq \emptyset \Rightarrow (P_i(G) \neq \emptyset \vee \exists (j, i) \in J\colon c_{ji}^G = \varepsilon \vee \exists (j, i) \in K\colon X_{ij}^+(G) \neq \emptyset \vee X_i(G) \cap V \neq \emptyset)\}$ and

$$pleap(G, J, K, V) = \{\, T \mid T \in leap(G) \wedge act(T) = \{i \in I \mid i \notin wait(G, J, K, V)\} \,\}$$
$$\text{if } wait(G, J, K, V) \subset I$$

$$pleap(G, J, K, V) = \{\, \{t\} \mid t \in X(G) \,\}$$
$$\text{otherwise.}$$

□

Definition 7.22

Let G be a global state of a protocol Π to be expanded during the DFS. Define

$$xpleap(G, J, K, V) = pleap(G, J, K, V) \cup \{T \cup \{t\} \mid t \in X(G) \text{ such that } act(t) \in wait(G, J, K, V), \text{ and } T \in pleap(G, J, K, V), \gamma \in lin(T) \text{ with } G \xrightarrow{\gamma}{}^{*} H \text{ and } H \text{ on the DFS stack}\}$$
$$\text{if } wait(G, J, K, V) \subset I$$

$$xpleap(G, J, K, V) = pleap(G, J, K, V)$$
$$\text{otherwise.}$$

□

Observe that when V includes the set $vis_f(\Pi)$ of visible transitions of a protocol Π wrt to some LTL formula f, the wait-set $wait(G, J, K, V)$ captures *at least* every process P_i whose set $X_i(G)$ does *not* qualify as an ample set in G with respect to conditions **C1** and **C3**, i.e. every process without executable transitions at G (violating the non-emptiness requirement on ample sets), or with potentially executable transitions at G (violating **C1**), or with executable transitions at G that are visible wrt f (violating **C3**). This attests that Definition 7.21 and Definition 7.22 indeed comply with Definition 7.9 and Definition 7.11 in Section 7.3, respectively. As a result, the reduced global state space obtained by the execution of the leap sets in $xpleap(G, J, K, V)$ in global states is adequate for deciding the absence of non-progress states, non-executable transitions, unspecified receptions wrt J and buffer overflows wrt K, as derived before in Chapter 5, and moreover for deciding the satisfiability of any LTL formula f with $vis_f(\Pi) \subseteq V$. In order to verify a protocol Π against a given LTL formula f, one would then typically set $V = vis_f(\Pi)$ and $J = K = \emptyset$. To sum up, by incorporating the proposed enhancement of POVAS into the formulation of LRA, we have established LRA as a uniform relief strategy for verifying both syntactic and semantic correctness properties of protocols in the CFSM model.

7.5 Experiments

The off-line versions of POVAS and its proposed enhancement have been implemented in the research tool package RELIEF, based on their formulation above for protocols in the CFSM model. Since the proposed enhancement of POVAS implements the execution of leap sets commensurate with LRA, in this section we will name it also LRA for short. Following the evaluation approach motivated in Chapter 6, the two model-checking techniques have been tested on the 400 sample protocols obtained with the automatic protocol synthesizer in RELIEF (see Section 6.3.1), and on the three real protocols taken from the literature: the X.21 call establishment/clear protocol [WZ78], the cache coherence protocol [Hol91] (see Appendix), and the alternating bit protocol with unreliable channels [Pac87] (see Section 6.3.2). The results of the experiments with the 400 synthesized protocols are given in Table 7.1, which compares the average percentages of reduction obtained with POVAS and LRA for off-line model-checking, per number of processes in a protocol and per concurrency level of a protocol. Recall that the latter was introduced in Chapter 6 as a conceivable measure for the degree of parallelism in a protocol. The first two rows of Table 7.1 show the reductions by POVAS and LRA over the conventional reachability analysis (CRA), respectively. The third row compares LRA directly to POVAS by normalizing the reductions by LRA with respect to those by POVAS. Table 7.2 gives the results of the experiments with the three real protocols. Overall, the numbers clearly indicate that using LRA instead of POVAS can further decrease both the memory and time resources needed for model-checking.

Table 7.1 LRA compared to POVAS for off-line model-checking.

Techniques		Average reductions (%) per number of processes							concurrency level			
		2	3	4	5	6	7	8	[0, 1]	(1, 2]	(2, 3]	(3, 4]
POVAS vs. CRA	states	36.11	51.16	61.67	69.55	77.30	81.79	89.13	38.19	64.29	85.52	93.41
	transitions	56.33	69.02	77.32	82.48	88.24	91.71	95.50	57.44	79.29	93.89	97.53
	space	36.03	51.23	61.72	69.68	77.41	81.87	89.15	38.16	64.37	85.58	93.37
	time	-42.64	10.61	36.48	52.85	65.33	66.06	81.36	-28.15	38.23	77.78	86.76
LRA vs. CRA	states	54.71	63.67	71.28	75.19	82.86	88.22	92.95	53.05	73.51	91.06	96.75
	transitions	63.59	73.24	80.34	84.32	89.88	93.61	96.33	63.25	82.33	95.29	98.30
	space	54.47	63.72	71.33	75.31	82.99	88.31	92.99	52.91	73.59	91.13	96.71
	time	30.61	41.61	51.17	59.87	68.41	68.04	84.52	27.26	53.27	79.01	92.22
LRA vs. POVAS	states	29.11	25.61	25.07	18.52	24.49	35.31	35.14	24.04	25.82	38.26	50.68
	transitions	16.62	13.62	13.32	10.50	13.95	22.92	18.44	13.65	14.68	22.91	31.17
	space	28.83	25.61	25.10	18.57	24.70	35.52	35.39	23.85	25.88	38.49	50.38
	time	51.35	34.68	23.13	14.89	8.88	5.83	16.95	43.24	24.35	5.54	41.24

Table 7.2 LRA and POVAS applied to three real protocols.

Protocol	Technique	States	Transitions	Space (MB)	Time (sec)
X.21 call establishment/clear	CRA	29868	64903	1.42	10.58
	POVAS	21805	32816	1.04	16.08
	LRA	15500	26882	0.75	6.33
Cache coherence	CRA	37037	126152	1.90	32.52
	POVAS	8760	11388	0.45	8.78
	LRA	5572	7966	0.28	5.45
Alternating bit	CRA	135352	626608	6.84	77.05
	POVAS	92414	266206	4.64	69.68
	LRA	63876	198583	3.24	65.90

7.6 Summary

In this chapter we studied the verification of temporal properties of finite-state concurrent systems and protocols. In particular, we addressed the state explosion problem in the context of LTL model-checking. LTL (linear-time temporal logic) is a propositional logic well suited for reasoning about semantic correctness properties of concurrent systems, including arbitrary safety and liveness properties. LTL model-checking refers to a fully automatic procedure, based on state exploration, for checking whether a given system satisfies some temporal property that can be expressed as a formula in LTL. In order to relieve the state explosion problem for LTL model-

checking, a series of so-called partial-order methods have been developed in recent years. It has been demonstrated that these methods can in many cases substantially reduce the space and time needed for LTL model-checking.

In this chapter we have built on the concepts underlying partial-order methods to yield an approach that enables further reductions in space and time for LTL model-checking. Specifically, we have proposed an enhancement of the partial-order method based on ample sets as described in [HP95, Pel96]. This method, which we referred to as POVAS (Partial Order Verification with Ample Sets), was chosen because it is generic in the sense that it can be readily adapted to capture the other partial-order methods (those based on persistent sets or stubborn sets), and because it is the most advanced partial-order method in terms of the properties that can be checked, the way fairness is dealt with, and the low overhead and high overall performance of its implementation [HP95, Pel96]. The idea behind the proposed enhancement stems from the principles underlying LRA in Chapter 5: instead of exploring a fixed interleaving order among concurrent operations, as does POVAS through the execution of ample sets, we abstain from any order altogether by executing leap sets that mimic a truly concurrent execution of these operations. Although POVAS and its proposed enhancement cannot be strictly compared in the sense that one does not subsume the other (i.e. their respective sets of reachable global states are not comparable by means of set inclusion), the experiments performed with both techniques confirmed that our approach to LTL model-checking is indeed an enhancement of POVAS. That is, our approach generally results in better space and time reductions and therefore widens the applicability of LTL model-checking to more complex concurrent systems and protocols.

