National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Verifying the Safety Properties of Concurrent Systems via Simultaneous Reachability

Kadir Özdemir

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**

Department of Computer Science
University of Ottawa
Ottawa, Ontario

* The Ph.D. program in Computer Science is a joint program with Carleton University, administrated by the Ottawa-Carleton Institute for Computer Science

ISBN 0-612-11588-7

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

This thesis proposes two techniques, *simultaneous reachability analysis* and *simultaneous product method*, to reduce the number of global states to be analyzed for verifying the safety properties of concurrent systems. Both techniques utilize the idea of simultaneous execution of transitions. Simultaneous reachability analysis is proposed for verifying a specific set of safety properties asserting absence of logical errors of communication protocols specified as a network of n ($n \geq 2$) processes communicating over error-free, bounded, FIFO queues, without placing any restrictions on the topology of the network and process structures. We prove that simultaneous reachability analysis not only verifies the absence of logical errors such as deadlock, nonexecutable transition, unspecified reception and buffer overflow (in a correct protocol) but also identifies every deadlock state, every nonexecutable transition, every missing receiving transition causing unspecified receptions and every channel at which a buffer overflow occurs (in an incorrect protocol). An empirical study is carried out to demonstrate the efficiency of simultaneous reachability analysis in terms of time and memory requirements. In this study, 300 protocols constructed by an automatic empirical protocol synthesizer are used and results are evaluated with respect to the characteristics of these protocols. Empirical results show that using simultaneous reachability analysis substantially reduces the number of global states.

Simultaneous product method is proposed for verifying general safety properties of finite-state concurrent programs. In this method, a concurrent program is specified as a

collection of processes represented by finite automata on finite words and the concurrent behavior of these processes is defined by usual operational semantics (CSP-style): actions that appear in several processes are synchronized, others are interleaved. Verification problem is formulated in the framework of automata-theoretic model-checking where the negation of a safety property is converted to a finite automaton on finite words and then an automaton is obtained by taking the simultaneous product of the automata representing processes and the automaton representing the negation of a safety property. We prove that any safety property for a finite-state concurrent program can be efficiently verified by using simultaneous product method.

Acknowledgements

I am most grateful to my supervisor, Prof. Hasan Ural, for discussions, support and numerous insightful comments and corrections on many drafts of this thesis. I would like to thank the School of Graduate Studies and Research of the University of Ottawa for providing a Differential Tuition Fee Wavier and Entrance Scholarship. Also, I would like to acknowledge that this work is supported by a grant from Natural Sciences and Engineering Research Council of Canada.

Finally, I am indebted to my wife, Perihan, for her encouragement and support, and to my little son, Baran, for his patience in the first two years of his life.

Table of Contents

List of Figures

List of Tables

Chapter 1
INTRODUCTION

Computer communication protocols and concurrent programs (such as operating systems, process control programs, etc.) are examples of concurrent systems. A concurrent system is commonly modeled as a collection of communicating processes. The type of communication among processes differs one concurrent model from another. We will consider two main communication types: communication by exchanging messages asynchronously and communication by executing common actions synchronously.

Reachability analysis is a widely used automatic verification method for concurrent systems. It was first proposed by West [West78a] for analyzing and validating computer communication protocols. Clarke et al. [ClEm86] show that it can also be used to verify temporal properties of finite state concurrent systems. Reachability analysis requires the generation of all the states of a concurrent system, i.e., all the *global states*, that are reachable from the initial global state of the system. The well-known problem of this analysis is that the number of global states to be generated for a concurrent system can be so large that it may be impossible to store them in the memory of an available computer. This is called the *state explosion* problem.

The main objective of this thesis is to reduce the number of global states to be analyzed for verifying properties of concurrent systems. We propose two techniques, *simultaneous reachability analysis* and *simultaneous product method*, for verifying the safety properties

of protocols and finite-state concurrent programs, respectively. In case of protocols we will consider a set of specific safety properties which assert the absence of logical errors such as deadlock, unspecified reception, buffer overflow and nonexecutable transition while in case of concurrent programs we will consider general safety properties. Although the formulations of these techniques are different from each other, both techniques utilize the same idea of simultaneous execution of transitions. In the first part, we address the problem of verifying the logical correctness of protocols. The second part is for the verification of safety properties of finite-state concurrent programs.

1.1 Verifying the Logical Correctness of Protocols

Communication protocols are widely specified in the communicating finite state machine (CFSM) model [BrZa83]. In this model, a protocol is represented as a network of processes that exchange messages over simplex channels where each process is represented by a finite state machine (FSM) and each channel is represented by an error free FIFO queue. When one process sends a message to another process, the message is added to the corresponding channel and the sender process may continue to make progress. The delay between the transmission and the reception of the message is unpredictable and unbounded. A state of a protocol (i.e., a *global state*) consists of the state of each process and the content of each channel in the protocol.

It is desirable for a protocol not to include erroneous global states such as deadlock, unspecified reception and buffer overflow states. At a *deadlock state*, all channels are empty and none of the processes is ready to send a message. At an *unspecified reception state*, the protocol does not specify what global state should be entered upon receiving the message at the head of a channel. Finally, at a *buffer overflow state*, there exists a process ready to send a message over a (bounded) channel which is full. It is also desirable that

each specified transition in a protocol is executable. The existence of an erroneous global state or a nonexecutable transition is called a *logical error* in a protocol. The types of logical errors to be considered in this study are deadlock, unspecified reception, buffer overflow and nonexecutable transition. A protocol is said to be *logically correct* if it does not include any logical error.

In the CFSM model, the logical correctness of a protocol can be mechanically verified by reachability analysis. In reachability analysis, all reachable global states of the protocol are systematically explored and checked against the first three types of logical errors. As a consequence, the absence of a nonexecutable transition can be decided after exploring all reachable global states. There are two main problems in the verification of logical correctness of a protocol specified in the CFSM model: *decidability* and *state explosion.*

For an n-process protocol (i.e., a protocol consisting of n, n>2, processes) with arbitrary topology and process structures, the problem of the absence of logical errors is undecidable if at least one of the channels is unbounded [BrZa83]. Decidability of the logical correctness problem for subclasses of protocols is reported in [BrZa83, Pach87, PePu90, LiMi94]. One subclass of protocols for which the verification problem is decidable consists of protocols where all the channels are bounded [BrZa83]. Therefore, practical reachability analysis places a bound on each channel capacity to avoid the undecidability.

However, even with the bounded channels, protocols usually have a very large number of reachable global states. Exhaustively generating every reachable global state by reachability analysis gives rise to the state explosion problem. Many strategies have been proposed to relieve the state explosion problem. These relief strategies often make some restricted assumptions such as (1) the simplex channels are always empty [Boch78], (2) the number of processes is two [GoYu84, GoHa85, RuWe82], (3) processes do not contain cycles other than the ones passing through their initial states [ItIc83], (4) protocols exhibit

certain structures which allow them to be decomposed into smaller components that can be analyzed independently [ChMi83, VuCo82], (5) a process at a state can either transmit or receive a message [YuGo82] and (6) the topology of the protocol is a unidirectional ring [LiMi94]; or analyze a subset of the state space of the protocol which does not guarantee to show the absence of a logical error [Holz88, MaSa87]. Interested readers are referred to [LiCh87, Yuan88] for survey papers.

In reachability analysis, called henceforth *sequential reachability analysis*, global state space is examined by the execution of a single transition in a protocol at a time. In the relief strategy proposed in this part, called *simultaneous reachability analysis*, at a global state, the executable transitions of distinct processes are executed simultaneously to generate the next global states. If there are some potentially executable transitions in the current state of a process, additional global states are also generated by simultaneously executing only the transitions of the other processes. The purpose of the additional global states is to enable the later execution of potentially executable transitions of a process by inhibiting the progress of this process.When several transitions are simultaneously executable at a global state, sequential reachability analysis simply considers every possible interleaving (or linearization) of these transitions. If there exists k such transitions then there exist $k!$ interleavings of these transitions. Even though the set of intermediate global states generated by one interleaving may not be equal to the set generated by another interleaving, these interleavings lead to a common global state. Simultaneous reachability analysis examines only part of the global state space by generating those global states that are reachable through the simultaneous execution of transitions that are executable at a global state and therefore prevents the generation of a large number of intermediate global states. Simultaneous reachability analysis is based on the idea of the simultaneous execution of transitions which is first proposed in [ItIc83] as a relief strategy, called reduced reachability analysis, for a very restricted class of protocols in which processes must synchronize at their initial states after executing a finite number of transitions. Reduced reachability

analysis (in which only receiving transitions can be potentially executable transitions) can only detect deadlock states and the global states blocked by some unspecified receptions. The contributions of this thesis by proposing simultaneous reachability analysis are

- extending the idea of simultaneous execution of transitions to n-process protocols ($n \geq 2$) with bounded channels without placing any restrictions on the topology of the network and process structures,
- introducing potentially executable sending transitions, and
- proving that
  - + simultaneous reachability analysis
    - verifies the absence of
      a) deadlocks,
      b) global states blocked by some unspecified receptions and/or buffer overflows, and
      c) nonexecutable transitions
    - or alternatively, identifies every
      a) deadlock state,
      b) global state blocked by some unspecified receptions and/or buffer overflows, and
      c) nonexecutable transition in a protocol,
  - + by augmenting a given protocol by some receiving transitions, simultaneous reachability analysis verifies the absence of unspecified receptions or

alternatively, identifies at which process states and for which messages the protocol has unspecified reception errors,

+ by augmenting simultaneous reachability analysis, the absence of buffer overflows is verified or alternatively, every channel which is overflowed is identified.

Extensive empirical results are provided to show that simultaneous reachability analysis substantially relieves the state explosion problem for the detection of logical errors.

Part 1 is organized as follows: Chapter 2 reviews the CFSM model and basic definitions related to sequential reachability analysis. In Chapter 3, an overview of relief strategies is given. Simultaneous reachability analysis is formulated in Chapter 4. An empirical study is described and the results of this study are presented in Chapter 5. Finally, concluding remarks are given in Chapter 6.

1.2 Verifying Safety Properties of Finite-State Concurrent Programs

Sequential programs such as batch, off-line data processing and other computational programs are called *transformational* because they receive their inputs at the beginning of their operation and yield their outputs at termination. The correctness of sequential programs can be formulated in terms of a *Precondition/Postcondition* pair in a formalism such as Hoare's Logic, where Precondition and Postcondition are assertions on the program variables upon entry and exit of the program, respectively. Concurrent programs such as operating systems, process control programs, seat reservation systems, are called *reactive* because they maintain a nonterminating computation while interacting with their environments [Pnue86]. Due to this nonterminating behavior, the methods used for

proving the correctness of sequential programs cannot be directly used for concurrent programs.

For specifying and verifying the correctness of concurrent programs, *temporal logic* is suggested by Pnueli [Pnue77]. Temporal logic is a special type of Modal Logic, which makes possible to state that a proposition is true now (at the current global state), eventually (in some global state that will eventually be reached), henceforth (at every subsequent global state) by using temporal operators such as *sometime* and *always*. These operators appear quite appropriate for describing the time-varying behavior of concurrent programs. There are two types of temporal logic regarding the underlying nature of time : *linear* and *branching* [Lamp80]. In linear temporal logic, at each moment there is only one possible future moment while in branching temporal logic, at each moment time may split into several possible future moments. It is a matter of debate as to whether branching or linear time is preferable [Emer90].

The first methods for verification of concurrent programs using temporal logic are proof-theoretic. Typically, one has to prove that a given program satisfies a formula (a desired property) specified in either a linear or branching temporal logic using various axioms and inference rules. This approach is usually quite tedious, and requires an intimate understanding of the program. The proof-theoretic verification is usually done manually, and there is no hope of constructing the proof completely algorithmically [VaWo86]. It is known that many concurrent programming problems have finite-state solutions. In [ClEm86], it is stated that proof construction is unnecessary in the case of finite-state concurrent programs, and can be replaced by a model checking approach which will mechanically determine if the program satisfies (or, is the model of) a formula expressed in a propositional temporal logic. A formula is a combination of boolean operators (e.g., *not* and *and*), temporal operators (e.g., *sometime* and *always*) and atomic propositions. Since each global state is characterized by a finite amount of information, this information can be

described by the set of propositions which are true in this state. This means that a finite-state concurrent program can be viewed as a finite structure which consists of a finite set of global states, a reachability relation between global states and a function labeling each global state with the set of atomic propositions true at the global state. Such a structure is called a *labeled transition system* or (for historical reasons [HuCr77]) a *Kripke structure*. Given a Kripke structure and a formula, an efficient algorithm can be given to determine whether the structure is a model of the formula. This is called *model checking*.

In [WoVa83], it is shown that for any linear time propositional temporal formula, one can construct a finite automaton on infinite words [Büch62] that accepts precisely the sequences satisfied by the formula. This result has led to the *automata-theoretic* model checking approach of [VaWo86, Vard87, Wolp89, GoWo94]. In this approach, a concurrent program is viewed a collection of finite automata on infinite words. Then, the negation of the formula is converted to a finite automaton on infinite words. Finally, the verification can be done by simply checking that the product of the automata describing the program and the automaton corresponding to the negation of the formula is empty. There are certain advantages to the automata theoretic approach. It enables the reduction of the problem of model checking for linear temporal logic to known automata-theoretic problems, yielding clean and asymptotically optimal algorithms. In addition, it provides a uniform framework in the area of concurrent program verification. Finally, it can directly be implemented using *on-the-fly* verification techniques which may require a relatively small part of the finite structure in case of verifying an incorrect program [VaWo86, JaJe89, CoVa92, Valm93]. In an on-the-fly technique, the construction the finite Kripke structure and verification of the negation of the formula are simultaneously performed. Then the construction guided by the negation of the formula does not necessarily need to be completed; it might be stopped after an error has been found or only the erroneous part of the structure might be investigated.

Model checking can be considered as an augmented reachability analysis, because it requires the enumeration of reachable global states. Therefore, it inherits the state explosion problem from reachability analysis. Recently a great deal of work is done for relieving the state explosion problem by using partial order model checking approaches, e.g., [GoWo93, GoWo94, Valm91, Valm93, Pele93, Pele94]. Most of the previous model checking methods are based on modeling concurrency by interleaving. In these models, the concurrent execution of k transitions is analyzed by exploring all $k!$ interleavings of these transitions. However, it has been proved that those $k!$ interleavings lead to a common end global state and the desired properties are in many cases insensitive to the interleaving order. Partial order methods avoid generating every possible interleaving, which reduces the size of the checked state space. They generate one interleaving, i.e., k global states, instead of $k!$ interleavings.

The automata-theoretic model checking method proposed in this part, called *simultaneous product method*, goes one step ahead by observing and proving that in many cases there is no need to even generate the intermediate k-1 global states by modeling concurrent execution of these k transitions by their simultaneous execution. Then, it generates one global state instead of k global states generated in partial order methods. Therefore, it is possible to obtain more reduction on the number of global states by using simultaneous product method.

For properties of concurrent programs, a classification into safety and liveness properties was first proposed by Lamport [Lamp77]. Intuitively, a safety property asserts that "something bad" never happens, and a liveness property asserts that "something good" eventually happens [Lamp80]. Restricting the verification problem to safety properties leads to better verification algorithms because safety properties can be checked by considering the finite behavior of a concurrent program whereas liveness properties are only meaningful for infinite behaviors [JaJe89, GoWo93]. This means that when the

verification of a safety property is considered, a concurrent program is viewed as a collection of finite automata on finite words and the negation of the formula (representing the safety property) is converted to a finite automaton on finite words. In this part, we verify safety properties of finite state concurrent programs by a automaton which is constructed by taking the simultaneous product of finite automata on finite words describing the program and the negation of the formula.

Part 2 is organized as follows: Chapter 7 reviews automatic verification of finite-state concurrent programs. An overview on the reduction methods in model checking is given in Chapter 8. Simultaneous product method is formulated in Chapter 9. Finally, concluding remarks are given in Chapter 10.

1.3 Main Contributions of the Thesis

In this thesis, two new methods, called *simultaneous reachability analysis* and *simultaneous product method*, are proposed in the area of verification of concurrent systems. Simultaneous reachability analysis is proposed for the verification of the logical correctness of protocols represented in the CFSM model. Simultaneous product method is proposed for the verification of the safety properties of the finite-state concurrent programs represented in the automata model. Although the proposed methods can be differentiated from each other in terms of their models, backgrounds and applications, both of the methods utilize the idea of simultaneous execution of transitions. The main contributions of this thesis can be listed as follows:

- Simultaneous reachability analysis can be applied to any protocol with bounded channels, i.e., does not place any restrictions on the topology of the protocol or on the structure of the processes.
- This thesis proves that

+ simultaneous reachability analysis

  - verifies the absence of

    a) deadlocks,

    b) global states blocked by some unspecified receptions and/or buffer overflows, and

    c) nonexecutable transitions

  - or alternatively, identifies every

    a) deadlock state,

    b) global state blocked by some unspecified receptions and/or buffer overflows, and

    c) nonexecutable transition in a protocol,

+ by augmenting a given protocol by some receiving transitions, simultaneous reachability analysis verifies the absence of unspecified receptions or alternatively, identifies at which process states and for which messages the protocol has unspecified reception errors,

+ by augmenting simultaneous reachability analysis, the absence of buffer overflows is verified or alternatively, every channel which is overflowed is identified.

• An empirical study is carried out as part of this thesis, which demonstrates that simultaneous reachability analysis reduces the number of global states need to be stored up to 70-80 %.

- This thesis proves that simultaneous product method, an automata-theoretic model-checking method, can be applied to any finite-state concurrent program.
- This thesis proves that simultaneous product method verifies safety properties.
- This thesis presents an efficient algorithm to implement simultaneous product method.
- This thesis analytically shows that simultaneous product method generates fewer global states without consuming more time when compared to the conventional method.
- This thesis shows that simultaneous product method has a better best case space complexity when compared to other methods.

Part I

Verifying the Logical Correctness of Protocols

Chapter 2
THE CFSM MODEL

A protocol is a set of rules that governs the orderly exchange of messages among interacting processes in a distributed system. Although this definition implies that protocols are related to all areas where interaction between processes is inherent, we will focus mainly on communication protocols.

In section 2.1, we review a simple but very popular representation model, the Communicating Finite State Machine (CFSM) model, for communication protocols. In section 2.2, we briefly outline the classes of protocols where the problem of detecting logical errors is decidable. Finally, in section 2.3, we give basic definitions and notations related to the CFSM model.

2.1 Introduction to the CFSM Model

In the CFSM model, a protocol is viewed as a collection of processes communicating with each other over error-free simplex channels. Each process is represented by a communicating finite state machine and each simplex channel is represented by a FIFO queue. In this model, the local data structures and internal operations are abstracted, and only operations related to sending and receiving messages are allowed in each process.

A communicating finite state machine can be depicted by a directed graph where an arc corresponds to either transmission or reception of a message. In Figure 2.1, an example of a simple protocol between a client (CLIENT) and a server (SERVER) in a computer

network is illustrated. This example is adapted from [BrZa83]. In this protocol, there are two simplex channels (C_{12} and C_{21}) and each channel has a capacity of one message. C_{12} (C_{21}) is the outgoing (incoming) channel for CLIENT and the incoming (outgoing) channel for SERVER. READY and IDLE are the initial states of processes CLIENT and SERVER, respectively. A state of a protocol, called a *global state*, consists of the state of each process and the content of each channel. A distinguished global state of a protocol is its initial global state where all processes are at their initial states and all channels are empty. For this example, the initial global state consists of two process states READY and IDLE and two empty sequences.

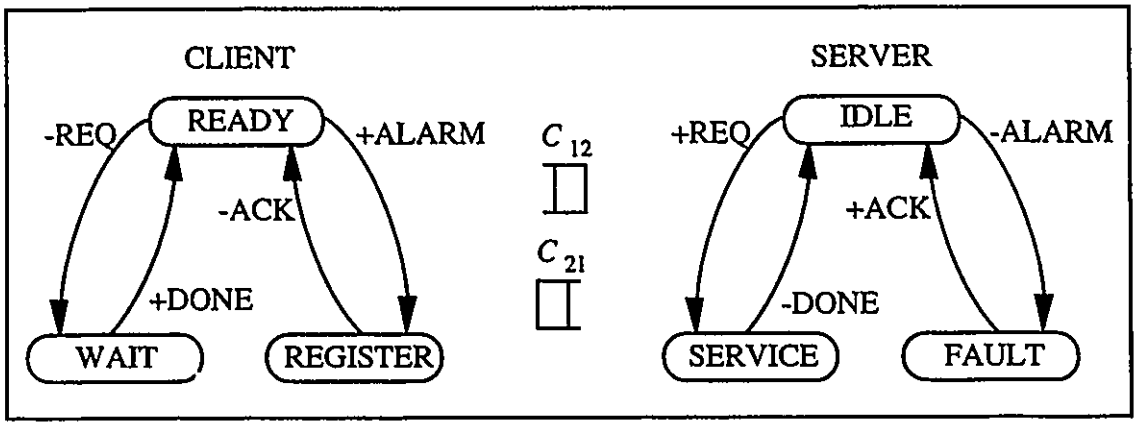


Figure 2.1 An example of a simple protocol adapted from [BrZa83].

An arc corresponding to a transmission (reception) of a message x is labeled by $-x$ ($+x$). When a process at a state from which an arc $-x$ leaves and the corresponding channel is not full then the process can traverse the arc. By traversing this arc the process enters the new state to which the arc points, and the message x is transmitted to the destination via the channel. For example, at the initial global state, CLIENT can traverse the arc labeled -REQ. Then CLIENT enters state WAIT and message REQ is transmitted to SERVER via channel C_{12}. There is no assumption on the time that a message spends in a channel. The delay between the transmission and the reception of a message is variable and unspecified. If and

when a message x arrives at a process that is at a state s from which an arc $+x$ leaves then the process can enter a new state by traversing the arc labeled $+x$ and message x is removed from the channel.

After receiving message REQ, SERVER enters state SERVICE; when it is finished with the service, it goes back to state IDLE while sending message DONE to CLIENT. Meanwhile CLIENT stays at state IDLE. By receiving message DONE, it returns to the initial state READY. While at state IDLE, SERVER can detect a fault in itself. If so, it informs CLIENT about it by sending message ALARM. When CLIENT receives message ALARM at state READY, it registers it and directs SERVER back to its state IDLE by message ACK.

It is desirable for a protocol not to include erroneous global states such as deadlock, unspecified reception and buffer overflow states. At a *deadlock state*, all channels are empty and none of the processes is ready to send a message. At an *unspecified reception state*, the protocol does not specify what global state should be entered upon receiving the message at the head of a channel. Finally, at a *buffer overflow state*, there exists a process ready to send a message over a (bounded) channel which is full. It is also desirable that each specified transition in a protocol is executable. The existence of an erroneous global state or a nonexecutable transition is called a *logical error* in a protocol. In this thesis, we consider four types of logical errors: deadlock, unspecified reception, buffer overflow and nonexecutable transition and we say that a protocol is *logically correct* if it does not include any logical error.

The protocol given in Figure 2.1 is not a logically correct protocol because it includes unspecified receptions and buffer overflows. To see unspecified receptions, consider the execution where CLIENT sends REQ and SERVER sends ALARM before either message arrives at its destination. Message ALARM arrives when CLIENT is at state WAIT, and message REQ arrives when SERVER is at state FAULT, but the protocol does not specify

what should happen in both situations. In each case, it is said that an unspecified reception occurs. To see buffer overflows, consider the case that SERVER sends ALARM at state IDLE and CLIENT receives ALARM at state READY and enters state READY by sending ACK to SERVER. It is possible that CLIENT then sends REQ before SERVER receives ACK. Since channel C_{12} can hold at most one message, this leads to an overflow in C_{12}.

The protocol given in Figure 2.2 is obtained by specifying missing receptions and increasing the capacities of channels of the protocol in Figure 2.1. However, this protocol is also logically incorrect. Consider the case that both REQ and ALARM are sent, and then received. In this case, the protocol reaches a global state where the process states are WAIT and FAULT, and the channels are empty. Since no message transmission is possible and the channels are empty, the protocol is at a deadlock state.

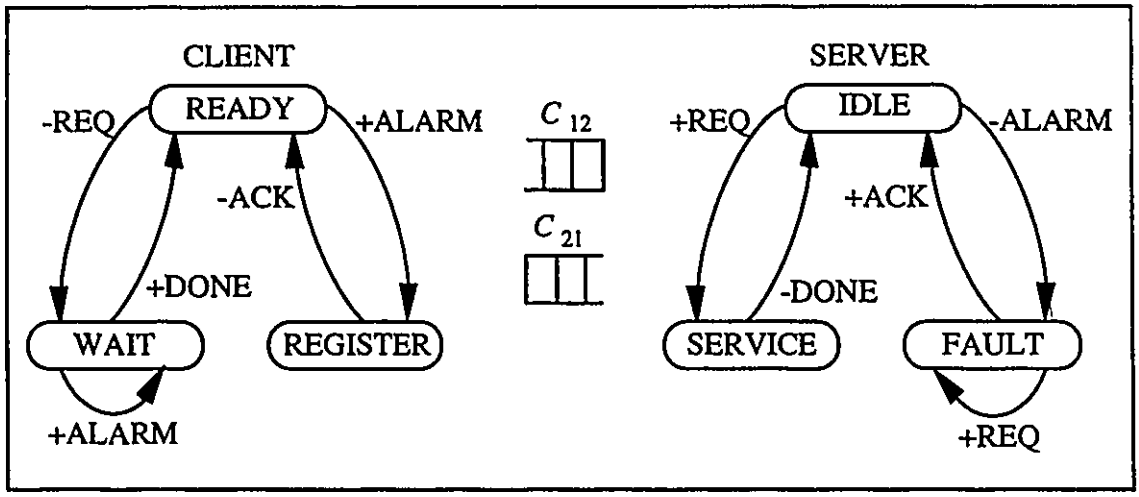


Figure 2.2 An example of a simple protocol adapted from [BrZa83].

2.2 Undecidability of Detecting Logical Errors

A protocol specified in the CFSM model can be viewed as a finite directed graph where nodes correspond to processes and arcs correspond to channels in the protocol. This

directed graph is called the *communication graph* of the protocol. When we refer to the *topology* of a protocol, we actually refer to the topology of its communication graph. A channel in a protocol is called an *unbounded* channel if the channel can hold an infinite number of messages. The problem of detecting of a logical error in a protocol is undecidable in the general case, i.e., for the class of protocols with arbitrary number of processes, arbitrary topology and unbounded channels [BrZa83]. The undecidability stems from the fact that even a two-process protocol with unbounded channels has the same computing power as a Turing machine. By viewing a protocol as a Turing machine, the halting problem can be reduced to the problem of detecting a logical error [BrZa81]. For the following subclasses of protocols, the detection problem is decidable:

- Protocols in which all channels are bounded [BrZa83].
- Two-process protocols with one bounded channel [BrZa83].
- Protocols with *recognizable channel property* [Pach87]. A channel is *recognizable* if the language of the channel (i.e., the set of all strings (sequences) of messages corresponding to the possible contents of the channel) is a regular language.
- Protocols with *monogenous channel property* [Fink88]. A channel is *monogenous* if the language of the channel is a set of prefixes of a finite number of words.
- Protocols with unidirectional ring topology and exactly one unbounded channel [PePu90].
- Protocols with unidirectional topology and channels which are not simultaneously unbounded [LiMi94].

Bypassing the undecidability problem by assuming that all channels are bounded is a popular strategy, e.g., [BrZa83, GoYu84, Holz85b, HuCh93, KaWa88, YuKe90]. In fact, Brand and Zafiropulo [BrZa83] observe that many protocols have all their channels

bounded. In the next section we will present basic terminology related to the CFSM model with bounded channels and the sequential (i.e., conventional) reachability analysis.

2.3 Preliminaries

In the CFSM model, a protocol is modeled by a collection of n processes exchanging messages over error-free bounded simplex channels.

Definition 2.3.1 A *protocol* Π which consists of a set $P = \{P_i \mid i = 1,2,...,n\}$ of processes communicating over error-free bounded simplex channels (represented by FIFO queues) is defined as a quintuple $\Pi = (\mathbf{S}, \mathbf{o}, \mathbf{M}, \delta, \mathbf{B})$ where

- $\mathbf{S}=\langle S_i \rangle_{i=1,2...,n}$ are n pairwise disjoint sets where S_i is the finite set of states of P_i.
- $\mathbf{o}=\langle o_i \rangle_{i=1,2...,n}$ where $o_i \in S_i$ is the initial state of P_i.
- $\mathbf{M}=\langle M_i \rangle_{i=1,2...,n}$ where $M_i = \{x \mid x \in M_{ij} \cup M_{ji}$ for $j=1,2,...,n$ and $i \neq j\}$ is the message set of P_i. Each M_{ij} represents the messages that can be sent from P_i to P_j such that $M_{ij} \cap M_{kl} = \varnothing$ for each $i,j,k,l = 1,2,...,n$ and $(i,j) \neq (k,l)$.
- $\delta=\langle \delta_i \rangle_{i=1,2...,n}$ where δ_i: $S_i \times M_i \rightarrow S_i$ is the partial transition function for P_i.
- $\mathbf{B}=\langle B_{ij} \rangle_{i,j=1,2...,n \text{ and } i \neq j}$ where B_{ij} is the maximum number of messages allowed in C_{ij} which represents the channel from P_i to P_j. □

Definition 2.3.2 In a process P_i of protocol Π, a triple $(s_i, x, \delta_i(s_i, x))$ with $s_i \in S_i$ and $x \in M_i$ is said to be a *transition* iff $\delta_i(s_i, x)$ is defined. When $x \in M_{ij}$, the transition is called *sending transition* and represents the transmission of message x from process P_i to process P_j. When $x \in M_{ji}$, the transition is called *receiving transition* and represents the reception of message x from process P_j by P_i. For readability purposes, a minus sign is used to identify a message transmission and plus sign is used to identify a message reception. □

Definition 2.3.3 A *global state* G_k of protocol Π is ($<s_i^k>_{i=1,2,...,n}$,$<c_{ij}^k>_{i,j=1,2,...,n \text{ and } i\neq j}$) such that s_i^k is the state of P_i and $c_{ij}^k \in M_{ij}^*$ is the content of C_{ij} at G_k where M_{ij}^* denotes the set of all finite sequences of messages from M_{ij} including empty sequence ε. If the content of C_{ij} is empty at G_k, it is represented by ε , i.e. c_{ij}^k =ε. The number of messages in C_{ij} is represented by $|c_{ij}|$. G_0 denotes the *initial global state* of Π where each process is at its initial state and all channels are empty, i.e., G_0=($<o_i>_{i=1,2,...,n}$, $<c_{ij}^0>_{i,j=1,2,...,n \text{ and } i\neq j}$) and c_{ij}^0=ε. ||

spec(s_i), *spec*(G_k), *spec*(P_i) and *spec*(P) denote all transitions at a process state, at a global state, in a process and in a set of processes, respectively i.e.

- $spec(s_i)=\{(s_i,x,\delta_i(s_i,x)) \mid \delta_i(s_i,x)$ is defined in $P_i\}$,
- $spec(G_k)=\bigcup_{i=1}^{n} spec(s_i^k)$,
- $spec(P_i)=\{t \mid s_i \in S_i \wedge t \in spec(s_i)\}$ and
- $spec(P)=\{t \mid P_i \in P \wedge t \in spec(P_i)\}$, respectively.

Definition 2.3.4 A global state G_l is a *sequential immediate successor* of G_k , denoted by $G_k \rightarrow G_l$ or $G_k \text{—}t\rightarrow G_l$, iff $\exists t \in spec(G_k)$ such that one of the following conditions is satisfied

1. $t=(s_i^k,-x,s_i^l) \wedge s_i^l=\delta_i(s_i^k,-x) \wedge c_{ij}^l=c_{ij}^k x \wedge |c_{ij}^k|<B_{ij} \wedge \forall(h=1,2,...,n)\ (h\neq i \Rightarrow s_h^l=s_h^k) \wedge \forall(h,m=1,2,...,n)\ (h\neq m \wedge \neg(h=i \wedge m=j) \Rightarrow c_{hm}^l=c_{hm}^k)$.
2. $t=(s_i^k,+x,s_i^l) \wedge s_i^l=\delta_i(s_i^k,+x) \wedge c_{ji}^k=xc_{ji}^l \wedge \forall(h=1,2,...,n)\ (h\neq i \Rightarrow s_h^l=s_h^k) \wedge \forall(h,m=1,2,...,n)\ (h\neq m \wedge \neg(h=i \wedge m=j) \Rightarrow c_{hm}^l=c_{hm}^k)$. []

These conditions correspond to a message transmission and a message reception by a process, respectively. Note that Definition 2.3.4, the sequential immediate successor, defines a binary relation between global states which is denoted by $\rightarrow$.

Definition 2.3.5 Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$.

1. G_m is *sequentially reachable* from G_k (or a *sequential successor* of G_k) iff $G_k \rightarrow^* G_m$.

2. G_m is *sequentially reachable* iff $G_0 \rightarrow^* G_m$.

$G_k \xrightarrow{t_1} G_{k(1)} \wedge G_{k(1)} \xrightarrow{t_2} G_{k(2)} \wedge \ldots \wedge G_{k(j-1)} \xrightarrow{t_j} G_m$ will sometimes be denoted by $G_k \xrightarrow{t_1 t_2 \ldots t_j} {}^* G_m$. □

Definition 2.3.6 For any $1 \le i,j \le n$ and $i \ne j$, a transition $t \in spec(P_i)$ is *executable* iff there exists a sequentially reachable global state G_k such that one of the following two conditions is satisfied:

1. $t=(s_i^k, -x, \delta_i(s_i^k, -x)) \wedge x \in M_{ij} \wedge |c_{ij}^k| < B_{ij}$.

2. $t=(s_i^k, +x, \delta_i(s_i^k, +x)) \wedge x \in M_{ji} \wedge c_{ji}^k = xX \wedge X \in M_{ji}^*$.

The set of executable transitions at G_k is denoted by $exec(G_k)$. □

Definition 2.3.7 In a protocol Π, deadlock, unspecified reception, nonexecutable transition and buffer overflow are logical errors and defined as follows:

1. t is a *nonexecutable transition* in Π if $t \in spec(P)$ and t is not executable.

Let $G_0 \rightarrow^* G_k$. At G_k,

2. a *deadlock* occurs if all channels are empty and there is no executable transition.

3. an *unspecified reception* of $x \in M_{ij}$ occurs if

$c_{ij}^k = xX \wedge X \in M_{ij}^* \wedge \delta_j(s_j^k, +x)$ is not defined.

4. a *buffer overflow* occurs in C_{ij} if
   $(s_i^k, -x, \delta_i(s_i^k, -x)) \in spec(G_k) \wedge x \in M_{ij} \wedge |c_{ij}^k| = B_{ij})$. ||

Definition 2.3.8 A protocol Π is logically correct if it does not contain a logical error. ||

Although we do not distinguish between "validating a protocol" and "verifying the logical correctness of a protocol", we prefer the latter expression for simultaneous reachability analysis, the method proposed in this thesis, to emphasize that it proves either the existence or absence of a logical error [Sajk85]. Notice that the logical correctness of a protocol does not require the functional correctness of the protocol. A protocol may be functionally incorrect but may not include any logical error. There is an analogy between the syntactic correctness of a program and the logical correctness of a protocol. "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" [Rudi88]. The logical errors are sometimes called *syntactic* errors of a protocol.

In sequential reachability analysis [Boch78, Haje78, West78a], logical errors in a given protocol are checked by generating all sequentially reachable global states from the initial global state with either a transmission or a reception of a message by a process at each step. If a global state includes a logical error, the error type and at least one transition sequence from the initial state to the erroneous global state are reported. During this analysis, a global state may be generated several times and thus to guarantee the termination of the analysis, the repeated occurrences of each global state must be recognized not to be examined again. An algorithm for the sequential reachability analysis is given in Figure 2.3. Sequentially reachable global state space of a protocol can be represented by a digraph, called *sequential*

reachability graph, in which nodes represent global states and arcs stand for the sequential immediate successor relation between global states.

In the algorithm given in Figure 2.3, A is the set of global states that have been checked against logical errors and W is the set of global states to be checked. As pointed out in [Holz92], if W is a stack then the algorithm performs a depth-first search, and if W is a queue then the algorithm performs a breadth-first search. If a breadth-first search is used, all transitions from the current global state are explored, and therefore all the immediate sequential successors of the current global states are generated before other global states analyzed. The global states are analyzed in the order they are generated (or reached). Breadth-first search algorithm guarantees that all global states are reached by one of the shortest paths leading them from the initial global state. Therefore, erroneous global states are discovered by the shortest execution sequences in this search. If the depth-first search is used, erroneous global states are unlikely to be reached by the shortest paths. However, unlike the breadth-first search, the depth-first search algorithm does not need to store information about successor relation between global states to report the paths from the initial global state to the erroneous ones because the global states on the path are currently stored in the stack. In addition, the depth-first search algorithm can minimize the memory requirements at the expense of running time by storing only global states in a single execution path from the initial global state to the current global state. Since such an algorithm cannot determine whether or not a newly generated global state was encountered before in a previously analyzed path, the algorithm may redundantly analyze the same global state several times. Holzmann studies variations of the depth-first search algorithms in [Holz92].

```
A=∅      /* A is the set of global states have already been analyzed */
W={G0}   /* W is the set of global states to be analyzed */
while W≠∅ do
begin
  remove an element Gk from W
  add Gk to A
  if Gk is a deadlock state then report deadlock
  for each transmission (s_i^k,-x,s') ∈ spec(Gk) where x∈Mij and i≠j do
    if |c_ij^k|=Bij then report buffer overflow
  for each c_ij^k (i≠j) do
    if c_ij^k=xX, X ∈ Mij* and δj(s_j^k,+x) is undefined then
      report unspecified reception
  for each Gm such that Gk→Gm
      if (Gm is not in A or W) add Gm to W
end
report each nonexecuted transition as a nonexecutable transition
```

Figure 2.3 Sequential reachability algorithm.

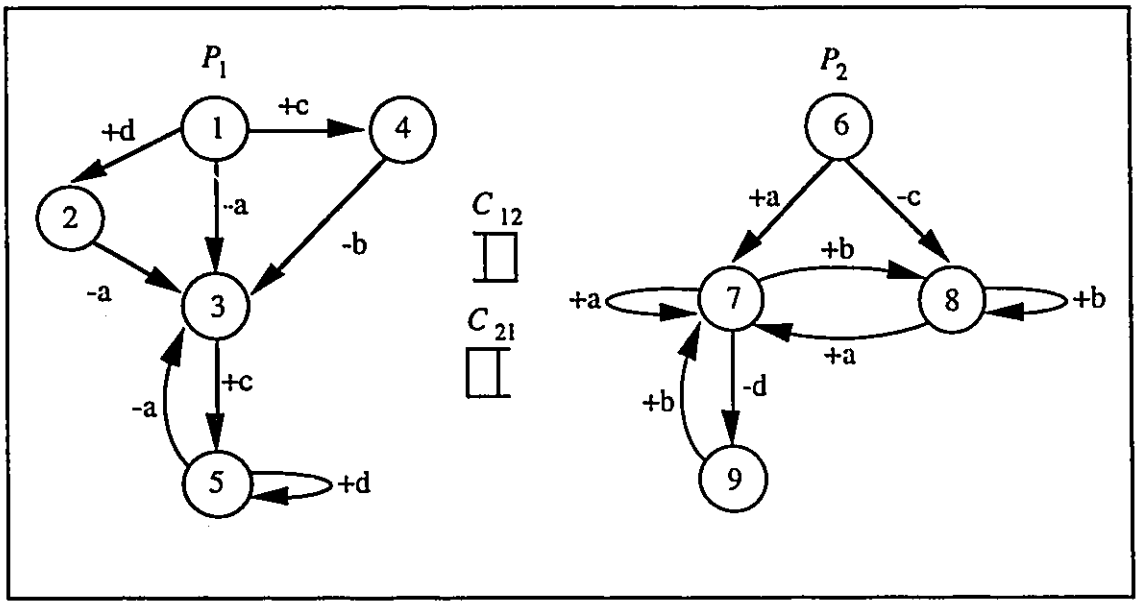


Figure 2.4 A protocol containing various logical errors.

The protocol given in Figure 2.4 includes four types of logical errors. As can be seen from the sequential reachability graph of this protocol given in Figure 2.5, this protocol includes 1 deadlock state, 4 unspecified reception states and 2 buffer overflow states. Since

transitions (1,+d,2) and (2,-a,3) in process P_1 and (7,+b,8) and (9,+b,7) are not executed during the sequential reachability analysis of this protocol, they are nonexecutable transitions of the protocol.

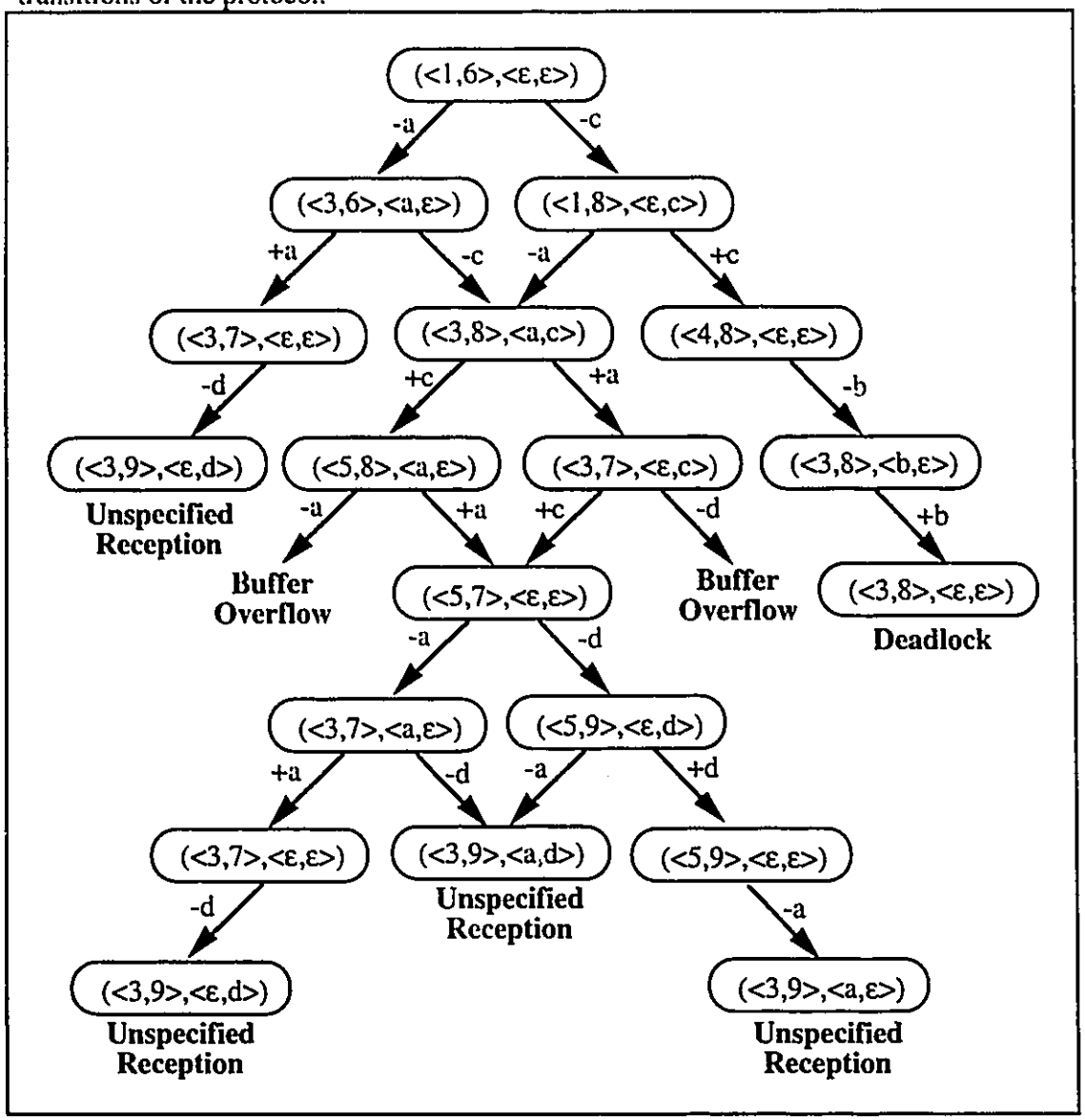


Figure 2.5 The sequential reachability graph of the protocol in Figure 2.4.