Chapter 8

Conclusions and future work

In this concluding chapter, we recapitulate the contributions of the thesis and indicate several directions for further research.

8.1 Summary of contributions

The work described in this thesis concerns the (design) verification of concurrent systems, and of communication protocols in particular. Communication protocols, which constitute an important class of concurrent systems, were assumed to be specified explicitly in the well-established CFSM (communicating finite state machine) model [Boc78, Wes78, WZ78, ZW+80, BZ83] as networks of finite-state machines that communicate asynchronously by sending and receiving messages over error-free FIFO queues. Concurrent systems at large were viewed more abstract as collections of concurrent, interacting sequential processes whose joint behavior can be formalized as a (labeled) transition system. Indeed, state transitions of concurrent systems in general may represent system events other than just message transmissions and receptions and, furthermore, processes may communicate not only by asynchronous message passing but also by synchronous "hand shaking".

The focus of our research has been on improving strategies proposed earlier to relieve the state explosion problem which arises during the verification of concurrent systems and protocols by state space exploration, or reachability analysis. This was motivated by the awareness that, despite the merit of existing relief strategies, pursuing further performance improvements in verification remains utterly important. Indeed, concurrent systems are inherently complex and this complexity is here to stay (and grow). Based on a thorough review of the state of the art in verification, it became our particular objective to seek improvements of relief strategies that are based on state exploration. Such relief strategies inherit the simplicity of conventional reachability analysis (CRA), but reduce its complexity by examining only a fraction of the state space of a system. In effect, they enable the verification of properties of concurrent systems without exploring all possible interleaving orders

of concurrent events/transitions, which is one of the foremost causes of state explosion. Existing state exploration based relief strategies differ in the classes of systems they can handle, the types of properties they can verify, and the savings in space and time they can yield. The most advanced and promising state exploration based relief strategies in these respects are fair reachability analysis, simultaneous reachability analysis and partial-order reduction methods. In this thesis, we have proposed an incremental improvement of each of these three relief strategies so as to broaden their applicability to yet more complex and larger concurrent systems and protocols. The potential practical impact of our contributions is that more realistic industrial-strength systems may become amenable to automated verification.

Fair reachability analysis (FRA) is a relief strategy which was first proposed for the verification of logical correctness properties of two-process protocols specified in the CFSM model [RW82, GH85]. It was recently generalized to cyclic protocols, in which two or more processes form a unidirectional ring [LM94, LM96]. We have further generalized the technique of FRA to so-called multi-cyclic protocols (Chapter 4) [SU95a, LM+96, SU96a]. A multi-cyclic protocol consists of any number of unidirectional rings, or component cyclic protocols, which are interconnected such that no two rings share more than one process. The class of multi-cyclic protocols has an interestingly wide applicability in practical protocol modeling. It captures not only protocols with a multi-ring topology (of which the two-process and cyclic protocols studied in [RW82, GH85, LM94, LM96] are special cases), but also protocols with other regular network topologies like a daisy-chain, a star or a tree. Moreover, any combination of these elementary topologies is allowed as long as no two rings in the resulting protocol have more than one process in common.

As for cyclic protocols [RW82, GH85, LM94, LM96], FRA in its basic form is effective and efficient for the detection of deadlocks in multi-cyclic protocols. The fair reachable global state space of a multi-cyclic protocol explored by FRA generally constitutes only a very small fraction of the complete state space of the protocol explored by CRA (cf. Table 6.4). Indeed, through proper formulation we established that it entails just those reachable global states of a multi-cyclic protocol in which for each ring all channels in the ring are of equal length. This so-called ring-wise equal channel length property captures in particular all deadlock states. As a result, FRA decides the deadlock detection problem for every multi-cyclic protocol whose fair reachable global state space is finite. We also determined two sufficient conditions for finiteness that relate to the boundedness aspect of channels (see propositions 4.39 and 4.41). Both these conditions allow the presence of unbounded channels, which indicates that FRA is important not only as a relief strategy but also as a state exploration technique capable of handling various unbounded (multi-cyclic) protocols. The ring-wise equal channel length property and the boundedness conditions for multi-cyclic protocols generalize the equal channel length property and the respective boundedness conditions given in [LM94, LM96] for cyclic protocols. This generalization proved necessary because multi-cyclic

protocols that are not cyclic (i.e. those that are composed of multiple rings) have "connector" processes with more than one incoming and one outgoing channel. A next step along this line would be an extension of FRA to protocols with yet more complex and perhaps even arbitrary communication topologies. However, we established that FRA is in fact infeasible beyond multi-cyclic protocols. More accurately, its effectiveness is limited to protocols that are fair-formed (see Definition 4.49). This negative result stems from the principle characteristics of FRA, forcing progress of at least two processes at each step during state exploration while preserving a global channel invariant.

Like FRA, simultaneous reachability analysis (SRA) employs the concept of executing multiple transitions in a single atomic step to reduce the number of global states and transitions explored, and thereby the space and time needed for verification. However, unlike FRA, SRA can be used to verify logical correctness properties of protocols specified in the CFSM model with totally arbitrary communication topologies [ÖU95, Özd95]. In essence, this generality was achieved by allowing processes in a protocol to progress concurrently (or simultaneously) in a more flexible way than FRA. Notwithstanding its novelty, we have proposed an incremental improvement of SRA which enables further savings in space and time for protocol verification (Chapter 5) [SU96b, SU98a]. This improvement, named leaping reachability analysis (LRA), governs the execution of sets of concurrent transitions (i.e. leap sets) similar as SRA to verify the absence of non-progress states (including deadlocks), non-executable transitions, unspecified receptions and buffer overflows in a protocol. Through an analytical comparison we established that, for every protocol, the fraction of the protocol state space explored by LRA is contained in the fraction of the state space explored by SRA, for each of these four logical correctness properties. That is, LRA never explores more global states and transitions than SRA. Moreover, LRA never incurs more run-time overhead and even eliminates the need for a protocol augmentation as required by SRA for detecting unspecified receptions. LRA is thus an absolutely risk-free improvement of SRA, i.e. using LRA instead of SRA for verifying logical correctness properties of protocols in the CFSM model is at no cost whatsoever, neither in space nor in time. This is a notable result by itself, since all too often one is confronted with a trade-off between space and time. Especially when attempting to improve the performance of an already efficient verification technique, extreme care must be taken to ensure that potential extra savings in space are not attended by unacceptable expenses in time.

The increased efficiency in space and time of LRA over SRA stems mainly from the fact that LRA selects fewer leap sets (called selected simultaneously executable sets in [ÖU95, Özd95]) than SRA for execution at any given global state G. The leap sets in G that are executed by LRA are conditioned to contain at most one executable transition from among the processes with potentially executable transitions at G (i.e. transitions that are not executable at G but may become executable

later at a global state reached from G), while those executed by SRA are allowed to have multiple executable transitions from such processes. Simple combinatorics attest that the number of extra leap sets executed in G by SRA is exponential in the number of processes with both executable and potentially executable transitions at G (see Section 5.6.1). In general, LRA can therefore be expected to employ (i.e. compute and execute) a significantly smaller number of leap sets during state exploration than SRA, especially for protocols whose state spaces manifest a relatively wide distribution of potentially executable transitions. We have complemented this rational finding with an empirical comparison of the performance of LRA and SRA on a large number of protocols (Chapter 6). Based on their implementation in the research tool package RELIEF [Özd95, Ngu97], LRA and SRA were tested on a set of 400 impartial sample protocols constructed with the automatic protocol synthesiser in RELIEF [ÖU95, Öz95], and on three real protocols from the literature. The experiments revealed that, overall, LRA is able to yield important incremental extra savings over SRA in space and especially in time. Remark that these extra savings should indeed be expected to be "only" incremental, since SRA itself can already yield significant savings. For the detection of non-progress states in a protocol, both the space and time savings by LRA over SRA can be quite substantial (cf. Table 6.2 and Table 6.3). For the detection of non-executable transitions, unspecified receptions and buffer overflows, the extra space savings by LRA turn out to be rather modest, but the extra time savings can still be very good (cf. Table 6.5-6.7). Add thereto that we also offered an optional refinement of LRA which exploits the special characteristics of a depth-first search (see Section 5.5). This refinement (called LRA2 in Chapter 6) enables more discrete space savings over SRA for the detection of non-executable transitions and unspecified receptions, and with just little extra computational overhead (cf. Table 6.5 and Table 6.6). Based on our analytical and empirical comparisons, and in view of the notorious state explosion problem, we may certainly conclude that LRA is a worthwhile improvement of SRA as a relief strategy for protocol verification.

Partial-order reduction methods [God90, Val90, HGP92, KP92a, Val92, Val93, GW93, GW94, HP95, God96, Pel96] are a collection of cognate state exploration techniques set to relieve the state explosion problem for the verification of (finite-state) concurrent systems in general. That is, these methods are largely independent of the model used for specifying concurrent systems. They are pertinent in principle to all specification models whose semantics induce labeled transitions systems [HP95, God96], including the CFSM model. Partial-order reduction methods have proved effective and efficient for verifying local and termination properties (e.g. deadlock-freedom and freedom non-executable transitions) and, moreover, for verifying linear-time temporal logic (LTL) properties of concurrent systems. This is known as LTL model-checking, and captures arbitrary (temporal) safety and liveness properties. To achieve space and time reduction for verification, partial-order

reduction methods aim at exploring just one fixed order out of all possible interleaving orders of concurrent independent transitions at a given global state, by executing at each step during state exploration only a selective subset of the transitions executable at the current state, rather than all of them (as is done in CRA). We have shown how to combine this idea with the concepts underlying LRA to yield an approach which enables further reductions in space and time for LTL model-checking in the general context of finite-state concurrent systems that can be formalized as labeled transition systems (Chapter 7) [Sch97, SU98b]. In particular, we have proposed an enhancement of the partial-order reduction method based on ample sets as described in [HP95, Pel96]. This method, which we referred to as POVAS (Partial Order Verification with Ample Sets), was chosen because it is generic in the sense that it can be readily adapted to capture other partial-order reduction methods (i.e. those based on persistent sets or stubborn sets), and because it is advocated as the most advanced in terms of the properties it can check, the way it deals with fairness, and the low overhead and high overall performance of its implementation [HP95, Pel96]. In essence, instead of exploring some fixed interleaving order among concurrent independent transitions at global states, as does POVAS or any other partial-order reduction method on the basis of ample sets, persistent sets or stubborn sets, our enhanced approach abstains whenever possible from any order altogether by executing leap sets that mimic truly concurrent executions of such transitions.

We have further shown how to realize POVAS and its proposed enhancement specifically in the context of the CFSM model. In particular, we determined that the general notion of dependency among transitions can be captured efficiently within the CFSM model in terms of the notion of potentially executable transitions. This made it possible to incorporate our enhancement of POVAS for LTL model-checking into the formulation of LRA, thereby establishing LRA as a uniform relief strategy for the verification of both logical (or syntactic) and functional (or semantic) correctness properties of protocols specified in the CFSM model. That is, given a protocol Π, exploring its state space by LRA is based uniformly on the notion of "leaping" and varies only with the property to be verified. The checked property induces the designated subset of leap sets in $leap(G)$ to be executed at each global state G encountered during state exploration, i.e. $pleap(G)$ for verifying indefinite progress and $xpleap(G, J, K, V)$ for verifying freedom of non-executable transitions, unspecified receptions wrt J, buffer overflows wrt K, and any LTL formula f with $vis_f(\Pi) \subseteq V$ (see Section 7.4). The resulting savings in space and time are commensurate with the complexity of the property. The experiments performed with POVAS and LRA for (off-line) LTL model-checking in the CFSM model, based on their implementation in the research tool package RELIEF, indicated that LRA can indeed yield considerable extra savings in space and time over POVAS. Hence, it widens the applicability of LTL model-checking to more complex and larger concurrent systems and protocols.

8.2 Future work

Naturally, several open problems arise from our investigations. These problems are discussed below, together with some other suggestions for future research.

FRA for multi-cyclic protocols beyond deadlock detection

As explained in Section 4.5, FRA in its basic form is inadequate for the detection of logical errors other than deadlocks, mainly because it does not ensure the exposure of all reachable process states of the different processes in a multi-cyclic protocol. At least a finite extension of the fair reachable global state space of a multi-cyclic protocol is thus needed to provide a more comprehensive logical error coverage. Such an extension is already in force for cyclic protocols. Specifically, for the class of cyclic protocols with a finite fair reachable global state space, a procedure has been proposed in [LM94b, LM96] that finitely augments this reduced state space for deciding (un)boundedness, freedom of non-executable transitions and freedom of unspecified receptions. In [LM95], three more reachability problems were solved in a similar way, viz. global state reachability, abstract state reachability and execution cycle reachability. We have described the extension procedure for cyclic protocols in detail, and argued that it is unfit for generalization to multi-cyclic protocols due to the possible interaction dependencies that may arise among processes in different rings in these protocols. Unfortunately, we have not succeeded in devising an alternative extension procedure for the class of multi-cyclic protocols with a finite fair reachable global state space. We did sketch an argument suggesting that logical correctness properties other than deadlock-freedom are in fact undecidable in general for this class by FRA plus finite extension, but this argument is informal by all means. Strictly speaking, it is therefore still open whether FRA can be used to achieve the same logical error coverage for multi-cyclic protocols as for cyclic protocols. We are left with either finding a finite extension procedure suitable for multi-cyclic protocols, or proving formally that such a procedure cannot exist.

Of importance in this respect is also the characterization established in [LM94b, LM96] of the logical correctness of a cyclic protocol: a cyclic protocol is free from logical errors if and only if its fair reachable global state space is free from logical errors. In other words, every logical error within the reachable global state space of a cyclic protocol implies the existence of a logical error (not necessarily the same) within the fair reachable global state space of the protocol. This offers the advantage of iterative verification: one can verify a cyclic protocol correct by repeatedly applying FRA, fixing errors after each single run, until no more logical errors are found. The time required for verification may of course increase substantially, but it can still be a practical way to circumvent

memory shortage. Establishing whether or not a similar result holds for multi-cyclic protocols is clearly beneficial.

Identifying classes of unbounded protocols with decidable properties

It is certainly of theoretical interest to identify classes of infinite-state systems with decidable properties (cf. Section 2.5.1). Both FRA and LRA turned out to be capable of deciding logical correctness properties for various such unbounded protocols in the CFSM model, but a complete characterization of these protocols remains to be determined. Regarding FRA, it seems that the existence of such a characterization for unbounded multi-cyclic protocols goes hand in hand with the ability to use FRA beyond deadlock detection. Indeed, in relation to the discussion above, the notion of weak boundedness (see Definition 4.40) is a necessary and sufficient condition for cyclic protocols to have a finite fair reachable global state space [LM94a, LM96], but not so for multi-cyclic protocols in general (see Figure 4.5). We speculate that certain structural conditions on the process graphs of individual processes may ultimately provide a complete characterization of the classes of unbounded (multi-cyclic) protocols amenable to FRA and LRA, respectively, but this requires further investigation.