Chapter 3

STATE EXPLOSION PROBLEM AND RELIEF STRATEGIES

3.1 State Explosion Problem

Reachability analysis is well-suited for verifying protocols against their logical errors, but it often suffers from *state explosion* problem. It is said that a state explosion occurs when the number sequentially reachable global states of a protocol becomes too large to be enumerated by the available computing resources. To formulate the possible maximum number of sequentially reachable global states of a protocol, the following parameters are defined:

- n is the number of processes in the protocol.
- s is the maximum number of states in a process.
- m is the maximum number of distinct messages sent from one process to another.
- c is the maximum number of messages (slots) in a channel, i.e., the channel capacity.

Let us first find the maximum number of channel states for a channel. Since the maximum number of slots in channel is c and each slot can contain any of m different messages, the

maximum number of channel states including empty channel is given by the following formula:

$$\sum_{i=0}^{c} m^i$$

Since there are n processes and at most $n*(n-1)$ channels in a given protocol, the maximum number of global states is given by the following formula:

$$s^n \left(\sum_{i=0}^{c} m^i \right)^{n*(n-1)}$$

Consider a protocol where n=5, s=10, m=5 and c=2. The maximum number of global states of the protocol according to the above formula can be $10^5*31^{20} \cong 6.72*10^{34}$. In sequential reachability analysis (Figure 2.3), all reachable global states are stored in the memory to recognize the repeated global states and therefore to guarantee the termination of the analysis. Then one may ask whether we can store $6.72*10^{34}$ global states in the memory. Unfortunately, the answer will be negative. In the rest of this section, we will answer the following questions:

- How many global states can be stored in the memory of a typical, available computer ?
- How much time is required to analyze all global states stored in the memory ?

Table 3.1 presents a number of parameters that will be used to answer these questions. The content of this table is adapted from [Holz90]. Let A denote the maximum number of global states that can actually be stored in the memory. Using Table 3.1, $A = mem/state = 10^5$. Assume that the memory is arranged as a balanced tree. Then searching a global state requires at most $time*\log(A)$ seconds. Since the average number of immediate successors of a global state is d, the total number of arcs in the sequential reachability graph is $d*A$.

Since we generate and search each global state pointed by an arc, the total number of searches is equal to the number of arcs in the sequential reachability graph. Then the total searching time is at most $d*time*A*\log(A)$. We can assume that inserting and analyzing a global state do not cost more time than searching the global state. This means that the total time required for searching, inserting and analyzing all global states is at most $(d+1)*time*A*\log(A)$ seconds. Using Table 1, the time required by sequential reachability analysis is calculated as 8.3 minutes. Therefore, one may expect that even for a simple protocol sequential reachability analysis requires more than the available memory. Note that time requirements are reasonable as long as the state space fits in the memory, i.e., the major issue in reachability based verification is to provide memory efficient algorithms.

Table 3.1 Parameters

Symbol	Description	Typical value
mem	Bytes of memory available for the search	10^7 bytes
state	Bytes of memory required to store one global state	10^2 bytes
d	Average number of immediate successors of a global state	2
time	CPU time required to compare two global states	10^4 seconds

3.2 Relief Strategies

Fortunately, the actual number of sequentially reachable global states of a real protocol is usually smaller than the number given by the formula presented the previous section. The more important fact is that most of these global states are redundant for verifying the protocol against the logical errors. This fact has motivated researchers to seek relief

strategies for the state explosion problem for more than one decade. In the remainder of this section, we survey the relief strategies (see also [LiCh87, Yuan88]) related to our work.

3.2.1 Fair Reachability Analysis

3.2.1.1 The Rubin-West Method

The first reduction technique that utilizes the execution of several transitions is proposed by Rubin and West [RuWe82] as a relief strategy for the analysis of two-process protocols. This technique (which is later called *fair reachability analysis* by [YuGo82]) produces and analyzes only *fairly reachable global states* which are reached by executing equal number of transitions in each process. That is, a global state is generated from another global state by executing only one ordering of a pair of transitions, one per each process. The authors only argue that fair reachability analysis can be used to decide whether the communication is free from deadlocks and unspecified receptions.

Fair reachability analysis considers only the behavior of a protocol obtained by equal progress speed of each process. However, a buffer overflow occurs when the sender process is faster than the receiver. Therefore, the fair reachability analysis does not guarantee to detect buffer overflows. In addition, a process can include a group of transitions (or a group of states) which can only be executed (or reached) if one process progresses faster than the other process does. Therefore, a transition not executed (or a process state not reached) during the fair reachability analysis may actually be an executable transition (or a reachable process state).

3.2.1.2 The Yu-Gouda Method

Using fair reachability analysis, Yu and Gouda [YuGo82] propose a deadlock detection algorithm which is polynomial in time and space for a special two-process protocol model. In this model, a process can send (or receive) one type of message and none of its states

has both sending and receiving transitions. Their model is too restricted to be applicable for real communication protocols.

3.2.1.3 The Gouda-Han Method

Gouda and Han [GoHa85] extend the Rubin-West method to decide boundedness for any two-process protocol whose fair reachability graph is finite. An example for the case where the fair reachability graph of a protocol is finite but the protocol is unbounded (i.e., at least one of its channels is unbounded) can easily be demonstrated. Consider the simple two-process protocol in Figure 3.1.

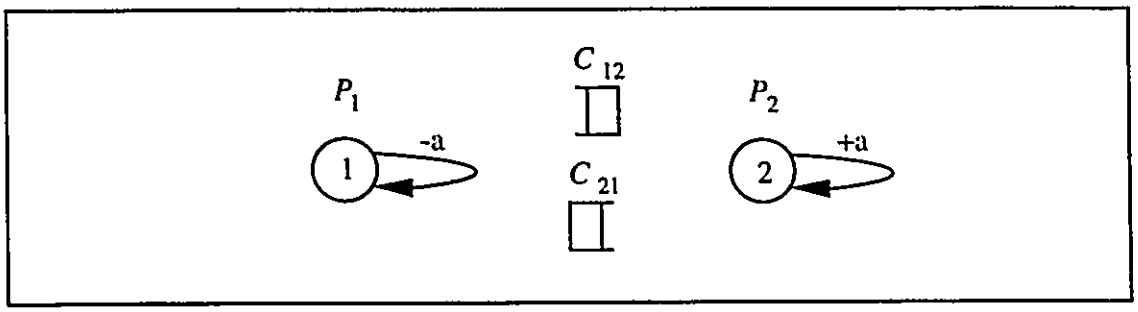


Figure 3.1 A simple protocol with a buffer overflow

The fair reachability graph of the protocol is given in Figure 3.2.

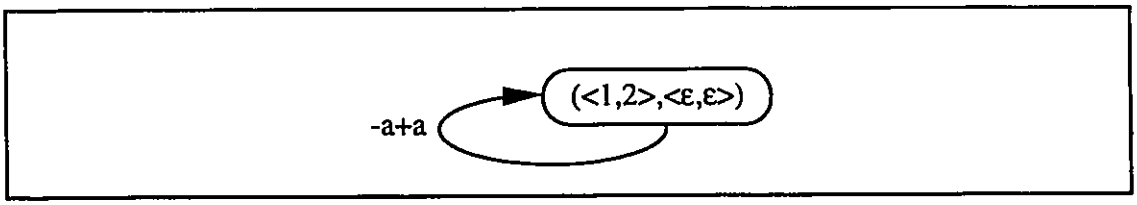


Figure 3.2 Fair reachability graph of the protocol in Figure 3.1.

If the sender (P_1) is faster than the receiver (P_2), the number of messages in the channel from the sender to the receiver will exceed any given bound for the channel after a finite

time. However, this buffer overflow is not detected by the Rubin-West method (see Figure 3.2).

Gouda and Han first present an algorithm to decide whether a process state is reachable. Consider the protocol partially given in Figure 3.3 where states 1 and 5 are the initial states of processes P_1 and P_2, respectively. By the execution of transition sequence (1,-a,2)(2,-c,3) during sequential reachability analysis, the protocol reaches global state (<3,5>,<ac,ε>). Since state 3 is contained in a reachable global state, it is a reachable process state of the protocol. However, the fair reachability graph of this protocol does not include a global state containing state 3. This can be seen from Figure 3.4. To find reachable states of a process, the first algorithm given in [GoHa85] augments the fair reachability graph of a given protocol for this process. This augmentation is performed by adding global states which are sequentially reachable by the sole progress of this process from an unspecified reception state. A process state is decided as an unreachable process state if the corresponding augmented fair reachability graph does not include a global state containing this process state. For example, global state (<2,6>,<a,b>) in Figure 3.4 is an unspecified reception state (because the reception of message a (b) is not specified at process state 6 (2)). When process P_1 solely executes transition (2,-c,3) at global state (<2,6>,<a,b>), the protocol reaches global state (<3,6>,<ac,b>) in the augmented fair reachability graph. Therefore, it is decided that state 3 is reachable from the augmented fair reachability graph.

Gouda and Han also prove that the outgoing channel of a process is unbounded in a protocol whose fair reachability algorithm is finite iff the finite state machine for the process includes a reachable state in a directed cycle of all sending transitions. Therefore, the second algorithm given in [GoHa85] finds each directed cycle of all transmissions in the finite state machine and checks whether the cycle includes a state which is decided as a reachable local state by the first algorithm. Finally, they present an algorithm which

augments the finite reachability graph of a protocol for each process to determine the possible smallest channel capacity of each channel. For the outgoing channel from of a process, the algorithm adds global states that can be reached in the fair reachability graph by only using transitions of this process. With the contribution of Yu and Gouda, fair reachability analysis can be used to detect all logical errors in two-process protocols.

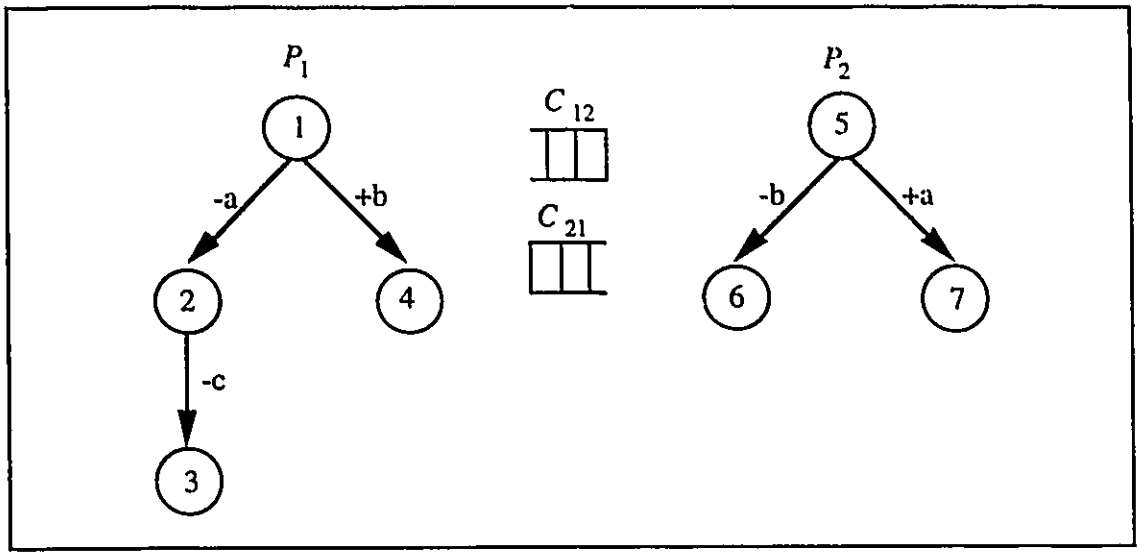


Figure 3.3 A two-process protocol.

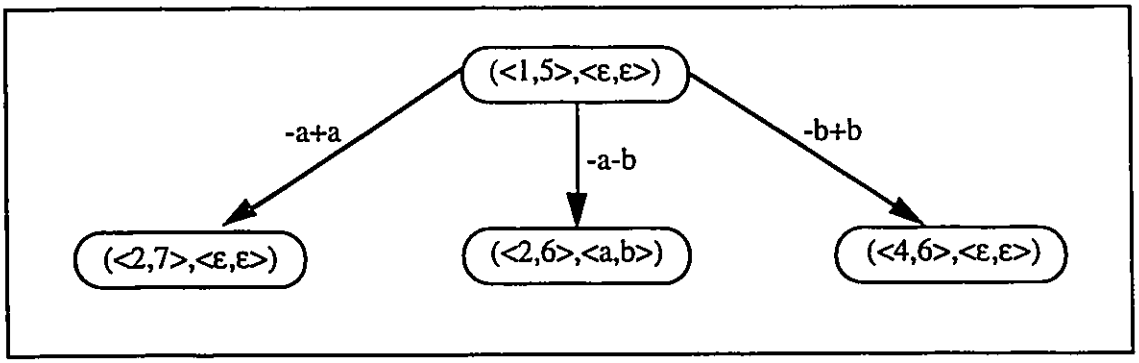


Figure 3.4 The fair reachability graph of the protocol given in Figure 3.3.

3.2.1.4 The Zhao-Bochmann Method

Zhao and Bochmann [ZhBo86] use an approach similar to fair reachability analysis using a different representation of the CFSM model. They represent protocols with process equations and reachability analysis with algebraic transformation rules. The proposed reachability algorithm utilizes both sequential and fair global state transitions to generate next global states. They prove that their method detects deadlocks and blocking receptions that are unspecified receptions occurring only at receiving process states (where all specified transitions are receptions). They also extend their method to detect unspecified receptions. However, the generalization of this method to protocols including more than two processes remains open.

3.2.1.5 The Liu-Miller Method

In [LiMi94] Liu and Miller generalize fair reachability analysis for n-process ($n > 2$) protocols with a unidirectional ring topology. Due to their special topology, such protocols are called *cyclic*. In a cyclic protocol, each process has only one input channel and one output channel. Notice that any 2-process protocol is a cyclic protocol and if $n > 2$, none of the processes in the cyclic protocol can directly send to and receive from the same process. They observe that at every fairly reachable global state for a two-process protocol, every channel has the same number of messages. They call this property the *equal channel length property*. Their generalization is based on preserving the equal channel length property for cyclic n-process protocols. Although the generalization does not preserve the equal progress property of fair reachability analysis, for protocols with $n = 2$ the equal channel length property leads to the same reduced reachability graph as the equal progress property.

In generalized fair reachability analysis, global states are generated by executing vectors of transitions called *fair progress vectors*. A vector of transitions (or transition vector) consists of one transition from each process. There are tree types of transitions that can be

included in transition vector: *executable*, *enabled* and *null*. While executable and null transitions are defined in the conventional manner, an enabled transition is defined as follows: An enabled transition is a receiving transition which is not executable at the current global state but it is executable at least at one of the immediate sequential successors of the current global state. Actually, there is only one possible case for a receiving transition of message x is enabled at a global state, which occurs if the sender process (of message x) is ready to send message x and the channel (in which message x will be in transit) is empty at the global state. A transition vector includes a null transition from a process only if there is no executable or enabled transition from the process at the current global state, otherwise it includes an executable or enabled transition. There are two types of fair progress vectors: *concurrency* and *synchronization*.

A concurrency vector is a transition vector in which all the transitions are either sending or receiving executable transitions. A synchronization vector contains k pairs of a sending and a receiving transition and n-k null transitions such that both transitions of a pair involve the same channel, at least one sequence (or order) of transitions of a pair is executable, $k > 0$, and the vector is maximal with respect to k. Obviously the sending transition of a pair in a synchronization vector must be executable whereas the receiving transition of the pair can be either enabled or executable. Notice that executions of both concurrency and synchronization vectors preserve the channel length property because the execution a concurrency vector of sending (receiving) transitions increase (decrease) the length of every channel by one and the execution of synchronization vector does change the length of any channel. Clearly at the initial global state and at every global state generated by executing a sequence of fair progress vectors from the initial global states, the equal channel length property holds.

A cyclic protocol is *simultaneously unbounded* if for any constant $K \geq 0$, there exists a reachable global state in which the length of every channel is greater than K, otherwise it is

not simultaneously unbounded. Liu and Miller report that a given cyclic protocol has a finite fair reachability graph iff the protocol is not simultaneously unbounded. They also report that deadlock and livelock detections are decidable for the class of cyclic protocols with finite reachability graphs. All these results are based on the theorem (Theorem 5.1 in [LiMi94]) stating that "any (sequentially) reachable global state with equal channel length is fair(ly) reachable". However, this theorem is contradictory with the definition of fair progress vectors because the execution of a synchronization vectors causes to skip some global states with equal channel length. We will present a cyclic protocol for which generalized fair reachability analysis does detect deadlocks because a synchronization vector causes to skip some global states with equal channel length.

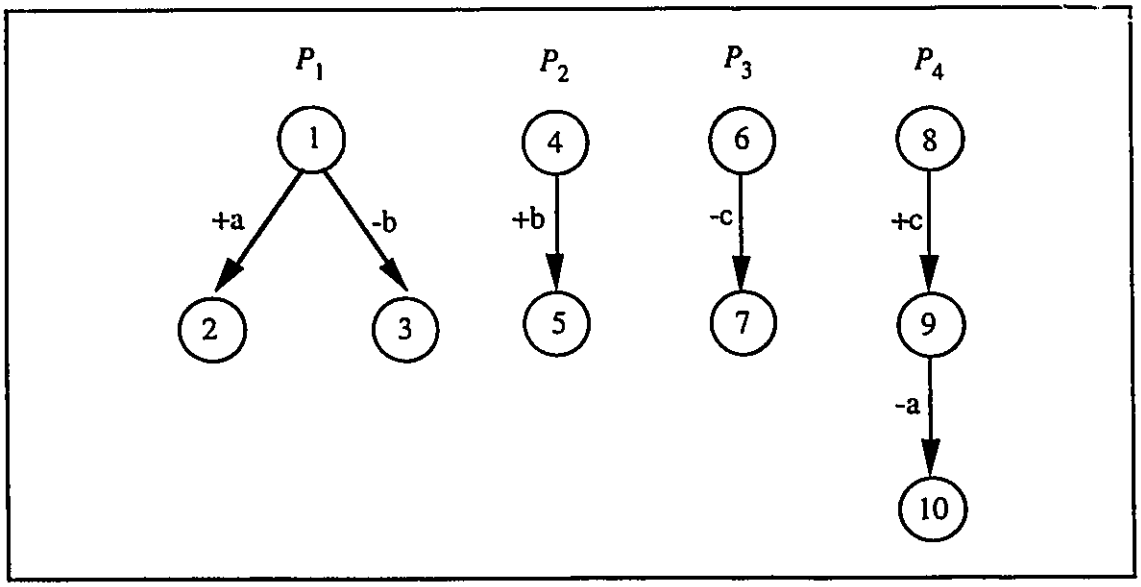


Figure 3.5 A cyclic protocol with a deadlock state

Consider the cyclic 4-process protocol is given in Figure 3.5. A global state G_k of this protocol is represented as a pair of two 4-tuple: $G_k = (\langle s_1^k, s_2^k, s_3^k, s_4^k \rangle, \langle c_{12}^k, c_{23}^k, c_{34}^k, c_{41}^k \rangle)$. This protocol has only one deadlock state which is $(\langle 2,4,7,10 \rangle, \langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle)$. This deadlock

state can sequentially be reached by executing transition sequence (6,-c,7)(8,+c,9)(9,-a,10)(1,+a,2) from the initial global state (<1,4,6,8>,<ε,ε,ε,ε>) (see Figure 3.6).

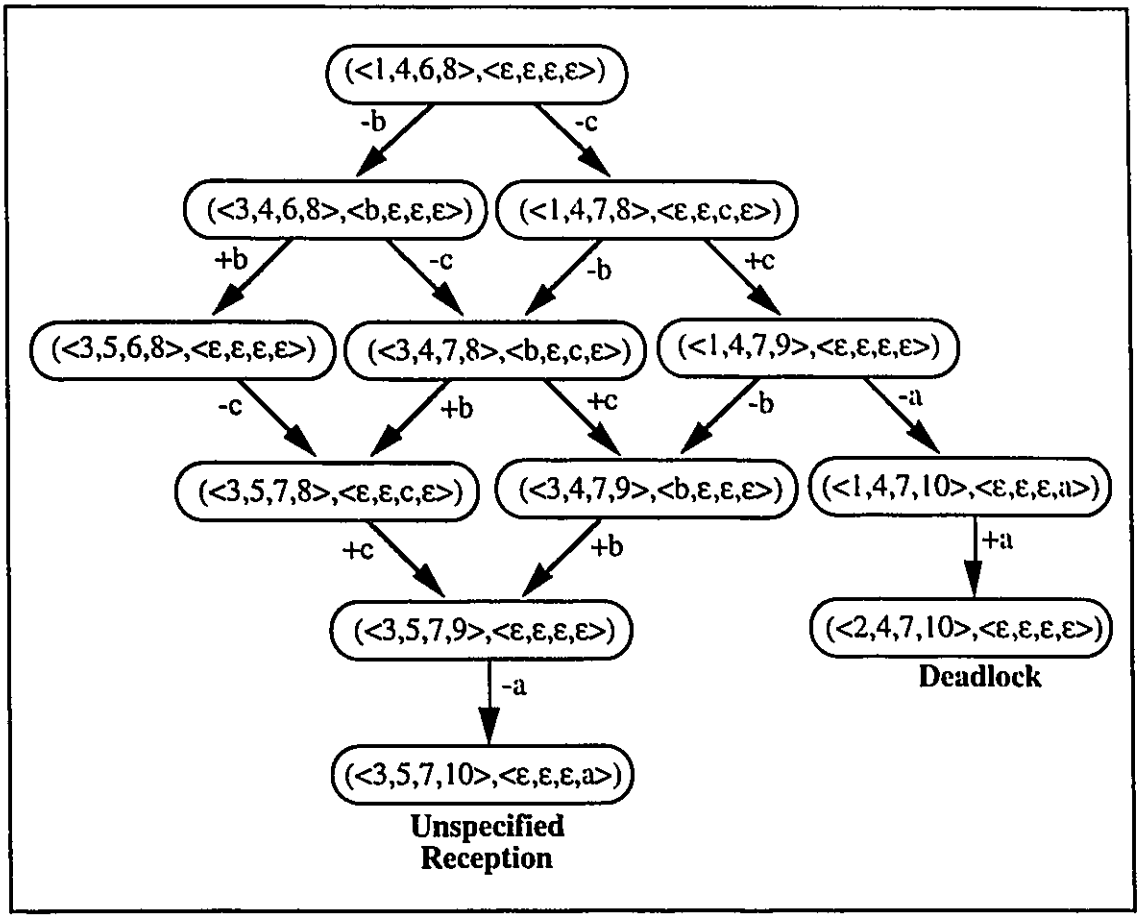


Figure 3.6 The sequential reachability graph of the protocol in Figure 3.5.

At the initial global state, transitions (1,-b,3) and (6,-c,7) are executable and transitions (4,+b,5) and (8,+c,9) are enabled. According to the definition of transition vectors, there is only one transition vector at the initial global state, which is ((1,-b,3),(4,+b,5),(6,-c,7),(8,+c,9)). This vector in fact is the only fair progress vector can be executed in the fair reachability graph of the protocol (see Figure 3.7) and the fair reachability graph does not include deadlock state (<2,5,7,10>,<ε,ε,ε,ε>). Notice that the execution of the

synchronization vector causes to ignore global states (<3,5,6,8>,<ε,ε,ε,ε>) and (<1,4,7,9>,<ε,ε,ε,ε>) with equal channel length.

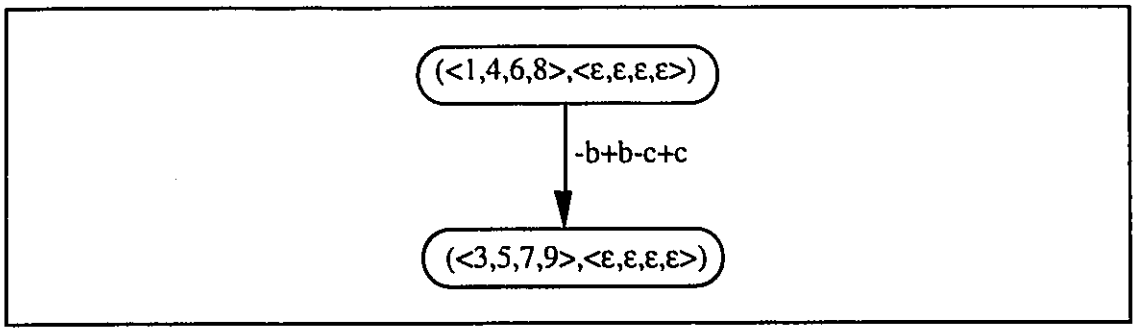


Figure 3.7 The fair reachability graph of the protocol in Figure 3.5.

3.2.2 Maximal Progress State Exploration

A technique called *maximal progress state exploration* is proposed by Gouda and Yu [GoYu84] for detecting deadlocks, blocking receptions and buffer overflows in two-process protocols specified by the CFSM model. The exploration is divided into two independent subtasks. In each subtask, only the global states reachable by forcing one process to make maximal progress are generated.

In the sequential reachability analysis, all possible progress speeds of processes are considered. However, for two-process protocols, considering only the case where one process progresses in maximal speed as in this method is sufficient to find deadlock and blocking receptions. Since deadlock and blocking receptions are the cases where a process cannot progress any more, they can be detected by considering maximal progress of each process independently. In addition, buffer overflows is also detected by maximal progress exploration because a buffer overflow occurs when one process transmits messages faster than the other process consumes them. Maximal progress state exploration may not be as efficient as fair reachability analysis in [RuWe82] but it detects also buffer overflows and

its independent tasks can be simultaneously executed to reduce the execution time. The generalization of this method to protocols consisting of more than two processes remains open.

3.2.3. Decomposing Process Graphs (FSMs) into Unilogues

Duologue Matrix analysis [Zafi78] is one the first automated protocol validation techniques. This technique is proposed for validating two-process protocols where the only loops in a process pass through the initial state of the process. A loop starting and terminating at the initial state is called a *unilogue*. Each process is viewed as a finite set of unilogues. A pair of unilogues, one from each process, forms a *duologue*. In this analysis, all possible duologues are analyzed and classified as *well-behaved*, *non-occurrable* or *erroneous*.

A duologue is well-behaved if all messages sent by one unilogue are received by the other and the duologue always terminates, i.e., processes return to their initial states. A duologue is non-occurrable if it can either never start or if any attempt to execute it results in the execution of another duologue. Finally, a duologue is erroneous if it is neither well-behaved nor non-occurrable. Let m_i (i=1,2) denote the number of unilogues in process P_i. Then all duologues is represented as a duologue matrix, denoted as D, of size $m_1 m_2$ such that its element d_{ij} is the duologue formed from ith and jth unilogues of P_1 and P_2, respectively. A validation function, *val* , is introduced as follows:

$$val(d_{ij}) = \begin{cases} +1 & \text{if } d_{ij} \text{ is well - behaved} \\ 0 & \text{if } d_{ij} \text{ is non - occurrable} \\ -1 & \text{if } d_{ij} \text{ is erroneous} \end{cases}$$

A validation matrix V is obtained from duologue matrix D and validation function *val* such that $v_{ij} = val(d_{ij})$. A protocol contains a logical error if its validation matrix V contains any -1 elements. The positions of the -1 elements in V identify the erroneous duologues of the protocol. If ith row (jth column) of V contains more than one +1 element,

then the behavior of P_2 (P_1) is ambiguous with respect to P_1 (P_2) when traversing ith (jth) unilogue. This technique was applied to the CCITT X.21 Recommendation [WeZa78]. Its application showed that the technique was useful for finding errors in a real protocol.

Zafiropulo [Zafi78] showed that the unilogues of a process can be obtained from repeated set theoretical matrix multiplications of the transition matrix representation of this process. Then, duologues are found by considering all pairs of unilogues, one from each process. A shortcoming of Duologue Matrix analysis is that it can not analyze the race conditions during the execution of a duologue, i.e., all possible execution sequences of the pair of unilogues. All possible execution sequences corresponding to a duologue are checked against the error conditions using phase diagrams proposed later in [West78b].

Itoh and Ichikawa [ItIc83] extended Zafiropulo's two-process protocol model [Zafi78] to specify n-process protocols, $n \geq 2$, and introduced so-called *reduced implementation sequences* to reduce the size of the global state space analyzed during reachability analysis. In this model, processes must synchronize on their initial states after a finite number of transitions and only the cycles passing through the initial states are allowed in each process. The verification method in [ItIc83] analyzes only the behavior of the protocol between two synchronization points. Since the number of states in each process is finite, there is no embedded cycle in each process, and the processes synchronize on their initial states, the number of global states are always finite.

In the reduced reachability analysis, a transition at a global state is said to be an *admissible event* if (1) it is a transmission or (2) it is a reception of a message from a channel and the first message in the channel is this message. A reception from a channel at a global state is called *potentially admissible event* if the channel is empty at the global state. In the reduced reachability analysis, at a global state, the admissible events of distinct processes are executed simultaneously to generate the next global states. If there is some

potentially admissible event in the current state of a process, additional global states are also generated by inhibiting the execution of all admissible events of this process. The purpose of the additional global states is to enable the later execution of potentially admissible events. A transition sequence generated from the initial global state to a terminating global state is called a *reduced implementation sequence*. Terminating global states are (1) deadlock states, (2) the global states blocked by some unspecified reception, i.e., where no admissible event exists but some channel is not empty, (3) the global states where all processes are at their initial states (some channel may not be empty).

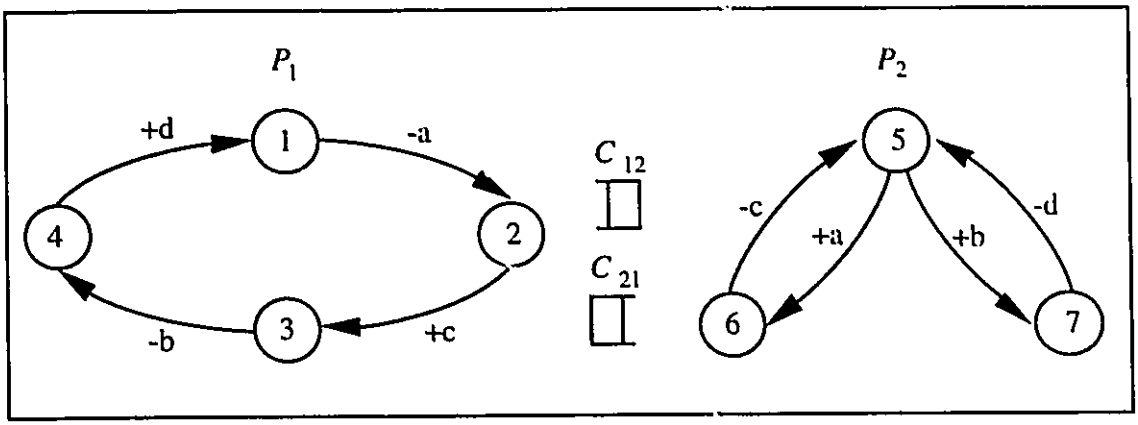


Figure 3.8 A two-process protocol.

They report an application of their technique to a four-process protocol, which results in a much shorter computation time and requires much less space than that required for the sequential reachability analysis. However, the protocol model has the same restriction as Zafiropulo's two-process model except the number of processes. As pointed out by West [West78b], excluding embedded loops from process graphs limits the application of the validation technique to real protocols. Even simple data transfer protocol such as alternating bit protocol [BaSc69] includes embedded loops. In addition, duologues or multilogues correspond to a concurrent behavior of protocols where their unilogues are traversed only once. Consider the simple two-process protocol given in Figure 3.8 which does not contain

any embedded loop. When process P_1 executes one unilogue, process P_2 executes two unilogues. This protocol does not contain any logical errors. However, the duologue matrix analysis or the reduced reachability analysis indicates an error for this protocol. An error is indicated because P_2 returns to the initial state without receiving all messages sent by P_1.

3.2.4 Acyclic Expansion of Protocols

Brand and Zafiropulo [BrZa83] propose a technique for detecting (missing pairs of a process state and a message causing) unspecified receptions, deadlocks and nonexecutable transitions. In this technique, they do not generate global states of a given protocol specified in the CFSM model. Instead, a tree protocol is constructed by transforming each process graph of the protocol into a tree (acyclic form) by tracing all possible execution sequences. In general, there is an infinite number of tree protocols for a given protocol, depending on how long we trace the executions. In addition, determining when the tree growth can be terminated is undecidable. Therefore, a transformation procedure that is guaranteed to terminate the tree growth if all channels are bounded is described in [BrZa83]. During this transformation, each repeated process state and transition are renamed so that all states and messages in different paths of the tree are distinct.

This technique is extended for detecting buffer overflows by Kakuda, Wakahara and Norigoe [KaWa88]. They extend the termination procedure given in [BrZa83] to obtain a smaller tree protocol. Recently, Yaung and Kershenbaum introduce a parallel version of this technique [YuKe90]. This is a two-phase algorithm. In the first phase trees are constructed while in the second phase error detection is performed. Each phase is applied to each process simultaneously and independently. They also extend termination procedure in [KaWa87] to prevent redundant constructions of trees by using and storing extra information.

The advantage of this technique is that the tree protocol version of a given protocol can be smaller than the sequential reachability graph of this protocol. However, the algorithm of this technique is much more complicated than the algorithm of the sequential reachability analysis. Since the global states are not generated, in order to obtain the global information three functions are defined for each process state s: $From_i(s)$, $To_i(s)$ and $L_i(s)$. $From_i(s)$ and $To_i(s)$ define the last message received from process P_i and the last message sent to process P_i, respectively, on the path from the initial state to s in P_i. $L_i(s)$ defines the last state process P_i that must have been reached before s can be reached. Realization of these functions requires complex tree searching procedures. The size of the tree protocol mostly depends on the termination produce. However, the proposed termination procedures do not guarantee to obtain the smallest tree protocol or a tree protocol whose size smaller than the size of the sequential reachability graph of the protocol. In addition, it is not known that for which type(s) of protocols this method provides a significant reduction in terms of memory requirements.

3.2.5 Partial Searches

It is known that the size of available memory is not sufficient for sequential reachability analysis to complete the search of the state space of a realistic protocol [Holz87a]. Let A and R represent the maximum number of global states that can be stored in an available memory and the number of all sequentially reachable global states, respectively. If $A < R$ then after searching "the first" A global states sequential reachability analysis halts by reporting that it run out of memory. In this case one can expect that the probability of finding any given error by sequential reachability analysis is A/R. However, in the case of $A < R$, the objective of a partial search strategy is to select (generate) A global states in such a way that the probability of finding any given error is better than A/R. Therefore, a subset of the sequential immediate successors of a global state is generated according to a selection criteria. The following srategies are major partial search strategies.

3.2.5.1 Scatter Search

Holzmann [Holz85b, Holz87b] propose a partial search strategy which is a depth-first search guided by some heuristics such as (1) not exploring all interleavings, (2) assigning priorities among concurrent transitions and (3) limiting channel capacities. For the first heuristic, the following example can be given: If two processes can concurrently execute two transitions, say a and b, only one of the sequence of ab and ba is explored. The second heuristic can be used to quickly detect deadlocks by giving priority to receive operations over send operations. Notice that at deadlock states, all channels are empty and therefore giving priority to receive operations over send operations guides the search to the deadlock states. Holzmann notes that unlike the breadth-first search, the depth-first search algorithm need only contain the global states in a single execution path from the initial global state to the current global state. Since such an algorithm can not determine whether or not a newly generated global state was encountered before in a previously analyzed sequence, the algorithm may analyze the same global state more than once. Therefore, a depth-first reachability algorithm attempts to minimize memory usage at the expense of running time by storing only global states in a single execution path from the initial to current global state. However, the scatter search can not demonstrate that a given protocol is free from a logical error, because none of its heuristics guarantees that the generated subset of all sequentially reachable global states contains any given error..

3.2.5.2 Random-Walk State Exploration

West [West 86] proposed a modified sequential reachability analysis, called *random-walk state exploration*, for detecting logical errors in complex protocols. In this technique, one of the sequential immediate successors of a global state is randomly generated and the generated global states are not stored. Therefore, the global states are explored in the form of a continuous sequence starting from the initial global state. During the analysis, the most

recently generated transitions are stored so that the exact sequence leading to an error can be constructed. If an error is detected, then the analysis is restarted from the initial global state.

The following observations are the main motivation of this technique. When a complex protocol is validated using a sequential reachability analysis, the majority of the errors is found many times over in different global states of the protocol. In general, the analysis of one sequence leading to a particular error is sufficient to identify the cause of the error and to correct it. These suggest that an analysis like a random-walk state exploration may be sufficient to identify a significant fraction of the errors without storing any generated global state.

However, there are two major disadvantages. First, this analysis has no well-defined termination. Second, since a subset of all global states is randomly generated, the analysis can not demonstrate that a given protocol does not include a logical error.

3.2.5.3 Probabilistic Search

In this technique [MaSa87], it is assumed that each transition in a protocol has a fixed probability of occurrence classified as "high" and "low". Using these two probability tags, the most probable execution sequences are analyzed. As indicated in [Rudi88, Holz90], the main difficulty of this technique is the estimation of the probabilities of transitions. Again, this technique can not verify the absence of a logical error in a protocol.

3.2.5.4 Supertrace

Holzmann [Holz88, Holz90] observed that when a protocol is large, more space is needed to store the global state descriptions and more time is needed to compare and analyze these global states. Storing more global states in a randomly accessed memory by reducing the global state description is the main objective of supertrace algorithm. The memory is considered as a bit array. Each generated global state is hashed into an index of this array.

If the corresponding bit in the array is on, then the algorithm considers that the generated global state has already been analyzed. Note that a collision may occur in the algorithm, i.e., two different global states might be hashed into the same bit of the memory. In that case, the last hashed global state will never be analyzed by supertrace algorithm even it is an erroneous global state. Since collision detection is ignored, each global state is stored in one bit in the randomly accessed memory and therefore, more global states can be stored and less time is required to search and compare global states. Many large systems can be analyzed by this technique. However, due to collisions, supertrace does not guarantee to analyze all reachable global states and to detect all logical errors.

3.2.6 Decomposition of Structured Protocols

Voung and Cowan observe that large, well-designed protocols often exhibit some structures which allow them to be decomposed into a number of smaller, more manageable components which can be validated independently to achieve the equivalent result of validating the entire protocol [VuCo82]. They represent protocols by the CFSM model and define three protocol structures for two-process protocols: *nested*, *sequential* and *parallel*.

A protocol exhibits a nested structure if a number of states in a process can be combined into a state called the *superstate*. The states inside the superstate are called *internal* states while the states of the process outside the superstate are called *external* states. There are some conditions that must be satisfied to construct a superstate, for example: the internal states must be strongly connected; if there is a transition from an internal state to an external state with action a then there must be a transition from every internal state to this external state with action a; the superstate must have the initial state of the process and no incoming transition.

A protocol exhibits a sequential structure if each process consists of two parts which are connected by only one state called the *synchronized* state and a process can reach its

second part only when all the processes are at their synchronized states. Then the protocol can be partitioned into two subprotocols such that in one protocol, each process ends with its synchronized state, in the other protocol, each process starts with its synchronized state.

A protocol exhibits a parallel structure if each process has both sending and receiving transitions at the initial state and contains no cycle which leads back to the initial state. Then the protocol can be decomposed into three subprotocols such that each process starts with sending transitions from its initial state in the first subprotocol, and one process starts with sending transitions and the other process start with receiving transitions from their initial states in the second and third subprotocols. Decomposing a protocol into smaller components by this technique may greatly simplify the validation of the protocol but the technique is applicable to only two-process protocols exhibiting the particular structures.

Choi and Miller present also a decomposition technique for protocol validation (and synthesis) [ChMi83, ChMi86]. Unlike the technique used by Vuong and Cowan, this technique is not based on certain protocol structures. They show that the technique either yields a maximal decomposition or reports that the protocol can not be decomposed. However, this decomposition technique is also limited to two-process protocols.

3.2.7 Pumping-Based Protocol Validation Technique

In [TaHo91], K.C. Tai, H.F. Ho and G.H. Chen propose a protocol validation technique using the pumping theorem for a finite state machine. The pumping theorem states that if xyz is a sentence of the language L accepted by a finite state machine M and, x and xy reach the same state from the initial state of M, then for $t > 1$, xy^tz is also a sentence of L. In order to use the pumping theorem, the reachability graph of a protocol is observed as a finite state machine by assuming that every global state in the reachability graph is a terminal state.

First, two sufficient conditions are presented such that if a transition sequence xyz of a protocol (i.e., xyz is executable from the initial global state of the protocol) satisfies any one of these conditions then for $t > 1$, xy^tz is also a transition sequence of the protocol. Then two more sufficient conditions are presented such that if a transition sequence xyz of a protocol which does not lead to a deadlock state (or a global state blocked by some unspecified receptions) satisfies any one of these conditions then for $t > 1$, xy^tz is also a transition sequence of the protocol which does not lead to a deadlock state (or a state blocked by some unspecified receptions). The main motivation of this approach is to detect deadlocks and blocking unspecified receptions of a large set of transition sequences (e.g., xy^tz, $t > 1$) by examining only a small subset of transition sequences (e.g., xyz).