More state reduction

The amount of state reduction obtained by LRA (and likewise by SRA and partial-order reduction methods) relies on the number of dependencies between the transitions (i.e. the coupling among the processes) of a concurrent system or protocol. The more dependencies, the more LRA degrades to conventional reachability analysis, thus yielding less reduction. Since the exact dependency relation between transitions is generally too hard to determine, in practice one must employ some upper approximation of this relation that can be easily computed. Such approximated dependency relation may then still be refined to obtain fewer dependencies and thus more state reduction at the expense of extra computational overhead (see e.g. [KP92b, Val92, GP93, God96]). For protocols in the CFSM model we implicitly used an approximated dependency relation, by way of the notion of potentially executable transitions. This notion is currently defined on the basis of global state information only, but it seems possible to refine it by exploiting the structural characteristics of individual processes. For instance, in FRA, certain potentially executable transitions are further classified as enabled transitions. Enabled transitions provide the means to look one step ahead during state exploration. FRA indeed benefits from this, as witnessed by Table 6.4: for various multi-cyclic protocols it can yield much better space reductions than LRA, without incurring totally unacceptable time penalties.

Alternative ways to improve the efficiency of LRA, and state exploration techniques in general, may be found by tackling other causes of the state explosion problem (i.e. other than the modeling of concurrency by interleaving), such as variables whose values range over a large and possibly infinite domain. For instance, studying the possibility of combining symbolic verification techniques [BC+90, Bry92, HD93, McM93], and structural or functional decomposition techniques [VC82, CM83, LS84, CGL85, CM86] with LRA is certainly worthwhile. On a different note, implementing LRA in conjunction with the bit-state hashing [Hol88, Hol90, Hol91] and/or state space caching [Hol85, Hol87, JJ91, GHP92] disciplines will further increase its efficiency as well, as already discussed in Section 3.6.

Verifying other properties with LRA

So far, we have developed LRA as a(n) (improved) relief strategy for verifying logical correctness properties, and for model-checking LTL properties of concurrent systems and protocols, including arbitrary safety and liveness properties. This covers many of the properties that will ever be verified in practice. Still, it would be commendatory to extend the scope of LRA to other types of properties, like properties expressed in branching-time temporal logics [Eme90, BVW94]. Several results on adapting partial-order reduction methods for branching-time temporal logic model-checking have already been reported [GK+95, WW96]. The work in [GK+95] follows POVAS very closely, and an improvement of this work in terms of an extension of LRA for branching-time temporal logic model-checking is therefore imminent. Along the same line of thought, another important direction for further research is the verification of "real-time" and "probabilistic" properties of systems specified in models that involve a quantitative notion of time. A large body of work is currently underway to develop efficient techniques for this purpose (see e.g. [PRO98]).

Other applications

The state explosion problem is a limiting factor not only in verification, but also in areas such as protocol synthesis [PS91] and protocol conversion [CL90], as well as in many other applications of computer science and engineering. Any technique that tackles the state explosion problem in a systematic manner may therefore be useful beyond verification. In particular, we think that the leaping concept underlying LRA can be set to work for any problem that can be reduced to a state exploration (or search) problem and that exhibits some form of concurrency.

Bibliography

[AHU74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[AS87] B. Alpern, F.B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, 2(3): 117-126, 1987.

[BC+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, "Symbolic model checking: 10^{20} states and beyond," in *Proc. 5th Symposium on Logic in Computer Science*, Philadelphia, PA, 1990, pp. 428-439.

[BD89] S. Budkowski, P. Dembinski, "An introduction to Estelle: a specification language for distributed systems," *Computer Networks and ISDN Systems*, 18: 25-59, 1989.

[BH88] F. Belina, D. Hogrefe, "The CCITT specification and description language SDL," *Computer Networks and ISDN Systems*, 16: 311-341, 1988.

[Boc78] G. von Bochmann, "Finite state description of communication protocols," *Computer Networks*, 2: 361-372, 1978.

[Bry92] R.E. Bryant, "Symbolic boolean manipulation with ordered binary decision diagrams," *ACM Computing Surveys*, 24(3): 293-318, 1992.

[Büc62] J.R. Büchi, "On a decision method in restricted second order arithmetic," in *Proc. Congress Logic, Method and Philosophical Science 1960*, Stanford University Press, 1962, pp. 1-12.

[BVW94] O. Bernholtz, M.Y. Vardi, P. Wolper, "An automata-theoretic approach to branching-time model checking," in *Proc. CAV'94, Lecture Notes in Computer Science 818*, Springer-Verlag, 1994.

[BZ81] D. Brand, P. Zafiropulo, "On communicating finite-state machines," Technical Report RZ 1053, IBM Zurich Research Lab., Rüschlikon, Switzerland, 1981.

[BZ83] D. Brand, P. Zafiropulo, "On communicating finite-state machines," *Journal of the ACM*, 30(2): 323-342, 1983.

[CCITT76] CCITT (International Telegraph and Telephone Consultative Committee), "Recommendation X.21 (revised)," *AP VI-No. 55-E*, Geneva, Switzerland, 1976.

[CES86] E.M. Clarke, E.A. Emerson, A.P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems*, 8(2): 244-263, 1986.

[CGL85] C.H. Chow, M.G. Gouda, S.S. Lam, "A discipline for constructing multi-phase communication protocols," *ACM Transanctions on Computer Systems*, 3(4): 315-343, 1985.

[CL90] K.L. Calvert, S.S. Lam, "Formal methods for protocol conversion," *IEEE Journal on Selected Areas in Communication*, 8(1): 127-142, 1990.

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to algorithms*, MIT Press, 1990.

[CM83] T.Y. Choi, R.E. Miller, "A decomposition method for the analysis and design of finite state protocols," in *Proc. ACM SIGCOMM*, 1983, pp. 167-176.

[CM86] T.Y. Choi, R.E. Miller, "Protocol analysis and synthesis by structured partitions," *Computer Networks and ISDN Systems*, 11: 367-381, 1986.

[CR93a] L. Cacciari, O. Rafiq, "On improving reduced reachability analysis," in M. Diaz and R. Groz (eds.), *Formal Description Techniques V*, Elsevier, North-Holland, 1993, pp. 137-152.

[CR93b] L. Cacciari, O. Rafiq, "Decidability issues in reduced reachability analysis," in *Proc. 1993 International Conference on Network Protocols*, San Francisco, CA, 1993, pp. 158-165.

[CV+92] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, 1: 275-288, 1992.

[Eme90] E.A. Emerson, "Temporal and modal logic," *Handbook of Theoretical Computer Science*, 1990, pp. 997-1072.

[Fin88] A. Finkel, "A new class of analyzable CFSMs with unbounded FIFO channels," in S. Aggarwal and K. Sabnani (eds.), *Protocol Specification, Testing and Verification VIII*, Elsevier, North-Holland, 1988, pp. 283-294.

[Fra86] N. Francez, *Fairness*, Springer-Verlag, 1986.

[GCL85] M.G. Gouda, C.H. Chow, S.S. Lam, "On the decidability of livelock detection in networks of communicating finite state machines," in Y. Yemini, R. Strom and S. Yemini (eds.), *Protocol Specification, Testing and Verification IV*, Elsevier, North-Holland, 1985, pp. 47-56.

[GC86] M.G. Gouda, C-K. Chang, "Proving liveness for networks of communicating finite state machines," *ACM Transanctions on Programming Languages and Systems*, 8(1): 154-182, 1986.

[GH85] M.G. Gouda, J.Y. Han, "Protocol validation by fair progress state exploration," *Computer Networks and ISDN Systems*, 9: 353-361, 1985.

[GH93] P. Godefroid, G.J. Holzmann, "On the verification of temporal properties," in A. Danthine, G. Leduc and P. Wolper (eds.), *Protocol Specification, Testing and Verification XIII*, Elsevier, North-Holland, 1993, pp. 109-124.

[GHP92] P. Godefroid, G.J. Holzmann, D. Pirottin, "State space caching revisited," in *Proc. CAV'92, Lecture Notes in Computer Science 663*, Springer-Verlag, 1992, pp. 178-191.

[GK+95] R. Gerth, R. Kuiper, D. Peled, W. Penczek, "A partial order approach to branching time logic model checking," in *Proc. 3rd Israel Symposium on Theory of Computing and Systems*, Tel Aviv, Israel, 1995, pp. 130-139.

[God90] P. Godefroid, "Using partial orders to improve automatic verification methods," in *Proc. CAV'90, Lecture Notes in Computer Science 531*, Springer-Verlag, 1990, pp. 176-185.

[God96] P. Godefroid, "Partial order methods for the verification of concurrent systems: an approach to the state explosion problem," *Lecture Notes in Computer Science 1032*, Springer-Verlag, 1996. [Doctoral Dissertation, University of Liège, Belgium, 1994]

[Gou84] M.G. Gouda, "Closed covers: to verify progress for communicating finite state machines," *IEEE Transactions on Software Engineering*, SE-10(6): 846-855, 1984.

[GP93] P. Godefroid, D. Pirottin, "Refining dependencies improves partial-order verification methods," in *Proc. CAV'93, Lecture Notes in Computer Science 697*, Springer-Verlag, 1993, pp. 438-449.

[GW93] P. Godefroid, P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," *Formal Methods in System Design*, 2(2): 149-164. 1993.

[GW94] P. Godefroid, P. Wolper, "A partial approach to model checking," *Information and Computation*, 110(2): 305-326, 1994.

[GY84] M.G. Gouda, Y.T. Yu, "Protocol validation by maximal progress state exploration," *IEEE Transactions on Communications*, COM-32(1): 94-97, 1984. [also as: "Maximal progress state exploration," in *Proc. ACM SIGCOMM*, 1983, pp. 68-75]

[HD93] A.J. Hu, D.L. Dill, "Efficient verification with BDD's using implicitly conjoined invariants," in *Proc. CAV'93, Lecture Notes in Computer Science 697*, Springer-Verlag, 1993, pp. 3-14.

[HGP92] G.J. Holzmann, P. Godefroid, D. Pirottin, "Coverage preserving reduction strategies for reachability analysis," in R.J. Linn, M.Ü. Üyar (eds.), *Protocol Specification, Testing and Verification XII*, Elsevier, North-Holland, 1992, pp. 349-364.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[Hol85] G.J. Holzmann, "Tracing protocols," *AT&T Technical Journal*, 64(10): 2413-2433, 1985.

[Hol87] G.J. Holzmann, "Automated protocol validation in Argos: assertion proving and scatter searching," *IEEE Transactions on Software Engineering*, SE-13(6): 683-696, 1987.

[Hol88] G.J. Holzmann, "An improved protocol reachability analysis technique," *Software, Practice and Experience*, 18(2): 137-161, 1988.

[Hol90] G.J. Holzmann, "Algorithms for automated protocol verification," *AT&T Technical Journal*, 69(1): 32-44, 1990.

[Hol91] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[HP95] G.J. Holzmann, D. Peled, "An improvement in formal verification," in D. Hogrefe and S. Leue (eds.), *Formal Description Techniques VII*, Elsevier, North-Holland, 1995, pp. 177-191.

[HPY96] G.J. Holzmann, D. Peled, M. Yannakakis, "On nested depth-first search," in [SPIN96].

[II83] M. Itoh, H. Ichikawa, "Protocol verification using reduced reachability analysis," *Transactions on the IECE of Japan*, E66(2): 88-93, 1983.

[JJ91] C. Jard, T. Jeron, "Bounded-memory algorithms for verification On-the-fly," in *Proc. CAV'91, Lecture Notes in Computer Science 575*, Springer-Verlag, 1991, pp. 192-202.

[KI+85] M. Kajiwara, H. Ichikawa, M. Itoh, Y. Yoshida, "Specification and verification of switching software," *IEEE Transactions on Commununications*, COM-33(3): 193-198, 1985.

[KP92a] S. Katz, D. Peled, "Verification of distributed programs using representative interleaving sequences," *Distributed Computing*, 6: 107-120, 1992.

[KP92b] S. Katz, D. Peled, "Defining conditional independence using collapses," *Theoretical Computer Science*, 101: 337-359, 1992.

[KPV97] I. Kokkarinen, D. Peled, A. Valmari, "Relaxed visibility enhances partial-order reduction," in *Proc. CAV'97, Lecture Notes in Computer Science 1254*, Springer-Verlag, 1997.

[KWN88] Y. Kakuda, Y. Wakahara, M. Norigoe, "An acyclic expansion algorithm for protocol validation," *IEEE Transactions on Software Engineering*, SE-14(8): 1059-1067, 1988.

[Lam77] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, SE-3(2): 125-143, 1977.

[Lam80] L. Lamport, " 'Sometime' is sometimes 'not never'," in *7th Annual ACM Symposium on Principles of Programming Languages*, 1980, pp. 174-185.

[Lam83] L. Lamport, "What good is temporal logic?," in *Proc. 9th IFIP Congress – Information Processing '83*, pp. 657-668.

[LCL87] F.J. Lin, P.M. Chu, M.T. Liu, "Protocol verification using reachability analysis: the state space explosion problem and relief strategies," *Computer Communication Review*, 17(5): 126-134, 1987.

[Liu97] H. Liu, personal communication, January 1997.

[LM94a] H. Liu, R.E. Miller, "Generalized fair reachability analysis for cyclic protocols: part 1," in S.T. Vuong (ed.), *Protocol Specification, Testing and Verification XIV*, Elsevier, North-Holland, 1994, pp. 271-286.

[LM94b] H. Liu, R.E. Miller, "Generalized fair reachability analysis for cyclic protocols: decidability for logical correctness problems," in *Proc. 1994 International Conference on Network Protocols*, Boston, MA, 1994, pp. 100-108.

[LM95] H. Liu, R.E. Miller, "Reachability problems for cyclic protocols," in *Proc. 4th International Conference on Computer Communications and Networks*, Las Vegas, NV, 1995, pp. 32-39.

[LM96] H. Liu, R.E. Miller, "Generalized fair reachability analysis for cyclic protocols," *IEEE/ACM Transactions on Networking*, 4(2): 192-204, 1996.

[LM+96] H. Liu, R.E. Miller, H. van der Schoot, H. Ural, "Deadlock detection by fair reachability analysis: from cyclic to multi-cyclic protocols (and beyond?)," in *Proc. 16th International Conference on Distributed Computing Systems*, Hong Kong, 1996, pp. 605-612.

[LP85] O. Lichtenstein, A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, LA, 1985, pp. 97-107.

[LS84] S.S. Lam, A.U. Shankar, "Protocol verification via projections," *IEEE Transactions on Software Engineering*, SE-10(4): 325-342, 1984.

[Maz86] A. Mazurkiewicz, "Trace theory," in *Advances in Petri Nets 1986, Part II, Lecture Notes in Computer Science 255*, 1986, pp. 279-324.

[McM93] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[Mil89] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[MK96] H. Miller, S. Katz, "Saving space by fully exploiting invisible transitions," in *Proc. CAV'96, Lecture Notes in Computer Science 1102*, Springer-Verlag, 1996, pp. 336-347.