In other words, the motivation is to find an upper bound on the depth of the reachability graph to be sufficient to detect deadlocks and blocking unspecified receptions (i.e., the length of the longest xyz). Having stated that it is difficult to find the length of the longest xyz, they provide an algorithm to find an upper bound which is guaranteed to be greater than the length of the longest xyz. To find an upper bound on the depth of the reachability, the algorithm takes the summation of the lengths of ys in all xyzs and the product of the numbers of states in the processes of the protocol. However, the bound found by the algorithm will likely be very large for protocols with n processes when n is greater than 2 because of the product of the numbers of states in the processes. Therefore, a reachability graph with this bound can be too large to be stored in memory.

Even though this pumping-based approach is significantly different from other protocol validation techniques, the provided algorithm for finding an upper bound on the depth of the reachability graph needs to be improved. In addition, nonexecutable transitions and buffer overflows are also important errors in protocol validation. This study does not mention how the approach can be used to detect these errors.

3.2.8 Reverse Reachability Analysis

Hung and Chen [HuCh93] present a technique for deadlock detection in protocols. They use the CFSM model to represent protocols. Their method is based on generating all reverse reachable paths starting from each possible deadlock state. If any of these reverse reachable paths can reach the initial global state, then the protocol includes a deadlock, otherwise the protocol is deadlock-free. They present experimental results showing that their method is better than sequential reachability analysis.

Holzmann uses a similar idea in [Holz85a] to detect deadlocks and unspecified receptions. However, Holzmann claims that sequential reachability analysis is better than the reverse reachability analysis. It is known that unreachable global state space is much larger than the reachable one. Since some of the potential deadlock states are unreachable, reverse reachability analysis likely explores unreachable global states. Therefore, Holzmann's claims is realistic. Moreover, the detection of nonexecutable transitions and buffer overflows can not be efficiently performed by reverse reachability analysis.

Chapter 4
SIMULTANEOUS REACHABILITY ANALYSIS

As in other models such as CSP [Hoar78], CCS [Miln80] and Petri nets [Pete77], each process is a sequential machine in the CFSM model, namely, a process can execute a single transition at a time. However, in the joint global behavior, these processes concurrently execute their transitions. In other words, several transitions each of which is from a distinct process may be executed at the same time in a protocol represented by the CFSM model. The set of such simultaneously executable transitions will be called a *simultaneously executable set*. Simultaneous reachability analysis proposed in this chapter utilizes simultaneously executable sets to reduce the global state space of a protocol to be analyzed. We will prove that a) simultaneous reachability analysis identifies every deadlock and every nonexecutable transition, b) by augmenting a given protocol with a very limited number of receiving transitions, simultaneous reachability analysis identifies every missing receiving transition causing an unspecified reception, c) by augmenting simultaneous reachability analysis, every channel which is overflowed is identified in a given protocol.

4.1 Equivalent Transition Sequences

In this section we define an equivalence relation between transition sequences, and present some properties of equivalent executable transition sequences. These properties will be used in the proofs of lemmas and theorems in the subsequent sections.

Definition 4.1.1 A sequence of transitions (or a transition sequence) $\sigma (=t_1 t_2 \ldots t_j)$ is *executable* iff $\exists G_k \ (G_k \text{—}\sigma\text{→}^* G_m)$. ||

Let $\sigma=t_1 \ldots t_j$, $\omega=t_1 \ldots t_i$ and $\mu=t_{i+1} \ldots t_j$ be transition sequences. Then $\sigma=\omega\mu$ where juxtapositions are used for concatenations and the length of σ (denoted by $|\sigma|$) is j. If $j=0$ then $\sigma=\varepsilon$, $\omega=\varepsilon$ and $\mu=\varepsilon$. If $i=0$ then $\omega=\varepsilon$. If $i=j$ then $\mu=\varepsilon$. If $\sigma=\omega\mu$ then ω is said to be a *prefix* of σ. The set of all prefixes of σ is denoted by *prefix*(σ). If σ includes a transition from *spec*(P_i) then P_i is said to be *active on* σ. We use *act*(t), *act*(σ) and *act*(T) to denote the set of all processes active on a transition t, a transition sequence σ, and a set T of transitions, respectively.

- $act(t)=\{P_i\}$ where $t \in spec(P_i)$
- $act(\sigma)=\bigcup_{i=1}^{j} act(t_i)$ where $\sigma=t_1 t_2 \ldots t_j$
- $act(T)=\bigcup_{i=1}^{j} act(t_i)$ where $T=\{t_1, t_2, \ldots, t_j\}$

$\sigma\downarrow_i$ will denote the *projection* of σ onto P_i, i.e. a sequence of transitions obtained by removing all transitions not in *spec*(P_i) from σ. Notice that, if P_i is active on σ (i.e. $P_i \in act(\sigma)$) then the projection of σ onto P_i is not an empty sequence.

Definition 4.1.2 Two transition sequences σ and ω are *equivalent*, denoted by $\sigma \equiv \omega$, iff $\forall P_i \in P \ (\sigma\downarrow_i = \omega\downarrow_i)$. []

The following properties are immediate from Definition 4.1.2

Property 4.1.1 $\forall(\sigma,\omega) \ (\sigma \equiv \omega \Rightarrow |\sigma| = |\omega|)$.

Property 4.1.2 $\forall(\sigma,\omega) \ (\sigma \equiv \omega \ \wedge \sigma=t_1 t_2 \ldots t_j \ \wedge \omega=t'_1 t'_2 \ldots t'_j \Leftrightarrow \exists\pi \ (\pi:\{1,2,\ldots j\} \rightarrow \{1,2,\ldots j\} \wedge \pi$ is a one to one and onto function $\wedge \forall i \ (1 \le i \le j \Rightarrow t_i = t'_{\pi(i)})))$.

The following lemma states that two equivalent executable transition sequences from a global state lead to a common global state.

Lemma 4.1.1

$$\forall(G_k,\sigma,\omega)\ (G_0 \rightarrow^* G_k \wedge G_k \text{—}\sigma\rightarrow^* G_l \wedge G_k \text{—}\omega\rightarrow^* G_m \wedge \sigma \equiv \omega \Rightarrow G_l=G_m).$$

Proof: From the definition of equivalent transition sequences, for each process P_i, projections of σ and ω onto this process are the same. This directly implies that $\forall$ $(i=1,2,...,n)$ $(s_i^l=s_i^m)$. Also, for each channel C_{ij}, the same sequence of messages is sent to and received from the channel during the execution of both σ and ω. This implies that $\forall$ $(i,j=1,2,...,n)$ $(i\neq j \Rightarrow c_{ij}^l=c_{ij}^m)$. □

Using Lemma 4.1.1, $\sigma_k \equiv_r \omega$ will be used when $G_k \text{—}\sigma\rightarrow^* G_r$, $\sigma\equiv\omega$ and $G_k \text{—}\omega\rightarrow^* G_r$. The following properties are immediate from the result of Lemma 4.1.1.

Property 4.1.3 $\forall(G_k,\sigma)\ (G_0 \rightarrow^* G_k \wedge G_k \text{—}\sigma\rightarrow^* G_r \Rightarrow \sigma_k \equiv_r \sigma)$.

Property 4.1.4 $\forall(G_k,\sigma,\omega,\mu)\ (G_0 \rightarrow^* G_k \wedge \sigma_k \equiv_q \omega \wedge G_q \text{—}\mu\rightarrow^* G_r \Rightarrow \sigma\mu_k \equiv_r \omega\mu)$.

Property 4.1.5 $\forall(G_k,\sigma,\omega,\mu)\ (G_0 \rightarrow^* G_k \wedge G_k \text{—}\mu\rightarrow^* G_q \wedge \sigma_q \equiv_r \omega \Rightarrow \mu\sigma_k \equiv_r \mu\omega)$.

Property 4.1.6 $\forall(G_k,\sigma,\omega,\mu,\rho)\ (G_0 \rightarrow^* G_k \wedge \sigma\mu_k \equiv_r \omega \wedge \sigma_k \equiv_q \rho \Rightarrow \rho\mu_k \equiv_r \omega)$.

The following lemma states that both interleavings of two executable transitions lead to the same global state (see Figure 4.1 for a graphical illustration).

Lemma 4.1.2

$$\forall(G_k,t,t')\ (G_0 \rightarrow^* G_k \wedge G_k \text{—}t\rightarrow G_l \wedge G_k \text{—}t'\rightarrow G_m \wedge act(t)\cap act(t')=\varnothing \Rightarrow$$
$$\exists G_r(tt'_k \equiv_r t't)).$$

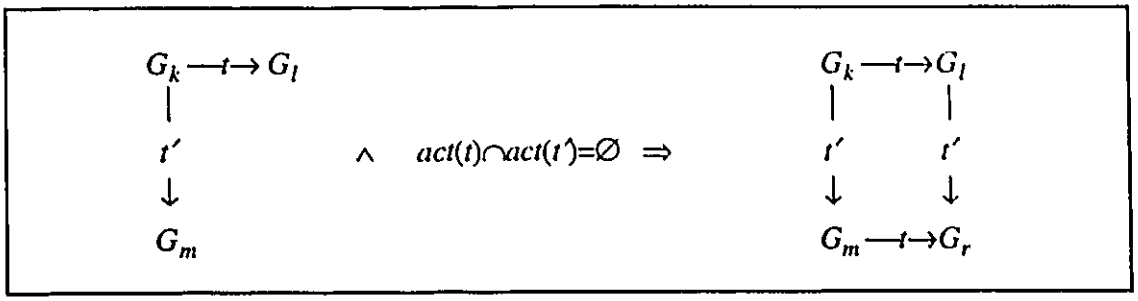


Figure 4.1 Illustration of Lemma 4.1.2.

Proof: Since $act(t) \cap act(t') = \emptyset$, the execution of t (t') may effect the execution $t'(t)$ iff their execution involve the same channel. Since $act(t) \cap act(t') = \emptyset$, their execution can involve the same channel iff one of them is a sending transition and the other is a receiving transition on the same channel. Since both t and t' are individually executable at G_k, the common channel is neither empty nor full. Since the channel is a non-empty and non-full FIFO queue, both first sending then receiving and first receiving then sending are possible. Therefore, the executions of both tt' and $t't$ are possible. It is clear that $tt' \equiv t't$. Using Lemma 4.1.1, $G_k \xrightarrow{t} G_l \wedge G_k \xrightarrow{t'} G_m \wedge tt' \equiv t't \Rightarrow G_l = G_m$. Thus, $\exists G_r \ (tt'_k \equiv_r t't)$. []

The following lemma states that an executable transition of a process P_i at a global state G_k remains executable at sequential successors of G_k as long as P_i does not execute a transition. Moreover, any interleaving of this transition and an executable transition sequence on which P_i is not active leads to the same global state (see Figure 4.2 for a graphical illustration).

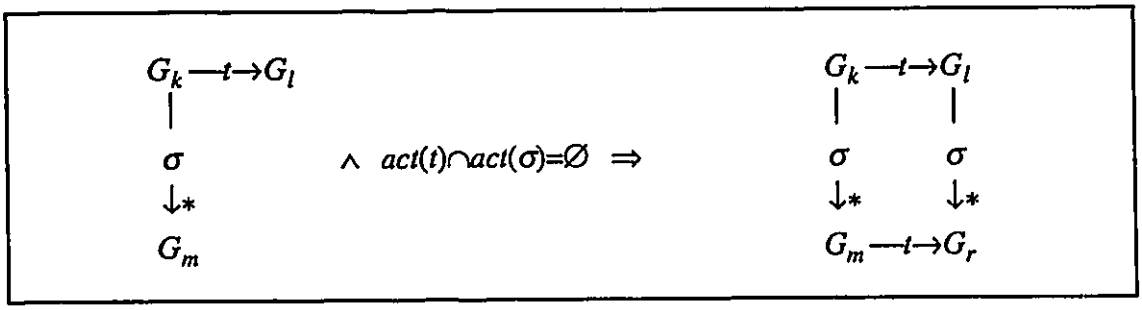


Figure 4.2 Illustration of Lemma 4.1.3.

Lemma 4.1.3

$$\forall(G_k,t,\sigma)\,(G_0 \rightarrow^* G_k \wedge G_k \text{—}t\rightarrow G_l \wedge G_k \text{—}\sigma\rightarrow^* G_m \wedge act(t)\cap act(\sigma)=\emptyset \Rightarrow$$

$$\exists G_r\,(t\sigma_{\,k}\equiv_r \sigma t)).$$

Proof: Let $\sigma=t_1t_2...t_j$ and $G_k \text{—}t_1\rightarrow G_{k(1)} \text{—}t_2\rightarrow \ ... \ \text{—}t_{j-1}\rightarrow G_{k(j-1)} \text{—}t_j\rightarrow G_m$. The lemma will be proven by induction on j.

Basis: Let j=0 then trivially $t_{\,k}\equiv_r t$.

Induction: For the induction step suppose the lemma holds for j-1, i.e.

$$\exists G_q\,(t_1t_2...t_{j-1}t_{\,k}\equiv_q tt_1t_2...t_{j-1}).$$

It will be proven that it also holds for j.

$G_k \text{—}t_1t_2...t_{j-1}\rightarrow^* G_{k(j-1)} \wedge G_k \text{—}t_1t_2...t_{j-1}t\rightarrow^* G_q \Rightarrow G_{k(j-1)} \text{—}t\rightarrow G_q$.

Using Lemma 4.1.2,

$G_{k(j-1)} \text{—}t_j\rightarrow G_m \wedge G_{k(j-1)} \text{—}t\rightarrow G_q \wedge act(t)\cap act(t_j)=\emptyset \Rightarrow \exists G_r\,(tt_{j\,k(j-1)}\equiv_r t_jt)$.

From Property 4.1.5,

$G_k \text{—}t_1t_2...t_{j-1}\rightarrow^* G_{k(j-1)} \wedge tt_{j\,k(j-1)}\equiv_r t_jt \Rightarrow t_1t_2...t_{j-1}tt_{j\,k}\equiv_r t_1t_2...t_{j-1}t_jt$.

From Property 4.1.6,

$t_1t_2...t_{j-1}t_{\,k}\equiv_q tt_1t_2...t_{j-1} \wedge t_1t_2...t_{j-1}tt_{j\,k}\equiv_r t_1t_2...t_{j-1}t_jt \Rightarrow tt_1t_2...t_{j\,k}\equiv_r t_1t_2...t_jt$. □

If two transition sequences are executable at a common global state and the set of active processes for these sequences are disjoint then interleavings of these sequences lead to the same global state. This is proven in the following lemma (see Figure 4.3 for a graphical illustration).

Lemma 4.1.4

$$\forall(G_k,\sigma,\omega)\,(G_0 \rightarrow^* G_k \wedge G_k \text{—}\sigma\rightarrow^* G_m \wedge G_k \text{—}\omega\rightarrow^* G_l \wedge act(\sigma)\cap act(\omega)=\emptyset \Rightarrow$$

$$\exists G_r\,(\omega\sigma_k\equiv_r \sigma\omega)).$$

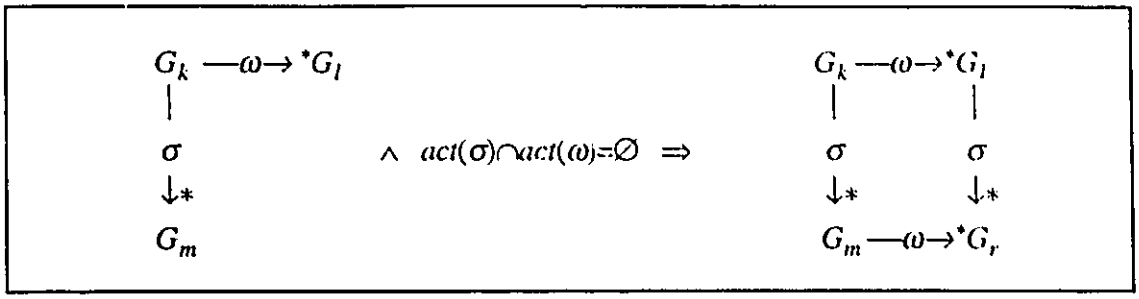


Figure 4.3 Illustration of Lemma 4.1.4.

Proof: Let $\sigma=t_1t_2...t_j$ and $G_k \xrightarrow{t_1} G_{k(1)} \xrightarrow{t_2} ... \xrightarrow{t_{j-1}} G_{k(j-1)} \xrightarrow{t_j} G_m$. The lemma will be proven by induction on j.

Basis: Let $j=0$ then $\sigma=\varepsilon$ and $\omega\ {}_k\equiv_r \omega$.

Induction: For the induction step suppose the lemma holds for j-1, i.e.

$$\exists G_q\ (t_1t_2...t_{j-1}\omega\ {}_k\equiv_q \omega t_1t_2...t_{j-1}).$$

It will be proven that it also holds for j.

$G_k \xrightarrow{t_1t_2...t_{j-1}} {}^*G_{k(j-1)} \wedge G_k \xrightarrow{t_1t_2...t_{j-1}\omega} {}^*G_q \Rightarrow G_{k(j-1)} \xrightarrow{\omega} {}^*G_q$.

Using Lemma 4.1.3,

$G_{k(j-1)} \xrightarrow{t_j} G_m \wedge G_{k(j-1)} \xrightarrow{\omega} {}^*G_q \wedge act(\omega)\cap act(t_j)=\varnothing \Rightarrow \exists G_r\ (t_j\omega\ {}_{k(j-1)}\equiv_r \omega t_j)$.

From Property 4.1.5,

$G_k \xrightarrow{t_1t_2...t_{j-1}} {}^*G_{k(j-1)} \wedge \omega t_j\ {}_{k(j-1)}\equiv_r t_j\omega \Rightarrow t_1t_2...t_{j-1}\omega t_j\ {}_k\equiv_r t_1t_2...t_{j-1}t_j\omega$.

From Property 4.1.6,

$t_1t_2...t_{j-1}\omega\ {}_k\equiv_q \omega t_1t_2...t_{j-1} \wedge t_1t_2...t_{j-1}\omega t_j\ {}_k\equiv_r t_1t_2...t_{j-1}t_j\omega \Rightarrow \omega t_1t_2...t_j\ {}_k\equiv_q t_1t_2...t_j\omega$. ||

4.2 Simultaneously Executable Sets

Definition 4.2.1 A nonempty subset of $exec(G_k)$ is called a *simultaneously executable set* of G_k if this set includes at most one transition from each process. The set of all simultaneously executable sets of G_k is denoted by $ses(G_k)$. ||

The following properties of simultaneously executable sets are immediate from Definition 4.2.1.

Property 4.2.1 $\forall(G_k,T)\ (T \in ses(G_k) \Rightarrow 1 \leq |T| \leq n)$ where $|T|$ is the cardinality of T.

Property 4.2.2 $\forall(G_k,T,T')\ (T \in ses(G_k) \wedge T' \subset T \wedge T' \neq \emptyset \Rightarrow T' \in ses(G_k))$.

Property 4.2.3 $\forall(G_k,T,t,t')\ (t,t' \in T \Rightarrow act(t) \cap act(t') = \emptyset)$.

In sequential reachability analysis, we observe the execution of a single transition in a protocol at a time. Therefore, when several concurrent transitions are simultaneously executable, the sequential reachability analysis simply considers every possible sequence of these transitions. Each such possible sequence is called a *linearization*. More specifically, the linearization of a set of transitions can be defined as follows. Let $T=\{t_1,t_2,\ldots,t_j\}$ be a simultaneously executable set of transitions and *Index* be a set of numbers from 1 to $|T|$ $(=j)$. Let π be any permutation function from *Index* to *Index*. A *linearization* of T, say γ, is a transition sequence such that $\gamma=t_{\pi(1)}t_{\pi(2)}\ldots t_{\pi(j)}$. Clearly, there exist $|T|!$ linearizations of T. The set of all linearizations of T is denoted by *linear*(T). The following lemma states that every linearization of a simultaneously executable set of a global state is an executable transition sequence from that global state.

Lemma 4.2.1

$$\forall(G_k,T,\gamma)\ (G_0 \rightarrow^* G_k \wedge T \in ses(G_k) \wedge \gamma \in linear(T) \Rightarrow \exists G_r\ (G_k \text{—}\gamma\text{→}^* G_r)).$$

Proof: Let $\gamma=t_1t_2\ldots t_j$ such that $T=\{t_1,t_2,\ldots,t_j\}$. $T \in ses(G_k)$ implies $T \subseteq exec(G_k)$.The lemma will be proven by induction on j.

Basis: Let j=1. Since $t_1 \in exec(G_k)$, there exists G_r such that $G_k \text{—}\gamma\text{→}^* G_r$.

Induction: For the induction step suppose the lemma holds for j-1, i.e. there exists G_q such that $G_k \text{—}t_1t_2\ldots t_{j-1}\text{→}^* G_q$. It will be proven that it also holds for j.

Since $t_j \in exec(G_k)$, there exists G_m such that $G_k \xrightarrow{t_j} G_m$. $T \in ses(G_k)$ implies $act(t_1 t_2 \ldots t_{j-1}) \cap act(t_j) = \emptyset$. Using Lemma 4.1.3, it can be concluded that there exists G_r such that $G_k \xrightarrow{\gamma}{}^* G_r$. ||

Using the result of Lemma 4.2.1, the following lemma states that every linearization of a simultaneously executable set leads to the same global state.

Lemma 4.2.2

$$\forall (G_k, T)\ (G_0 \rightarrow^* G_k \wedge T \in ses(G_k) \wedge \gamma \in linear(T) \wedge G_k \xrightarrow{\gamma}{}^* G_r \Rightarrow$$
$$\forall (\gamma' \in linear(T))\ (\gamma_k \equiv_r \gamma').$$

Proof: It is clear that for each linearization γ' of T, $\forall P_i \in P\ (\gamma\downarrow_i = \gamma'\downarrow_i)$. By the definition of the equivalence relation, $\gamma \equiv \gamma'$. From Lemma 4.2.1, there exists G_r such that $G_k \xrightarrow{\gamma}{}^* G_r$. Using Lemma 4.1.1, it can be concluded that $\gamma_k \equiv_r \gamma'$. ||

Consider $ses(G_k)$ of a global state G_k. From Property 4.2.2, every nonempty subset of a simultaneously executable set $T \in ses(G_k)$ is a simultaneously executable set. Since there are $2^{|T|}-1$ nonempty subsets of T and each subset may lead to a distinct global state, $2^{|T|}-1$ distinct global states may be generated in the sequential reachability analysis through the execution of $|T|!$ linearizations of $|T|$ transitions. On the other, when $|T|$ transitions are simultaneously executed, the linearizations of these transitions are not considered, and hence $2^{|T|}-2$ global states are not generated. Therefore, in order to maximize the reduction in the number of global states generated, simultaneous reachability analysis selects the maximal simultaneously executable sets in $ses(G_k)$.

Unfortunately, selecting only the maximal simultaneously executable sets in $ses(G_k)$ is not sufficient for verification purposes. This is due to the fact that a transition t of a process $P_i \in act(T)$, for a selected maximal $T \in ses(G_k)$, that is not executable at G_k may become executable at a sequential successor G_m of G_k and the execution of t at G_m may

lead to a deadlock state. However, during the execution of T, an executable transition from P_i will be executed. But, in order to detect the deadlock state, P_i must not make any progress until t becomes executable. Such a transition t from P_i is defined below as a *potentially executable transition* at G_k.

4.3 Potentially Executable Transitions And Selected Simultaneously Executable Sets

Consider the protocols given in Figures 4.4 and 4.7, which will be used to illustrate the cases where the maximal simultaneously executable sets are not sufficient to detect logical errors. For the protocol given in Figure 4.4, processes P_1 and P_2 are initially at states 1 and 5, respectively. After executing transition sequence (5,-b,6)(1,-a,2)(6,+a,7)(2,-c,3)(7,+c,8)(3,+b,3) from the initial global state, the protocol reaches global state (<3,8>,<ε,ε>) which is a deadlock state (Figure 4.5). The reduced reachability graph constructed by only executing maximal simultaneously executable sets does not include this deadlock state (Figure 4.6).

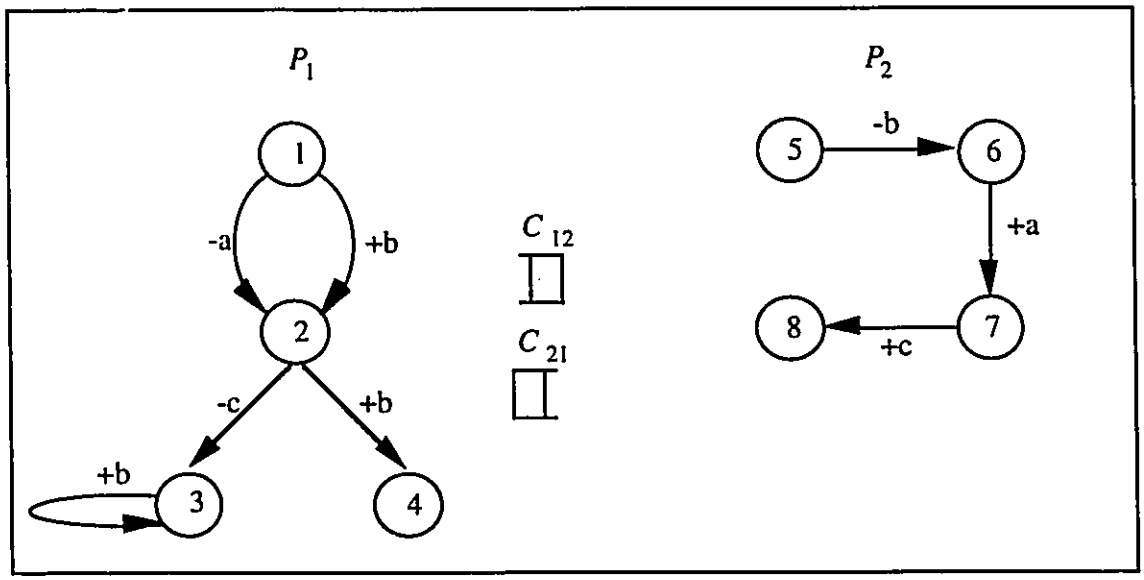


Figure 4.4 A protocol containing deadlocks.

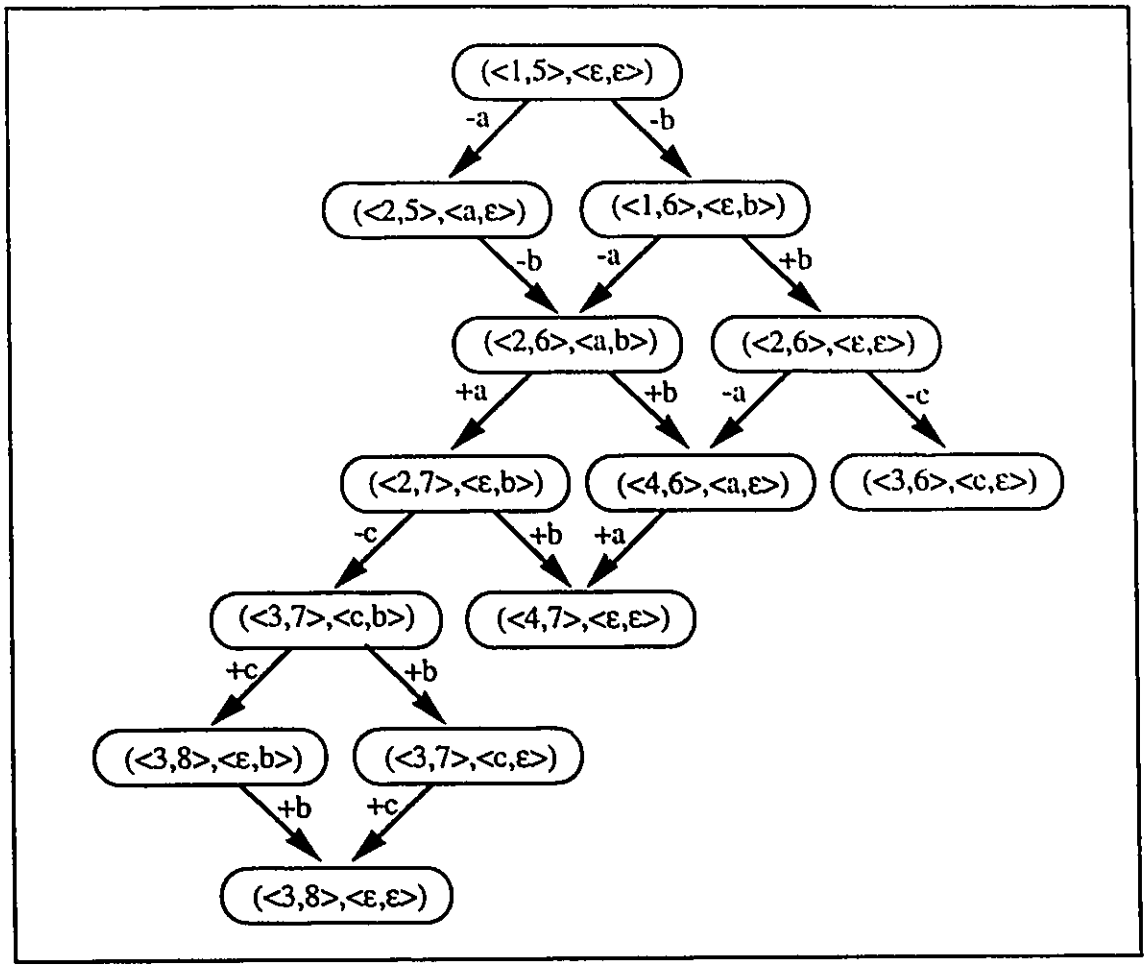


Figure 4.5 The sequential reachability graph for the protocol given in Figure 4.4.

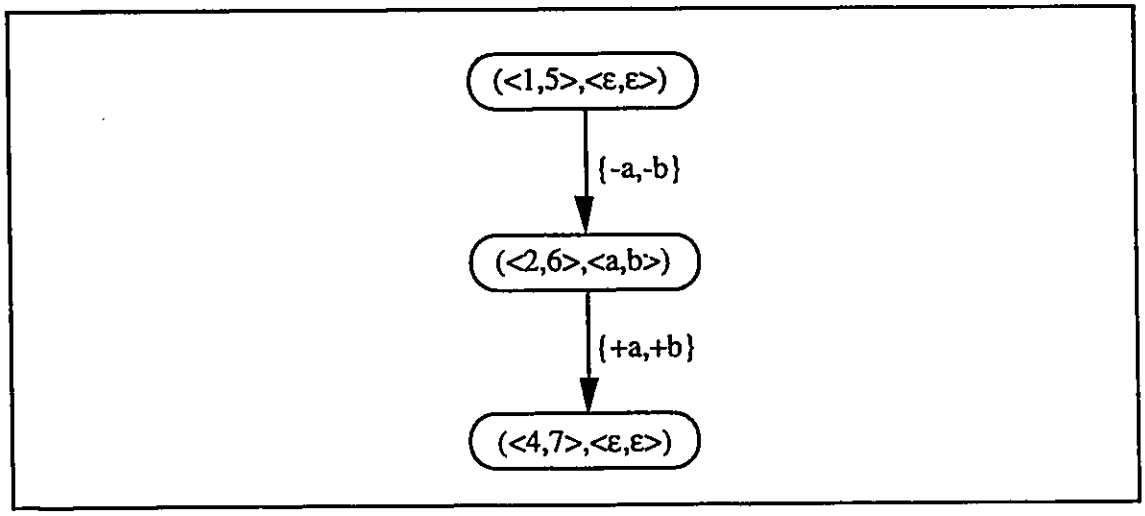


Figure 4.6 A reduced reachability graph for the protocol given in Figure 4.4.

Similarly, states 1 and 3 are the initial states of processes P_1 and P_2, respectively, for the protocol given in Figure 4.7. The protocol reaches deadlock state (<2,5>,<ε,ε>) by executing transition sequence (1,-a,2)(3,+a,5) from the initial global state (Figure 4.8). However, the reduced reachability graph constructed by only executing maximal simultaneously executable sets does not include this deadlock state (Figure 4.9).

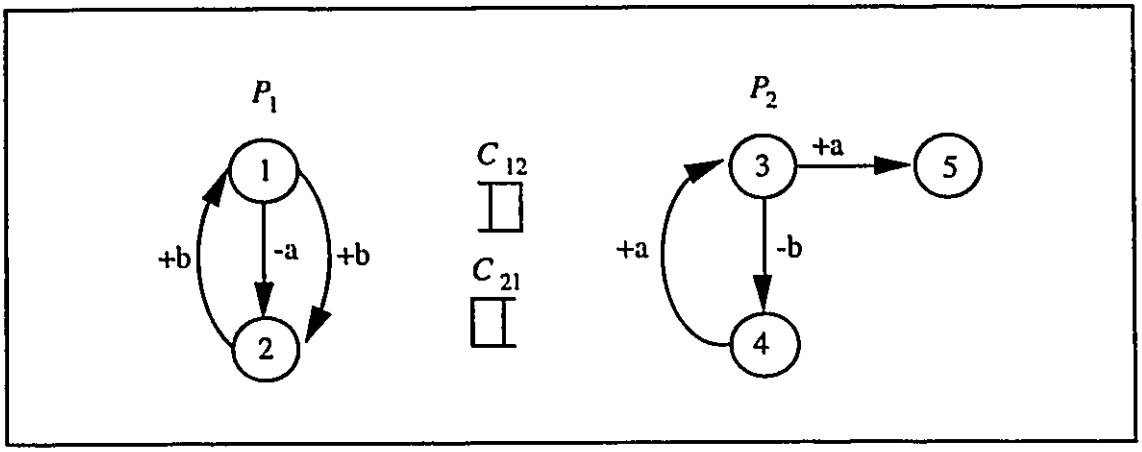


Figure 4.7 A protocol containing deadlocks.

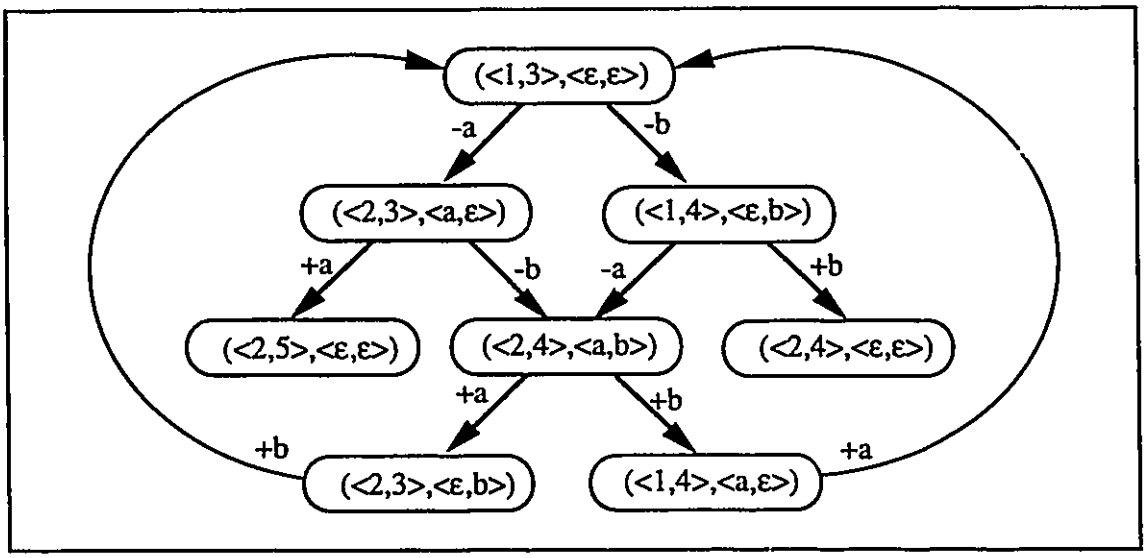


Figure 4.8 The sequential reachability graph for the protocol in Figure 4.7.

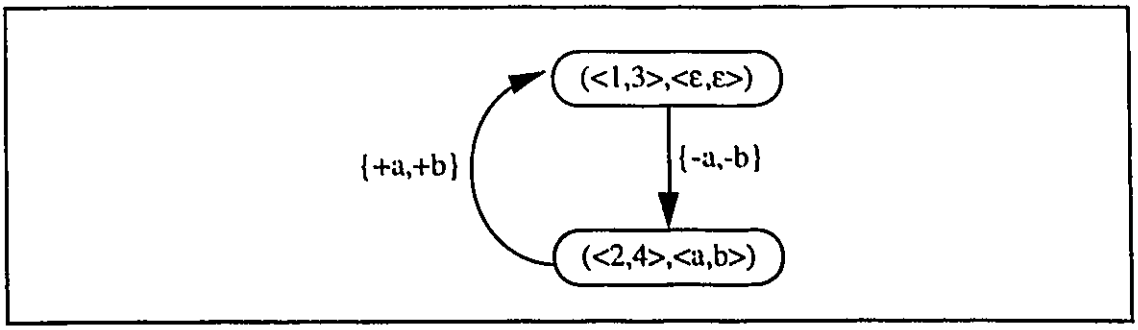


Figure 4.9 A reduced reachability graph for the protocol in Figure 4.7.

Thus, simultaneous reachability analysis needs to select some non-maximal simultaneously executable sets in addition to maximal ones. There are some transitions specified at a global state such that they are not executable at this global state but they may be executable at the global states sequentially reachable from this global state by the execution of a transition sequence on which the processes of these transitions do not make any progress. Such transitions will be called *potentially executable transitions*. Potentially executable transitions will be used in determining which non-maximal simultaneously executable sets are selected in addition to maximal ones.

Definition 4.3.1 For any $1 \le i,j \le n$ and $i \ne j$, a transition $t \in spec(P_i)$ is called *a potentially executable transition at* G_k iff one of the following conditions is satisfied :

1. $t=(s_i^k,-x,\delta_i(s_i^k,-x)) \wedge x \in M_{ij} \wedge |c_{ij}^k| = B_{ij}$.

2. $t=(s_i^k,+x,\delta_i(s_i^k,+x)) \wedge x \in M_{ji} \wedge c_{ji}^k = \varepsilon$.

The set of all potential transitions at G_k is denoted by $pot(G_k)$. []

The first condition of Definition 4.3.1 states that a sending transition (t) is potentially executable if the corresponding channel (C_{ij}) is full. Assume that just after this condition is satisfied some of the processes except process P_i make progress and this progress leads to

the reception of a message from channel C_{ij}. Then channel C_{ij} contains room for another message. Since the state of the process P_i does not change and channel C_{ij} is not full any more, the potentially executable transition becomes an executable transition at a successor global state. For example, the protocol given in Figure 4.4 includes a potentially executable sending transition at the simultaneously reachable global state (<2,6>,<a,b>). Transition (2,-c,3) is not executable at this global state because C_{12} is full, but according to Definition 4.3.1, this sending transition is potentially executable at this global state. Notice that if process P_2 receives message a at this global state, the protocol reaches global state (<2,7>,<ε,b>). Even though process P_1 does not make any progress between global states (<2,5>,<a,ε>) and (<2,7>,<ε,b>), transition (2,-c,3) which is not executable (but potentially executable) at (<2,5>,<a,ε>) becomes executable at (<2,7>,<ε,b>) (Figure 4.7).

The second condition of Definition 4.3.1 states that a receiving transition is potentially executable if the corresponding channel (C_{ji}) is empty. We can outline a similar scenario for the potentially executable receiving transitions. Assume that just after the second condition is satisfied some of the processes except process P_i make progress and this progress leads to the transmission of message x as the next message to channel C_{ji}. Then channel C_{ji} contains only message x. Since the state of process P_i does not change and message x is the first message in channel C_{ji}, the potentially executable transition becomes an executable transition at a successor global state. For the protocol given in Figure 4.7, transition (3,+a,5) is not executable at the initial global state. According to Definition 4.3.1, this receiving transition is potentially executable because the corresponding channel is empty. Consider that process P_1 sends message a, and therefore the protocol reaches global state (<2,3>,<a,ε>). At this global state, transition (3,+a,5) becomes executable even though P_2 has not made progress.

Consider the case where the following conditions are satisfied:

- There are both executable and potentially executable transitions from the same process P_i at a global state G_k.

- One of these potentially executable transitions becomes executable at a global state G_m which is sequentially reachable from G_k by the execution of a transition sequence which does not include a transition from P_i.

- The execution of this transition leads to a deadlock state.

If we select only maximal simultaneously executable sets then in every selected simultaneously executable set of G_k we include an executable transition from process P_i. This means that for each generation of a next global state from G_k, P_i executes a transition. However, in order to detect the deadlock state, the process should not make any progress until a potentially executable transition becomes executable, i.e., from G_k to G_m. As a result, the maximal simultaneously executable sets are not sufficient for the detection of the logical errors because of the existence of potentially executable transitions.

For example, at the simultaneously reachable global states (<2,6>,<a,b>) (in Figure 4.6) and (<1,3>,<ε,ε>) (in Figure 4.9) of protocols given in Figures 4.4 and 4.7, respectively, we have both executable and potentially executable transitions from process P_1. From Figures 4.6 and 4.9, it can be seen that we detect deadlock states (<3,8>,<ε,ε>) and (<2,5>,<ε,ε>) from these simultaneously reachable global states only when the potentially executable transitions (2,-c,3) and (3,+a,5) of P_1 becomes executable at the global states sequentially reachable from (<2,6>,<a,b>) and (<1,3>,<ε,ε>), respectively.

Hence, if there are both executable and potentially executable transitions of P_i at G_k then for every selected maximal simultaneously executable set including an executable transition from P_i, one needs to select another simultaneously executable set which includes the same set of transitions except the executable transition from P_i. This means that a simultaneously executable set selected from $ses(G_k)$ may or may not contain an

executable transition from a process having both executable and potentially executable transitions at G_k but must contain one executable transition from each process having only executable transitions at G_k.

Definition 4.3.2 Let $pot(s_i^k)$ and $exec(s_i^k)$ denote $pot(G_k) \cap spec(s_i^k)$ and $exec(G_k) \cap spec(s_i^k)$, respectively. $T \in ses(G_k)$ is a *selected simultaneously executable set* of G_k if $\forall(i=1,2,...,n)$ $(pot(s_i^k)=\varnothing \wedge exec(s_i^k)\neq\varnothing \Rightarrow exec(s_i^k) \cap T \neq \varnothing)$.

The set of all selected simultaneously executable sets of G_k is denoted by $selected(G_k)$. □

The algorithm given in Figure 4.10 is an optimal algorithm for constructing $selected(G_k)$ because its time complexity is equal to the cardinality of $selected(G_k)$. The following properties of selected simultaneously executable sets are immediate from the definitions given in this section.

Property 4.3.1 $\forall(G_k,T,t,t')$ $(T \in selected(G_k) \wedge t \in T \wedge t' \in exec(G_k) \wedge act(t)=act(t') \Rightarrow ((T \setminus \{t\}) \cup \{t'\}) \in selected(G_k))$.

Property 4.3.2 $\forall(G_k,T,t,t')$ $(T \in selected(G_k) \wedge t \in T \wedge t' \in pot(G_k) \wedge act(t)=act(t') \Rightarrow (T \setminus \{t\}) \in selected(G_k))$.

Property 4.3.3 $\forall(G_k,t)$ $(t \in exec(G_k) \Rightarrow \exists T\ (T \in selected(G_k) \wedge t \in T))$.

4.4. Simultaneous Reachability Analysis

In the simultaneous reachability analysis of a protocol Π, a global state, say G_k, produces another global state, say G_l, if there exist a selected simultaneously executable set of G_k and the processes execute simultaneously the transitions in that set.

Definition 4.4.1 A global state G_l is a *simultaneous immediate successor* of G_k, denoted by $G_k \longrightarrow\!\!\!\!\rightarrow G_l$ or $G_k \xrightarrow{T}\!\!\!\!\rightarrow G_l$, iff there exists $T \in selected(G_k)$ such that $\gamma \in linear(T)$ and $G_k \xrightarrow{\gamma}{}^{*} G_l$. □

```
/* e is the number of processes having executable transitions at G_k*/
/* p is an array of indices of such processes                       */

e=0
for i=1 to n do
  if exec(s_i^k)≠∅ then
  begin
    e=e+1
    p(e)=i
  end

/* selected(G_k) includes maximal simultaneously executable sets    */

selected(G_k)=exec(s_p(1)^k)×exec(s_p(2)^k)×...×exec(s_p(e)^k)

/* if a process has both executable and potentially executable      */
/* transitions at G_k then for every selected simultaneously        */
/* executable set including an executable transition from this      */
/* process, select another simultaneously executable set which      */
/* includes the same set of transitions except that executable      */
/* transition.                                                      */

for h=1 to e do
  if pot(s_p(h)^k)≠∅ then
  begin
    extra=∅
    for each T∈ selected(G_k) do
      extra=extra∪{T\{t}} such t∈T and t∈exec(s_p(h)^k)
    selected(G_k)=selected(G_k)∪extra
  end
```

Figure 4.10 An algorithm for constructing *selected*(G_k).

Note that Definition 4.4.1, the simultaneous immediate successor, defines a binary relation between global states which is denoted by —».

Definition 4.4.2. Let —»* be the reflexive and transitive closure of —».

1. G_m is simultaneously reachable from G_k (or a simultaneous successor of G_k) iff

   G_k—»*G_m.

2. G_m is simultaneously reachable iff G_0—»*G_m.

G_k —T_1—»$G_{k(1)}$ ∧ $G_{k(1)}$ —T_2—»$G_{k(2)}$ ∧ … ∧ $G_{k(r-1)}$ —T_r—»G_m will sometimes be denoted by G_k —$T_1T_2\ldots T_r$→*G_m. □

An algorithm for the simultaneous reachability analysis is given in Figure 4.11. Simultaneously reachable global state space of a protocol can be represented by a digraph, called *simultaneous reachability graph*, in which nodes represent global states and arcs stand for the simultaneous immediate successor relation between global states.

```
A=∅      /* A is the set of global states have already been analyzed */
W={G0}   /* W is the set of global states to be analyzed */
while W≠∅ do
begin
  remove an element Gk from W
  add Gk to A
  if Gk is a deadlock state then report deadlock
  for each transmission (s_i^k,-x,s') ∈ spec(Gk) where x∈Mij and i≠j do
    if |c_ij^k|=Bij then report buffer overflow
  for each c_ij^k (i≠j) do
    if c_ij^k=xX, X ∈ Mij*, and δj(s_j^k,+x) is undefined then
      report unspecified reception
  for each T∈selected(Gk) do    /* see the formation of selected(Gk) */
  begin                         /* in Figure 4.1 */
    generate Gm such that Gk—T—»Gm
    if (Gm is not in A or W) add Gm to W
  end
end
report each nonexecuted transition as a nonexecutable transition
```

Figure 4.11 Simultaneous reachability algorithm.

Consider the part of a protocol specification given in Figure 4.12 where G_1 = (<1,4,7>,<ε,ε,ε,ε,a,ce>) is a reachable global state and bound is 2 for each channel. Lett_1=(1,+a_{31},2),t_2=(4,+c_{32},5),t_3=(4,-d_{23},6),t_4=(7,-b_{31},8),t_5=(7,+d_{23},7),t_6=(7,-e_{32},8). Then

$exec(G_1)=\{t_1,t_2,t_3,t_4\}$,

$pot(G_1)=\{t_5,t_6\}$,

$ses(G_1)=\{\{t_1\},\{t_2\},\{t_3\},\{t_4\},\{t_1,t_2\},\{t_1,t_3\},\{t_1,t_4\},\{t_1,t_2,t_4\},\{t_1,t_3,t_4\}\}$,

$selected(G_1)=\{\{t_1,t_2,t_4\},\{t_1,t_3,t_4\},\{t_1,t_2\},\{t_1,t_3\}\}$.

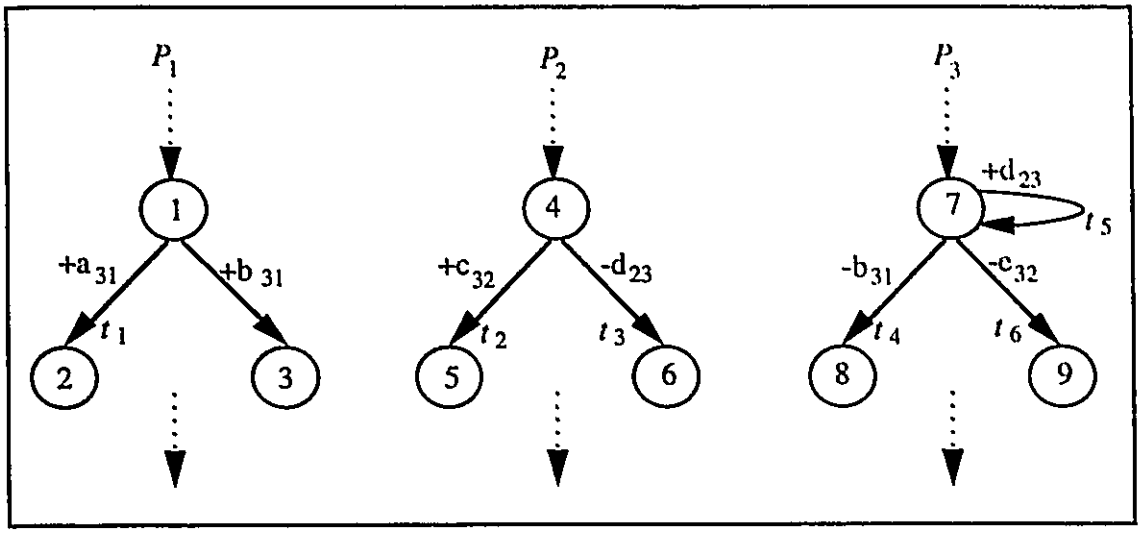
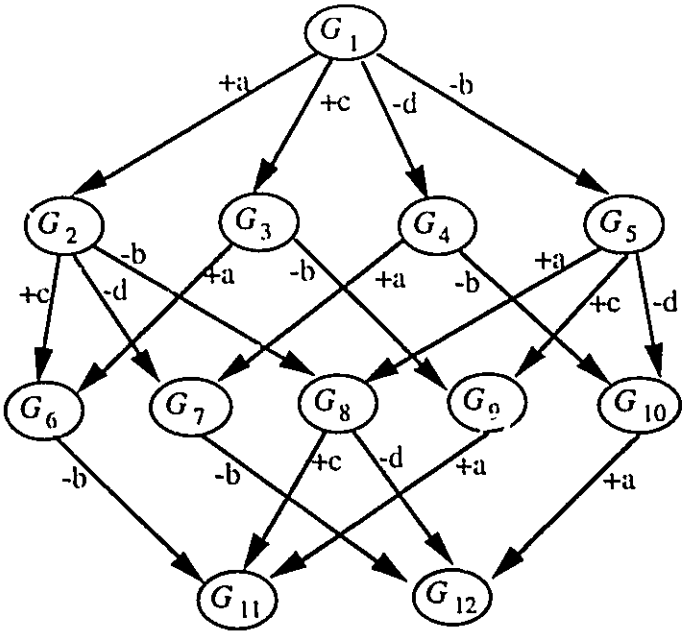


Figure 4.12 A part of the specification of a protocol.

Notice that if the algorithm given in Figure 4.10 is applied, one first obtains maximal simultaneously executable sets which are $\{t_1,t_2,t_4\},\{t_1,t_3,t_4\}$. Since only P_3 includes potentially executable transitions, one obtains two more simultaneously executable sets which are $\{t_1,t_2\},\{t_1,t_3\}$ by removing the executable transition of P_3 from the maximal simultaneously executable sets. Sequential reachability analysis enumerates all possible sequences of these executable transitions such that each sequence includes one transition from each process. There are 12 distinct sequences such as $t_1t_2t_3$, $t_1t_3t_4$, $t_1t_4t_2$, etc. During the execution of these sequences, 20 global states are generated and 11 of them are stored as distinct global states (see Figure 4.13). On the other hand, simultaneous reachability analysis generates G_6, G_7, G_{11} and G_{12} from G_1, i.e. G_1 —$\{t_1,t_2\}$—»G_6, G_1 —$\{t_1,t_3\}$ —»G_7, G_1 —$\{t_1,t_2,t_4\}$—»G_{11} and G_1 —$\{t_1,t_3,t_4\}$—»G_{12}. This is shown in Figure 4.14.



$G_k = (<s_1^k, s_2^k, s_3^k>, <c_{12}^k, c_{13}^k, c_{21}^k, c_{23}^k, c_{31}^k, c_{32}^k>)$

G_1 = (<1,4,7>,<ε,ε,ε,ε,a,ce>)

G_2 = (<2,4,7>,<ε,ε,ε,ε,ε,ce>)

G_3 = (<1,5,7>,<ε,ε,ε,ε,a,e>)

G_4 = (<1,6,7>,<ε,ε,ε,d,a,ce>)

G_5 = (<1,4,8>,<ε,ε,ε,ε,ab,ce>)

G_6 = (<2,5,7>,<ε,ε,ε,ε,ε,e>)

G_7 = (<2,6,7>,<ε,ε,ε,d,ε,ce>)

G_8 = (<2,4,8>,<ε,ε,ε,ε,b,ce>)

G_9 = (<1,5,8>,<ε,ε,ε,ε,ab,e>)

G_{10} = (<1,6,8>,<ε,ε,ε,d,ab,ce>)

G_{11} = (<2,5,8>,<ε,ε,ε,ε,b,e>)

G_{12} = (<2,6,8>,<ε,ε,ε,d,b,ce>)

Figure 4.13 Sequential reachability graph for the partial protocol in Figure 4.12.

In the next section we will prove that simultaneous reachability analysis detects every deadlock state and every nonexecutable transition. Before proving this fact, for the sake of completeness let us apply simultaneous reachability analysis to the protocols given in Figures 4.4 and 4.7. Each of these protocols includes deadlock states (see Figures 4.5 and 4.8). The simultaneous reachability graphs of these protocols are given in Figures 4.15 and

4.16. Notice that all the deadlock states included in the sequential reachability graphs are also included in these graphs.

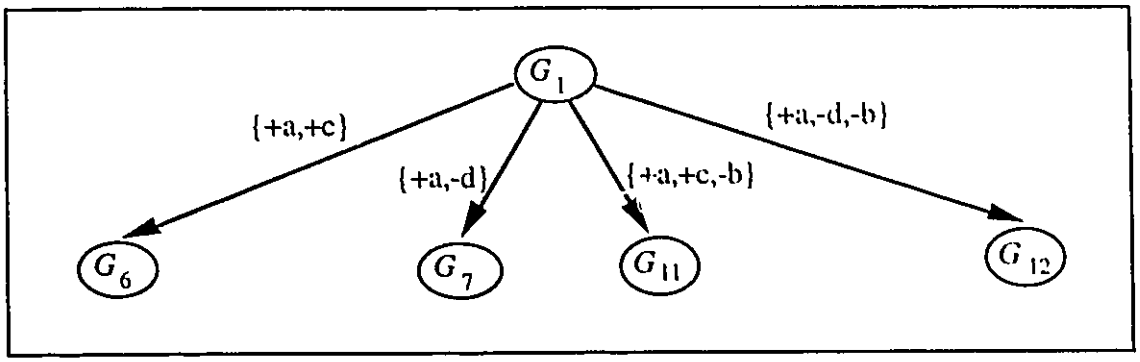


Figure 4.14 Simultaneous reachability graph for the partial protocol in Figure 4.12.

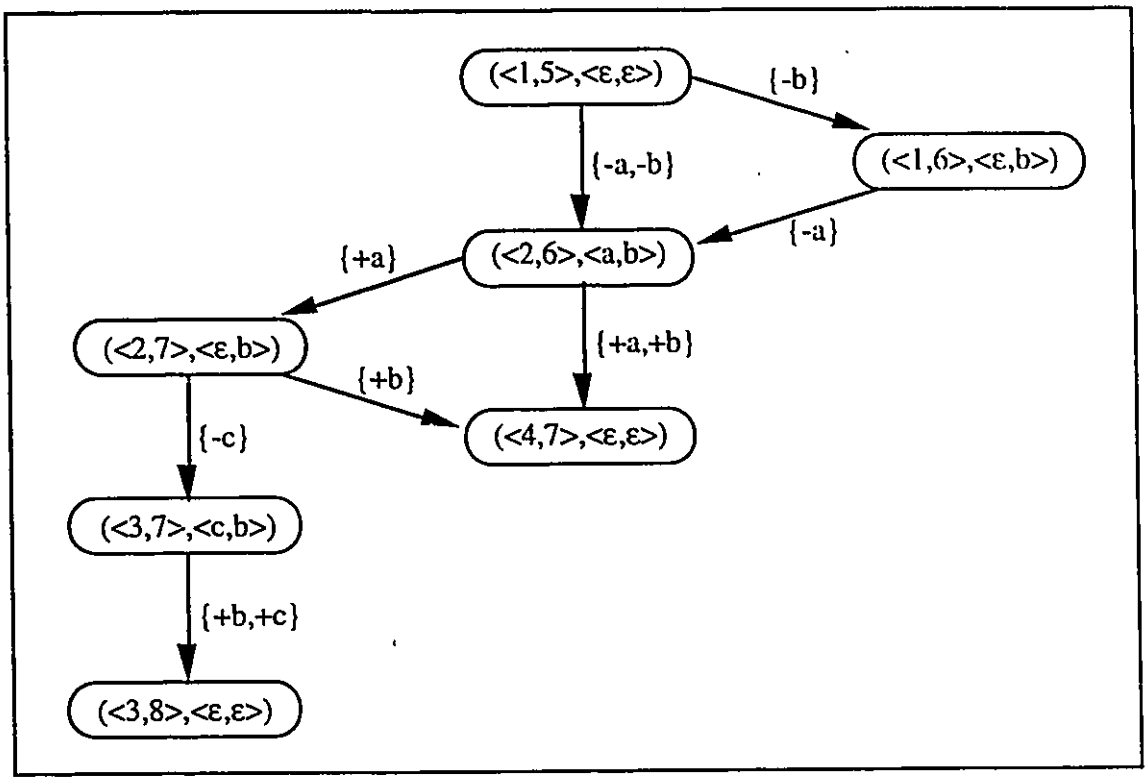


Figure 4.15 Simultaneous reachability graph for the protocol given in Figure 4.4.

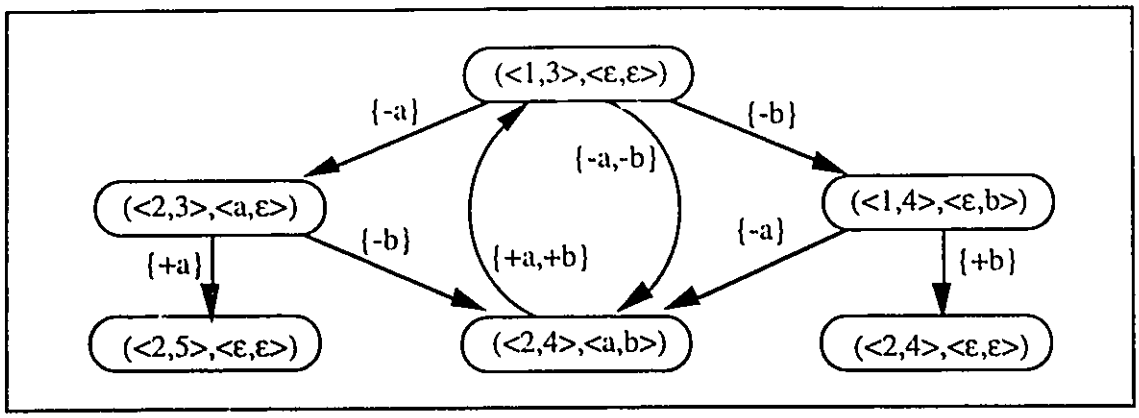


Figure 4.16 Simultaneous reachability graph for the protocol given in Figure 4.7.

4.5 Deadlock and Nonexecutable Transition Detection

The following theorem states that every simultaneously reachable global state is a sequentially reachable global state.

Theorem 4.5.1 $\forall(G_k)\ (G_0 \text{—»}^* G_k \Rightarrow G_0 \rightarrow^* G_k)$.

Proof: Let $G_0 \text{—} T_1 T_2 \ldots T_j \text{—»}^* G_k$ and $\gamma_i \in linear(T_i)$ for i=1,2,...,j. Using Lemma 4.2.2, $G_0 \text{—} \gamma_1 \gamma_2 \ldots \gamma_j \rightarrow^* G_k$. □

Given any sequentially reachable global state G_k and a transition sequence σ such that $G_k \text{—} \sigma \rightarrow^* G_m$, the following lemma states that there exists a global state G_r such that G_r is sequentially reachable from both G_m and one of the simultaneous immediate successors of G_k (see Figure 4.17 for a graphical illustration).

Lemma 4.5.1

$$\forall(G_k,\sigma)\ (G_0 \rightarrow^* G_k \wedge G_k \text{—} \sigma \rightarrow^* G_m \wedge \sigma \neq \varepsilon \Rightarrow$$

$$\exists(T,\omega,\rho,G_r)\ (T \in selected(G_k) \wedge \gamma \in linear(T) \wedge \sigma\omega \equiv_r \gamma\rho \wedge |\sigma| > |\rho|)).$$

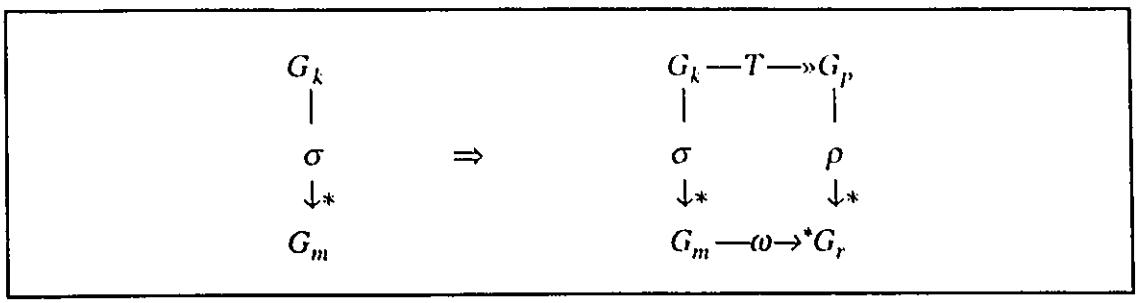


Figure 4.17 Illustration of Lemma 4.5.1.

Proof: Let *First* be the set of transitions from σ such that transition $t \in First$ iff $\exists \mu$ ($\mu t \in prefix(\sigma) \wedge act(\mu) \cap act(t) = \emptyset$). Let t_1 be the first transition in σ. Clearly, $t_1 \in First$. Then $\sigma \neq \varepsilon$ implies $First \neq \emptyset$. From Property 4.3.3, $\forall t$ ($t \in exec(G_k) \Rightarrow \exists T$ ($T \in selected(G_k) \wedge t \in T$)). Since $t_1 \in exec(G_k)$, $\exists T$ ($T \in selected(G_k) \wedge T \cap First \neq \emptyset$).

Case 1. $\exists T$ ($T \in selected(G_k) \wedge T \subseteq First$).

Let $T=\{t_1,t_2,...,t_j\}$. Since $T \subseteq First$, $\sigma = t_1\mu_1t_2\mu_2...t_j\mu_j$ such that $act(t_1\mu_1t_2\mu_2...t_i\mu_i) \cap act(t_{i+1}) = \emptyset$ for i=1,2,...,j-1 and $j<n$.

Let G_k —$t_1\mu_1$→ $G_{k(1)}$ —$t_2\mu_2$→* ... —$t_{j-1}\mu_{j-1}$→* $G_{k(j-1)}$ —$t_j\mu_j$→* G_m.

$t_1\mu_1t_2\mu_2...t_j\mu_j \; k \equiv_m t_1t_2...t_j\mu_1\mu_2...\mu_j$ will be proven by induction on j.

Basis: Let j=1 then from Property 4.1.3, $t_1\mu_1 \; k \equiv_m t_1\mu_1$.

Induction: For the induction step suppose the claim holds for j-1, i.e.

$$t_1\mu_1t_2\mu_2...t_{j-1}\mu_{j-1} \; k \equiv_{k(j-1)} t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}. \quad (1)$$

It will be proven that it also holds for j.

Using Property 4.1.4, $G_{k(j-1)}$ —$t_j\mu_j$→* $G_m \wedge$ (1) $\Rightarrow$

$$t_1\mu_1t_2\mu_2...t_{j-1}\mu_{j-1}t_j\mu_j \; k \equiv_m t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\mu_j \quad (2)$$

Using (1), let G_k —$t_1t_2...t_{j-1}$→* G_l —$\mu_1\mu_2...\mu_{j-1}$→* $G_{k(j-1)}$. Using Lemma 4.1.3,

$t_j \in exec(G_k) \wedge G_k$ —$t_1t_2...t_{j-1}$→* $G_l \wedge act(t_j) \cap act(t_1t_2...t_{j-1}) = \emptyset \Rightarrow t_j \in exec(G_l)$.

Again using Lemma 4.1.3,

$t_j \in exec(G_l) \wedge G_l$ —$\mu_1\mu_2...\mu_{j-1}$→* $G_{k(j-1)} \wedge act(\mu_1\mu_2...\mu_{j-1}) \cap act(t_j) = \emptyset \Rightarrow$

$\exists G_q$ ($\mu_1\mu_2...\mu_{j-1}t_j \; l \equiv_q t_j\mu_1\mu_2...\mu_{j-1}$).

From Property 4.1.5,

$G_k \text{—}t_1t_2...t_{j-1}\rightarrow^* G_l \wedge \mu_1\mu_2...\mu_{j-1}t_j\, l \equiv_q t_j\mu_1\mu_2...\mu_{j-1} \Rightarrow$

$t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\, k \equiv_q t_1t_2...t_{j-1}t_j\mu_1\mu_2...\mu_{j-1}$. (3)

$G_k \text{—}t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\mu_j\rightarrow^* G_m$ (from 2) $\wedge\ G_k \text{—}t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\rightarrow^* G_q$ (from 3) $\Rightarrow G_q \text{—}\mu_j\rightarrow^* G_m$.

From Property 4.1.4,

$G_q \text{—}\mu_j\rightarrow^* G_m \wedge$ (3) $\Rightarrow t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\mu_j\, k \equiv_m t_1t_2...t_j\mu_1\mu_2...\mu_{j-1}\mu_j$. (4)

From Property 4.1.6,

(2) $\wedge$ (4) $\Rightarrow t_1\mu_1t_2\mu_2...t_{j-1}\mu_{j-1}t_j\mu_j\, k \equiv_m t_1t_2...t_{j-1}t_j\mu_1\mu_2...\mu_{j-1}\mu_j$.

Let $\gamma=t_1t_2...t_j$ and $\rho=\mu_1\mu_2...\mu_j$ then $\sigma_k \equiv_m \gamma\rho$ and $\gamma \in linear(T)$. Finally, let $\omega=\varepsilon$ and $G_r=G_m$ then $\sigma\omega_k \equiv_r \gamma\rho$. $|\gamma|>1$, $|\omega|=0$ and $|\sigma|=|\gamma\rho|$ (from Property 4.1.1) imply $|\sigma|>|\rho|$.

Case 2. $\exists T\,(T \in selected(G_k) \wedge T \subseteq First)$.

It is known that $\exists T\,(T \in selected(G_k) \wedge T \cap First \neq \varnothing)$. Let $T \in selected(G_k)$ such that $T \setminus First$ is minimal. Notice that $T \setminus First \neq \varnothing$, otherwise $T \subseteq First$. $act(T \setminus First) \cap act(\sigma)=\varnothing$ will be proven by contradiction.

Suppose $act(T \setminus First) \cap act(\sigma) \neq \varnothing$, i.e, $\exists P_i\ (P_i \in act(T \setminus First) \cap act(\sigma))$. Let $t,t' \in spec(P_i)$ such that $t \in (T \setminus First)$, $\mu t' \in prefix(\sigma)$ and $P_i \notin act(\mu_j)$. Since $P_i \notin act(\mu)$, either $t' \in exec(G_k)$ or $t' \in pot(G_k)$.

Case 2.1. $t' \in exec(G_k)$.

From Property 4.3.1, $T \in selected(G_k) \wedge t \in T \wedge t' \in exec(G_k) \wedge act(t)=act(t') \Rightarrow$ $((T \setminus \{t\}) \cup \{t'\}) \in selected(G_k)$.

However, $|((T \setminus \{t\}) \cup \{t'\}) \setminus First|+1=|T \setminus First|$. This means that $T \setminus First$ is not minimal, a contradiction.

Case 2.2. $t' \in pot(G_k)$.

From Property 4.3.2, $T \in selected(G_k) \wedge t \in T \wedge t' \in pot(G_k) \wedge act(t)=act(t') \Rightarrow$ $(T \setminus \{t\}) \in selected(G_k)$.

However, $|(T \setminus \{t\}) \setminus First| + 1 = |T \setminus First|$. This means that $T \setminus First$ is not minimal, a contradiction.

Since both Case 2.1 and Case 2.2 are contradictory cases, it can be concluded that $act(T \setminus First) \cap act(\sigma) = \emptyset$.

Let ω be a linearization of $T \setminus First$. Using Lemma 4.2.1, $(T \setminus First) \in ses(G_k)$ implies that there exists G_h such that $\omega \in linear(T \setminus First) \wedge G_k \text{—}\omega\text{→}^* G_h$.

$\omega \in linear(T \setminus First) \wedge act(T \setminus First) \cap act(\sigma) = \emptyset \Rightarrow act(\sigma) \cap act(\omega) \neq \emptyset$.

$G_k \text{—}\omega\text{→}^* G_h \wedge G_k \text{—}\sigma\text{→}^* G_m \wedge act(\sigma) \cap act(\omega) \neq \emptyset \Rightarrow \exists G_r (\sigma\omega \,{}_k\!\equiv_r \omega\sigma)$ (Lemma 4.1.4).

Let $\sigma' = \sigma\omega$. Let $First'$ be the set of transitions from σ' such that transition $t \in First'$ iff $\exists \mu$ $(\mu t \in prefix(\sigma') \wedge act(\mu) \cap act(t) = \emptyset)$. From Property 4.2.3,

$$\forall (t,t') \; (t,t' \in (T \setminus First) \Rightarrow act(t) \cap act(t') = \emptyset). \qquad (5)$$

$act(\sigma) \cap act(\omega) \neq \emptyset \wedge \omega \in linear(T \setminus First) \wedge (5) \Rightarrow First' = First \cup (T \setminus First)$.

$First' = First \cup (T \setminus First) \wedge First \neq \emptyset \wedge T \neq \emptyset \Rightarrow T \subset First'$. Therefore, Case 2 can be transformed into Case 1 by letting $\sigma = \sigma'$. Then $\exists (\gamma, \omega, \rho, G_r)$ $(\sigma\omega \,{}_k\!\equiv_r \gamma\rho)$. Since $(T \setminus First) \subset T$, $|\omega| < |\gamma|$. Finally $|\omega| < |\gamma|$ and $|\sigma\omega| = |\gamma\rho|$ (from Property 4.1.1) imply $|\sigma| > |\rho|$. ||

The following lemma is a generalization of Lemma 4.5.1. Given any sequentially reachable global state G_k and a transition sequence σ such that $G_k \text{—}\sigma\text{→}^* G_m$, the lemma states that there exists a global state G_r such that G_r is sequentially reachable from G_m and simultaneously reachable from G_k (see Figure 4.18 for a graphical illustration).

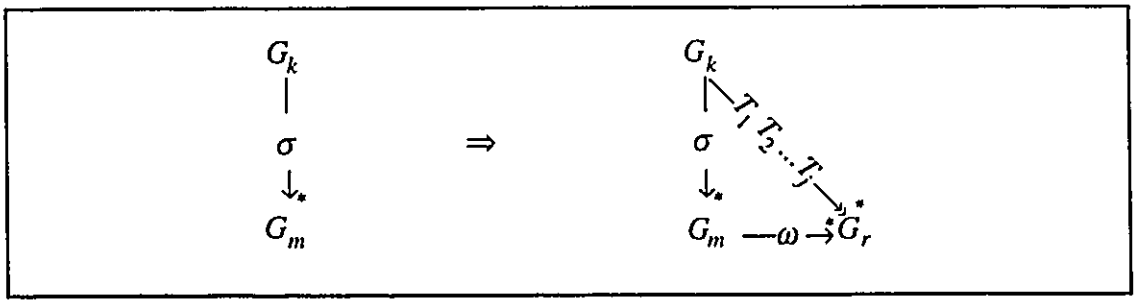


Figure 4.18 Illustration of Lemma 4.5.2.

Lemma 4.5.2

$$\forall(G_k,\sigma)\ (G_0 \rightarrow^* G_k \wedge G_k \text{—}\sigma\rightarrow^* G_m \wedge \sigma\neq\varepsilon \Rightarrow$$

$$\exists(G_r,\omega j)\ (G_k \text{—}T_1\text{—}\rightarrow G_{k(1)} \text{—}T_2\text{—}\rightarrow G_{k(2)} \wedge \ldots \wedge G_{k(j-1)} \text{—}T_j\text{—}\rightarrow G_r \wedge$$

$$\sigma\omega_{\ k}\equiv_r \gamma_1\gamma_2\ldots\gamma_j \wedge \gamma_i \in linear(T_i)\ (i=1,2,\ldots,j)).$$

Proof: Using Lemma 4.5.1, $\exists(T_i,\gamma_i,\omega_i,\rho_i,G_{r(i)}\ (i=1,2,\ldots,j))$

$$((\gamma_1 \in linear(T_1) \wedge \sigma\omega_{1\ k}\equiv_{r(1)} \gamma_1\rho_1 \wedge |\sigma| > |\rho_1|) \wedge$$

$$(\gamma_2 \in linear(T_2) \wedge \rho_1\omega_{2\ k(1)}\equiv_{r(2)} \gamma_2\rho_2 \wedge |\rho_1| > |\rho_2|) \wedge \ldots \wedge$$

$$(\gamma_j \in linear(T_j) \wedge \rho_{j-1}\omega_{j\ k(j-1)}\equiv_{r(j)} \gamma_j\rho_j \wedge |\rho_{j-1}| > |\rho_j|)).$$

Clearly, $G_{r(i)} \text{—}\omega_{\ i+1}\rightarrow^* G_{r(i+1)}$ for i=1,2, ...j-1.

$\sigma\omega_1\omega_2\ldots\omega_{j\ k}\equiv_{r(j)} \gamma_1\gamma_2\ldots\gamma_j\rho_j$ will be proven by induction on j.

Basis: Let j=1 then $\sigma\omega_{1\ k}\equiv_{r(1)} \gamma_1\rho_1$.

Induction: For the induction step suppose the claim holds for j-1, i.e.

$$\sigma\omega_1\omega_2\ldots\omega_{j-1\ k}\equiv_{r(j-1)} \gamma_1\gamma_2\cdots\gamma_{j-1}\rho_{j-1} \qquad (1)$$

It will be proven that it also holds for j.

$$(1) \wedge G_{r(j-1)} \text{—}\omega_{\ j}\rightarrow^* G_{r(j)} \Rightarrow \sigma\omega_{\ 1}\omega_{\ 2}\ldots\omega_{j-1}\ \omega_{j\ k}\equiv_{r(j)} \gamma_1\gamma_2\ldots\gamma_{j-1}\rho_{j-1}\omega_j. \qquad (2)$$

$$(2) \wedge \rho_{j-1}\omega_{\ j\ k(j-1)}\equiv_{r(j)} \gamma_j\rho_j \Rightarrow \sigma\omega_1\omega_2\ldots\omega_{j\ k}\equiv_{r(j)} \gamma_1\gamma_2\ldots\gamma_j\rho_j.$$

Since $|\sigma|$ is finite, $|\sigma| > |\rho_1|$, for i=1,2, ...j-1, $|\rho_i| > |\rho_{i+1}|$, and Lemma 4.5.1 can be applied to ρ_i as long as $|\rho_i| > 0$, it can be assumed that $|\rho_j| = 0$. Then one can write $\sigma\omega_{\ k}\equiv_r \gamma_1\gamma_2\ldots\gamma_j$ where $\omega=\omega_1\omega_2\ldots\omega_j$ and $G_{r(j)}=G_r$. []

Theorem 4.5.2 A global state is a deadlock state in sequential reachability analysis iff it is a deadlock state in simultaneous reachability analysis.

Proof: *If Part.* Straightforward from Theorem 4.5.1. Since every simultaneously reachable global state is sequentially reachable, every deadlock state in simultaneous reachability analysis is also a deadlock state in sequential reachability analysis.

Only If Part. Let $G_0 \text{—}\sigma\text{→}^* G_d$ and G_d is a deadlock state. From Lemma 4.5.2, $\sigma\omega_{\,0} \equiv_r \gamma_1\gamma_2...\gamma_j$ where $G_0 \text{—}T_1T_2...T_j\text{—»}^* G_r$ and $\gamma_i \in linear(T_i)$ for i=1,2,...,j. Since G_d is a deadlock state, ω must be empty. Therefore, $\sigma_{\,0} \equiv_d \gamma_1\gamma_2...\gamma_j$. ||

A global state G_b is said to be a *blocked state* if $exec(G_b)=\varnothing$. Notice that a deadlock state is a blocked state but a blocked state is not a deadlock state if at least one channel is not empty. The following result is immediate from the proof of Theorem 4.5.2.

Corollary 4.5.2 A global state is a blocked state in sequential reachability analysis iff it is a blocked state in simultaneous reachability analysis.

Theorem 4.5.3 A transition is executable at a sequentially reachable global state iff it is executable at a simultaneously reachable global state.

Proof: *If Part.* Straightforward from Theorem 4.5.1. Since every simultaneously reachable global state is sequentially reachable, every transition executable in simultaneous reachability analysis is also executable in sequential reachability analysis.

Only If Part. Let t be any transition such that t is a transition in σ and $G_0 \text{—}\sigma\text{→}^* G_m$ so that t is an executable transition at a sequentially reachable global state. From Lemma 4.5.2, $\sigma\omega_{\,0} \equiv_r \gamma_1\gamma_2...\gamma_j$ where $G_0 \text{—}T_1T_2...T_j\text{—»}^* G_r$ and $\gamma_i \in linear(T_i)$ for i=1,2,...,j. From Property 4.1.2, t is also in $\gamma_1\gamma_2...\gamma_j$. ||

Corollary 4.5.3 A transition is nonexecutable in protocol Π iff there is no simultaneously reachable global state at which this transition is executable.

4.6 Unspecified Reception Detection

A protocol has an unspecified reception at a global state G if a message x is at the head of an incoming queue of a process which is at a state s, but the protocol does not specify what

state the process should enter upon receiving x at state s (see Definition 2.3.7). Here, let us distinguish between the global state G and the pair (s,x) by referring the global state G as an *unspecified reception state* and the pair (s, x) as an *unspecified reception pair*. Unspecified reception pairs are the causes of unspecified reception states. Due to one unspecified reception pair, a protocol may have many unspecified reception states. Simultaneous reachability analysis given in Figure 4.11 does not guarantee the verification of the absence of unspecified reception states. In this section, an augmentation of a given protocol by some receiving transitions will be presented. When the given protocol is augmented with receiving transitions, simultaneous reachability analysis (in Figure 4.11) detects at least one unspecified reception state for every unspecified reception pair.

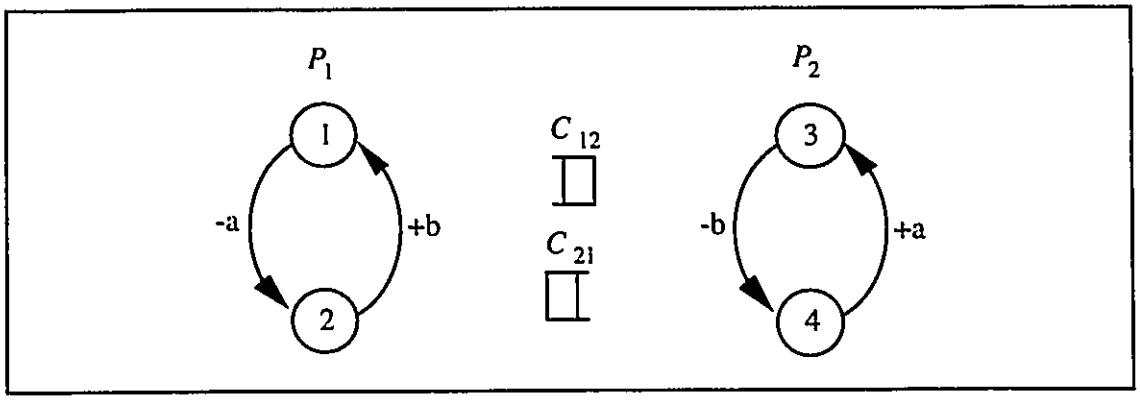


Figure 4.19 A protocol with unspecified receptions.

Consider the protocol given in Figure 4.19. States 1 and 3 are the initial states of the processes P_1 and P_2, respectively. P_1 can send message a at state 1 and message a can reach P_2 when P_2 is at state 3. Since reception of message a at state 3 is not specified, (<2,3>,<a,ε>) is an unspecified reception state and (3,a) is an unspecified reception pair for this protocol. Similarly, P_2 can send message b at state 3 and message b can reach P_1 when P_1 is at state 1. Since reception of message b at state 1 is not specified, (<1,4>,<ε,b>) is an unspecified reception state and (1,b) is an unspecified reception pair

for this protocol. However, simultaneous reachability analysis given in Figure 4.11 does not identify any of these unspecified reception states and pairs. Simultaneous reachability graph for this protocol is given Figure 4.20.

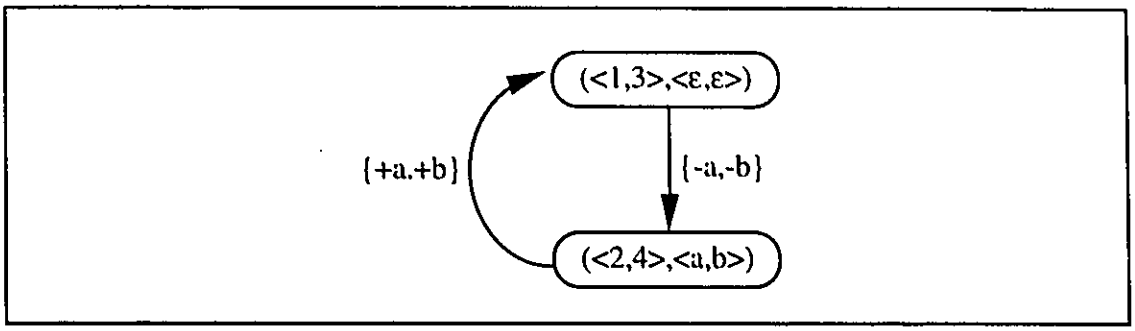


Figure 4.20 Simultaneous reachability graph of the protocol in Figure 4.19.

As we design a protocol for which simultaneous reachability analysis does not detect any unspecified reception pair, we can also design a protocol for which simultaneous reachability analysis detects all unspecified reception pairs by generating at least one simultaneously reachable unspecified reception state for each unspecified reception pair. To identify in which case the simultaneous reachability analysis detects unspecified reception pairs, we present the following lemma. The lemma states that if there exists a receiving transition at a state s of a process P_j for a message x sent from a process P_i then there exists at least one simultaneously reachable unspecified reception state for every unspecified reception pair (s,y) where y is a message sent from P_i.

Lemma 4.6.1

$\forall(i,j,x,y,s)$ $(i,j=1,2,...,n \ \wedge \ i\neq j \ \wedge x,y \in M_{ij} \ \wedge \ \delta_j(s,+x)$ is defined in $P_j \ \wedge \ \delta_j(s,+y)$ is not defined in $P_j \ \Rightarrow$

$(\exists(G_k,X)(G_0 \rightarrow^* G_k \ \wedge \ X \in M_{ij}^* \ \wedge \ s_j^k=s \ \wedge \ c_{ij}^k=yX) \ \Leftrightarrow$

$\exists(G_m,Y)(G_0 \text{—»}^* G_m \ \wedge \ Y \in M_{ij}^* \ \wedge \ s_j^m=s \ \wedge \ c_{ij}^m=yY)))$.

Proof: *If Part:* The claim holds from the statement of Theorem 4.5.1 which is

$$\forall(G_m)\ (G_0 \text{—»}^* G_m \ \Rightarrow\ G_0 \rightarrow^* G_m).$$

Only If Part. Let $G_0 \text{—}\sigma\rightarrow^* G_k$. From Lemma 4.5.2,

$$G_0 \text{—}\sigma\rightarrow^* G_k \ \wedge\ \sigma \neq \varepsilon \ \Rightarrow$$

$$\exists(G_r,\omega,l)\ (G_1 \text{—}T_1\text{—»}G_1 \text{—}T_2\text{—»}G_2 \wedge \ \ldots\ \wedge\ G_{l-1} \text{—}T_l\text{—»}G_r \wedge$$

$$\sigma\omega\ {}_0\equiv_r \gamma_1\gamma_2\ldots\gamma_l \ \wedge\ \gamma_a \in linear(T_a)\ (a=1,2,\ldots,l)).$$

Let $\gamma=\gamma_1\gamma_2\ldots\gamma_l$. Since $\sigma\omega\ {}_0\equiv_r \gamma$, for $h=1,2,\ldots,n$, $\sigma\downarrow_h \in prefix(\gamma\downarrow_h)$. This implies that there exist l' and l'' such that $l' \le l$, $l'' \le l$, $\sigma\downarrow_i=(\gamma_1\gamma_2\ldots\gamma_{l'})\downarrow_i$ and $\sigma\downarrow_j=(\gamma_1\gamma_2\ldots\gamma_{l''})\downarrow_j$. Clearly, $\sigma\downarrow_j=(\gamma_1\gamma_2\ldots\gamma_{l''})\downarrow_j$ implies $s_j^{l''}=s_j^k$. Let ZyX be a sequence of messages that P_i sends to P_j during the execution of σ (and $(\gamma_1\gamma_2\ldots\gamma_{l'})\downarrow_i$). Then Z is the sequence of messages P_j receives from P_i during the execution of σ (and $(\gamma_1\gamma_2\ldots\gamma_{l''})\downarrow_j$).