[MP92] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.

[MS87] N.F. Maxemchuk and K. Sabnani, "Probabilistic verification of communication protocols," in H. Rudin and C.H. West (eds.), *Protocol Specification, Testing and Verification VII*, Elsevier, North-Holland, 1987, pp. 307-320.

[MW84] Z. Manna, P. Wolper, "Synthesis of communicating processes from temporal logic specifications," *ACM Transanctions on Programming Languages and Systems*, 6(1): 68-93, 1984.

[Ngu97] T. Nguyen, "RELIEF version 3.5," Department of Computer Science, University of Ottawa, Canada, 1997.

[Oku88] K. Okumura, "Protocol analysis from language structure," in S. Aggarwal and K. Sabnani (eds.), *Protocol Specification, Testing and Verification VIII*, Elsevier, North-Holland, 1988, pp. 113-124.

[ÖU94] K. Özdemir, H. Ural, "Deadlock detection in CFSM models via simultaneously executable sets," in *Proc. 1994 International Conference on Communication and Information*, Peterborough, Ontario, Canada, 1994, pp. 673-688.

[ÖU95] K. Özdemir, H. Ural, "Protocol validation by simultaneous reachability analysis," *Computer Communications*, 20: 772-788, 1997. [also: Technical Report, Department of Computer Science, University of Ottawa, Canada, March 1995]

[Özd95] K. Özdemir, *Verifying the safety properties of concurrent systems via simultaneous reachability*, Doctoral Dissertation, Department of Computer Science, University of Ottawa, Canada, 1995.

[Pac87] J. Pachl, "Protocol description and analysis based on a state transition model with channel expressions," in H. Rudin and C.H. West (eds.), *Protocol Specification, Testing and Verification VII*, Elsevier, North-Holland, 1987, pp. 207-219.

[Pel93] D. Peled, "All from one, one for all: on model checking using representatives," n *Proc. CAV'93, Lecture Notes in Computer Science 697*, Springer-Verlag, 1993, pp. 409-423.

[Pel96] D. Peled, "Combining partial order reductions with on-the-fly model checking," *Formal Methods in System Design*, 8: 39-64, 1996.

[Pet81] J. L. Peterson, *Petri Net theory and the Modeling of Systems*, Prentice Hall, 1981.

[Pnu77] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Symposium on Foundations of Computer Science*, Providence, RI, 1977, pp. 46-57.

[PP90] W. Peng, S. Purushothaman, "A unified approach to the deadlock detection problem in networks of communicating finite state machines," in *Proc. CAV'90, Lecture Notes in Computer Science 531*, Springer-Verlag, 1990, pp. 243-252.

[PRO98] *Proceedings of the 1998 workshop on probabilistic methods in verification*, Indiana University, Indianapolis, IN, 1998.

[PS91] R.L. Probert, K. Saleh, "Synthesis of communicating protocols: survey and assessment," *IEEE Transactions on Computers*, 40(4), 1991.

[Rud88] H. Rudin, "Protocol engineering: a critical assessment," in S. Aggarwal and K. Sabnani (eds.), *Protocol Specification, Testing and Verification VIII*, Elsevier, North-Holland, 1988, pp. 3-16.

[Rud92] H. Rudin, "Protocol development success stories: part I," in R.J. Linn, M.Ü. Uyar (eds.), *Protocol Specification, Testing and Verification XII*, Elsevier, North-Holland, 1992, pp. 149-160.

[RW82] J. Rubin, C.H. West, "An improved protocol validation technique," *Computer Networks and ISDN Systems*, 6: 65-73, 1982.

[Sch97] H. van der Schoot, "Partial-order verification in SPIN can be more efficient," in [SPIN97].

[Sis85] A.P. Sistla, "On characterization of safety and liveness properties in temporal logic," in *Proc. 4th Annual ACM Symposium on Principles of Distributed Computing*, 1985, pp. 39-48.

[SPIN95] *Proceedings of the 1st SPIN workshop*, J. Grégoir (ed.), INRS-Télécommunications, Montréal, Québec, Canada, October 1995.

[SPIN96] *Proceedings of the 2nd SPIN workshop*, J. Grégoir, G.J. Holzmann, and D. Peled (eds.), Rutgers University, New Brunswick, NJ, August 1996.

[SPIN97] *Proceedings of the 3rd SPIN workshop*, R. Langerak (ed.), University of Twente, Enschede, The Netherlands, April 1997.

[SU95a] H. van der Schoot, H. Ural, "On generalizing fair reachability analysis to protocols with arbitrary topologies," in *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ontario, Canada, 1995, p. 267.

[SU95b] H. van der Schoot, H. Ural, "On improving simultaneous reachability analysis for the efficient verification of deadlock-freedom," Technical Report, Department of Computer Science, University of Ottawa, Canada, December 1995.

[SU96a] H. van der Schoot, H. Ural, "Deciding deadlock-freedom of daisy-chain protocols by fair reachability analysis," in *Proc. 11th International Symposium on Computer and Information Sciences*, Antalya, Turkey, 1996, pp. 579-588.

[SU96b] H. van der Schoot, H. Ural, "Protocol verification by leaping reachability analysis," in *Proc. 5th International Conference on Computer Communications and Networks*, Rockville, MD, 1996, pp. 334-339.

[SU98a] H. van der Schoot, H. Ural, "On improving reachability analysis for verifying progress properties for networks of CFSMs," in *Proc. 18th International Conference on Distributed Computing Systems*, Amsterdam, the Netherlands, 1998. [abridged version of the full paper: "A uniform approach to tackle state explosion in verifying progress properties for networks of CFSMs," Technical Report, Department of Computer Science, University of Ottawa, Canada, July 1997.]

[SU98b] H. van der Schoot, H. Ural, "An improvement of partial-order model checking," *Software Testing, Verification and Reliability* (to appear).

[Val90] A. Valmari, "Stubborn sets for reduced state space generation," in *Advances in Petri Nets 1990, Lecture Notes in Computer Science 483*, 1990, pp. 491-515.

[Val92] A. Valmari, "A stubborn attack on state explosion," *Formal Methods in System Design*, 1: 297-322, 1992.

[Val93] A. Valmari, "On-the-fly verification with stubborn sets," in *Proc. CAV'93, Lecture Notes in Computer Science 697*, Springer-Verlag, 1993, pp. 397-408.

[VC82] S.T. Vuong, D.D. Cowan, "A decomposition method for the validation of structured protocols," in *Proc. IEEE INFOCOM*, 1982, pp. 209-220.

[VW86] M.Y. Vardi, P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proc. 1st Symposium on Logic in Computer Science*, Cambridge, 1986, pp. 322-331.

[Wes78] C.H. West, "General technique for communications protocol validation," *IBM Journal of Research and Development*, 22(3): 393-404, 1978.

[Wes86] C.H. West, "Protocol verification by random state exploration," in G. von Bochmann (ed.) *Protocol Specification, Testing and Verification VI*, Elsevier, North-Holland, 1986, pp. 233-242.

[Wol89] P. Wolper, "On the relation of programs and computations to models of temporal logic," in *Proc. Temporal Logic in Specification, Lecture Notes in Computer Science 398*, Springer-Verlag, 1989, pp. 75-123.

[WVS83] P. Wolper, M.Y. Vardi, A.P. Sistla, "Reasoning about infinite computation paths," in *Proc. 24th IEEE Symposium on Foundations of Computer Science*, Tucson, AZ, 1983, pp. 185-194.

[WW96] B. Willems, P. Wolper, "Partial-order methods for model checking: from linear time to branching time," in *Proc. 11th Symposium on Logic in Computer Science*, New Brunswick, NJ, 1996.

[WZ78] C.H. West, P. Zafiropulo, "Automated validation of a communications protocol: the CCITT X.21 recommendation," *IBM Journal of Research and Development*, 22(1): 60-71, 1978.