Case 1 $l' \le l''$.

This case implies that P_i sends ZyX before or when P_j receives Z during the execution of γ. After P_j receives Z, the protocol reaches $G_{l''}$ at which the content of C_{ij} will be yX. Since $c_{ij}^{l''}=yX$ and $s_j^{l''}=s_j^k=s$, let $G_m=G_{l''}$ and $Y=X$. Thus, the claim holds.

Case 2 $l'' < l'$.

This case implies that P_j receives Z before P_i sends ZyX during the execution of γ. However, Z can only be received by P_j if it is sent by P_i. When P_j receives Z, the content of C_{ij} (i.e. $c_{ij}^{l''}$) will be either yY or empty, where $Y \in prefix(X)$.

Case 2.1 $c_{ij}^{l''}=yY$.

Since $c_{ij}^{l''}=yX$ and $s_j^{l''}=s_j^k=s$, let $G_m=G_{l''}$. Thus, the claim holds.

Case 2.2 $c_{ij}^{l''}=\varepsilon$.

Since, in this case P_j receives Z before P_i sends yX, P_i sends yX independently from P_j. This implies that there always exists an executable transition from the processes in $P \setminus P_j$ until P_i sends yX.

Let $l'' \le q < l'$ such that $c_{ij}^a=\varepsilon$ for $a=l'', l''+1,\ldots,q$ and $c_{ij}^{q+1}=y$. It will be proven that there exists $\gamma_{l''}\gamma_{l''+1}\ldots\gamma_q$ such that $P_j \notin act(\gamma_{l''}\gamma_{l''+1}\ldots\gamma_q)$ by induction on q.

Basis: Let $q=l''$. Since $c_{ij}^{q}=\varepsilon$ and $s_j^{l''}=s_j^{k}=s$, $(s,+x,\delta_j(s,+x))$ is a potentially executable transition at $G_{l''}$. Clearly, there exists an executable transition from the processes in $P \setminus P_j$ at $G_{l''}$. Then, it is immediate that there exist $\gamma_{l''}$ such that $P_j \notin \mathrm{act}(\gamma_{l''})$.

Induction: For the induction step suppose that the claim holds for q-1, i.e. $P_j \notin \mathrm{act}(\gamma_{l''}\gamma_{l''+1}...\gamma_{q-1})$. It will be proven that it also holds for q. $P_j \notin \mathrm{act}(\gamma_{l''}\gamma_{l''+1}...\gamma_{q-1})$ implies $s_j^{q}=s_j^{l''}=s_j^{k}=s$. Since $c_{ij}^{q}=\varepsilon$ and $s_j^{q}=s$, $(s,+x,\delta_j(s,+x))$ is a potentially executable transition at G_q. Clearly, there exists an executable transition from the processes in $P \setminus P_j$ at G_q. Then it is immediate that there exist γ_q such that $P_j \notin \mathrm{act}(\gamma_q)$ and therefore, $P_j \notin \mathrm{act}(\gamma_{l''}\gamma_{l''+1}...\gamma_q)$. This completes the induction.

$P_j \notin \mathrm{act}(\gamma_{l''}\gamma_{l''+1}...\gamma_q)$ implies that $s_j^{q+1}=s$. Since $s_j^{q+1}=s$ and $c_{ij}^{q+1}=y$, let $G_m=G_{q+1}$ and $Y=\varepsilon$. Thus, the claim holds. ||

From Lemma 4.6.1, we immediately conclude that we need only the existence of a receiving transition at a state s of a process P_j for a message x sent from a process P_i to detect at least one simultaneously reachable unspecified reception state for every unspecified reception pair (s, y) where y is a message sent from P_j. Notice that the lemma does not require that the receiving transition $(s,+x,\delta_j(s,+x))$ should be executable. The result of Lemma 4.6.1 directs us to a simple protocol augmentation algorithm in which we add *extra receiving transitions* to the original protocol. By replacing protocol Π by its augmented version, simultaneous reachability analysis identifies every unspecified reception pair in protocol Π. The following is a constructive definition of extra receiving transitions of a protocol Π.

Definition 4.6.1 Let $@_{ij}$ represents the extra message for M_{ij}, which is not included in the message set of any process in Π. For every $1 \le i,j \le n$ $(i \neq j)$ and $s \in Q_j$, receiving transition $(s,+@_{ij},s)$ is said to be *extra receiving transition* for Π if P_j does not include a

receiving transition at state s of process P_j for any message x from M_{ij} (i.e. sent from process P_i). Let E be set of all extra receiving transitions for Π. []

Using Definition 4.6.1, we directly obtain the algorithm given in Figure 4.21 to obtain E for a protocol Π. Since for each process state, there exist at most n-1 extra receiving transitions, $|E|$ can be at most $|S|*(n-1)$ where $|S|$ is the total number of process states in Π, i.e. $\bigcup_{j=1}^{n} S_j = S$.

```
E=∅
for each process Pj of protocol Π do
  for each state s of Pj do
    for each process Pi such that Pi≠Pj do
      if ((Mij≠∅) and (δj(s,+x) is undefined for each x∈Mij)) then
        E=E∪(s,+@ij,s)
```

Figure 4.21 Finding extra receiving transitions for a protocol.

Let h=1,2,...,l and E_h be subsets of E, which is obtained by partitioning E into l disjoint subsets. Let Π' and Π'_h be the protocols obtained by adding the transitions in E and E_h, respectively, to Π. The following results are immediate from the facts that

- $\bigcup_{h=1}^{l} E_h = E$ includes all necessary transitions to satisfy the condition of Lemma 4.6.1
- detection of unspecified reception states for an unspecified reception pair is independent of the detection of unspecified receptions states for the other pairs.

Corollary 4.6.1 The simultaneous reachability analysis of Π' identifies every unspecified reception pair in Π.

Corollary 4.6.2 The simultaneous reachability analyses of all Π'_h identify every unspecified reception pair in Π.

Let R^{Π}_{Seq} (R^{Π}_{Sim}), $R^{\Pi'_h}_{Seq}$ ($R^{\Pi'_h}_{Sim}$), and $R^{\Pi'}_{Seq}$ ($R^{\Pi'}_{Sim}$) be the sets of sequentially (simultaneously) reachable global state spaces of Π, Π'_h and Π', respectively. Since all extra receiving transitions in E_h (or E) are nonexecutable transitions in Π'_h (or Π') $R^{\Pi}_{Seq}=R^{\Pi'}_{Seq}=\bigcup_{h=1}^{l} R^{\Pi'_h}_{Seq}$. However, this is not the case for simultaneously reachable global state space. In the simultaneous reachability analysis of Π'_h (or Π'), extra receiving transitions may turn into potentially executable transitions and potentially executable transitions increase the number of selected simultaneously executable sets. Therefore, these extra receiving transitions may lead to generation of extra global states. As a result, $R^{\Pi}_{Sim} \subseteq R^{\Pi'_h}_{Sim} \subseteq R^{\Pi'}_{Sim}$, for h=1,2,...,l. Due to the fact l independent runs can be executed in parallel and the space requirement is the major problem in reachability analysis, one should partition E into several disjoint subsets.

In chapter 5, we provide empirical results to demonstrate the effectiveness of simultaneous reachability analysis. For the detection of unspecified receptions, we divide the unspecified reception detection task into n independent runs where in each run we augment the protocol with extra receiving transitions of one process. The augmentation algorithm for a process P_j is given in Figure 4.22.

```
Π'j=Π
for each transition (s,+@ij,s)∈E do
  add (s,+@ij,s) to Π'j
```

Figure 4.22 Protocol augmentation algorithm for process P_j.

For the protocol given in Figure 4.19, E={(1,+@$_{21}$,1),(3,+@$_{12}$,3)}. By using the augmentation algorithm given in Figure 4.22, we obtain two augmented protocols Π'_1 (see Figure 4.23) and Π'_2 (see Figure 4.25) for process P_1 and P_2, respectively. The

simultaneous reachability graph for Π'_1 is given in Figure 4.24 where unspecified reception state (<1,4>,<ε,b>) and therefore unspecified reception pair (1,b) are detected. Similarly, the simultaneous reachability graph for Π'_2 is given in Figure 4.26 where unspecified reception state (<2,3>,<a,ε>) and therefore unspecified reception pair (3,a) are detected.

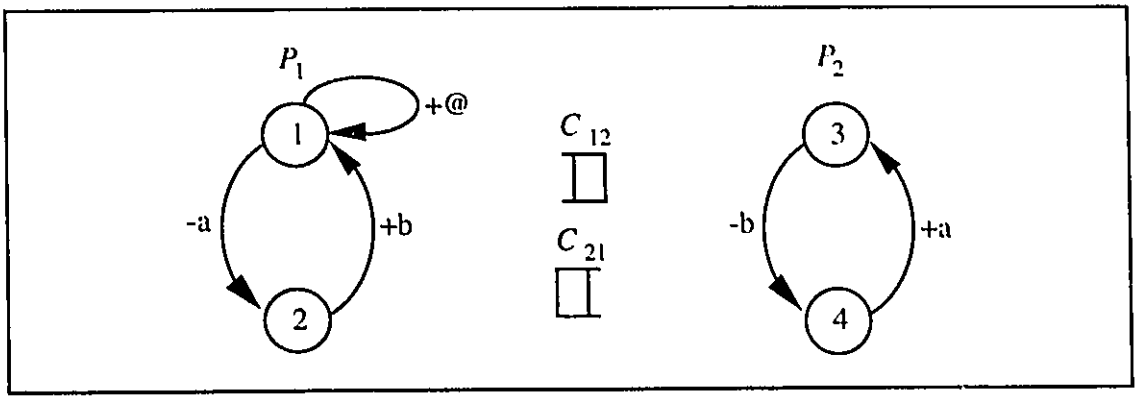


Figure 4.23 Augmented version Π'_1 of the protocol in Figure 4.19 for process P_1.

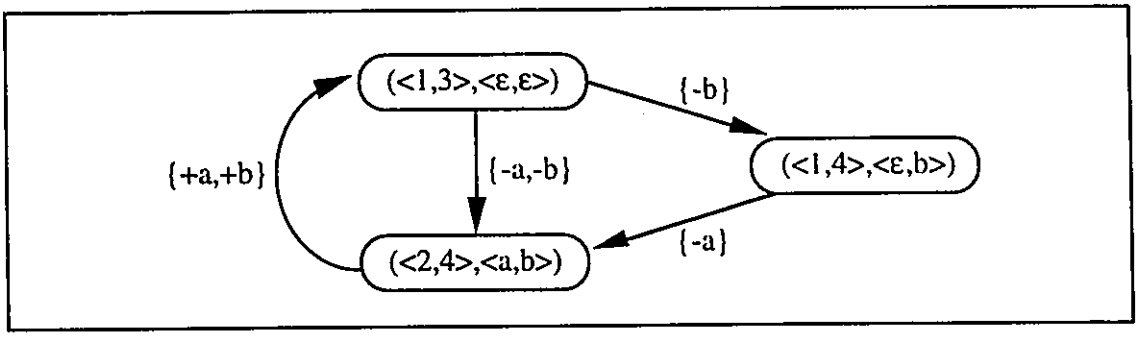


Figure 4.24 The simultaneous reachability graph of the protocol in Figure 4.23.

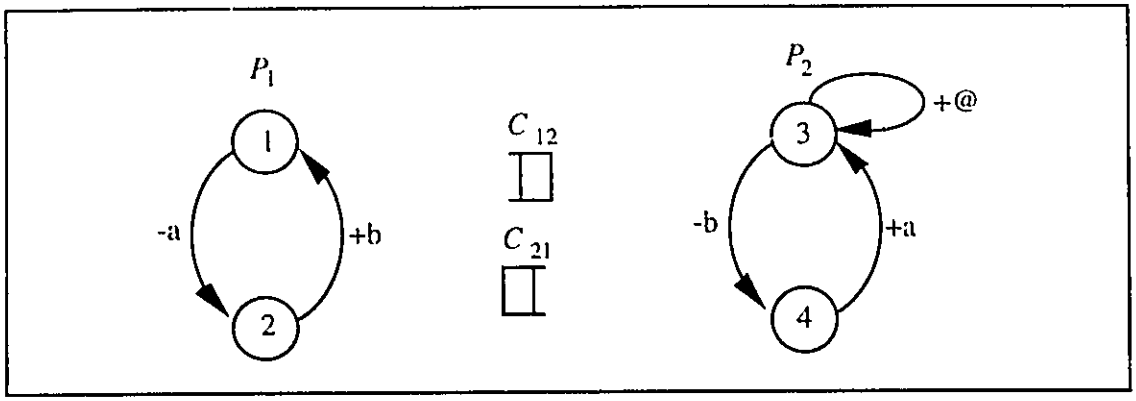


Figure 4.25 Augmented version Π'_2 of the protocol in Figure 4.19 for process P_2.

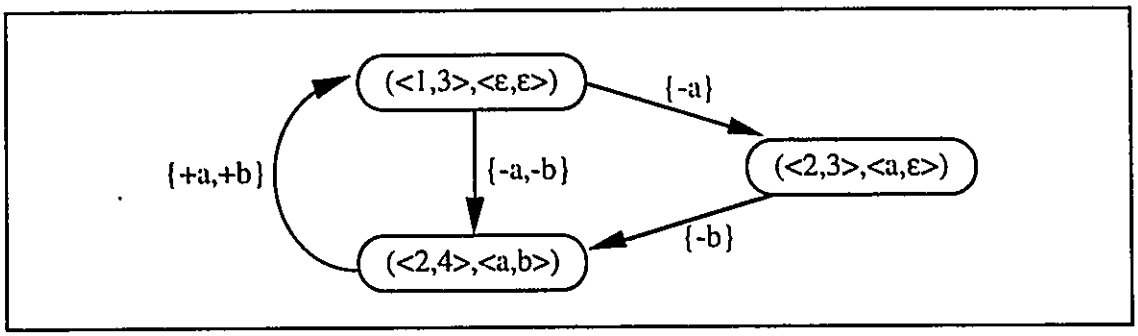


Figure 4.26 The simultaneous reachability graph of the protocol in Figure 4.25.

4.7 Buffer Overflow Detection

A buffer overflow usually occurs when one process sends messages faster than the other process consumes those messages. Simultaneous reachability analysis given in Figure 4.11 does not guarantee the verification of the absence of buffer overflows because several processes are forced to execute transitions at the same time. In the rest of this section, an augmentation of simultaneous reachability analysis will be proposed and it will be proven that the augmented simultaneous reachability analysis detects all channels which are overflowed.

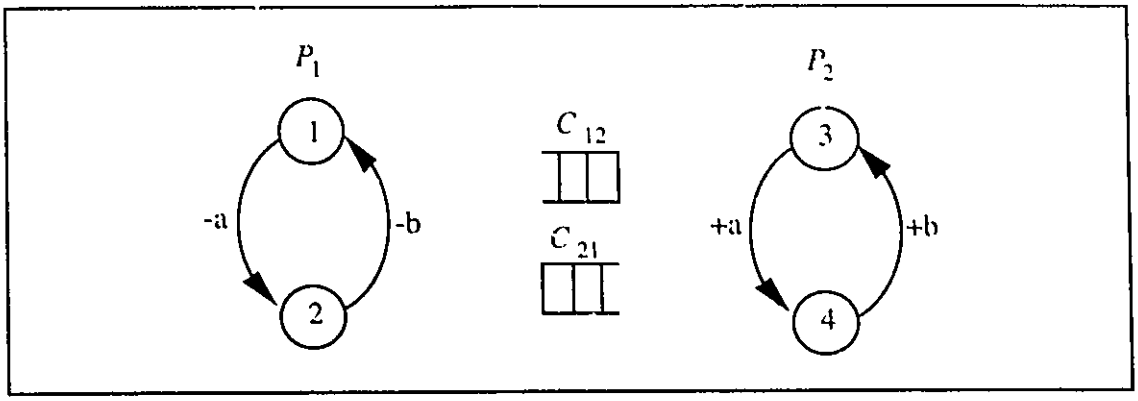


Figure 4.27 A protocol with buffer overflows.

Consider the protocol given in Figure 4.27. If process P_2 is not fast enough to consume (receive) messages a and b immediately after P_1 sends them, channel C_{12} will eventually be overflowed. However, simultaneous reachability analysis given in Figure 4.11 does not detect this buffer overflow. The simultaneous reachability graph of this protocol is given in Figure 4.28.

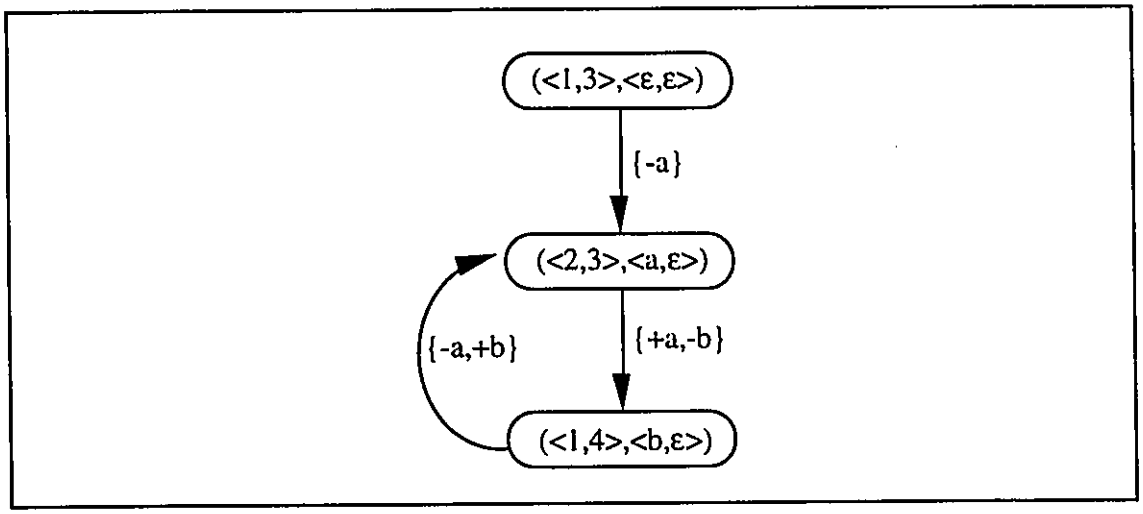


Figure 4.28 Simultaneous reachability graph of the protocol given in Figure 4.27.

The augmentation of simultaneous reachability analysis is based on a modification of the algorithm given in Figure 4.10 for the construction of selected simultaneously

executable sets. The modified algorithm for *selected*(G_k) is obtained by replacing `if` pot($s^k_{p(h)}$)$\neq\emptyset$ `then` with `if` (pot($s^k_{p(h)}$)$\neq\emptyset$ `or` ($c^k_{ip(h)}\neq\varepsilon$ `and` `i`$\neq$`p(h))) then` in Figure 4.10. The result of this modification is that, for each channel, if a selected simultaneously executable set includes a transition for the reception of a message from the channel then one more simultaneously executable set will be selected by removing that receiving transition from the set. This means that if channel C_{ij} is not empty, the progress of P_j will be blocked as long as the other process have executable transitions. Therefore, the possibility of overflows in each channel will be analyzed by forcing the channel overflow. However, this requires the application of simultaneous reachability analysis once for detecting at least one buffer overflow in each channel C_{ij}, henceforth denoted by ASRA_{ij}. It is important to note that all ASRA_{ij} utilizing the modified algorithm for *selected*(G_k) can be independently executed which implies that at most $n*(n-1)$ independent runs are required. In fact, for the empirical study reported in section 7, the detection of buffer overflows is based on the application of all ASRA_{ij} for each protocol used in the study.

Theorem 6.1 A buffer overflow (if exists) in C_{ij} is detected by ASRA_{ij}.

Proof: Let $G_0 \text{—}\sigma\text{→}^* G_k$, $|c^k_{ij}| = B_{ij}, \delta_i(s^k_i, -x)$ and $x \in M_{ij}$ be defined so that σ leads to a buffer overflow in C_{ij} at G_k. From Lemma 4.5.2,

$$G_0 \text{—}\sigma\text{→}^* G_k \wedge \sigma\neq\varepsilon \Rightarrow$$
$$\exists(G_r,\omega,l)\ (G_0 \text{—}T_1T_2...T_l\text{—»}^* G_r \wedge \sigma\omega_0 \equiv_r \gamma_1\gamma_2...\gamma_l \wedge \gamma_a \in \mathit{linear}(T_a)$$
$$(a=1,2,...,l)).$$

Let $\gamma=\gamma_1\gamma_2...\gamma_l$. Since $\sigma\omega_0 \equiv_r \gamma$, for $h=1,2,...,n$, $\sigma\downarrow_h \in \mathit{prefix}(\gamma\downarrow_h)$. This implies that there exist l' such that $l'\leq l$ and $\sigma\downarrow_j=(\gamma_1\gamma_2...\gamma_{l'})\downarrow_j$. Let $\sigma\downarrow_j\mu=\gamma\downarrow_j$. First, it will be shown that μ does not include any transition for the reception of a message from C_{ij} by induction on $d=l-l'$.

Basis: Let $d=0$ then $\sigma\downarrow_j=\gamma\downarrow_j$ and the claim trivially holds.

Induction: For the induction step suppose that the claim holds for d-1. It will be proven that it also holds for d. Let $G_0 \text{—}\gamma_1\gamma_2...\gamma_{l-1}\rightarrow^* G_{l-1}$. The content of C_{ij} at G_{l-1} is either empty or nonempty.

Case 1 $c_{ij}^{l-1}=\varepsilon$.

C_{ij} is empty, no transition for the reception of a message from C_{ij} is executable at G_{l-1}. Then γ_l and therefore μ do not include a transition for the reception of a message from C_{ij}.

Case 2 $c_{ij}^{l-1}\neq\varepsilon$.

In $ASRA_{ij}$, if a selected simultaneously executable set includes an executable transition for the reception of a message from the channel then one more simultaneously executable set will be selected by removing that receiving transition from the set. This guarantees that there exists a γ_l and therefore μ such that they do not include a transition for the reception of a message from C_{ij}.

Therefore, $\sigma\downarrow_j=\gamma\downarrow_j$ and μ does not include any transition for the reception of a message from C_{ij}. Since $\sigma\downarrow_i \in prefix(\gamma\downarrow_i)$, there exist l'' such that $l''\leq l$ and $\sigma\downarrow_i=(\gamma_1\gamma_2...\gamma_{l''})\downarrow_i$. Let $G_0 \text{—}\gamma_1\gamma_2...\gamma_{l''}\rightarrow^* G_{l''}$. Since it has just been shown that $\sigma\downarrow_j\mu=\gamma\downarrow_j$ and μ does not include any transition for the reception of a message from C_{ij}, it must be $|c_{ij}^{l''}|=B_{ij}$. Clearly, $\sigma\downarrow_i=\gamma_1\gamma_2...\gamma_{l''}\downarrow_i$ implies $s_i^{l''}=s_i^k$. It will be shown that $\sigma\downarrow_i=\gamma\downarrow_i$, $|c_{ij}^{r}|=B_{ij}$ and $s_i^r=s_i^k$ by induction on $d'=l-l''$.

Basis: Let $d'=0$ then trivially $\sigma\downarrow_i=\gamma\downarrow_i$.

Induction: For the induction step suppose that the claim holds for d'-1. It will be proven that it also holds for d'. Let $G_0 \text{—}\gamma_1\gamma_2...\gamma_{l-1}\rightarrow^* G_{l-1}$ then $|c_{ij}^{l-1}|=B_{ij}$ and $s_i^{l-1}=s_i^k$ from the hypothesis. This implies that $(s_i^k,-x,\delta_i(s_i^k,-x))$ is a potentially executable transition at G_{l-1}. Thus, it can be guaranteed that there exist γ_l such that it does not include a transition of P_i. So $\sigma\downarrow_i=\gamma\downarrow_i$, $|c_{ij}^{r}|=B_{ij}$ and $s_i^r=s_i^k$. []

The $ASRA_{12}$ of the protocol in Figure 4.27 is given in Figure 4.29. Notice that at global state (<2,3>,<a,e>), {(2,-b,1),(3,+a,4)} is a selected simultaneously executable set.

Since channel C_{12} is not empty, we obtain another selected simultaneously executable set {(2,-b,1)} by removing the transition (3,+a,4) of process P_2 from {(2,-b,1),(3,+a,4)}. The execution of this extra set leads to detection of a buffer overflow in C_{12}.

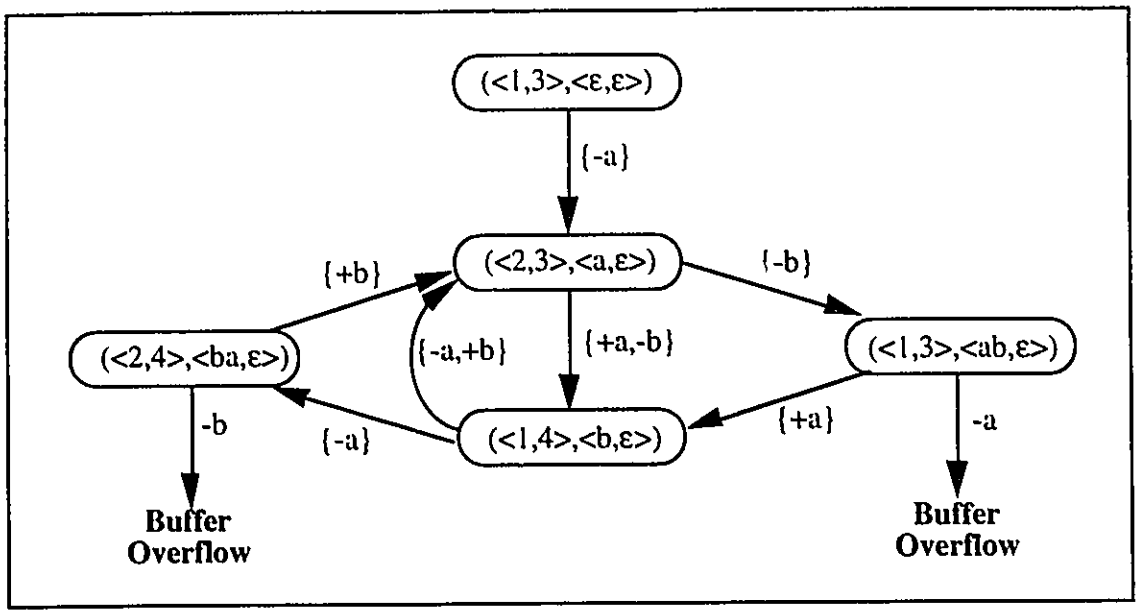


Figure 4.29 The $ASRA_{12}$ of the protocol in Figure 4.27.

Chapter 5
EMPIRICAL RESULTS

This chapter presents the empirical results obtained by applying both sequential and simultaneous reachability analyses to the same set of protocols and by comparing both analyses in terms of their space and time requirements. The set of protocols used in the study are constructed by an automatic protocol synthesizer, called APS.

The aim of protocol synthesis methods is to construct a correct protocol from an incomplete protocol description. Synthesis methods can be classified as service oriented and non-service oriented synthesis methods [PrSa91]. Service oriented methods construct functionally correct protocols while non-service oriented methods construct logically correct protocols. The protocols that we automatically synthesize will be used as input to empirical comparison of reachability based analysis methods for the verification of the logical correctness of protocols. Therefore, our synthesis method will be a non-service oriented synthesis method. Although, the synthesis methods intend to obtain correct protocols, our aim is not to obtain totally error free protocols. Since, in some sense our synthesizer simulates the protocol designer, the constructed protocols may contain some logical errors. We require that our synthesizer construct n CFSMs such that the compositional behavior of these processes includes a significant number of reachable global states and very limited number of logical errors for the purpose of comparing sequential and simultaneous reachability analyses. APS builds protocols from a set of parameters by using a random number generator function.

APS uses a set of parameters such as $MinP$, $MaxP$, $MinS(n)$, $MaxS(n)$ where $n=MinP, MinP+1,...,MaxP$ and $MinP$ ($MaxP$) is the minimum (maximum) number of processes in a protocol. For each n-process protocol, $MinS(n)$ ($MaxS(n)$) is the minimum (maximum) number of states in a process. 300 protocols have been constructed by choosing the values of these parameters as follows: $MinP$=2, $MaxP$=6, $MinS(n)$ =2 and $MaxS(n)$=22-2*n for $n=MinP, MinP+1,...,MaxP$. For every protocol synthesized in this experiment, the channel bounds are set to 3 messages.

APS requires a random number generator function $RS(low, high)$ which randomly selects an integer between *low* and *high* where *low* and *high* are integers and $low \leq high$. We implemented this function in C using random number generator function lrand48 in UNIX. APS consists of two main procedures. The first procedure produces an incomplete specification of the protocol by randomly specifying transition function for sending transitions. The second procedure completes the incomplete protocol specification by specifying necessary receiving transitions using sequential reachability analysis. The first procedure of APS includes 5 steps.

1. Randomly select the number n of processes in the protocol ($n=RS(MinP, MaxP)$).

2. Randomly specify communicating pair of processes such that every process communicates with at least one process. We define an $n*n$ symmetric matrix CM such that $CM(i,j)=CM(j,i)=1$ indicates that P_i communicates with P_j.

3. For each process, randomly select the number of process states (for $i=1,2,...,n$ $|S_i|=RS(MinS(i),MaxS(i))$.

4. For each process, randomly select the maximum number of distinct messages that can be sent from this process. The number of messages is limited with the half of the number of process states. Then for each process, create the message sets to be

sent and for each message, randomly select the process to which this message is to be sent.

```
m=1
for i=1 to n do
  for k=1 to RS(MinS(i),|S_i|)*0.5 do
  begin
    Randomly select j such that CM(i,j)=1
    M_ij=M_ij∪{m}
    m=m+1
  end
```

Figure 5.1 Step 4 of APS.

5. Define the transition function for sending transitions such that the process graph does not include any cycle of sending transitions. In sequential reachability analysis if any sending transition in a cycle of sending transitions becomes executable then this cycle can be continuously executed until one of the corresponding channels is overflowed. Therefore, such a cycle either leads to a buffer overflow or becomes a set of nonexecutable transitions in a protocol. This implies this assumption is not only an admissible assumption but also a necessary condition for a logically correct protocol. A process state is a *sending* (*receiving*) state if all the transitions specified in this state are sending (receiving) transitions. A process state is a *mixed* state if both sending and receiving transitions are specified in this state. In order to create approximately equal number of send, receive and mixed process states, APS attempts to define sending transitions $|S_i|*0.66$ times for process P_i. In each attempt, it randomly select two process states and a message for the sending transition. The following is the algorithm for this step:

```
for i:=1 to n do
  for k:=1 to 0.66*|S_i| do
  begin
    Randomly select the current state (CS=RS(1,|S_i|))
    Randomly select m∈M_i such that δ_i(CS,-m) is not defined
    if there is such m then
    begin
      Randomly select the next state NS such that the edge (CS,NS) does
      not lead to a cycle in the process graph
      If there exists such NS
        Define the transition function such that δ_i(CS,-m)=NS
    end
  end
```

Figure 5.2 Step 5 of APS.

The second procedure of APS consists of the following two steps:

6. Let US_i be the set of states of process P_i such that the states in US_i are not reachable from the initial state of P_i. For i=1,2,...,n, construct US_i by applying depth-first search algorithm to the process graph of P_i.

7. Systematically generate all sequentially reachable global states of the protocol. If a global state does not include a receiving transition for a message at the head of a channel then specify a receiving transition for this message as follows:

Constructing protocols from an incomplete description by using sequential reachability analysis is also used in some protocol synthesis methods, for example [Sidh82, ZaTa88]. Such a synthesis method does not prevent deadlock (when n>2) and buffer overflow states. In order to generate unspecified reception states and nonexecutable transitions, the specification of 25% of necessary receiving transitions is ignored in the second procedure.

```
A=∅
W={G0}
while (W≠∅) do
begin
  remove an element (Gk) from W
  add Gk to A
  for each c^k_ij (i≠j) do
   if (c^k_ij=xX and δj(s^k_j,+x) is undefined) then
    if RS(1,1000)>250 then /* Define with the probability of 0.75 */
    begin
      /* Find the next process state to define the transition */
      if USj ≠ ∅ then
      begin
        randomly select next_state from USj
        remove next_state from USj
      end
      else
        randomly select next_state from Sj
      δj(s^k_j,+x)=next_state
    end
  end
  for each Gm such that Gk→Gm do
   if (Gm is not in A or W) add Gm to W
```

Figure 5.3 Step 7 of APS.

Thus, it is guaranteed that the synthesized protocols can contain every type of logical errors. As expected, the number of global states of some of the synthesized protocols is either too small or too large. Such protocols are eliminated by setting lower and upper bounds on the number of global states. For n-process protocols, the lower bound is $500*2^n$ and the upper bound is 300000 global states. APS synthesized roughly equal number of protocols for each value of n: the numbers of synthesized protocols for

n=2,3,...,6 are 55,61,68,57,59, respectively. For each n-process protocol, the structural properties of processes such as the number of process states, the number of sending transitions and the number of receiving transitions disperse very well. This can be seen from Table 5.1 where standard deviations for these properties are relatively high with respect to their averages.

Both sequential and simultaneous reachability analyses as well as APS were implemented using C under UNIX. The generated reachability graphs are stored in the memory as balanced trees. All three implementations share the same data structures and functions for maintaining and searching a balanced tree. The results of sequential reachability analyses of the synthesized protocols are summarized in Table 5.2. The standard deviations show that the numbers of sequentially reachable global states of protocols disperse very well in each class of n-process protocols. Table 5.2 also presents results on the amount of time consumed and logical errors detected by sequential reachability analysis. The last row of Table 5.2 shows the results on the average number of processes having executable transitions at a global state in a protocol. This average can be viewed as the *concurrency level* in a protocol. Table 5.3 shows that the reduction obtained by using simultaneous reachability analysis is proportional with the number of processes in the synthesized protocols, more specifically, with the level of concurrency. This fact is more clearly presented in Table 5.4 where protocols are categorized with respect to concurrency level. Table 5.4 directly supports our motivation for simultaneous reachability analysis. As anticipated, the higher the concurrency level, the larger the reduction in the number of global states. As can be from the last row of Table 5.2, the concurrency levels are at most half of the number of processes in the synthesized protocols. From Table 5.3 and Table 5.4, one can observe that with this level of concurrency, a reduction up to 70-80% in the number of reachable global states is obtained by using simultaneous reachability analysis in place of sequential reachability analysis.

A significant reduction in time for deadlock and nonexecutable transition detection is obtained by using simultaneous reachability analysis (see Tables 5.5 and 5.6). However, in Table 5.5, it is shown that simultaneous reachability analysis consumes up to 17.2% more time for unspecified reception detection when compared to sequential reachability analysis. The reason for that is the following. A given protocol is augmented with extra receiving transitions for detecting unspecified receptions. The augmentation algorithm does not guarantee that each of these extra transitions is necessary for unspecified reception detection. Some of the "unnecessary" extra receiving transitions may turn into potentially executable transitions, and therefore, lead to more selected simultaneously executable sets to be computed during simultaneous reachability analysis. This may increase the time to be consumed by simultaneous reachability analysis to detect unspecified receptions especially for the protocols with low concurrency level (for which simultaneous reachability analysis cannot reduce the state space substantially).

The reason for the small increase in time (up to 6.81% in Table 5.5) for buffer overflow detection is the modification of the algorithm for selection of simultaneous executable sets to generate extra simultaneously executable sets. The modified algorithm does not guarantee that every generated extra simultaneously executable set is necessary for buffer overflow detection. Because of the computation of "unnecessary" extra selected simultaneous executable sets, simultaneous reachability analysis may consume more time than sequential reachability analysis does for the detection of buffer overflows especially for the protocols with low concurrency level.

As can be seen from Table 5.6, when the concurrency level is high (see the last column in Table 5.6) simultaneous reachability analysis does not consume more time than sequential reachability analysis does for the detection of all the logical errors considered in this study. Note that simultaneous reachability analysis reduces the number of global states substantially when the concurrency level is high, in other words, when simultaneous

reachability analysis reduces the number of global states substantially, it does not consume more time than sequential reachability analysis does. Since the concurrency is the major contributor to the state explosion problem, simultaneous reachability analysis relieves the major problem without costing extra time.

From Figure 5.4, it is observed that the synthesized protocols are well-distributed with respect to the number sequentially reachable global states in the classes of 5 and 6-process protocols. For 2, 3 and 4-process protocols, the distribution has a similar pattern but is skewed to the left due to the relatively small average (see Table 5.2). This well-distribution property of protocols makes possible to infer the expected efficiency of simultaneous reachability analysis from the results presented in this section. Consequently, simultaneous reachability analysis reduces substantially the number of global states without consuming more time than consumed by the sequential reachability analysis.

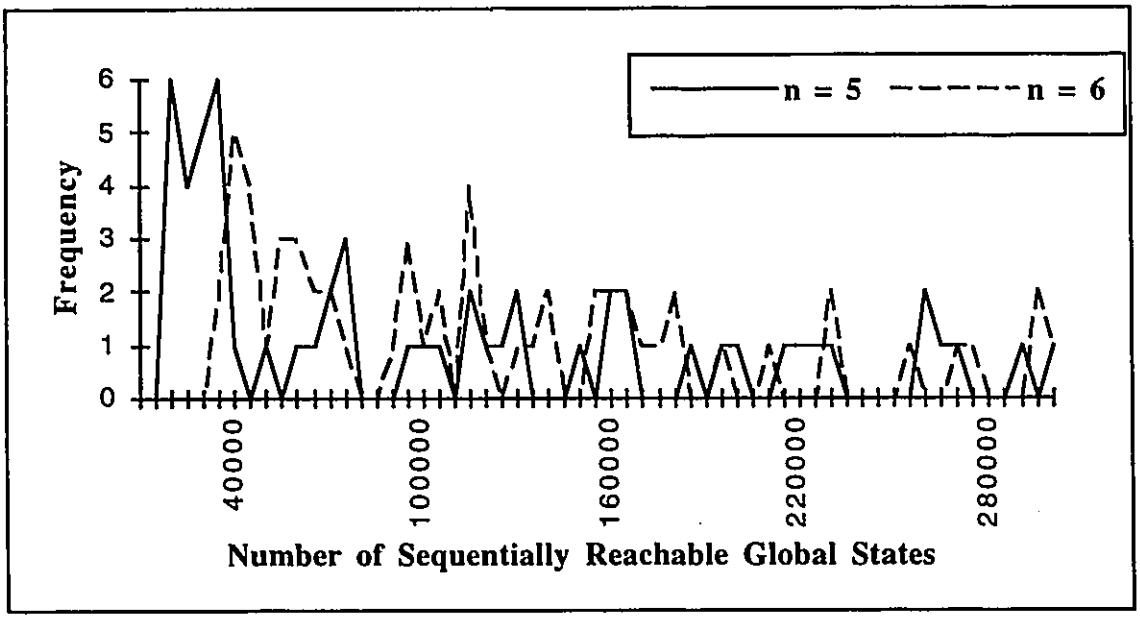


Figure 5.4 Frequency distribution of the number of sequentially reachable global states.

Table 5.1 Summary of the properties of the synthesized protocols

Attributes	A : Average SD: Standard Deviation	Results for 2 Process Protocols	3	4	5	6
Number of processes communicating with a process	A	1.00	1.38	1.43	1.68	1.63
	SD	0.00	0.24	0.34	0.47	0.55
Number of states in a process	A	11.70	8.68	7.28	6.46	5.58
	SD	16.65	15.44	11.18	8.03	5.05
Number of sending transitions in a process state	A	2.37	1.54	1.22	0.94	0.89
	SD	1.70	1.75	1.72	1.24	1.18
Number of receiving transitions in a process state	A	0.69	0.69	0.72	0.71	0.70
	SD	0.55	0.49	0.47	0.47	0.43

Table 5.2 Summary of the results of the reachability analysis of the synthesized protocols

Attributes	A : Average SD: Standard Deviation	Results for 2 Process Protocols	3	4	5	6
Number of sequentially reachable global states	A	36498	51773	67702	104654	119511
	SD	43614	56886	78625	85852	75656
Time consumed by sequential reachability analysis (in sec)	A	21.30	41.72	65.55	117.60	155.32
	SD	27.65	51.12	85.87	104.65	124.91
Percentage of deadlock states in sequentially reachable global states	A	0.17	0.14	0.06	0.03	0.01
	SD	0.22	0.20	0.08	0.05	0.02
Ratio of the number of unspecified receptions to the number of specified transitions	A	0.24	0.22	0.21	0.18	0.19
	SD	0.08	0.07	0.08	0.07	0.07
Ratio of the number of overflowed channels to the number of channels	A	1.00	0.67	0.50	0.37	0.32
	SD	0.00	0.18	0.17	0.11	0.09
Percentage of the number of nonexecutable transitions to the number of specified transitions	A	0.05	4.45	10.33	13.01	13.40
	SD	0.34	4.71	7.94	7.51	8.84
Average number of processes having executable transitions at a global state in a protocol	A	0.38	1.23	2.05	2.63	2.90
	SD	0.06	0.26	0.23	0.28	0.32

Table 5.3 Global state reduction obtained by using simultaneous reachability analysis and classified with respect to the number of processes in protocols.

Logical Errors	Average Reduction in the Number of Global States for Process Protocols (%)				
	2	3	4	5	6
Deadlock and Nonexecutable Transition	48.47	57.65	63.55	68.32	73.63
Unspecified Reception	48.42	51.02	50.27	54.11	55.56
Buffer Overflow	21.89	34.95	42.79	56.70	60.74

Table 5.4 Global state reduction obtained by using simultaneous reachability analysis and classified with respect to the concurrency level in protocols.

Logical Errors	Average Reduction in the Number of Global States for Concurrency Levels (%)			
	[0,1]	(1,2]	(2,3]	(3,4]
Deadlock and Non-executable Transition	51.93	56.30	68.41	80.85
Unspecified Reception	51.49	46.91	51.85	70.97
Buffer Overflow	25.23	35.69	53.06	70.02

Table 5.5 Time reduction obtained by using simultaneous reachability analysis and classified with respect to the number of processes in protocols.

Logical Errors	Average Reduction in Time for Process Protocols (%)				
	2	3	4	5	6
Deadlock and Nonexecutable Transition	43.14	47.06	46.36	41.66	43.54
Unspecified Reception	43.06	35.59	16.23	1.21	-17.22
Buffer Overflow	-3.21	-1.61	-6.81	4.30	-3.66

Table 5.6 Time reduction obtained by using simultaneous reachability analysis and classified with respect to the concurrency level in protocols.

Logical Errors	Average Reduction in Time for Concurrency Levels (%)			
	[0,1]	(1,2]	(2,3]	(3,4]
Deadlock and Non-executable Transition	45.20	43.97	42.55	56.11
Unspecified Reception	44.28	25.75	-5.46	22.14
Buffer Overflow	-4.69	-0.31	-4.47	11.32

Chapter 6
CONCLUDING REMARKS FOR PART I

A method called simultaneous reachability analysis has been proposed in the first part of the thesis to reduce the number of global states need to be analyzed for the verification of the logical correctness of n-process ($n \geq 2$) protocols with bounded channels without placing any restrictions on the topology of the protocol and on the structure of the processes. The method is based on generating global states by means of simultaneously executing several transitions at a global state. The set of such simultaneously executable transitions is called a simultaneously executable set. A distinction is made between executable and potentially executable transitions at a global state. A simultaneously executable set consists of at most one executable transition from each process and there is no restriction on the relative number of sending or receiving transitions in this set.

It is observed that selecting larger simultaneously executable sets leads to a better reduction in the number global states. Therefore, simultaneous reachability analysis always selects maximal simultaneously executable sets. However, it is shown that the existence of potentially executable transitions requires the selection of extra simultaneously executable sets to detect logical errors. Consequently, simultaneous reachability analysis derives extra simultaneously executable sets from maximal ones using the information about the existence of potential transitions at a global state.

'. ıe idea of the simultaneous execution of transitions was previously proposed by Itoh and Ichikawa in [ItIc83]. Their method is limited to a restricted class of protocols where processes must synchronize on their initial states after a finite number of transitions and only the cycles passing through the initial states are allowed in each process. The verification problem in [ItIc83] is described as follows: Given a set of multilogues (each of which is an n-tuple of paths such that ith path is a path from the initial state to the initial state in the directed graph of process P_i), check whether for each multilogue, there exists an execution sequence from the initial global state to the initial global state, where each process traverses its path in that multilogue. Apparently, their verification problem is different from the verification problem described in this thesis. While utilizing the idea of the simultaneous execution of transitions in a more general protocol model for a more general verification problem, we investigate potentially executable sending transitions. Moreover, the verification of the absence of unspecified receptions has required a new algorithm for preprocessing of the given protocol structure while buffer overflows has required the augmentation of simultaneous reachability analysis.

It is proven that simultaneous reachability identifies every deadlock, every nonexecutable transition, every missing pair of process state and message causing an unspecified reception, and every channel which is overflowed in a protocol. In the empirical study reported in this part, 300 protocols are constructed by an automatic protocol synthesizer using a random number generator. The average number of processes having executable transitions at a global state in a protocol is viewed as the concurrency level in the protocol. The results indicate that the reduction (in the number global states) obtained by simultaneous reachability analysis is proportional to the concurrency level of a protocol. The concurrency levels are at most half of the number of processes in the synthesized protocols. With this level of concurrency, a reduction up to 70-80% in the number of reachable global states is obtained by using simultaneous reachability analysis in place of

sequential reachability analysis without consuming more time than consumed by the sequential reachability analysis.

Simultaneous reachability and fair reachability analyses can be considered in the same class of reduction methods because both analyses reduce the number of global states by jumping from one global state to another by means of executing a set and vector of several transitions, respectively. The first version of fair reachability analysis [RuWe82] is proposed for two-process protocols by forcing two processes progress with equal speed. This property is called the *fair progress property*. However, due to the existence of potentially executable transitions, this property is not sufficient to verify the absence of deadlocks in n-process protocols, $n>2$ (see section 4.3). In fact, Liu and Miller [LiMi94] use another property called the *equal channel property* of fair reachability analysis to generalize it for n-process cyclic protocols. In cyclic protocols, each process has exactly one input and one output channel while in n-process protocols, a process may have several input and output channels. In order to preserve the equal channel length property in n-process protocols with arbitrary topology, if a process send a message over an output channel then it must send one message over all channels. However, we always assume that a process is a sequential machine, and therefore, cannot execute more than one transition at a time. This implies that this property cannot be preserved in n-process protocols with arbitrary topology. Although any empirical or analytic study is not available for generalized fair reachability analysis, it is not difficult to see that in many cases simultaneous reachability analysis leads to a better reduction than generalized fair reachability analysis does because the synchronization vectors in fair reachability analysis is not considered to be maximal. Moreover, generalized fair reachability analysis is topologically restricted to only cyclic protocols.

It is believed that the higher the number of potentially executable transitions in the global state space of a protocol, the lower the reduction in the number of global states. In

this study, only the information available in a global state is used to define the potentially executable transitions. A finer definition of potentially executable transitions using the information available in process structures may yield a small number of potentially executable transitions and therefore, a better reduction. This remains to be investigated.

In simultaneous reachability analysis, many intermediate (sequentially reachable) global states are not considered and therefore fewer global states are stored. Due to the cyclic behavior of processes, during simultaneous reachability analysis one of these previously skipped intermediate global states may be generated. This leads to storing redundant global states. The suitability of backtracking in simultaneous reachability analysis needs to be studied not to store the redundant global states.

There are protocols with unbounded channels for which the number of simultaneous reachable global states is finite. Currently we do not know the characteristics of these protocols. Theoretically it will be interesting to the define the characteristics of these protocols.

Part II

Verifying Safety Properties of Finite-State Concurrent Programs

Chapter 7
AUTOMATIC VERIFICATION OF FINITE-STATE CONCURRENT PROGRAMS

The problem definition of the second part of thesis is stated as follows: Given a finite-state concurrent program and a safety property, verify whether or not the property holds for the program. The aim of this chapter is to formulate this problem and present the conventional solution based on state exploration. We will give an overview on temporal logic and model checking to that end.

7.1 Temporal Logic

A sequential program can be viewed as a function from the initial state to final state, so it can be specified and verified in terms of input and output conditions. For example, in Hoare's Logic [Hoar69], the correctness of a sequential programs can be formulated in terms of a *Precondition/Postcondition* pair, where Precondition and Postcondition are assertions on the program variables upon entry and exit of the program, respectively.

This is not the case for concurrent programs. The following illustrative example is taken from [Lamp83]. Consider the two program statements given in Figure 7.1, where the angle brackets denote atomic operations. An atomic operation is indivisible with respect to concurrently executed operations.

```
S1:  <x := x +1>

S2:  begin
        <x := x + y +1>;
        <x := x - y>
     end
```

Figure 7.1 Two program statements.

Since statements S_1 and S_2 produce the same mapping from initial state to final state, they are completely equivalent when they are used in a sequential program. However, consider the concurrent program given in Figure 7.2, where the **cobegin** indicates that its two statements separated by [] are to be executed concurrently. Note that the statements to be executed concurrently in a **cobegin** structure are called *processes*.

```
cobegin
   <y := y - 7>   []   S
coend
```

Figure 7.2 A concurrent program [Lamp83].

Substituting S_1 for S yields a program that increments x by one, while substituting S_2 for S yields a program that increments x either by one or eight. The latter possibility occurs if the statement $\langle y := y - 7 \rangle$ is executed between the execution of the two statements of S_2. Hence, S_1 and S_2 are not equivalent when they are in concurrent programs.

This example shows that when verifying concurrent programs one must consider what happens during its execution. The other important point is that concurrent programs such as

operating systems, process control programs, seat reservation systems, usually maintains a *nonterminating* computation while the computation of a sequential program is always expected to terminate. Therefore, the methods used for proving the correctness of sequential programs cannot be directly used for concurrent programs.

Pnueli [Pnue77] introduces (Linear) Temporal logic for specifying and verifying concurrent programs. Temporal Logic is a special type of a Modal Logic [HuCr77], and is designated to discuss the time varying behavior of concurrent programs without explicitly introducing time. It describes the order in which things must happen rather than the actual times at which they happen. There are two types of temporal logic regarding the underlying nature of time : *linear* and *branching* [Lamp80]. In linear temporal logic, at each moment there is only one possible future moment while in branching temporal logic, at each moment time may split into several possible future moments. Both linear and branching temporal logic have been extensively studied. It is a matter of debate as to whether branching or linear time is preferable [Emer90].

In the application of temporal logic to concurrent programs, time is discrete where the present moment is the current state of the program and the next moment is the program's immediate successor state. Thus, the temporal structure corresponding to a program execution is an infinite sequence of program states. Notice that for a terminating execution of the program, an infinite sequence of states is obtained by repeating the final state. Such infinite sequence of states is called a *computation* of the program.

In the rest of the thesis, we only consider Propositional Linear Temporal Logic (PLTL) [Pnue80]. PLTL is a classical propositional logic extended with four temporal operators: X (*next*), U (*until*), F (*sometime*) and G (*always*).

7.1.1 Syntax

PLTL formulas are built from:

- A set AP of atomic propositions: $p_1, p_2, p_3, \ldots$
- Boolean connectives: $\neg$, $\wedge$.
- Temporal operators: X, U.

The formation rules are:

- An atomic proposition $p \in AP$ is a formula.
- If f_1 and f_2 are formulas , so are $\neg f_1, f_1 \wedge f_2, \mathsf{X} f_1, f_1 \mathsf{U} f_2$.

7.1.2 Semantic

Informally, the formula $f_1 \mathsf{U} f_2$, read as "f_1 until f_2", asserts that f_2 eventually holds and f_1 will hold everywhere prior to f_2. The formula $\mathsf{X} f$, read as "next time f", holds now iff f holds at the next moment.

The other operators F and G are defined as abbreviations. The formula $\mathsf{F} f$, read as "sometime f" or "eventually f" and asserting that at some future moment f is true, equals to *true* Uf. The formula $\mathsf{G} f$, read as "always f" or "henceforth f" *and* asserting that at all moments f is true, equals to $\neg \mathsf{F}(\neg f)$.

In PLTL, a concurrent program is considered as a set of computations (i.e., infinite sequences of states). The semantic of a PLTL formula is defined with respect to the linear-time structure corresponding to a computation. A structure for an PLTL formula (with set AP of atomic propositions) is a triple $M=(S,x,\pi)$ where

- S is a finite set of states
- $x = s_0 s_1 s_2 \ldots$ is a computation where $s_i \in S$ for $i = 0,1, \ldots$.
- π: $S \rightarrow 2^{AP}$ is a labeling of each state with the set of atomic propositions in AP true at the state.

When $x=s_0 s_1 s_2 \ldots$, the notation $x(0) = s_0, x(1) = s_1, x(2) = s_2, \ldots$ and $x^i = s_i s_{i+1} s_{i+2} \ldots$ is used. $M,x \models p$ means that "x (in M) satisfies p" or "M is a model of p". When M is understood, one writes $x \models p$. For a structure M,

- $x \models p$ iff $p \in \pi(x(0))$, for $p \in AP$.
- $x \models \neg f$ iff it is not the case that $x \models f$.
- $x \models f_1 \wedge f_2$ iff $x \models f_1$ and $x \models f_2$.
- $x \models \mathsf{X} f$ iff $x^1 \models f$.
- $x \models f_1 \mathsf{U} f_2$ iff $\exists j$ ($x^j \models f_2$ and $\forall i \leq j$ ($x^i \models f_1$).
- $x \models \mathsf{F} f$ iff $\exists j$ ($x^j \models f$).
- $x \models \mathsf{G} f$ iff $\forall j$ ($x^j \models f$).

A formula f is said to be *satisfiable* if there exists a linear-time structure $M=(S,x,\pi)$ such that $M,x \models f$. f is said to be *valid* iff for all linear-time structures M, $M,x \models f$.

7.2 Automatic Verification of Finite-State Concurrent Programs: Model Checking

A state of a program is an interpretation of its variables which associates each variable with a value from its domain. If the program variables range over finite domains then the

program has a finite number of states. Model checking approach is based on an important observation: many concurrent systems can be represented as finite-state concurrent programs. The aim of a model-checking algorithm is to mechanically determine if a finite-state concurrent program satisfies (or, is the model of) a formula expressed in a propositional linear temporal logic.

A formula is defined in terms of boolean operators (e.g., *not* and *and*), temporal operators (e.g., *sometime* and *always*) and atomic propositions. Since each global state is characterized by a finite amount of information, this information can be described by the set of propositions which are true in this state. This means that a finite-state concurrent program can be viewed as a finite structure which consists of a finite set of global states, a reachability relation between global states and a function labeling each global state with the set of atomic propositions true at the global state. Such a structure is called a *labeled transition system* or (for historical reasons [HuCr77]) a *Kripke structure*. Given a Kripke structure and a formula, an efficient algorithm can be given to determine whether the structure is a model of the formula. This is so-called *model checking* [ClGr87].

The term "model checking" is coined by Clarke and Emerson [ClEm81] who propose a polynomial time model checking algorithm for the branching temporal logic called Computation Tree Logic (CTL). In [LiPn84], Lichtenstien and Pnueli give a model-checking algorithm for PLTL with a complexity exponential in the length of the formula but linear in the size of the program structure. They argue that since formulas are generally quite short while the program structures are usually quite large, the exponential complexity in the size of the formula can be discounted.

7.3 Automata-Theoretic Model Checking

Automata-theoretic model-checking approach was initiated by Wolper, Vardi and Sistla [WoVa83] showing that for any linear time propositional temporal formula, one can

construct a finite automata on infinite words, a Büchi automaton [Büch62], that accepts precisely the sequences satisfied by the formula. This result has led to the *automata-theoretic* model checking approach of [VaWo86, Vard87, Wolp89, GoWo94]. In this approach, a concurrent program is viewed a collection of finite automata on infinite words. Then, the negation of the formula is converted to a finite automaton on infinite words. Finally, the verification can be done by simply checking that the product of the automata describing the program and the automaton corresponding to the negation of the formula is empty.

A Büchi automaton is a tuple $A=(\Sigma,S,\rho,s_0,F)$ where

- Σ is an alphabet,
- S is a set of states,
- ρ: $S\times\Sigma\rightarrow 2^S$ is a nondeterministic transition function,
- $s_0\in S$ is an initial state, and
- $F\subseteq S$ is a set of accepting states.

A *run* of A over an infinite word $\omega= a_1a_2...$ is an infinite sequence $s_0s_1...$, where s_0 is the initial state and $s_i\in \rho(s_{i-1},a_i)$ for all $i\geq 0$. A run $s_0s_1...$ is *accepting* if there is some designated state that repeats infinitely often, i.e., for some $s\in F$ there are infinitely many s_i such that $s_i=s$. The infinite word w is accepted by A if there is an accepting run of A over w. The set of infinite words accepted by A is denoted $L(A)$.

Consider the graphical representation of a Büchi automaton given in Figure 7.3, where vertices and arcs respectively represent the states and transitions of the automaton. The initial state is marked with the symbol > and the final state is distinguished by a double circle. This automaton accepts the language $(a^*bb^*)^\omega$ where * denotes finite repetition (zero

or more) and ω denotes infinite repetition. Intuitively, a word over the alphabet $\{a,b\}$ is accepted by this automaton if it contains b infinitely often. Note that the words of the form $(a^*bb^*)a^\omega$ are not accepted.

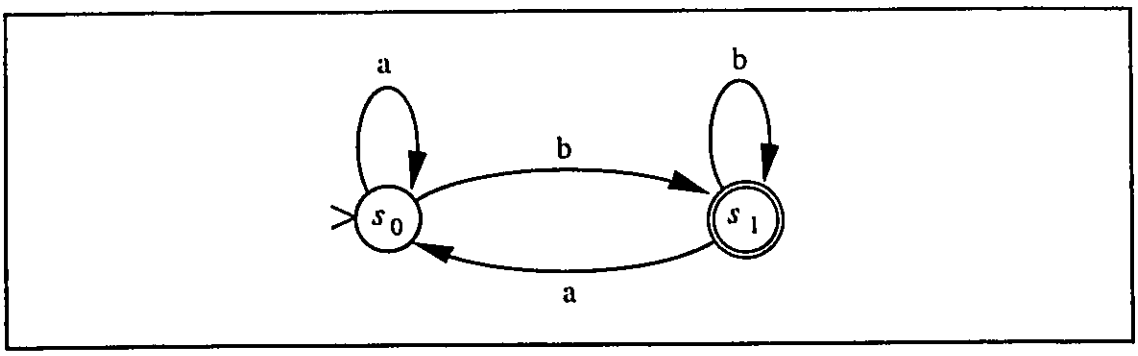


Figure 7.3 A Büchi automaton.

Theorem [WoVa83] Given an (P)LTL formula f, one can build a Büchi automaton $A_f=(\Sigma,S,\rho,s_0,F)$ where $\Sigma=2^{AP}$ and $|S|=2^{O(|f|)}$, such that $L(A)$ is exactly the set of sequences satisfying the formula f.

The Büchi automaton given in Figure 7.4 accepts every computation satisfying Fp of a program where p is a atomic proposition. Recall that in PLTL, a program is considered as a set of computations and a computation is an infinite sequence of states of the program. A PLTL structure corresponding to a computation x is (S,x,π) where S is a finite set of states and $\pi: S\rightarrow 2^{AP}$ is a labeling of each state with the set of atomic propositions in AP true at the state. Since AP is the set of atomic propositions in a given formula, $AP=\{p\}$ for the formula Fp. Note that the Büchi automaton in Theorem [WoVa83] reads an infinite sequence of sets of atomic propositions. This means that one may consider the interpretation of a computation, obtained by mapping every state in the computation to a set of atomic propositions by π, (rather than the computation itself) as the input to the Büchi automaton. Since $AP=\{p\}$, each state is mapped to either $\{p\}$ or $\varnothing$. Therefore, the Büchi

automaton given in Figure 7.4 accepts the set of infinite words over the alphabet $\Sigma=\{\varnothing,\{p\}\}$ in which the symbol $\{p\}$ appears at least once.

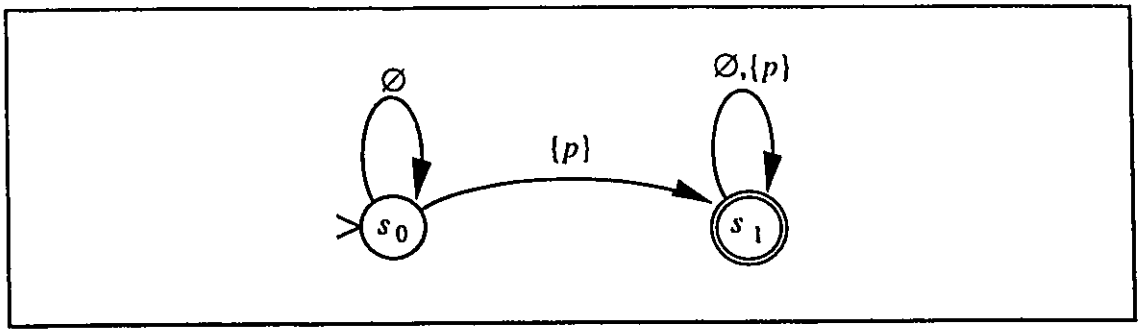


Figure 7.4 A Büchi automaton for Fp

The Büchi automaton of Figure 7.3 accepts the set of computations satisfying the formula GFp when a and b are replaced by $\varnothing$ and $\{p\}$. Note that a computation satisfies the formula GFp if it contains infinitely many program states which satisfy $\{p\}$.

Each process P_i in a concurrent program is represented by a Büchi automaton such that each state in the automaton is an accepting state. The Büchi automaton P representing the joint behavior of the processes P_i (or the behavior of the program consisting of these processes) can be computed by taking product of the automata representing processes such that actions that appear in several processes are synchronized and others are interleaved. Formally, the product of n Büchi automata $P_1 \times P_2 \ldots \times P_n$, where $P_i = (\Sigma_i, S_i, \rho_i, s_{0i}, F_i)$ for $i=1,2,\ldots,n$, is the automaton $P = (\Sigma, S, \rho, s_0, F)$ defined by

- $S=S_1\times S_2\times\ldots\times S_n$
- $\Sigma=\Sigma_1\cup\Sigma_2\cup\ldots\cup\Sigma_n$
- $F=F_1\times F_2\times\ldots\times F_n$
- Δ is defined by $((s_1,s_2,\ldots,s_n),a,(s'_1,s'_2,\ldots,s'_n))$ iff

$(s_i, a, s'_i) \in \Delta_i$ for each i such that $a \in \Sigma_i$

$s_i = s'_i$ for each i such that $a \notin \Sigma_i$

Automata-theoretic model-checking approach can be formulated as follows: Given a program P described as a set of processes P_i represented by Büchi automata and a propositional (linear) temporal logic formula f, check that all infinite computations of P satisfy f by using the following steps [VaWo86]:

1. Build the Büchi automaton for the negation of the formula f (one uses the negation of the formula since this yields a more efficient algorithm). The resulting automaton is $A_{\neg f}$.

2. Compute the global behavior of the program P : $P = P_1 \times P_2 \ldots \times P_n$, .

3. Take the product of the program P and the automaton $A_{\neg f}$: $P \times A_{\neg f}$

4. Check if this product is nonempty.

Note that one can implement the steps 2 and 3 together. This means that the construction of the program structure and verification of the negation of the formula are simultaneously performed. This is called *on-the-fly* verification. The construction of the program structure guided by the negation of the formula does not necessarily need to be completed; it might be stopped after an error has been found or only the erroneous part of the structure might be investigated. Hence, an on-the-fly technique may require a relatively small part of the finite structure in case of verifying an incorrect program [JaJe89, CoVa92, Valm93].

The above formulation of the automata-theoretic model-checking has a mismatching problem. Note the alphabet of the processes is the set of actions of the program while the alphabet of the automata representing the negation of the formula is the set of subsets of

atomic propositions of the formula. However, this problem is only superficial and can be solved during the implementation by labeling all the edges leaving a state of the program with the set of atomic propositions true at that state [Wolp89].

7.4 Properties of Concurrent Programs

For properties of concurrent programs, a classification into safety and liveness properties is first introduced by Lamport [Lamp77]. Lamport intuitively defines that a safety property asserts that "something bad" never happens and a liveness property asserts that a 'good thing' eventually happens. These properties are first formalized by Alpern and Schneider [AlSc85, AlSc87].

7.4.1 Safety Properties

Mutual exclusion, deadlock freedom and partial correctness are well-known examples of safety properties. *Mutual exclusion* states two different processes are never in their critical section at the same time, where the 'bad thing' is two processes executing in critical sections at the same time. *Deadlock freedom* states that the program never enters a state in which no further progress is possible, where the 'bad thing' is deadlock. *Partial correctness* states that if the program starts with the precondition is true, then it can never terminate with the post condition is false, where the 'bad thing' is terminating in a state not satisfying the postcondition after having been started in a state satisfying the precondition.

Let S be set of program states, S^{ω} be the set of infinite sequences of program states, i.e., the set of computations, and S^{*} be the set of finite sequences of program states, i.e., the set of *partial* computations. Let f be a safety property. If f does not hold for a computation, then at some point some 'bad thing' must happen. Such a 'bad thing' must be irremediable because a safety property states that the 'bad thing' never happens during execution. Thus, f is a *safety property* iff the following holds [AlSc85]:

$$\forall x\,(x \in S^{\omega} \wedge x \not\models f \Rightarrow \exists i\,(\,0 \leq i \;\wedge \forall y\,(y \in S^{\omega} \wedge x^{i}y \not\models f))).$$

A Büchi automaton is said to be reduced if from every state there is a path to an accepting state. For a reduced Büchi automaton A, its *closure* $cl(A)$ is the corresponding Büchi automaton in which every state has been made into an accepting state. It is said that an *undefined* transition occurs at state s of a Büchi automaton A when A is at state s and next symbol read from the input is a but the transition function of A is not defined for (s,a).

Theorem [AlSc87]. A reduced Büchi automaton A specifies a safety property iff $L(A) = L(cl(A))$.

This theorem states that if a reduced Büchi automaton specifies a safety property then every state of the Büchi automaton is an accepting state, i.e., it is *prefix-closed*. This implies that the Büchi automaton accept every infinite sequence unless an undefined transition occurs. Thus, a program does not satisfy this safety property only if the Büchi automaton attempts an undefined transition when reading a computation of the program. This means that one does not need a Büchi automaton for specifying and verifying a safety property and can use a finite automata on finite words [JaJe89, GoWo93].

7.4.2 Liveness Properties

Examples of liveness properties include starvation freedom, termination, and guaranteed service. *Starvation freedom* states that a process makes progress infinitely often, where the 'good thing' is making progress. *Termination* states that a program eventually halts, the 'good thing' is completion of the final instruction. Finally, *guaranteed service* states that every request for a service is eventually satisfied, where the 'good thing' is receiving service.

The thing to observe about a liveness property is that no partial computation is irremediable: it always remains possible for the required 'good thing' to occur in the future. Thus, f is a *liveness property* iff the following holds [AlSc85]:

$$\forall x\ (x \in S^* \Rightarrow \exists y\ (y \in S^\omega \wedge xy \models f)).$$

Theorem [AlSc87]. A reduced Büchi automaton A specifies a liveness property iff $L(cl(A)) = L(S^\omega)$.

This theorem states that if a reduced Büchi automaton specifies a liveness property then the closure of the Büchi automaton accepts every computation of the program. This means that a reduced Büchi automaton specifying a liveness property reads every computation of the program without attempting an undefined transition. Thus, a program satisfies the liveness property only if for every computation of the program, the Büchi automaton enters an accepting state infinitely often .

7.4.3 Other Properties

Although there are many temporal properties of concurrent programs which are neither safety nor liveness, every property is the intersection of a safety property and a liveness property [AlSc85]. For example, *total correctness* is the intersection of partial correctness (a safety property) and termination (a liveness property).

7.5 Automata-Theoretic Model-Checking Restricted for Safety Properties

This section reviews the formulation of the automata-theoretic model-checking approach for verification of the safety properties by using finite automata on finite words for representing both the concurrent program and the safety property.

A concurrent program P consists of n processes P_i.

Definition 7.5.1 A *process* P_i is a finite automaton on finite words : $P_i = (\Sigma_i, S_i, \Delta_i, s_{0i})$, where

- Σ_i is a finite set of actions,
- S_i is a finite set of states,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and
- $s_{0i} \in S_i$ is the initial state. □

In the following definition, the joint global behavior of the processes is defined by usual operational semantics (CSP-style): actions that appear in several processes are synchronized, others are interleaved.

Definition 7.5.2 Let $P_i = (\Sigma_i, S_i, \Delta_i, s_{0i})$ be a process for $i = 1,2,\ldots,n$. The finite automaton on finite words which represents the joint global behavior of processes P_i obtained by taking the *sequential product* (denoted by $\otimes$) of automata P_i is $P_{seq} = (\Sigma, G, \Delta_{seq}, g_0) = P_1 \otimes P_2 \otimes \ldots \otimes P_n$ where

- $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n$
- $G = S_1 \times S_2 \times \ldots \times S_n$
- Δ_{seq} is defined by $((s_1, s_2, \ldots, s_n), a, (s'_1, s'_2, \ldots, s'_n))$ iff

  $(s_i, a, s'_i) \in \Delta_i$ for each i such that $a \in \Sigma_i$

  $s_i = s'_i$ for each i such that $a \notin \Sigma_i$

- $g_0 = (s_{01}, s_{02}, \ldots, s_{0n})$ □

Note that to distinguish between a state of P_i and a state of P, we denote states of processes by s ($\in S_i$) and states of the program by g($\in G$). Henceforth, a state of a program called *global state* while a state of a process is called *local state*.

Definition 7.5.3 The relation "$\rightarrow$" is defined between global states: $g \rightarrow g'$ iff $\exists a \in \Sigma$ $((g,a,g') \in \Delta_{seq})$. If $g \rightarrow g'$ then g' is said to be *sequential immediate successor* of g. ||

Definition 7.5.4 The relation "$\rightarrow^*$" is the reflexive transitive closure of "$\rightarrow$". g' is *sequentially reachable from* g if $g \rightarrow^* g'$ and g' is *sequentially reachable* if $g_0 \rightarrow^* g'$. ||

An automaton can be represented by a directed graph where the nodes correspond to the global states and the edges correspond to the transitions.

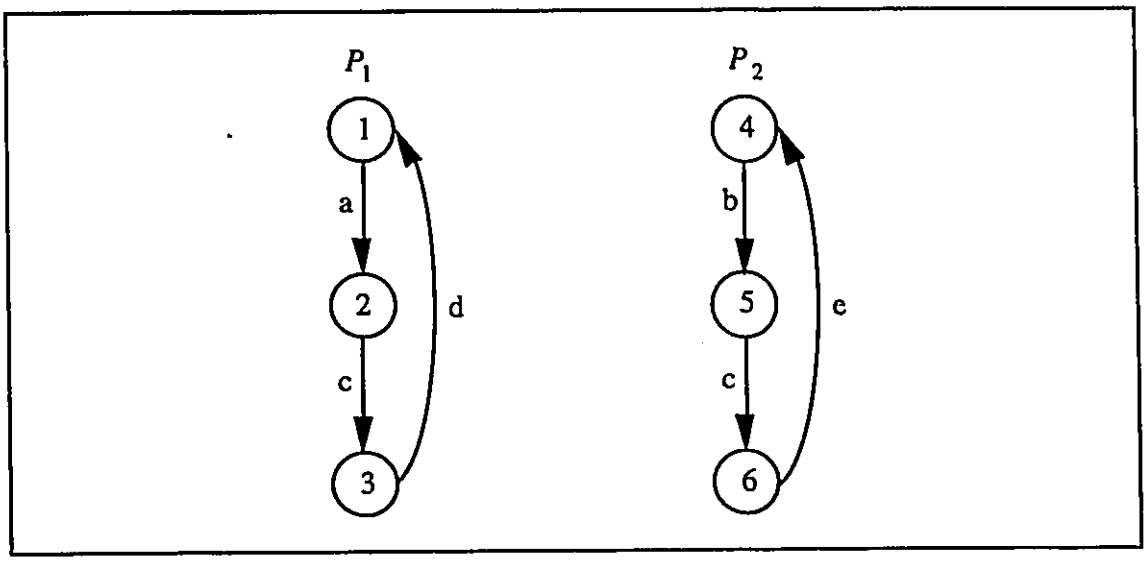


Figure 7.5 A concurrent program.

Consider the concurrent program composed of two processes given in Figure 7.5. Action c is synchronized and actions a, b, d and e are interleaved in the joint global behavior. The automaton P_{seq} representing the joint global behavior of the two processes is given in Figure 7.6. At the initial state (1,4), a and b are enabled actions. These actions are

interleaved. After both interleaving sequences ab and ba, the system reaches global state (2,5). Then processes synchronize on c and the program reaches (3,6). Final d and e are interleaved and the system returns to the initial state.

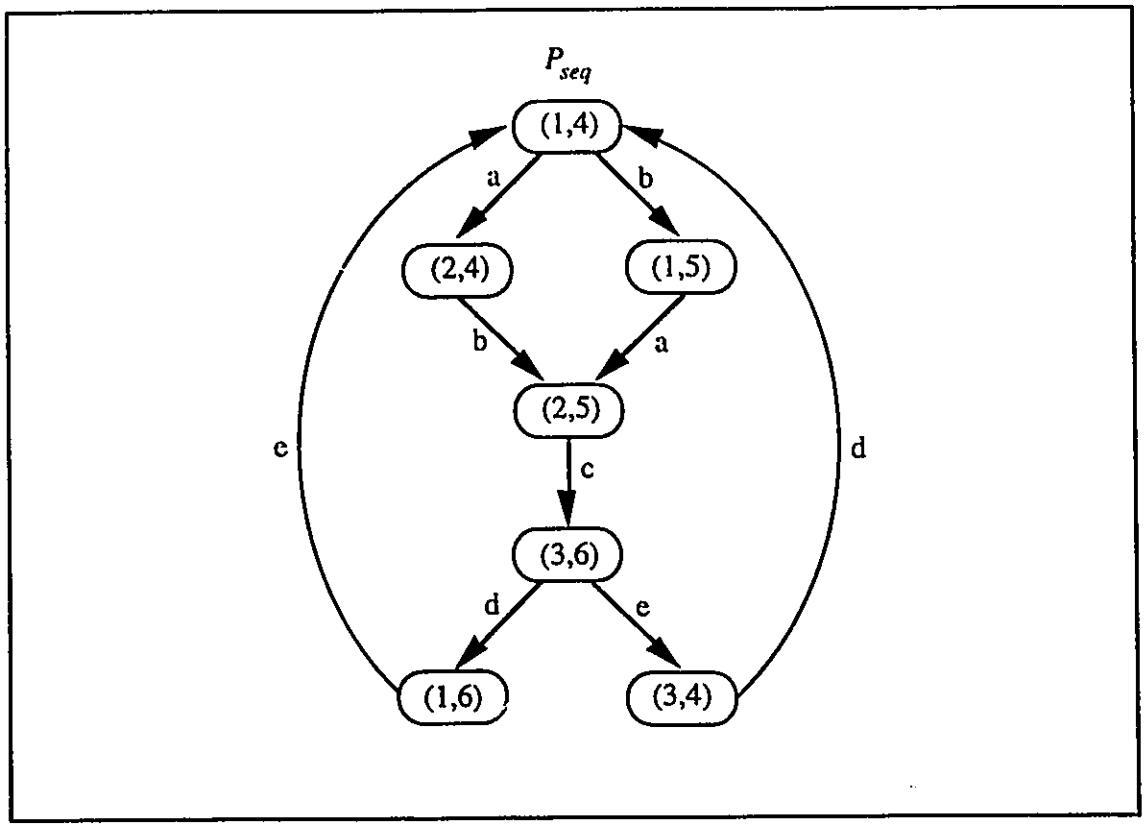


Figure 7.6 The sequential automaton for the program in Figure 7.5.

Henceforth, we assume that the sets of states of automata P_i are pairwise disjoint so that we can also view the global state $g=(s_1,s_2,...,s_n)$ as a set $g=\{s_1,s_2,...,s_n\}$ with fixed cardinality n. The local state s is said to be sequentially reachable iff there exists a sequentially reachable state g such that $s \in g$.

Using the fact that safety properties can be represented by prefix-closed finite automata on finite words, automata-theoretic model-checking approach can be reformulated for safety properties as follows [GoWo93]: Given a program P described as a set of processes

P_i represented by finite automata on finite words and a safety property f represented by a prefix-closed finite automaton on finite words A_f, check whether P satisfies f by using the following steps:

1. Build the automaton $A_{\neg f}$ corresponding to the complement of A_f. Since A_f is prefix-closed, $A_{\neg f}$ is naturally an automaton with only one accepting state (denoted X).

2. Check if the local sate X is sequentially reachable in the concurrent system obtained by taking the sequential product of processes in the program and $A_{\neg f}$: $P_1 \otimes P_2 \ldots \otimes P_n \otimes A_{\neg f}$.

Definition 7.5.5 For each global transition $\delta=(g,a,g')\in \Delta_{seq}$ with $g=(s_1,s_2,\ldots,s_n)$ and $g'=(s'_1,s'_2,\ldots,s'_n)$ the sets

- ${}^\bullet\delta=\{s_i \in g \mid (s_i,a,s'_i)\in \Delta_i\}$
- $\delta^\bullet=\{s'_i \in g' \mid (s_i,a,s'_i)\in \Delta_i\}$

are called the *preset* and the *postset* of the global transition δ, respectively. □

Intuitively, the preset (postset) of a global transition δ of P_{seq} is the set of local states that synchronize on a before (after) this transition. We say that a process P_i is *active* on a global transition δ if the preset for δ includes one state from S_i. Note that in this case postset for δ also includes one state from S_i.

Definition 7.5.6 Two global transitions $\delta_1=(g_1,a_1,g'_1)$, $\delta_2=(g_2,a_2,g'_2)$ are said to be equivalent (denoted by "≡") iff ${}^\bullet\delta_1={}^\bullet\delta_2 \wedge \delta_1{}^\bullet=\delta_2{}^\bullet \wedge a_1=a_2$. □

Equivalent global transitions are the distinct occurrences of the same "program" transition. The occurrences can only differ by the states of the P_i's that are not active on the transition. The set of equivalence classes defined over Δ_{seq} by ≡ is denoted by $\hat{\Delta}_{seq}$, i.e., $\hat{\Delta}_{seq}=\{({}^\bullet\delta,a,\delta^\bullet) \mid \delta=(g,a,g')\in\Delta_{seq}\}$. Henceforth, we will refer the elements of $\hat{\Delta}_{seq}$

as *program transitions* and preset (postset) of a program transition $t = ({}^\bullet\delta, a, \delta^\bullet) \in \hat{\Delta}_{seq}$ will be ${}^\bullet t = {}^\bullet\delta\ (t^\bullet = \delta^\bullet)$ where $\delta=(g,a,g') \in \Delta_{seq}$. Note that we can obtain the transitions of the program from the process automata without generating the global states of the program. We list all transitions of the program for Figure 1 as follows: ({1},a,{2}), ({2,5},c,{3,6}), ({3},d,{1}), ({4},b,{5}), ({6},c,{4}).

In the sequel, program transitions will be called transitions.

Definition 7.5.7 A transition $t \in \hat{\Delta}_{seq}$ is *related* to global state g iff ${}^\bullet t \cap g \neq \varnothing$.

related(g) will be used to denote the set of all transitions related to g. □

Definition 7.5.8 A transition $t \in$ *related*(g) is *enabled* at state g iff ${}^\bullet t \subseteq g$.

enabled(g) ($\subseteq$ *related*(g)) will be used to denote the set of all enabled transitions at g. A transition $t \in$ *related*(g) \ *enabled*(g) is said to be *not-enabled* at g. □

Let $t' = (g_k, a, g_k) \in \Delta_{seq}$ and $t = ({}^\bullet t', a, t'^\bullet) \in \hat{\Delta}_{seq}$ then we write $g_k \xrightarrow{t} g_k$. $g_k \xrightarrow{t_1} g_{k(1)} \wedge g_{k(1)} \xrightarrow{t_2} g_{k(2)} \wedge \ldots \wedge g_{k(j-1)} \xrightarrow{t_j} g_m$ will sometimes be denoted by $g_k \xrightarrow{t_1 t_2 \ldots t_j} {}^* g_m$.

Let $\sigma = t_1 t_2 \ldots t_j$, $\omega = t_1 t_2 \ldots t_i$ and $\mu = t_{i+1} t_{i+2} \ldots t_j$ be transition sequences. Then $\sigma = \omega\mu$ (where juxtapositions are used for concatenations) and the length of σ (denoted by $|\sigma|$) is j. If $j = 0$ then $\sigma = \varepsilon$, $\omega = \varepsilon$ and $\mu = \varepsilon$. If $i = 0$ then $\omega = \varepsilon$. If $i = j$ then $\mu = \varepsilon$. If $\sigma = \omega\mu$ then ω is said to be a *prefix* of σ. The set of all prefixes of σ is denoted by *prefix*(σ). We denote the set of active processes on a transition t, a sequence σ of transitions and a set T of transitions by *act*(t), *act*(σ) and *act*(T), respectively, as follows:

- $act(t) = \{P_i \mid {}^\bullet t \cap S_i \neq \varnothing\}$.
- $\sigma = t_1 t_2 \ldots t_j \Rightarrow act(\sigma) = \bigcup_{i=1}^{j} act(t_i)$.

- $T = \{t_1, t_2, \ldots, t_j\} \Rightarrow act(T) = \bigcup_{i=1}^{j} act(t_i)$.

The following definition is adapted from [KaPe92]:

Definition 7.5.9 Two transitions t_1 and t_2 of a program P are independent if the following is true at all sequentially reachable global sates g of P (otherwise they are said to be dependent):

1. if t_1 (t_2) is enabled at g and $g \text{—}t_1\rightarrow g'$ ($g \text{—}t_2\rightarrow g'$), then t_2 (t_1) is enabled at g iff t_2 (t_1) is enabled at g' (independent transitions can neither disable nor enable each other); and

2. if t_1 and t_2 are enabled at g, then there is a unique g' such that both $g \text{—}t_1t_2\rightarrow g'$ and $g \text{—}t_2t_1\rightarrow g'$ (commutativity of enabled independent transitions). []

Note that for the model given in this section, two transitions t_1 and t_2 are independent if the set active processes of t_1 is disjoint from that of t_2. In fact, such an easily checkable syntactic condition can be given in most programming models [WoGo94].

Definition 7.5.10 Two transitions t and t' are *independent* (*dependent*) iff $act(t) \cap act(t') = \emptyset$ ($act(t) \cap act(t') \neq \emptyset$). []

Notice that there exists a binary relation between two independent (dependent) transitions, which is called an *independence* (*dependence*) relation and determined from the syntactic structure of a concurrent program. If two transitions t and t' are independent (dependent) then we may say that t is independent (dependent) with t'. The definitions and notations introduced in this section will be used through the rest of the thesis.

Chapter 8
REDUCTION METHODS FOR STATE EXPLOSION

A conventional automatic verifier for a concurrent program can be illustrated as in Figure 8.1. It consists of a generator and a model checker. The generator receives the syntactic structure of a concurrent program which is actually a set of processes, and produces the reachability graph of the program. The reachability graph is also named

state space, labeled state transition graph, kripke structure or program structure. The model checker takes two inputs: the reachability graph and the temporal logic specification of a property. Its outcome stating whether the given property is valid or not is the output of the verifier. The time complexity of the model checker is linear with the size of the reachability graph. However, the size of reachability graph can be exponential with size of the syntactic structure of the program. Note that if s represent the maximum number of states in a process, then the reachability graph may have up to s^n nodes (where n is the number of processes). Therefore, model checking inherits the state explosion problem from reachability analysis. In this chapter, we give an overview on the reduction methods for the state explosion in model checking.