[YG82] Y.T. Yu, M.G. Gouda, "Deadlock detection for a class of communicating finite state machines," *IEEE Transactions on Commununications*, COM-30(12): 2514-2518, 1982.

[Yua88] M.C. Yuang, "Survey of protocol verification techniques based on finite state models," in *Proc. Computer Networking Symposium*, Washington, DC, 1988, pp. 164-172.

[Zaf78] P. Zafiropulo, "Protocol validation by duologue-matrix analysis," *IEEE Transactions on Communications*, COM-26(8): 1187-1194, 1978.

[ZB86] J. Zhao, G. von Bochmann, "Reduced reachability analysis of communication protocols: a new approach," in G. von Bochmann (ed.) *Protocol Specification, Testing and Verification VI*, Elsevier, North-Holland, 1986, pp. 243-254.

[ZW+80] P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan, D. Brand, "Towards analyzing and synthesizing protocols," *IEEE Transactions on Communications*, COM-28(4): 651-661, 1980.

Appendix

This appendix contains the cache coherence protocol which was used as one of the three real protocols in the empirical study discussed in Chapter 6 and Chapter 7 of the thesis. The original Promela [Hol91] description of the protocol is given, as well as the translated description in the CFSM model that was provided as input to the research tool package RELIEF.

The cache coherence protocol in Promela

```
/*                              */
/* cache coherence protocol */
/*                              */

#define QUEUE_SIZE      2
#define W               1
#define R               2
#define X               3

/*                */
/* message types */
/*                */
mtype = {r, w, raw, RD, WR, RX, MX, MXdone, req0, req1, CtoB, BtoC, grant, done};

/*          */
/* channels */
/*          */
chan tocpu0     = [QUEUE_SIZE] of { byte };
chan fromcpu0   = [QUEUE_SIZE] of { byte };
chan tobus0     = [QUEUE_SIZE] of { byte };
chan frombus0   = [QUEUE_SIZE] of { byte };
chan tocpu1     = [QUEUE_SIZE] of { byte };
chan fromcpu1   = [QUEUE_SIZE] of { byte };
chan tobus1     = [QUEUE_SIZE] of { byte };
chan frombus1   = [QUEUE_SIZE] of { byte };
chan grant0     = [QUEUE_SIZE] of { byte };
chan grant1     = [QUEUE_SIZE] of { byte };
chan claim0     = [QUEUE_SIZE] of { byte };
chan claim1     = [QUEUE_SIZE] of { byte };
chan release0   = [QUEUE_SIZE] of { byte };
chan release1   = [QUEUE_SIZE] of { byte };
```

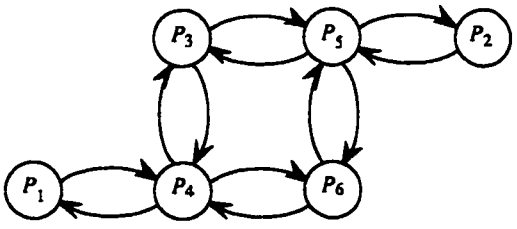
```
/*           */
/* processes */
/*           */

/* process 1: cpu0 */
proctype cpu0()
{
        xs fromcpu0;
        xr tocpu0;
        do
        :: fromcpu0!r   -> tocpu0?done
        :: fromcpu0!w   -> tocpu0?done
        :: fromcpu0!raw -> tocpu0?done
        od
}

/* process 2: cpu1 */
proctype cpu1()
{
        xs fromcpu1;
        xr tocpu1;
        do
        :: fromcpu1!r   -> tocpu1?done
        :: fromcpu1!w   -> tocpu1?done
        :: fromcpu1!raw -> tocpu1?done
        od
}

/* process 3: busarbiter */
proctype busarbiter()
{
        xs grant0;
        xs grant1;
        xr claim0;
        xr claim1;
        xr release0;
        xr release1;

        do
        :: claim0?req0 -> grant0!grant; release0?done
        :: claim1?req1 -> grant1!grant; release1?done
        od
}

/* process 4: cache0 */
proctype cache0()
{       byte state = X;
        byte which;

        xr frombus0;
        xr fromcpu0;
        xs tocpu0;
        xs tobus0;
        xr grant0;
        xs claim0;
        xs release0;
resume:
        do
        :: frombus0?RD ->
                if
                :: (state == W) -> state = R; tobus0!CtoB
                :: (state != W) -> tobus0!done
                fi
        :: frombus0?MX -> state = X; tobus0!MXdone
```

```
        :: frombus0?RX ->
                if
                :: (state == W) -> state = X; tobus0!CtoB
                :: (state == R) -> state = X; tobus0!done
                :: (state == X) -> tobus0!done
                fi

        :: fromcpu0?r ->
                if
                :: (state != X) -> tocpu0!done
                :: (state == X) -> which = RD; goto buscycle
                fi
        :: fromcpu0?w ->
                if
                :: (state == W) -> tocpu0!done
                :: (state != W) -> which = MX; goto buscycle
                fi
        :: fromcpu0?raw ->
                if
                :: (state == W) -> tocpu0!done
                :: (state != W) -> which = RX; goto buscycle
                fi
        od;
buscycle:
        claim0!req0;
        do
        :: frombus0?RD ->
                if
                :: (state == W) -> state = R; tobus0!CtoB
                :: (state != W) -> tobus0!done
                fi
        :: frombus0?MX -> state = X; tobus0!MXdone
        :: frombus0?RX ->
                if
                :: (state == W) -> state = X; tobus0!CtoB
                :: (state == R) -> state = X; tobus0!done
                :: (state == X) -> tobus0!done
                fi
        :: grant0?grant ->
                if
                :: (which == RD) -> state = R
                :: (which == MX) -> state = W
                :: (which == RX) -> state = W
                fi;
                tocpu0!done;
                break
        od;
        release0!done;

        if
        :: (which == RD) -> tobus0!RD -> frombus0?BtoC
        :: (which == MX) -> tobus0!MX -> frombus0?done
        :: (which == RX) -> tobus0!RX -> frombus0?BtoC
        fi;
        goto resume
}

/* process 5: cache1 */
proctype cache1()
{       byte state = X;
        byte which;

        xr frombus1;
        xr fromcpu1;
```

```
        xs tobus1;
        xs tocpu1;
        xr grant1;
        xs claim1;
        xs release1;
resume:
        do
        :: frombus1?RD ->
                if
                :: (state == W) -> state = R; tobus1!CtoB
                :: (state != W) -> tobus1!done
                fi
        :: frombus1?MX -> state = X; tobus1!MXdone
        :: frombus1?RX ->
                if
                :: (state == W) -> state = X; tobus1!CtoB
                :: (state == R) -> state = X; tobus1!done
                :: (state == X) -> tobus1!done
                fi

        :: fromcpu1?r ->
                if
                :: (state != X) -> tocpu1!done
                :: (state == X) -> which = RD; goto buscycle
                fi
        :: fromcpu1?w ->
                if
                :: (state == W) -> tocpu1!done
                :: (state != W) -> which = MX; goto buscycle
                fi
        :: fromcpu1?raw ->
                if
                :: (state == W) -> tocpu1!done
                :: (state != W) -> which = RX; goto buscycle
                fi
        od;
buscycle:
        claim1!req1;
        do
        :: frombus1?RD ->
                if
                :: (state == W) -> state = R; tobus1!CtoB
                :: (state != W) -> tobus1!done
                fi
        :: frombus1?MX -> state = X; tobus1!MXdone
        :: frombus1?RX ->
                if
                :: (state == W) -> state = X; tobus1!CtoB
                :: (state == R) -> state = X; tobus1!done
                :: (state == X) -> tobus1!done
                fi
        :: grant1?grant ->
                if
                :: (which == RD) -> state = R
                :: (which == MX) -> state = W
                :: (which == RX) -> state = W
                fi;
                tocpu1!done;
                break
        od;
        release1!done;

        if
        :: (which == RD) -> tobus1!RD -> frombus1?BtoC
```

```
        :: (which == MX) -> tobus1!MX -> frombus1?done
        :: (which == RX) -> tobus1!RX -> frombus1?BtoC
        fi;
        goto resume
}

/* process 6: bus */
proctype bus()          /* models real bus + main memory */
{
        xs frombus1;
        xs frombus0;
        xr tobus0;
        xr tobus1;

        do
        :: tobus0?CtoB -> frombus1!BtoC
        :: tobus1?CtoB -> frombus0!BtoC

        :: tobus0?done -> /* M -> B */ frombus1!BtoC
        :: tobus1?done -> /* M -> B */ frombus0!BtoC

        :: tobus0?MXdone -> /* B -> M */ frombus1!done
        :: tobus1?MXdone -> /* B -> M */ frombus0!done

        :: tobus0?RD -> frombus1!RD
        :: tobus1?RD -> frombus0!RD

        :: tobus0?MX -> frombus1!MX
        :: tobus1?MX -> frombus0!MX

        :: tobus0?RX -> frombus1!RX
        :: tobus1?RX -> frombus0!RX
        od
}

init {
        atomic {
                run cpu0(); run cpu1();
                run cache0(); run cache1();
                run bus(); run busarbiter()
        }
}
```