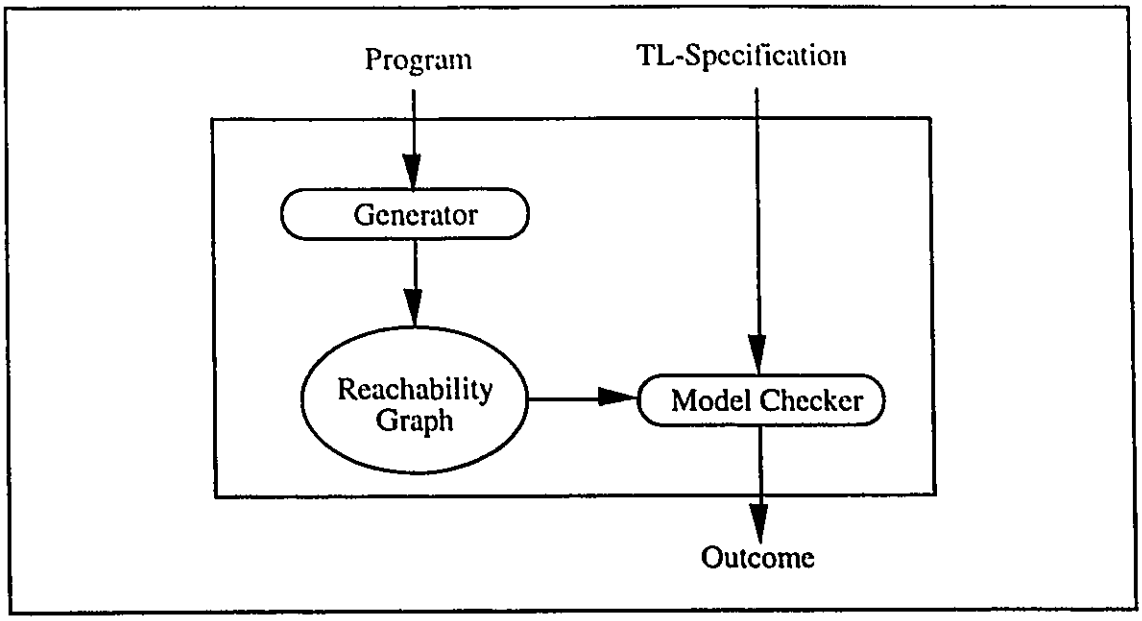


Figure 8.1 Conventional verifier.

8.1 On-the-fly Methods

In conventional verification, first the reachability graph (or the state space) of a concurrent program is generated, then the property is checked over the state space. In on-the-fly verification, the generation of the state space and the checking of the property are performed simultaneously. On-the-fly verification has some benefits over conventional verification:

1. The generation of the state space can be stopped after finding the first violation against the property.

2. The property to be verified may not require the investigation of some branches in the state space.

3. The state space can be divided into many subspaces such that each subspace can be verified independently. Therefore, an on-the-fly verification method may not require to record the whole state space.

Benefits 1 and 2 come naturally with intersecting the generation and checking steps but Benefit 3 requires a particular search of the state space which is a depth-first search.

In [Holz85b, Holz87b], Holzmann shows that a depth-first search algorithm can minimize the memory requirements at the expense of running time by storing only global states in a single execution path from the initial global state to the current global state. It is known that the depth of a reachability graph is much smaller than the size of the graph [Holz88]. When there exists enough memory to store the global states of the longest execution path from the initial global state, the algorithm guarantees a complete exploration of finite state space even though the state space is too large to be stored in the available memory. Since such an algorithm cannot determine whether or not a newly generated global state was encountered before in a previously analyzed path, the algorithm may redundantly analyze the same global state several times. This dramatically increases the run-time requirements of this type of strategy. However, the strategy of storing all states of the current path and storing as many other states as possible in the remaining available amount of memory reduces the run-time requirements. This strategy is called *state space caching* [Holz85b].

Intuitively, an implementation of checking a safety property involves searching "bad" states (i.e., the states violating the safety property) in the state space while an implementation of checking a liveness property involves searching "bad" cycles (i.e., the computations violates the liveness property).

In [JaJe89], Jard and Jéron rediscover the state space caching strategy to verify the safety properties of concurrent programs. In [JaJe91], they use this strategy to verify any

property specified by a deterministic Büchi automaton. Note that a Büchi automaton is non-deterministic if more than one transition is possible from some state for some input symbol, otherwise deterministic.

Let P be labeled transition system representing a finite state concurrent program and A be a deterministic Büchi automaton representing a property of the program. Consider P as a Büchi automaton such that every state is an accepting state. Assume that A is complete. Note that an automaton is complete if from every state there exists a transition for every input symbol. If A is not complete, we define all missing transitions by adding a new state so that A becomes complete. Let B be the Büchi automaton corresponding to the product of P and A, and F be the accepting states of B. Since A is complete, for every computation of P, there exists one run in B. In fact, there exists one and only one run for each computation because A is deterministic. When B is viewed as a (finite) directed graph, then every run corresponds to a cycle of the graph. If a run is accepting then the corresponding cycle contains a state of F. Finally, the property represented by A is a valid property of P if every cycle of the directed graph of B includes a state of F. This is equivalent to say that the subgraph B' obtained from B by removing all states of F (and the corresponding edges) is acyclic. And B' is acyclic if and only if a depth-first traversal of B' does not detect any cycle.

In [JaJe91], the following algorithm is given to check whether B' is acyclic during a depth-first traversal of subgraphs of B. Let *Start* be the set of roots of the depth-first traversals of sub-graphs. Initially, *Start* includes the initial state of B. A state is removed from *Start* and then a depth-first traversal is performed from this state. In the traversal, if a new state from F is reached, this state is added to *Start* and the successors of this state are not explored. If a cycle detected by visiting a state not from F, which has been previously visited in the current traversal, then B' is not acyclic, otherwise another depth-first traversal

is performed by removing a state from *Start*. This procedure continues as long as *Start* is not empty.

There are properties of concurrent programs such that those can not be specified by deterministic Büchi automata. In that case, the negation of the given property is converted to a (nondeterministic) Büchi automaton $A_{\neg}$. Let B is the Büchi automaton corresponding to the product of P and $A_{\neg}$, and F be the accepting states of B. Then the verification can be done by simply checking that B is empty. In [CoVa92], an algorithm is given to check nonemptiness of Büchi automata. More specifically, the algorithm is provided for the following problem: "Given a directed graph G, start node s_0, distinguished set of accepting nodes F, determine whether there is a member of F that is reachable from s_0 and belongs to a cycle, or equivalently, to a nontrivial strongly connected component". The algorithm consists of two depth-first searches. In the first search, it marks the member of F that are reachable from s_0, and order them according to last visit. In the second search, it checks that a marked member of F is in a cycle. This algorithm utilizes the technique called supertrace [Holz88] to check that a state has been previously visited. In supertrace the memory is arranged as a bit array. Each generated state is hashed into an index of this array. If the corresponding bit in the array is on, then the algorithm considers that the generated global state has already been analyzed. Since collision detection is ignored, the algorithm should be viewed more as a systematic debugging tool rather than as a verification tool [CoVa92].

8.2 Partial Order Methods

In all possible executable transition sequences of a concurrent program, if transition t always occurs before transition t', then there exists a casual dependency between t and t'. More specifically, t is the cause and t' is the effect. In this case there is a linear order between t and t', i.e., tt'. Due to the concurrency, some transitions are independent of each

other, i.e., all transitions may not be put in one linearly ordered sequence. Thus, there exists a partial order between transitions.

Each transition sequence can be viewed as a linearly ordered multiset of transitions since repetitions of transitions are possible in the sequence. Consider that we are given a transition sequence of a concurrent program and the dependency relation between transitions in this sequence. The given transition sequence is an interleaving of the corresponding multiset of transitions, i.e., a linearization of the corresponding multiset of transitions which preserves the partial order between transitions. From the given sequence and the dependency relation, we can generate all possible interleavings of the multiset of transitions by successively exchanging the ordering of adjacent independent transitions. It is shown that all these interleavings leads to the same state of the program. Mazurkiewicz [Mazu86] considered all these interleavings are equivalent and named an equivalent class of sequences as a *trace*.

The fact that all interleavings of a trace leads to the same global state is the main inspiration of the partial order methods. Since all interleavings of a trace leads to the same global state, if one interleaving of a trace leads to a deadlock state then every interleaving of the trace leads to the same deadlock state. Therefore, it is concluded that one interleaving is enough for deadlock detection. These methods usually attack first deadlock detection problem and then the solution for deadlock detection is adapted for verifying other properties of concurrent programs. The main objective of the partial order methods is to check a desired property as efficiently as possible. While checking a property, they may generate several redundant interleavings of the same trace or they may not generate any interleaving for some traces. Hence, their aim is not to represent a concurrent program by Mazurkiewicz's traces.

The partial order methods search a subset of the reachable state space by executing a subset of enabled transitions at a global state. In other words, they select some of the

enabled transitions at a global state to be used in generating next global states. In [WoGo94], the sets of selected enabled transitions are classified as persistent sets and sleep sets, and the methods utilizing one or both of these sets are called *selective searches*.

8.2.1 Persistent Sets

Intuitively, a persistent set T of a global state g is a subset of the set of enabled transitions of g such that each transition in T remains enabled at every global state reached from g by the execution of transitions which does not belong to T. In other words, whatever one does from g, while remaining outside of T, does not effect the executability of transitions in T. A selective search method computes a persistent set T in each global state and generate the next global states by only executing the transitions in T. Since T is usually a proper set of the set of enabled transitions of g, not all of the successors of g are generated. Therefore, selective search methods generate a subset of the (reachable) state space of a concurrent program. The definition of persistent sets is given in our notation as follows (adapted from [WoGo94]):

Definition 8.2.1.1 Let $T \subseteq enabled(g)$. T is *persistent* at g iff $\forall(\sigma, g') \; (g \xrightarrow{} t_1 t_2 \ldots t_j \rightarrow^* g' \wedge \forall(i = 1,2,\ldots j) \; (t_i \notin T) \Rightarrow \forall(t \in T) \; (t \text{ and } t_j \text{ are independent})$. □

In order to present a definition consistent with the intuition behind persistent sets, we redefine them and prove that our definition is equivalent to Definition 8.2.1.1.

Definition 8.2.1.2 Let $T \subseteq enabled(g)$. T is *persistent* at g iff $\forall(\sigma, g') \; (g \xrightarrow{} t_1 t_2 \ldots t_j \rightarrow^* g' \wedge \forall(i = 1,2,\ldots j) \; (t_i \notin T) \Rightarrow T \subseteq enabled(g')$. □

Lemma 8.2.1.1 Definitions 8.2.1.1 and 8.2.2.2 are equivalent.

Proof. Let T be *persistent* at g and g' be sequentially reachable from g by the execution of transitions not in T. Then let $g \xrightarrow{} t_1 t_2 \ldots t_j \rightarrow^* g'$ and $\forall(i = 1,2,\ldots j) \; (t_j \notin T)$. From Definition 8.2.1.1, it is immediate that t_i and t are independent for $i = 1,2,\ldots j$ and each $t \in$

T. Using the commutativity of enabled independent transitions, we show that for each $t \in T$, $t \in enabled(g')$ by induction on j.

Basis: Let $j = 0$. Trivially, $t \in enabled(g')$.

Induction: For the induction step suppose the lemma holds for j-1, i.e., there exists g'' such that $t \in enabled(g'')$ and $g \text{—}t_1t_2\ldots t_{j-1} \rightarrow^* g''$. We will prove that it also holds for j. Since t and t_j are enabled independent transitions at g'', tt_j and t_jt are executable from g''. This implies that $t \in enabled(g')$. ||

Note that T is trivially persistent if $T = enabled(g)$, because in this case no global state is reachable from g by transitions not in T. Let g_d be a sequentially reachable deadlock state. Then g_d is also reached in a persistent-set selective search [WoGo94]. Why? Let $g_0 \text{—}t_1t_2\ldots t_j \rightarrow^* g_d$ and T_0 be a persistent set of g_0. Since $enabled(g_d)$ is empty, Definition 8.2.1.2 implies that there exist i such that $1 \leq i \leq j$ and t_i is a member of T_0 is a persistent-set of g_0. Assume t_i is the first such a transition on $t_1t_2\ldots t_j$. From Definition 8.2.1.1, for all $k = 1,2,\ldots,i\text{-}1$, t_k is independent with t_i. Then t_i can be moved to the first position by successively changing the ordering of adjacent independent transitions. This implies that $t_it_1t_2\ldots t_{i-1}t_{i+1}t_{i+2}\ldots t_j$ and $t_1t_2\ldots t_j$ are equivalent, i.e., $g_0 \text{—}t_i\, t_1t_2\ldots t_{i-1}t_{i+1}t_{i+2}\ldots t_j \rightarrow^* g_d$. Let $g_0 \text{—}t_i \rightarrow g_1$ then $g_1 \text{—}t_1t_2\ldots t_{i-1}t_{i+1}t_{i+2}\ldots t_j \rightarrow^* g_d$ and g_1 is a reachable global state in a persistent-set selective search. Obviously, after a finite number of applications of the above reordering, g_d will become reachable in a persistent-set selective search.

Note that if for each g reached in persistent-set selective search, $enabled(g)$ is selected as the persistent set at g then all sequentially reachable global states are enumerated in the search. Therefore, the reduction in terms of the number of global states gained by such a search directly depends on the algorithm computing persistent sets. The aim of these algorithms is to obtain the smallest possible persistent sets. The computation for smaller persistent set requires more information about the program and a higher time complexity [HoGo92]. In fact, always choosing the persistent sets with the smallest number of

enabled transitions does not necessarily lead to the smallest number of global states [Valm91].

Persistent sets are called *stubborn sets* by Valmari who intensively studies different computing algorithms for persistent sets (for example see [Valm91, Valm92, Valm93]. Stubborn sets can be defined in our notation as follows (adapted from [Valm93]):

Definition 8.2.1.3 T is a stubborn set of g iff $T \cap enabled(g) \neq \emptyset \wedge \forall(t \in T)$ ((1) $\wedge$ (2)) where

1. $t \notin enabled(g) \Rightarrow \exists(P_i \in act(t))\,(s_i \notin {}^\bullet t \wedge \{t' \mid t' \in related(g) \wedge s_i \notin {}^\bullet t'\} \subseteq T)$.
2. $t \in enabled(g) \Rightarrow \forall(P_i \in act(t))\,(\{t' \mid t' \in related(g) \wedge s_i \in {}^\bullet t'\} \subseteq T)$. □

That is, a stubborn set T of a global state g is a set of transitions related to g such that T contains at least one enabled transition, (1) to enable a not enabled transition in T it is necessary to execute at least one transition in T, and (2) enabled transitions in T are independent of the related transitions outside T. It can be shown that one obtain a persistent set by restricting a stubborn set to enabled transitions.

Godefroid and Wolper [GoWo93] present a partial order method which utilizes both persistent and sleep sets (see the next section for sleep sets). Let two transition t and t' be referred to as being *in conflict* iff ${}^\bullet t \cap {}^\bullet t' \neq \emptyset \wedge \exists (t_1,t_2,\ldots,t_j)$ (${}^\bullet t \cap {}^\bullet t_1 \neq \emptyset \wedge {}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset \wedge \ldots \wedge {}^\bullet t_{j-1} \cap {}^\bullet t_j \neq \emptyset \wedge {}^\bullet t_j \cap {}^\bullet t' \neq \emptyset$). The following definition of persistent sets (adapted from [GoWo93]) is given in our notation:

Definition 8.2.1.4 $T \subseteq enabled(g)$ is a persistent set of g iff (1) $\vee$ ($\neg$(1) $\wedge$ $T = enabled(g)$) where (1): $\forall(t,t')$ ($t \in T \wedge t$ and t' are in conflict $\Rightarrow t' \in T$) □

That is, a persistent set is a proper subset of enabled transitions at a global state if there is no transition outside of the subset is in conflict with a transition inside the subset, otherwise the persistent set includes every enabled transition.

Although the selective search algorithms are sufficient to check deadlock freedom, they usually have to be modified to check properties other than deadlock freedom. It is well-known that the verification of any safety property can be reduced to checking the sequential reachability of a local state (see section 7.5). However, the selective search algorithms are not sufficient to check the sequential reachability of a local state. Consider the concurrent program given in Figure 8.2 where the initial global state is (1,2). (1,a,1) and (2,b,3) are enabled transitions at the initial global state. A selective search algorithm can compute {(1,a,1)} as a persistent set. After exploring transition (1,a,1) the algorithm stops since this transition leads back to the initial global state. Note that transition (2,b,3) has been never executed and therefore local state 3 is not reached even though (2,b,3) is enabled at initial global state and therefore local state 3 is sequentially reachable. This problem is called the *ignoring problem* [Valm91]. The solutions proposed to solve this problem require selection of some enable transitions in addition to the transitions in persistent sets (e.g. [Valm91, HoGo92]).

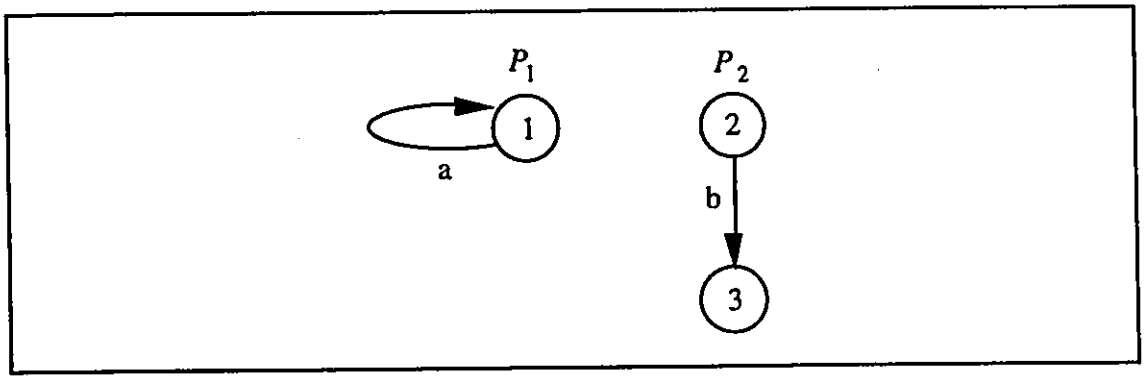


Figure 8.2 A concurrent program.

The verification of general properties can be reduced to checking reachable cycles containing specific local states [VaWo86, Wolp89, JaJe91, CoVa92]. This requires further more enabled transition to be selected by selective search algorithms [Valm93, Pele94].

8.2.2 Sleep Sets

Consider the concurrent program given in Figure 8.3. Let t_1 = (1,a,2) and t_2 =(3,b,4). Since t_1 and t_2 are enabled independent transitions at the initial global state (1,3), both transitions t_1t_2 and t_2t_1 are executable and lead to the same global state (Figure 8.4). This is potentially wasteful since both of these interleavings lead to the same global state. In order to prevent this, the sleep set method prevents the exploration of t_1 at (1,4).

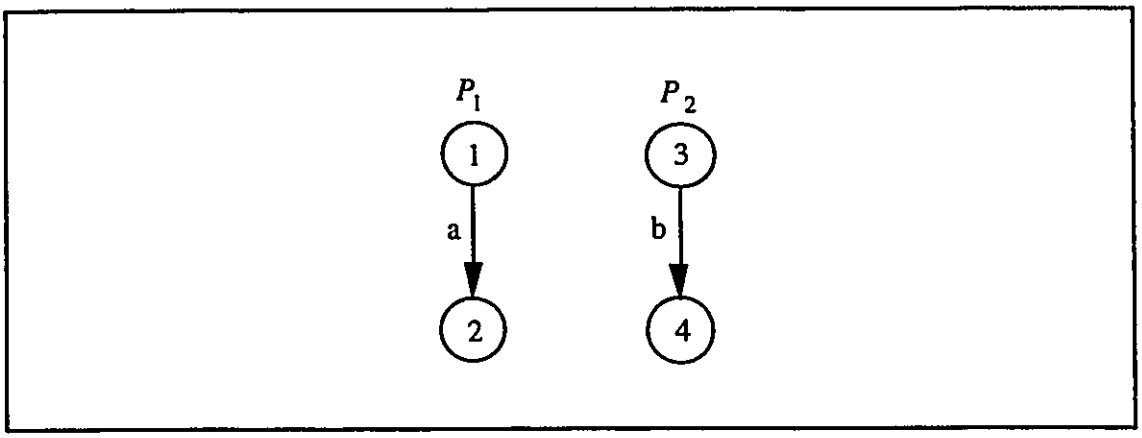


Figure 8.3 A concurrent program.

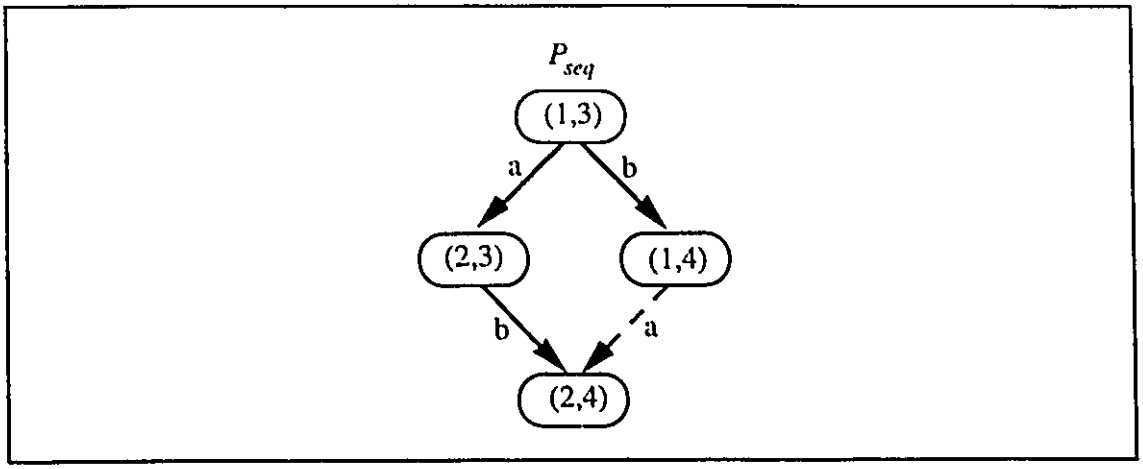


Figure 8.4 The sequential automata for the program in Figure 8.3.

A *sleep set* for a global state g is a set of enabled transitions at g that will not be executed from g. The sleep set for the initial global state is empty. Let T be the set of transitions that have been selected to be explored from g. Take a first transition t_1 out of T. Let g_1 be reached by exploring t_1 from g. Then the sleep set of g_1 is obtained from the sleep set of g by removing transitions dependent with t_1. Let t_2 be the second transition taken out of T and g_2 be reached by exploring t_2 from g. The sleep set of g_2 is obtained from the sleep set of g by removing transitions dependent with t_2 and adding t_1. Finally, let t_i be the ith transition taken out of T and g_i be reached by exploring t_i from g. The sleep set of g_i is obtained from the sleep set of g by removing transitions dependent with t_i and adding transitions $t_1, t_2, \ldots, t_{i-1}$.

When sleep sets are used alone, they lead to reduction in the number of transitions but not the number of global states [GoHo92]. However, when combined with persistent set techniques, they further reduce the number of global states [GoWo93,WoGo94].

8.3 Optimal Simulation

Janicki et al. [JaLa86] formally define and analyze the semantics expressed intuitively as "execute as much as possible simultaneously". They show that in general, this semantic is not sufficiently expressive to verify the properties of concurrent programs. The class of concurrent programs for which this semantic is sufficient is also characterized in [JaLa86].

Janicki and Kounty in [JaKo89] define a technique called *optimal simulation* to obtain a subset of possible computations of a concurrent program. They show that optimal simulation is minimal for verifying a particular set of formulas. It is known that this set is sufficient to express deadlock-freedom and some other safety properties but the relative power of the set remains open. In addition, an efficient implementation of the reachability graph for optimal simulation remains open in [JaKo89].

In [JaKo90b], they claim that "Unfortunately, as opposed to the full reachability graph, in the general case it is not clear how to generate the reachability graph for optimal simulation in an efficient way" and outline how such a graph can be constructed for Petri nets which can be decomposed into finite state machines. The formal definition of this construction is given in [JaKo90a]. However, the construction algorithm has two major drawbacks. The algorithm does not directly construct the reachability graph for optimal simulation. It first generates an auxiliary reachability graph then prunes the redundant arcs and nodes to obtain the graph for optimal simulation. However, it is not known how much the auxiliary graph is bigger than the graph for optimal simulation. The other drawback is that the graph for optimal simulation can be bigger than the full reachability graph (see [JaKo90a]). Therefore, an efficient implementation of optimal simulation is still an open problem.

Chapter 9
SIMULTANEOUS PRODUCT FOR VERIFYING SAFETY PROPERTIES

In this chapter we will formulate a new product called *simultaneous product* to generate a subset of the joint global behavior of the processes of a concurrent program. This subset will be proven to be sufficient to verify safety properties of the concurrent program. When the simultaneous product of processes of a concurrent program is taken, the global states are generated by simultaneously executing several (global) transitions, i.e., by allowing several synchronizations to occur in the concurrent program at a time. When several transitions are simultaneously executed, the interleaving sequences of these transitions are not generated, that is, the states that are supposed to be generated during the execution of these sequences are not generated. Therefore, replacing single (or sequential) executions by simultaneous executions reduces the number of states to be generated.

9.1 Equivalent Transition Sequences

In this section we define an equivalence relation between transition sequence and present some properties of equivalent executable transition sequences. These properties will be used in the proof of lemmas and theorems in the subsequent sections.

Definition 9.1.1 A sequence of transitions (or a transition sequence) $\sigma (= t_1 t_2 \ldots t_j)$ is *executable* iff $\exists g_k \, (g_k \text{—}\sigma \rightarrow^* g_m)$. □

We use $\sigma\downarrow_i$ to denote the *projection* of σ onto P_i, i.e., the sequence of transitions obtained by removing all transitions whose set of active processes do not include P_i from σ. Notice that, if P_i is active on σ (i.e., $P_i \in act(\sigma)$) then the projection of σ onto P_i is not an empty sequence.

Definition 9.1.2 Two transition sequences σ and ω are *equivalent*, denoted by $\sigma \equiv \omega$, iff $\forall P_i \in P\ (\sigma\downarrow_i = \omega\downarrow_i)$. □

The following properties are immediate from Definition 9.1.2.

Property 9.1.1 $\forall(\sigma,\omega)\ (\sigma \equiv \omega \Rightarrow |\sigma| = |\omega|)$.

Property 9.1.2 $\forall(\sigma,\omega)\ (\sigma \equiv \omega \wedge \sigma = t_1 t_2 ... t_j \wedge \omega = t'_1 t'_2 ... t'_j \Leftrightarrow \exists\pi\ (\pi:\{1,2,...j\} \rightarrow \{1,2,...j\} \wedge \pi$ is a one to one and onto function $\wedge\ \forall i\ (1 \le i \le j \Rightarrow t_i = t'_{\pi(i)})))$.

The following lemma states that two equivalent executable transition sequences from a global state lead to a common global state.

Lemma 9.1.1

$\forall(g_k,\sigma,\omega)\ (g_0 \rightarrow^* g_k \wedge g_k \text{—}\sigma\rightarrow^* g_l \wedge g_k \text{—}\omega\rightarrow^* g_m \wedge \sigma \equiv \omega \Rightarrow g_l = g_m)$.

Proof: From the definition of equivalent transition sequences, for each process P_i, projections of σ and ω onto this process are the same. This directly implies that $g_l = g_m$. □

Using Lemma 9.1.1, we write $\sigma\ {}_k\!\equiv_r \omega$ when $g_k \text{—}\sigma\rightarrow^* g_r$, $\sigma \equiv \omega$ and $g_k \text{—}\omega\rightarrow^* g_r$. The following properties are immediate from the result of Lemma 9.1.1.

Property 9.1.3 $\forall(g_k,\sigma)\ (g_0 \rightarrow^* g_k \wedge g_k \text{—}\sigma\rightarrow^* g_r \Rightarrow \sigma\ {}_k\!\equiv_r \sigma)$.

Property 9.1.4 $\forall(g_k,\sigma,\omega,\mu)\ (g_0 \rightarrow^* g_k \wedge \sigma\ {}_k\!\equiv_q \omega \wedge g_q \text{—}\mu\rightarrow^* g_r \Rightarrow \sigma\mu\ {}_k\!\equiv_r \omega\mu)$.

Property 9.1.5 $\forall(g_k,\sigma,\omega,\mu)\ (g_0 \rightarrow^* g_k \wedge g_k \text{—}\mu\rightarrow^* g_q \wedge \sigma\ {}_q\!\equiv_r \omega \Rightarrow \mu\sigma\ {}_k\!\equiv_r \mu\omega)$.

Property 9.1.6 $\forall(g_k,\sigma,\omega,\mu,\rho)\ (g_0 \rightarrow^* g_k \wedge \sigma\mu_{\ k}\equiv_r \omega \wedge \sigma_{\ k}\equiv_q \rho \Rightarrow \rho\mu_{\ k}\equiv_r \omega)$.

If two transition sequences are executable at a common global state and the set of active processes for these sequences are disjoint then interleavings of these sequences lead to the same global state. This is proven in the following lemma (see Figure 9.1 for a graphical illustration).

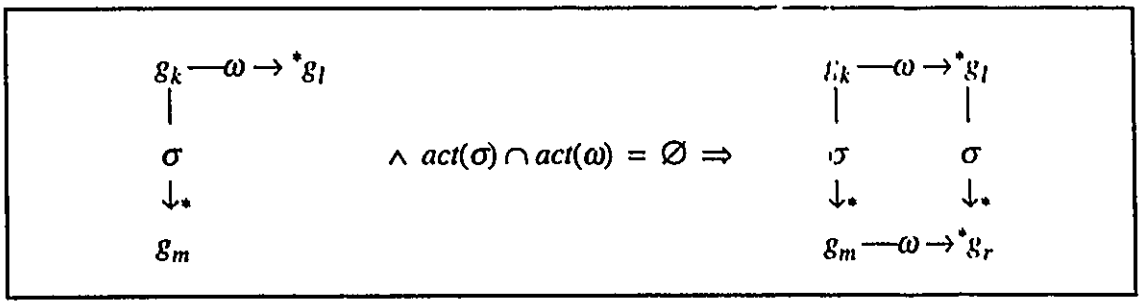


Figure 9.1 Illustration of Lemma 9.1.2.

Lemma 9.1.2

$\forall(g_k,\sigma,\omega)\ (g_0 \rightarrow^* g_k \wedge g_k$ —σ→ $^*g_m \wedge g_k$ —ω→ $^*g_l \wedge act(\sigma) \cap act(\omega) = \emptyset \Rightarrow$
$\exists g_r\ (\omega\sigma_{\ k}\equiv_r \sigma\omega))$.

Proof. Since the processes active on $\sigma\ (\omega)$ is passive on $\omega\ (\sigma)$, the execution of $\sigma\ (\omega)$ does not prevent the execution of $\omega\ (\sigma)$, i.e., there exist g_r and $g_{r'}$ such that g_k —$\sigma\omega$→ $^*g_r$ and g_l —$\omega\sigma$→ $^*g_{r'}$. Since $act(\sigma) \cap act(\omega) = \emptyset$, $\forall P_i \in P\ ((\sigma\omega)\downarrow_i = (\omega\sigma)\downarrow_i)$. This implies that $\sigma\equiv\omega$. Using Lemma 9.1.1 we conclude that $g_r = g_{r'}$. []

9.2 Simultaneously Executable Sets

It is clear that a set of transitions can be simultaneously executable only if the transitions of this set are mutually independent because a (sequential) process cannot execute more than one transition at a time.

Definition 9.2.1 A nonempty set T of transitions enabled at a global state g_k is said to be a *simultaneously executable set* of g_k iff $\forall(t,t')\ (t,t' \in T \Rightarrow t = t' \vee act(t) \cap act(t') = \emptyset)$.

$ses(g_k)$ represents the set of all simultaneously executable sets of g_k. □

The following properties of simultaneously executable sets are immediate from the definition.

Property 9.2.1. $\forall(g_k,T)\ (T \in ses(g_k) \Rightarrow 1 \leq |T| \leq n)$ where $|T|$ is the cardinality of T.

Property 9.2.2. $\forall(g_k,T,T')\ (T \in ses(g_k) \wedge T' \subset T \wedge T' \neq \emptyset \Rightarrow T' \in ses(g_k))$.

We observe the execution of a single transition at a time in the behavior of a concurrent program obtained by taking sequential product of the processes. Therefore, when several transitions are simultaneously executable (at a global state), every possible sequence of these transitions is considered in this behavior. Each such possible sequence is called a *linearization*. More specifically, the linearization of a set of transitions can be defined as follows. Let $T = \{t_1,t_2,\ldots,t_j\}$ be a simultaneously executable set of transitions and *Index* be a set of numbers from 1 to $|T|$ ($= j$). Let π be any permutation function from *Index* to *Index*. A *linearization* of T, say γ, is a transition sequence such that $\gamma = t_{\pi(1)}t_{\pi(2)}\ldots t_{\pi(j)}$. Clearly, there exist $|T|!$ linearizations of T. The set of all linearizations of T is denoted by $linear(T)$. The following lemma states that every linearization of a simultaneously executable set of a global state is an executable transition sequence from that global state.

Lemma 9.2.1

$$\forall(g_k,T,\gamma)\ (g_0 \rightarrow^* g_k \wedge T \in ses(g_k) \wedge \gamma \in linear(T) \Rightarrow \exists g_r\,(g_k \text{—}\gamma\text{→}^* g_r)).$$

Proof. Let $\gamma = t_1t_2\ldots t_j$ such that $T = \{t_1,t_2,\ldots,t_j\}$. $T \in ses(g_k)$ implies $T \subseteq enabled(g_k)$.We prove the lemma by induction on j.

Basis: Let $j = 1$. Since $t_1 \in enabled(g_k)$, there exists g_r such that $g_k \text{—}\gamma\text{→}^* g_r$.

Induction: For the induction step suppose the lemma holds for j-1, i.e., there exists g_q such that $g_k \text{—} t_1 t_2 \ldots t_{j-1} \rightarrow^* g_q$. We will prove that it also holds for j.
Since $t_j \in enabled(g_k)$, there exists g_m such that $g_k \text{—} t_j \rightarrow g_m$. By definition, $T \in ses(g_k)$ implies $act(t_1 t_2 \ldots t_{j-1}) \cap act(t_j) = \emptyset$. Using Lemma 9.1.2, we conclude that there exists g_r such that $g_k \text{—} \gamma \rightarrow^* g_r$. ||

Using the result of Lemma 9.2.1, the following lemma states that every linearization of a simultaneously executable set leads to the same global state.

Lemma 9.2.2

$$\forall (g_k, T)\ (g_0 \rightarrow^* g_k \wedge T \in ses(g_k) \wedge \gamma \in linear(T) \wedge g_k \text{—} \gamma \rightarrow^* g_r \Rightarrow$$
$$\forall (\gamma' \in linear(T))\ (\gamma_k \equiv_r \gamma').$$

Proof. It is clear that for each linearization γ' of T, $\forall P_i \in P\ (\gamma\downarrow_i = \gamma'\downarrow_i)$. By the definition of the equivalence relation, $\gamma \equiv \gamma'$. From Lemma 9.2.1, we know that there exists g_r such that $g_k \text{—} \gamma \rightarrow^* g_r$. Using Lemma 9.1.1, we conclude that $\gamma_k \equiv_r \gamma'$. ||

Consider $ses(g_k)$ of a global state g_k. From Property 9.2.2, every nonempty subset of a simultaneously executable set $T \in ses(g_k)$ is a simultaneously executable set. Since there are $2^{|T|}$-1 nonempty subsets of T and each subset may lead to a distinct global state, $2^{|T|}$-1 distinct global states may be generated in sequential product through the execution of |T|! linearizations of |T| transitions. On the other hand, when |T| transitions are simultaneously executed, the linearizations of these transitions are not considered, and hence $2^{|T|}$-2 global states are not generated.

The idea of considering only the maximal simultaneously executable sets in generating next states is very appealing for obtaining a good reduction in number of global states. However, in general it is not sufficient for verification purposes. Thus, sometimes we need to select non-maximal simultaneous executable sets. In the next section, we provide

examples to illustrate the intuition behind why and how sometimes we select non-maximal simultaneously executable sets in place of maximal ones.

9.3 Properties of Simultaneously Executable Sets

Example 9.3.1 Consider the concurrent program given in Figure 9.2. At the initial global state (1,4), t_a = ({1},a,{2}) and t_b = ({4},b,{5}) are the enabled transitions and t_c = ({2,4},c,{3,6}) is the not-enabled transition. There exists only one maximal simultaneously executable set at the initial global state, which is $\{t_a,t_b\}$. When we simultaneously execute the transitions in this set, the program reaches global state (2,5). At this global state, there exists only one enabled transition which is ({2,5},d,{1,4}). By executing this transition, the program returns to initial global state (see Figure 9.3). However, there exists one deadlock state, which is (3,6), in this program. This deadlock state is reached if we execute first t_a and then t_c (see Figure 9.4). By simultaneously executing t_a and t_b we prevent the execution of t_c after t_a. Note that both t_a and t_b are dependent with t_c.

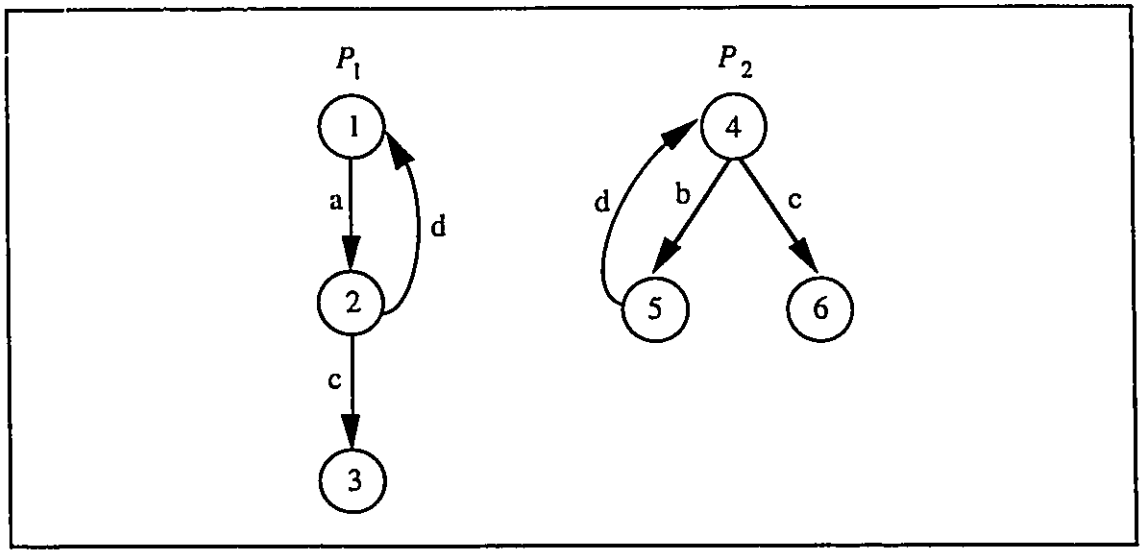


Figure 9.2 A concurrent program.

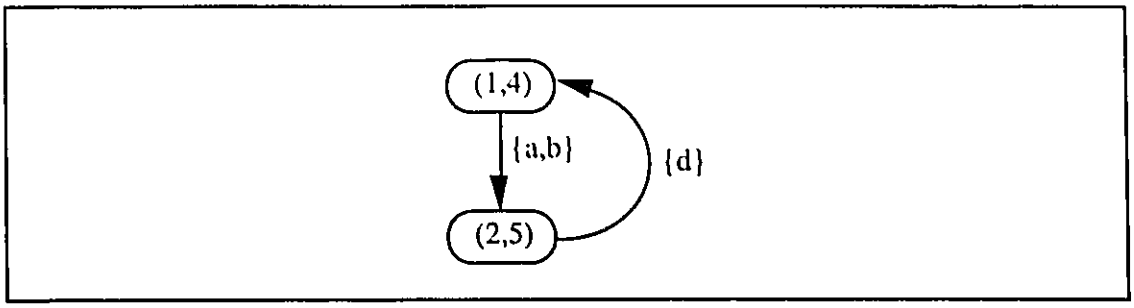


Figure 9.3 A reduced automaton for the program in Figure 9.2.

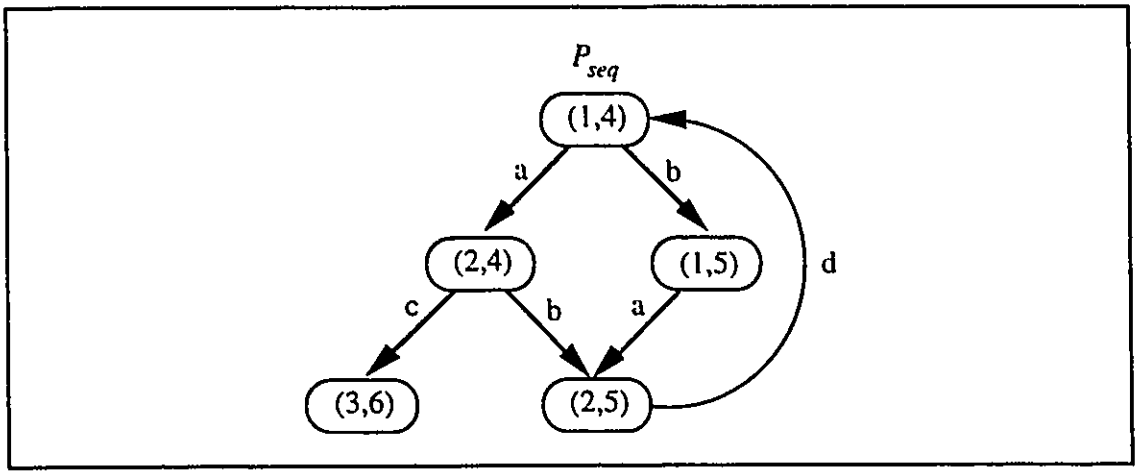


Figure 9.4 The sequential automaton for the program in Figure 9.2.

From Example 9.3.1, we observe the following:

Property 9.3.1 A *selected* simultaneously executable set of a global state should not include a pair of transitions which are dependent with a common not-enabled transition of that global state.

Example 9.3.2 Consider the concurrent program given in Figure 9.5. At the initial global state (1,3,5,8), all transitions in the program are enabled. These are t_a = ({1,3},a,{2,3}), t_b = ({3,5},b,{4,6}) and t_c = ({5,8},c,{7,9}). $\{t_a,t_c\}$ and $\{t_b\}$ are the simultaneously executable sets satisfying Property 9.3.1 at the initial global state. By executing t_a and t_c

together, the program reaches global state (2,3,7,9), a deadlock state. By executing t_b, we reach global state (1,4,6,8) which is also a deadlock state (see Figure 9.6). However, there exists one more deadlock state, (2,4,6,8), which is reached by executing first t_a and then t_b (see Figure 9.7). By simultaneously executing t_a and t_b we prevent the execution of t_c after t_a. Note that t_b, which is an enabled transition (at the initial global state), is dependent with both t_a and t_c.