Given above is the specification of the cache coherence protocol in Promela [Hol91]. The protocol consists of six processes, which communicate over 2-slot FIFO channels according to the following topology graph:



Process P_1 and P_2 represent the two CPUs, process P_3 represents the bus arbiter, process P_4 and P_5 represent the two cache memories, and process P_6 represents the bus.

The cache coherence protocol in the CFSM model

The Promela specification of the cache coherence protocol given above can be translated directly into the CFSM model, since each of the six processes is in fact a communicating finite state machine adhering to the semantics of this model. The result of this translation is shown below. Given is an equivalent specification of the cache coherence protocol in the CFSM model, following the input format required by the research tool package RELIEF. Various comments have been added in-line to clarify the input format and to show the correspondence between this specification and the Promela specification above. The documentation on the research tool package RELIEF, including the tool input format requirements, is available on the World Wide Web at http://www.csi.ural.ca/~ural/c/RELIEFv35.

```
/* protocol 1a */
1

/* number of processes followed by the process las */
6 1 2 3 4 5 6

/* process 1: cpu0 */
2 0 1          /* two process states: state 0 and state 1 */
3                  /* state 0: 3 transitions */
r - 4 1                /* send message r to process 4 and move to state 1 */
w - 4 1                /* send message w to process 4 and move to state 1 */
raw - 4 1              /* send message raw to process 4 and move to state 1 */
1                  /* state 1: 1 transition */
done - 4 0             /* receive message done from process 4 and move to state 0 */

/* process 2: cpu1 */
2 0 1
3
r - 5 1
w - 5 1
raw - 5 1
1
done - 5 0

/* process 3: busarbiter */
5 0 1 2 3 4    /* five process states: states 0, 1, 2, 3 and 4 */
2                  /* state 0: 2 transitions */
req0 - 4 1             /* receive message req0 from process 4 and move to state 1 */
req1 - 5 2             /* receive message req1 from process 5 and move to state 2 */
1                  /* state 1: 1 transition */
grant - 4 3            /* send message grant to process 4 and move to state 3 */
1                  /* state 2: 1 transition */
grant - 5 4            /* send message grant to process 5 and move to state 4 */
1                  /* state 3: 1 transition */
done - 4 0             /* receive message done from process 4 and move to state 0 */
1                  /* state 4: 1 transition */
done - 5 0             /* receive message done from process 5 and move to state 0 */

/* process 4: cache0 */
40 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
6
MX - 6 1
RX - 6 2
```

```
RD - 6 2
r - 1 3
w - 1 4
raw - 1 5
1
MXdone - 6 0
1
done - 6 0
1
req0 - 3 6
1
req0 - 3 13
1
req0 - 3 20
1
MX - 6 7
RX - 6 8
RD - 6 8
grant - 3 9
1
MXdone - 6 6
1
done - 6 6
1
done - 1 10
1
done - 3 35
1
BtoC - 6 12
6
MX - 6 1
RX - 6 2
r - 1 28
RD - 6 29
w - 1 38
raw - 1 39
4
MX - 6 14
RX - 6 15
RD - 6 15
grant - 3 16
1
MXdone - 6 13
1
done - 6 13
1
done - 1 36
1
MX - 6 18
1
done - 6 19
6
MX - 6 1
RD - 6 32
RX - 6 33
r - 1 34
w - 1 34
raw - 1 34
4
MX - 6 21
RX - 6 22
RD - 6 22
grant - 3 23
```

```
1
MXdone - 6 20
1
done - 6 20
1
done - 1 37
1
RX - 6 25
1
BtoC - 6 19
4
MX - 6 14
RX - 6 15
grant - 3 16
RD - 6 30
4
MX - 6 21
RX - 6 22
grant - 3 23
RD - 6 31
1
done - 1 12
1
done - 6 12
1
done - 6 26
1
done - 6 27
1
CtoB - 6 12
1
CtoB - 6 0
1
done - 1 19
1
RD - 6 11
1
done - 3 17
1
done - 3 24
1
req0 - 3 26
1
req0 - 3 27

/* process 5: cache1 */
40 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
6
MX - 6 1
RX - 6 2
RD - 6 2
r - 2 3
w - 2 4
raw - 2 5
1
MXdone - 6 0
1
done - 6 0
1
req1 - 3 6
1
req1 - 3 13
1
req1 - 3 20
```

```
4
MX - 6 7
RX - 6 8
RD - 6 8
grant - 3 9
1
MXdone - 6 6
1
done - 6 6
1
done - 2 10
1
done - 3 35
1
BtoC - 6 12
6
MX - 6 1
RX - 6 2
r - 2 28
RD - 6 29
w - 2 38
raw - 2 39
4
MX - 6 14
RX - 6 15
RD - 6 15
grant - 3 16
1
MXdone - 6 13
1
done - 6 13
1
done - 2 36
1
MX - 6 18
1
done - 6 19
6
MX - 6 1
RD - 6 32
RX - 6 33
r - 2 34
w - 2 34
raw - 2 34
4
MX - 6 21
RX - 6 22
RD - 6 22
grant - 3 23
1
MXdone - 6 20
1
done - 6 20
1
done - 2 37
1
RX - 6 25
1
BtoC - 6 19
4
MX - 6 14
RX - 6 15
grant - 3 16
RD - 6 30
```

```
4
MX + 6 21
RX + 6 22
grant + 3 23
RD + 6 31
1
done - 2 12
1
done - 6 12
1
done - 6 26
1
done - 6 27
1
CtoB - 6 12
1
CtoB - 6 0
1
done - 2 19
1
RD - 6 11
1
done - 3 17
1
done - 3 24
1
req1 - 3 26
1
req1 - 3 27

/* process 6: bus */
11 0 1 2 3 4 5 6 7 8 9 10
12
CtoB + 4 1
done + 4 1
MXdone + 4 2
MXdone + 5 3
RD + 4 4
RD + 5 5
done + 5 6
CtoB + 5 6
RX + 5 7
RX + 4 8
MX + 5 9
MX + 4 10
1
BtoC - 5 0
1
done - 5 0
1
done - 4 0
1
RD - 5 0
1
RD - 4 0
1
BtoC - 4 0
1
RX - 4 0
1
RX - 5 0
1
MX - 4 0
```

```
:
MX - 5 0

/* QUEUE_SIZE */
2
```