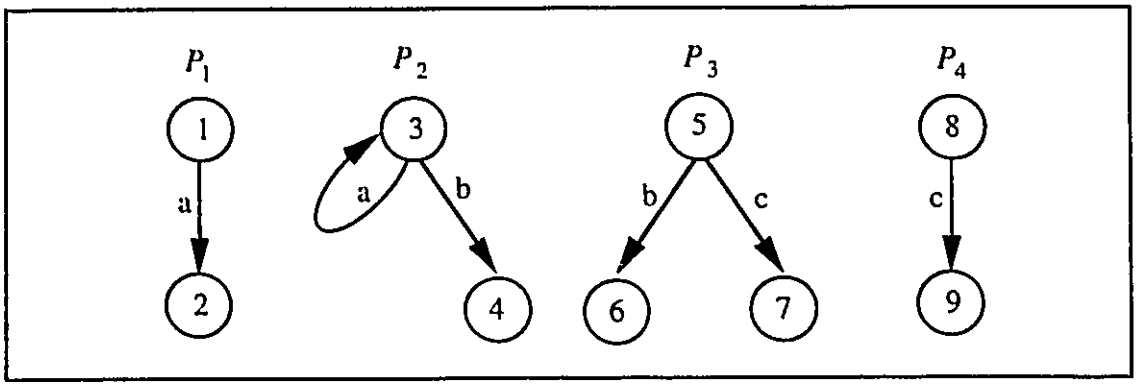


Figure 9.5 A concurrent program.

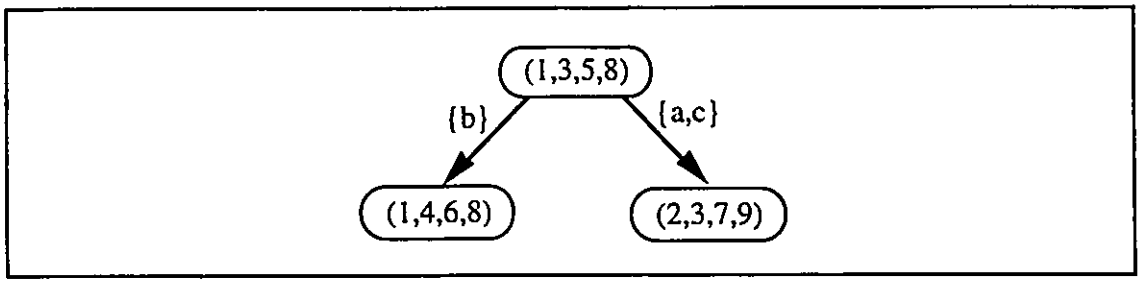


Figure 9.6 A reduced automaton for the program in Figure 9.5.

From Property 9.3.1 and Example 9.3.2, we observe the following:

Property 9.3.2 A *selected* simultaneously executable set of a global state should not include a pair of transitions dependent with a common transition related to that global state.

Note that Property 9.3.1 is implied by Property 9.3.2.

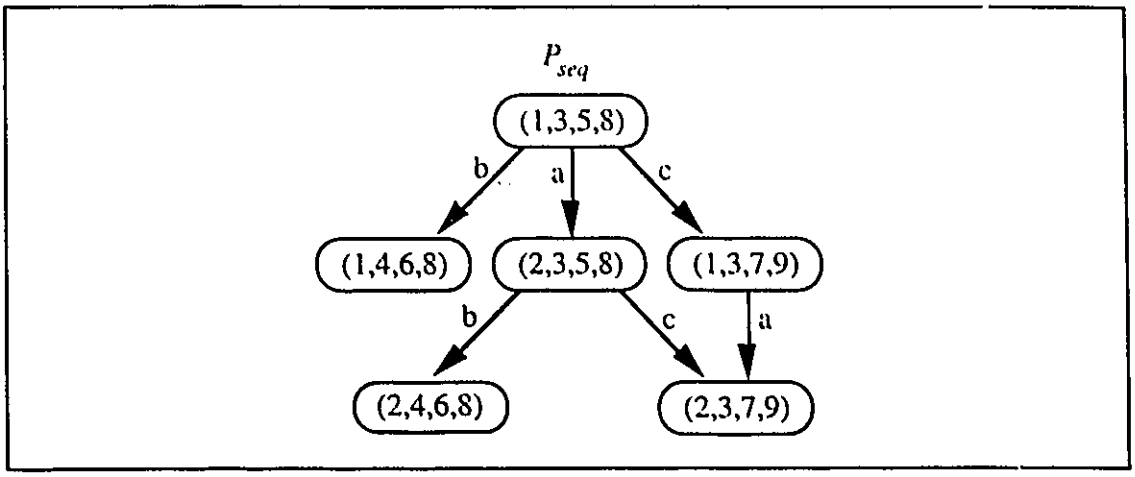


Figure 9.7 The sequential automaton for the program in Figure 9.5.

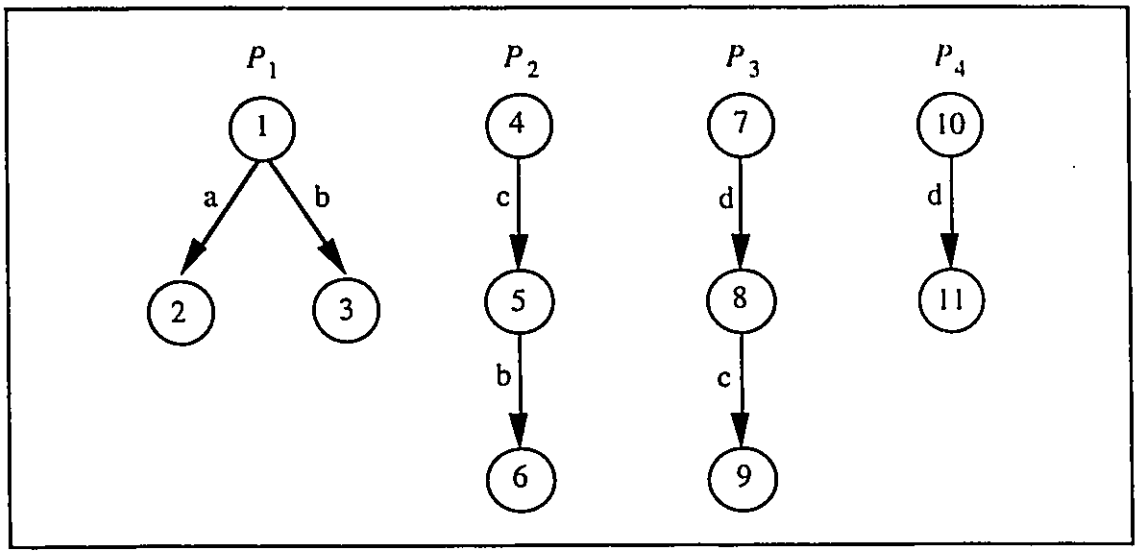


Figure 9.8 A concurrent program.

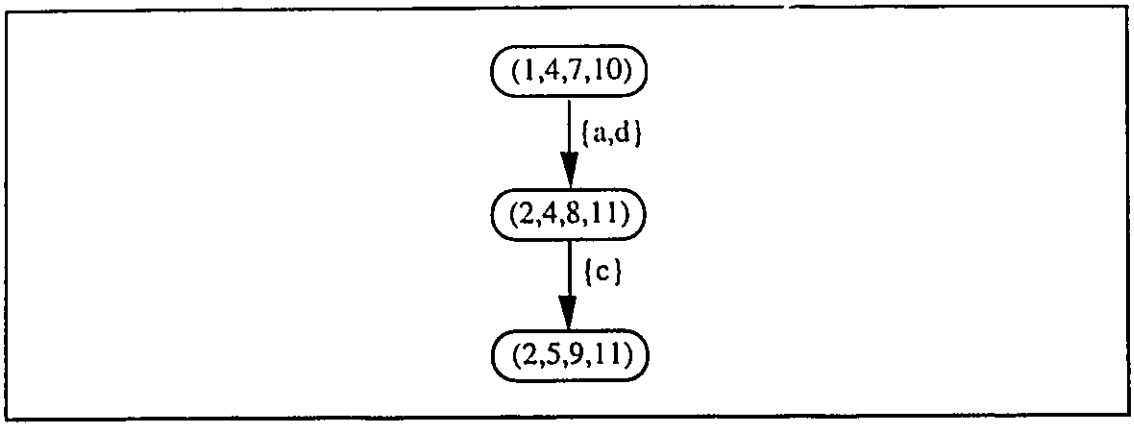


Figure 9.9 A reduced automaton for the program in Figure 9.8.

Example 9.3.3 Consider the concurrent program given in Figure 9.8. t_a = ({1},a,{2}) and t_d = ({7,10},d,{8,11}) are enabled transitions; t_b = ({1,5},b,{3,6}) and t_c = ({4,8},c,{5,9}) are not-enabled transitions at the initial global state (1,4,7,10). Note that $\{t_a,t_d\}$ is a simultaneously executable set satisfying Property 9.3.2. By simultaneously executing t_a and t_d we reach global state (2,4,8,11). Then we can only execute t_c and reach global state (2,5,9,11) (see Figure 9.9). However, deadlock state (3,6,9,11) is sequentially reached by executing first t_d, then t_c and finally t_b (see Figure 9.10).

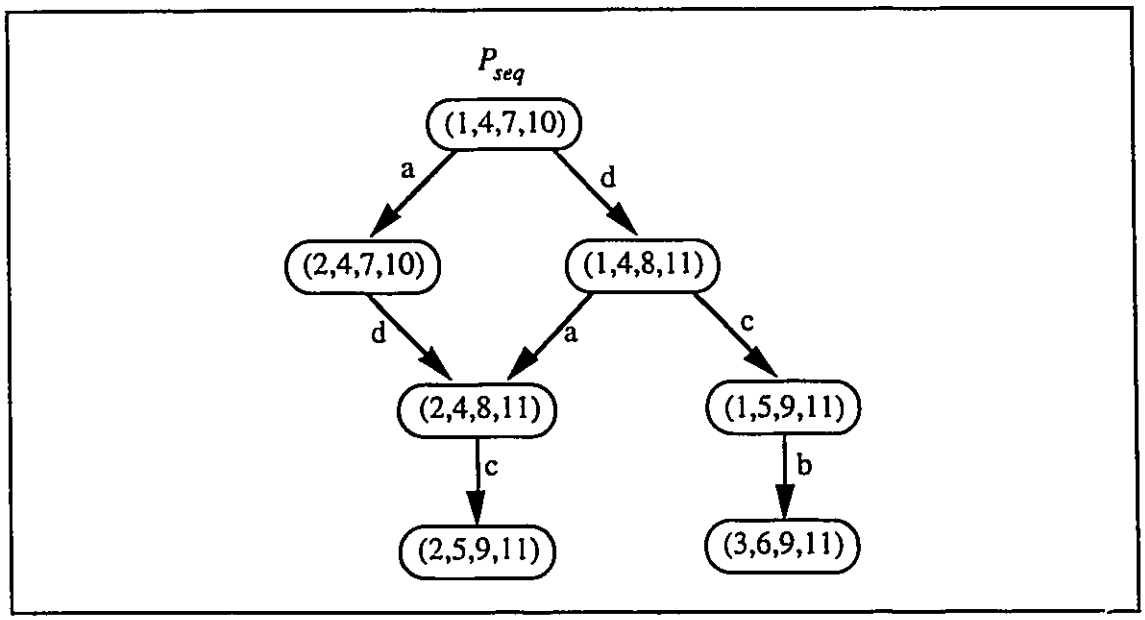


Figure 9.10 The sequential automaton for the program in Figure 9.8.

Before stating the property of selected simultaneously executable sets concluded from this example, we present another example and then give a property concluded from both examples.

Example 9.3.4 Consider the concurrent program given in Figure 9.11, where all related transitions at the initial global state (1,3,5,7) are enabled. These transitions are t_a = ({1,3},a,{2,3}), t_b = ({3,5},b,{4,5}), t_c = ({5,7},c,{6,8}) and t_d = ({7},d,{9}). Note that $\{t_a,t_d\}$, $\{t_b\}$ and $\{t_c\}$ are the simultaneously executable sets of the initial global state, satisfying Property 9.3.2. After executing these simultaneously executable sets, we can execute t_b, t_d and t_a, respectively, and then detect three deadlock states (see Figure 9.12). However, there exists one more deadlock state, (2,4,6,8), sequentially reached by executing first t_a, then t_b and finally t_c (see Figure 9.13).

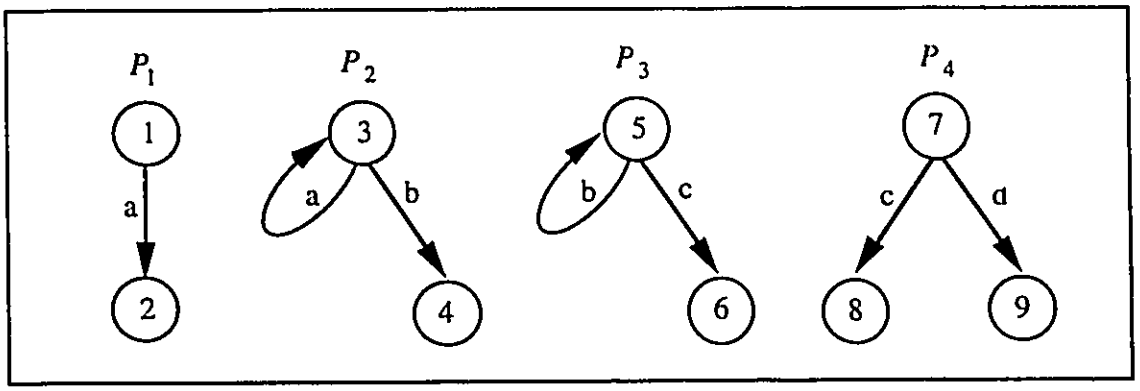


Figure 9.11 A concurrent program.

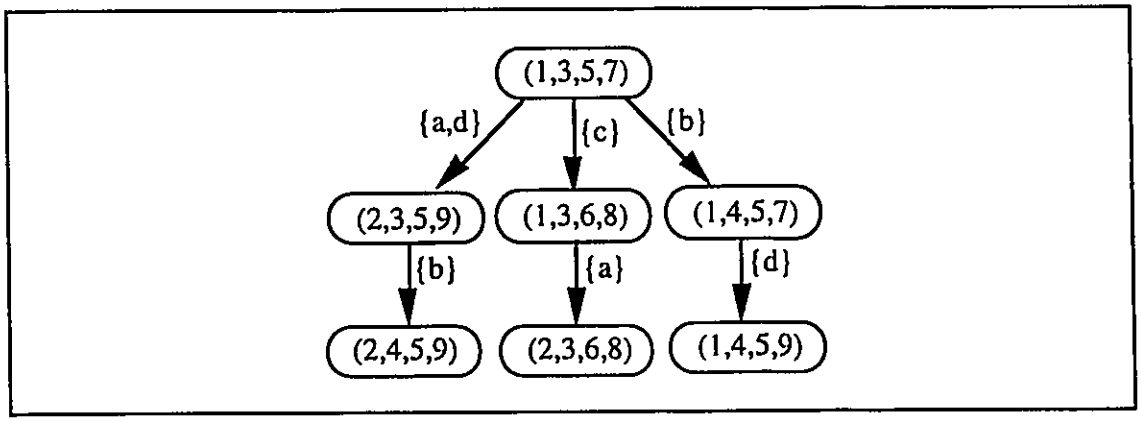


Figure 9.12 A reduced automaton for the program in Figure 9.11.

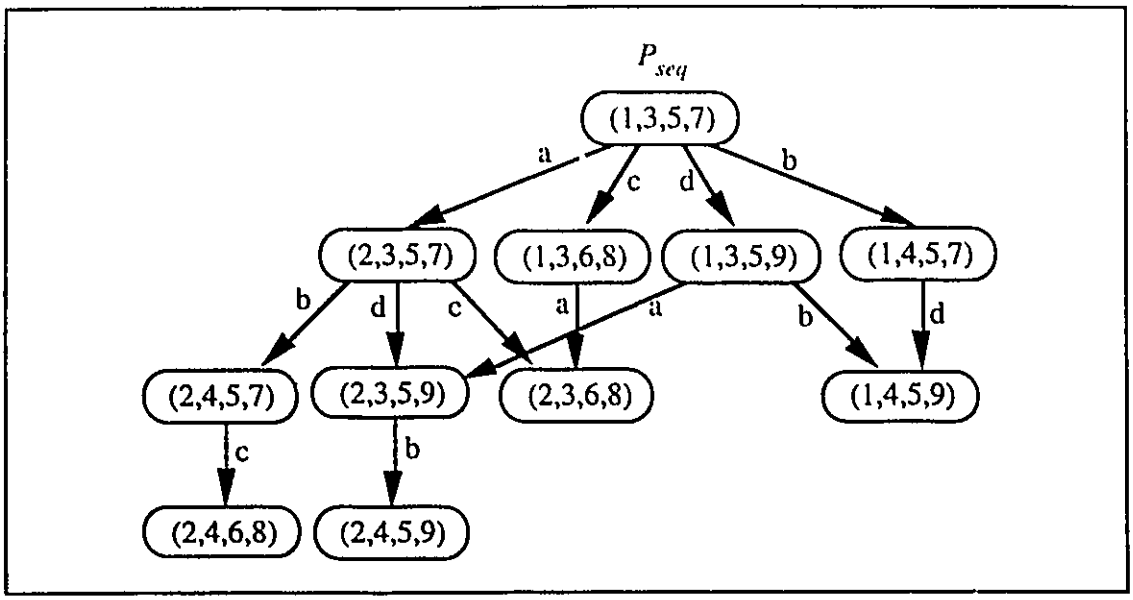


Figure 9.13 The sequential automaton for the program in Figure 9.11.

To write the property concluded from the last two examples, we need to define another relation between transitions, called *connected*, which is basically derived from the symmetric and transitive closure of dependence relation.

Definition 9.3.1 Two transitions $t,t' \in related(g)$ are *connected* at g iff $act(t) \cap act(t') \neq \emptyset \vee \exists(t_1,t_2,\ldots,t_r)\ (r \geq 1 \wedge t_1,t_2,\ldots,t_r \in related(g) \wedge act(t) \cap act(t_1) \neq \emptyset \wedge act(t_1) \cap act(t_2) \neq \emptyset \wedge \ldots. \wedge act(t_{r-1}) \cap act(t_r) \neq \emptyset \wedge act(t_r) \cap act(t') \neq \emptyset)$.

$con(t,t',g)$ denotes that t and t' are connected at g. □

In Example 9.3.3, we could not detect deadlock state (3,6,9,11) because we simultaneously executed transitions t_a and t_d at the initial global state. Note that since (t_a,t_b), (t_b,t_c), (t_c,t_d) are pairs of dependent transitions, t_a and t_d are connected (over not-enabled transitions t_b and t_c) at the initial global state. Similarly, in Example 9.3.4, we could not detect deadlock state (2,4,6,8) because we simultaneously executed transitions t_a and t_d at the initial global state. Note that since (t_a,t_b), (t_b,t_c), (t_c,t_d) are pairs of dependent

transitions, t_a and t_d are connected (over enabled transitions t_b and t_c) at the initial global state. Consequently, we decide that the following property should be satisfied.

Property 9.3.3 A *selected* simultaneously executable set of a global state should not include a pair of transitions which are connected at that global state.

Note that Property 9.3.2 is implied by Property 9.3.3.

9.4 Selected Simultaneously Executable Set

Using Property 9.3.3, we give the following definition:

Definition 9.4.1 $T \in ses(g_k)$ is a *selected simultaneously executable set* of g_k iff $\forall(t,t')$ $(t,t' \in T \Rightarrow t = t' \vee \neg con(t,t',g))$ and T is maximal.

$selected(g_k)$ represents the set of all selected simultaneously executable sets of g_k. ||

Lemma 9.4.1

$\forall(g_k,t)\ (t \in enabled(g_k) \Rightarrow \exists T\ (T \in selected(g_k) \wedge t \in T))$.

Proof. From Definition 9.4.1, $\forall T$ ((1) $T \in ses(g_k) \wedge$ (2) $\forall(t,t')\ (t,t' \in T \Rightarrow t = t' \vee \neg con(t,t',g)) \wedge$ (3) T is maximal (satisfying (1) and (2)) $\Rightarrow T \in selected(g_k)$). By definition, $\forall(g_k,t)\ (t \in enabled(G_k) \Rightarrow \{t\} \in ses(g_k))$. Therefore, $\{t\}$ satisfies (1). Since $\{t\}$ consists of one transition, it also satisfies (2). There are two cases to be considered according to whether $\{t\}$ may or may not satisfy (3).

Case 1: $\{t\}$ satisfies (3).

Then $\{t\} \in selected(g_k)$, and thus the lemma holds for this case.

Case 2: $\{t\}$ does not satisfy (3).

Then $\{t\}$ is not maximal. Since $\{t\}$ satisfies (1) and (2), and $\{t\}$ is not maximal, there exists a $T \in selected(g_k)$ such that $\{t\} \subset T$. Thus, the lemma holds for this case. []

The following lemma claims that if a transition t of a simultaneously executable set is connected to an enabled transition t' at a global state then replacing t by t' we obtain another simultaneously executable set of that global state.

Lemma 9.4.2

$\forall(T,t,t')\ (T \in selected(g_k) \wedge t \in T \wedge t' \in enabled(g_k) \wedge con(t,t',g_k) \Rightarrow (T \setminus \{t\}) \cup \{t'\} \in selected(g_k))$.

Proof. Let $\bigcup_{i=1}^{h} C_i = enabled(g_k)$ such that $\forall(i = 1,2,..,h)\ ((1) \wedge (2) \wedge (3))$ where

(1) $C_i \neq \emptyset$.

(2) $\forall(t,t' \in C_i)\ (con(t,t',g_k))$.

(3) $\forall(t,t' \in enabled(g_k))\ (t \in C_i \wedge con(t,t',g_k) \Rightarrow t' \in C_i)$.

That is, we divide $enabled(g_k)$ into h subsets C_i such that for $i = 1,2,..,h$, (1) C_i is not empty; (2) C_i is a set of mutually connected transitions; and (3) C_i is maximal.
Let $T = \{t_1,t_2,\ldots,t_h\}$ be any set such that $\forall(i = 1,2,..,h)\ (t_i \in C_i)$. Obviously, $\forall(t,t')\ (t,t' \in T \Rightarrow \neg con(t,t',g_k))$ and T is maximal. From Definition 9.4.1, $selected(g_k) = \{\{t_1,t_2,\ldots,t_h\} \mid \forall(i = 1,2,..,h)\ (t_i \in C_i)\}$. Thus, the lemma holds. []

Note that the proof of Lemma 9.2.1 is constructive. The algorithm for an implementation of this constructive proof ,which can used to efficiently obtain $selected(g_k)$ is outlined as follows:

1. A undirected graph R is constructed from $related(g_k)$. The nodes of R correspond to the transitions in $related(g_k)$. We say that a node t in R is enabled if the transition t is enabled. An edge between two nodes t_1 and t_2 is drawn iff the transitions t_1 and t_2 are dependent.

2. An enabled node, say node t, is arbitrarily selected. Starting from the node t, a depth-first search algorithm in R is applied. Clearly, all visited nodes correspond to the transitions connected to t. Let C_1 be the set of the enabled transitions connected to t. If there exists an enabled node t' which is not visited, the depth-first search algorithm starting from this node is applied. Let C_2 be the set of the enabled transitions connected to t'. We apply the same procedure until all enabled nodes are visited. Assume that at the end, we obtain $C_1, C_2, ..., C_h$. Note that $h \leq n$.

3. We construct $|C_1|*|C_2|* \ldots *|C_h|$ distinct sets with cardinality h by selecting one transition from each C_i where $i = 1,2,..,h$. These sets are the selected simultaneously executable sets of g_k.

Note that steps 1 and 2 can be implemented together, that is, we can construct R during the depth-first traversal. Moreover, a connected component of R might not contain any enabled transition. If steps 1 and 2 are implemented together then such connected components are not considered in the construction of R. Thus, it is wise to combine steps 1 and 2 as in Figure 9.14.

9.5 Simultaneous Product

Simultaneous product is obtained when the enabled individual transitions in sequential product are replaced by selected simultaneously executable sets. For the formal definition of simultaneous product, we denote the set of actions corresponds to a simultaneously executable set T by *actions*(T), i.e., *actions*(T) = $\{ a \mid \exists (t \in T)\ (t = ({}^\bullet t, a, t^\bullet)\}$. In addition we extend the definitions of preset and postset for simultaneously executable sets such that the preset and postset of a simultaneously executable set is the union of presets and postset of its transitions, respectively. Therefore, ${}^\bullet T = \bigcup_{t \in T} {}^\bullet t$ and $T^\bullet = \bigcup_{t \in T} t^\bullet$.

```
/* Steps 1 and 2 */
V = related(g_k); Start = enabled(g_k); h = 0
while Start ≠ ∅ do
begin
  remove an element t from Start
  Stack is empty
  push t onto Stack
  h = h + 1
  C_h = {t}
  Visited = ∅
  while Stack is not empty do
  begin
    pop t from Stack
    if t ∉ Visited then
    begin
      Visited = Visited ∪ {t}
      V = V \ {t}
      for each t'∈ V such that act(t) ∩ act(t')≠∅ do
        if t' ∉ Visited then
        begin
          push t' onto Stack
          if t'∈ Start then
          begin
            Start = Start \ {t'}
            C_h = C_h ∪ {t'}
          end
        end
    end
  end
end
/* Step 3 */
selected(g_k) = C_1 × C_2 × ... × C_h
```

Figure 9.14 An algorithm to obtain *selected*(g_k).

Definition 9.5.1 Let $P_i = (\Sigma_i, S_i, \Delta_i, s_{0i})$ for $i = 1,2,\ldots,n$ be processes of a concurrent program P. The finite automaton on finite words which represents the joint global behavior of processes P_i obtained by taking the *simultaneous product* (denoted by $\oplus$) of automata P_i is $P_{sim} = (\Sigma_{sim}, G, \Delta_{sim}, g_0) = P_1 \oplus P_2 \oplus \ldots \oplus P_n$ (called *simultaneous automaton*) where

- $\Sigma_{sim} = 2^{\Sigma'}$ where $\Sigma' = \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n$
- $G = S_1 \times S_2 \times \ldots \times S_n$
- Δ_{sim} is defined by (g_k, Σ'', g_l) with $g_k, g_l \in G$ iff
  - $\exists T \in selected(g_k)\ (\Sigma'' = actions(T))$ and
  - $g_l = (g_k \setminus {}^\bullet T) \cup T^\bullet$.
- $g_0 = (s_{01}, s_{02}, \ldots, s_{0n})$ ||

In the behavior of a concurrent program obtained by taking simultaneous product of its processes, a global state, say g_k, produces another global state, say g_l, if there exists a selected simultaneously executable set of g_k and the processes execute simultaneously the transitions in that set.

Definition 9.5.2 A global state g_l is a *simultaneous immediate successor* of g_k, denoted by $g_k \longrightarrow\!\!\!\!\!\rightarrow g_l$ or $g_k \text{—}T\text{—}\!\!\!\!\!\rightarrow g_l$, iff there exists $T \in selected(g_k)$ such that $\gamma \in linear(T)$ and $g_k \text{—}\gamma\!\rightarrow^* g_l$. ||

Note that Definition 9.5.2, the simultaneous immediate successor, defines a binary relation between global states which is denoted by $\longrightarrow\!\!\!\!\!\rightarrow$.

Definition 9.5.3 Let $\longrightarrow\!\!\!\!\!\rightarrow^*$ be the reflexive and transitive closure of $\longrightarrow\!\!\!\!\!\rightarrow$.

- g_m is simultaneously reachable from g_k (or a simultaneous successor of g_k) iff $g_k \longrightarrow\!\!\!\!\!\rightarrow^* g_m$.

- g_m is simultaneously reachable iff $g_0 —»^* g_m$.

$g_k —T_1 —»g_{k(1)} \wedge g_{k(1)} —T_2 —»g_{k(2)} \wedge \ldots \wedge g_{k(r-1)} —T_r —»g_m$ will sometimes be denoted by $g_k —T_1T_2...T_r \rightarrow^* g_m$. []

In the following theorem we prove that every simultaneously reachable global state is a sequentially reachable global state.

Theorem 9.5.1 $\forall(g_k)\ (g_0 —»^* g_k \Rightarrow g_0 \rightarrow^* g_k)$.

Proof: Let $g_0 —T_1T_2...T_j —»^* g_k$ and $\gamma_i \in linear(T_i)$ for $i = 1,2,...j$. Using Lemma 9.2.2, $g_0 —\gamma_1\gamma_2...\gamma_j \rightarrow^* g_k$. []

In the rest of this section, we analyze the reduction in the number of global states when simultaneous product method is used instead of sequential product method and compare the time complexities of both methods.

Let g_k be a simultaneously reachable global state and $C_1, C_2, \ldots C_h$ be obtained from steps 1 and 2 of the algorithm given in Figure 9.14. For the sake of simplicity, we assume that $|C_i| = C$ for all $i = 1,2,..,h$. Let F be the set of simultaneous immediate successors of a global state g_k, i.e., $F = \{ g_m \mid g_k —» g_m \}$. Then $|F| = C^h$. Since in sequential product method a global state is generated by executing a single transition and each global state in F is sequentially reached from g_k by executing h transitions, the sequential product method has to generate a set I of global states (which are skipped in simultaneous product method by jumping from g_k to global states in F) before generating the ones in F. Assume that we divide I into $h-1$ subsets I_j such that I_j is the set of global states sequentially reached from g_k by the execution j transitions, where $j = 1,2,..,h-1$ (see Figure 9.15). Since every linearization of a simultaneously executable set is executable and every nonempty subset of a selected simultaneously executable set is a simultaneosly executable set, formally, $I = \bigcup_{j=1}^{h-1} I_j$ such that for $j = 1,2,..,h-1$, $I_j = \{ g_m \mid g_k —t_1t_2...t_j \rightarrow^* g_m \wedge \exists(T \in ses(g_k))$

($\{t_1,t_2,\ldots,t_j\} \subset T$). Since every linearization of a simultaneously executable set leads to the same global state, for $j = 1,2,..,h-1$, the number of global states in I_j is equal to the number of subsets of selected simultaneously executable sets, consisting of j transitions. Since each selected simultaneously executable set is obtained by selecting one transition from each connected set C_i where $i = 1,2,..,h$, a subset of a selected simultaneously executable set consisting of j transitions is obtained first selecting j connected sets and then selecting one transition from each connected set which is selected. Since we can select $\frac{h*(h-1)*\ldots*(h-j+1)}{j!}$ distinct sets consisting of j connected sets from h connected sets and C^j simultaneously executable sets from j connected sets, we write

$$|I_j| = \frac{h*(h-1)*\ldots*(h-j+1)*C^j}{j!}$$

Then

$$|I| = \sum_{j=1}^{h-1}(|I_j|$$

$$= \sum_{j=1}^{h-1} \frac{h*(h-1)*\ldots*(h-j+1)*C^j}{j!}$$

$$= (C+1)^{h-1} - 1$$

Since the global states in I is ignored while generating the ones in F, and $|I| > |F|$ when h > 2 and C > 2, one can except a large number of reduction in the number of global states gained by simultaneous product method.

In the following comparison of the time complexities of both methods, we assume that checking whether any given transition is enabled at a global state requires constant time. As can be seen from Definition 7.5.8, actually it requires $O(n)$ time but our assumption does not effect the result of the comparison. Let $R = (V, E)$ where $V = related(g_k)$ and E be the set of edges representing dependence relations between transitions in $related(g_k)$. Since a depth-first traversal of R *can* be implemented in $O(|V| + |E|)$ time and steps 1 and 2 are

combined in the algorithm in Figure 9.14, these steps can be completed in $O(|V| + |E|)$ time. Concurrency is known as the major contributor for state explosion. Simultaneous product method is proposed for reducing the state explosion which mostly occurs due to the concurrency among the processes of a concurrent program. Concurrency means that several transitions can be executed simultaneously at a global state. In other words, we observe more independence, i.e., less dependence among transitions. Therefore, we assume that $|E| = O(|V|)$. This assumption is reasonable when the simultaneous product method is used because the simultaneous product method usually does not reduce the size of state space when there is no concurrency. Note that a similar assumption is also made for the stubborn set method in [Valm92]. Therefore, steps 2 and 3 are completed in $O(|V|)$ time.

If each C_i is viewed as an array then a simple h nested loops is sufficient to print the selected simultaneously executable sets, where ith loop iterates $|C_i|$ times. Since h nested loops iterates $|C_1|*|C_2|* \ldots *|C_h|$ times and each simultaneously executable set includes h transitions, Step 3 can be completed in $O(h*C^h)$ time. Let $Time_{sim}$ be the time complexity of generating all the global states in F from g_k in simultaneous product method. Since the generation a global state by executing a simultaneously executable set is implemented by executing h transitions, we write that

$$Time_{sim} = O(|V|+h*C^h)$$

In order to compare the time complexity of sequential and simultaneous product methods, let $Time_{seq}$ be the time complexity of generating all the global states in F from g_k in sequential product method. At a global state, first, enabled transitions are determined by searching all the transitions in $related(g_k)$ and then sequential immediate successors are generated in sequential product method. Then searching for enabled transitions requires $O(|V|)$ while generating sequential immediate successors requires $O(|enabled(g_k)|)$. Note that $|enabled(g_k)| = h*C$. For the sake of simplicity, we assume that searching for enabled

transitions requires |V| for all global states in I. Let g' be a global state in I_j. Without loss of generality, assume that $g_k \text{—}t_1t_2\ldots t_j \rightarrow^* g'$ and for $i = 1,2,\ldots j$, $t_i \in C_i$. By definition of I_{j+1}, $\forall(t \in \bigcup_{i=j+1}^{h} C_i)\,(g' \text{—}t \rightarrow g'' \Rightarrow g'' \in I_{j+1})$. This implies that every global state in I_j has $(h-i)*C$ sequential immediate successors which are member of I_{j+1}. In other words, at every global state in I_j, $(h-j)*C$ global states have to be generated in sequential product method, where $j = 1,2,\ldots, h-1$ (Figure 9.15). Therefore,

$$Time_{seq} = O(h*C + |V| + \sum_{j=1}^{h-1}(|I_j|*(|V| + (h-j)*C)))$$

$$= O(h*C + |V|*(1 + \sum_{j=1}^{h-1}\frac{h*(h-1)*\ldots*(h-j+1)*C^j}{j!}) + \sum_{j=1}^{h-1}\frac{h*(h-1)*\ldots*(h-j)*C^{j+1}}{j!})$$

$$= O(|V|*(1+(C+1)^{h-1}-1) + h*C + \sum_{j=1}^{h-1}\frac{h*(h-1)*\ldots*(h-j+1)*C^{j+1}}{j!})$$

$$= O(|V|*(C+1)^{h-1} + \sum_{j=1}^{h}\frac{h*(h-1)*\ldots*(h-j+1)*C^j}{(j-1)!})$$

Obviously, $Time_{sim} \leq Time_{seq}$. As a result, when compared to sequential product method, simultaneous product method generates less global states without consuming more time.

9.6 Deadlock Detection

In this section, we prove that the simultaneous automaton of a concurrent program P is sufficient to find all deadlocks in P. Before giving these theorems we will prove two important lemmas. Given any sequentially reachable global state g_k and a transition sequence σ such that $g_k \text{—}\sigma \rightarrow^* g_m$, the following lemma states that there exists a global state g_r such that g_r is sequentially reachable from both g_m and one of the simultaneous immediate successor of g_k (see Figure 9.16 for a graphical illustration).

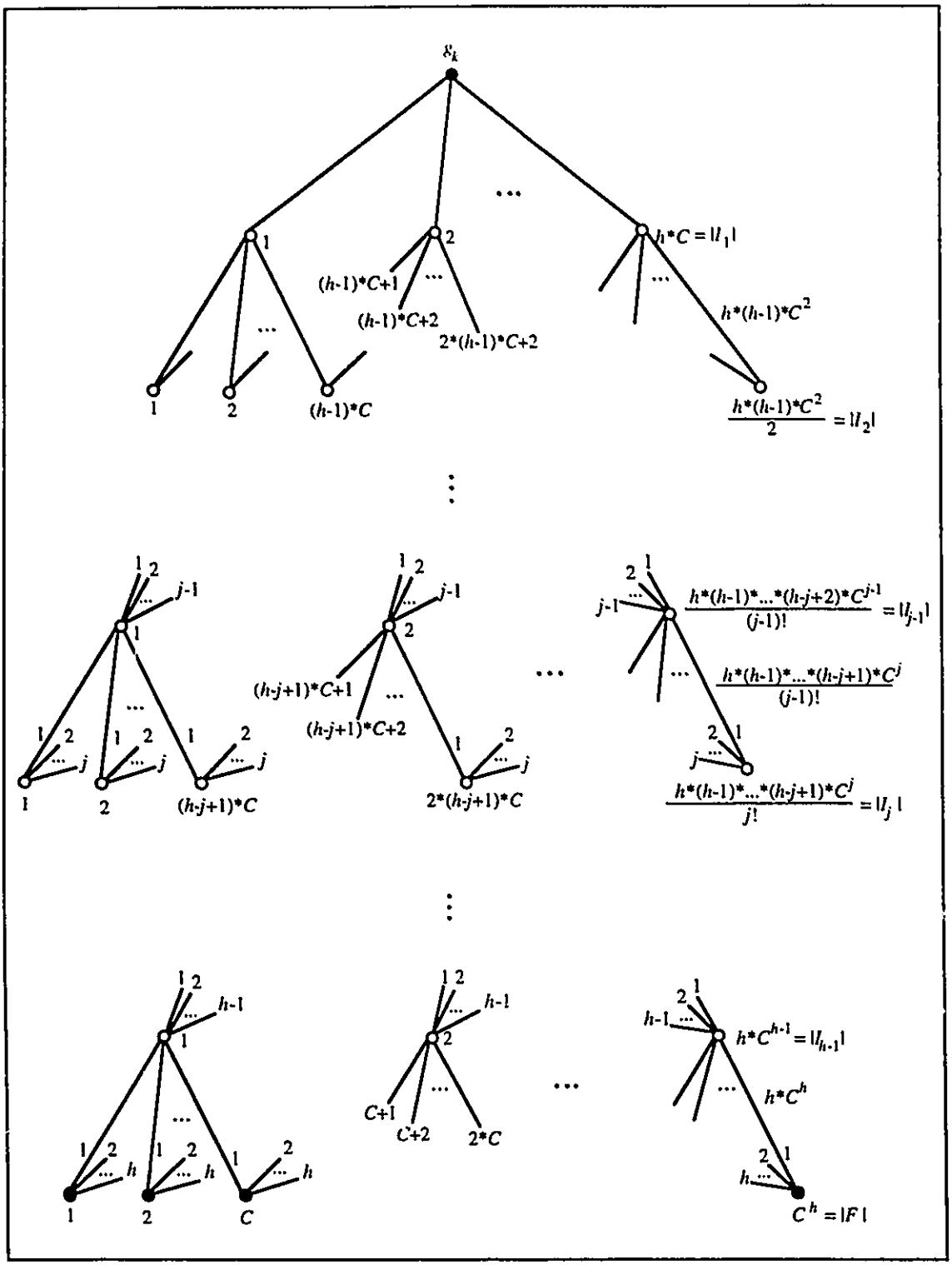


Figure 9.15 Illustration of the complexity analyses.

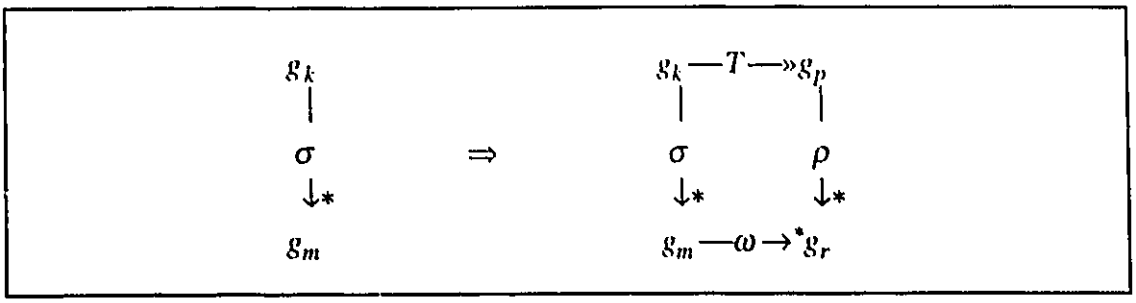


Figure 9.16 Illustration of Lemma 9.6.1.

Lemma 9.6.1

$\forall(g_k,\sigma)\ (g_0 \rightarrow^* g_k \wedge g_k \text{—}\sigma\rightarrow^* g_m \wedge \sigma \neq \varepsilon \Rightarrow$

$\exists(T,\omega,\rho,g_r)\ (T \in selected(g_k) \wedge \gamma \in linear(T) \wedge \sigma\omega_k \equiv_r \gamma\rho \wedge |\sigma| > |\rho|))$.

Proof: Let *First* be the set of transitions from σ such that transition $t \in First$ iff $\exists\mu\ (\mu t \in prefix(\sigma) \wedge act(\mu) \cap act(t) = \varnothing)$. Let t_1 be the first transition in σ. Clearly, $t_1 \in First$. Then $\sigma \neq \varepsilon$ implies $First \neq \varnothing$. From Lemma 9.4.1, $\forall t\ (t \in enabled(g_k) \Rightarrow \exists T\ (T \in selected(g_k) \wedge t \in T))$. Since $t_1 \in enabled(g_k)$, $\exists T\ (T \in selected(g_k) \wedge T \cap First \neq \varnothing)$.

Case 1: $T \subseteq First$.

Let $T = \{t_1,t_2,...,t_j\}$. Since $T \subseteq First$, $\sigma = t_1\mu_1t_2\mu_2...t_j\mu_j$ such that $act(t_1\mu_1t_2\mu_2...t_i\mu_i) \cap act(t_{i+1}) = \varnothing$ for $i = 1,2,...,j\text{-}1$ and $j < n$.

Let $g_k \text{—}t_1\mu_1 \rightarrow g_{k(1)} \text{—}t_2\mu_2 \rightarrow^* ... \text{—}t_{j-1}\mu_{j-1} \rightarrow^* g_{k(j-1)} \text{—}t_j\mu_j \rightarrow^* g_m$.

$t_1\mu_1t_2\mu_2...t_j\mu_j \equiv_{k,m} t_1t_2...t_j\mu_1\mu_2...\mu_j$ will be proven by induction on j.

Basis: Let $j = 1$ then from Property 9.1.3, $t_1\mu_1 \equiv_{k,m} t_1\mu_1$.

Induction: For the induction step suppose the claim holds for j-1, i.e.

$$t_1\mu_1t_2\mu_2...t_{j-1}\mu_{j-1} \equiv_{k(j-1)} t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}. \qquad (1)$$

It will be proven that it also holds for j.

Using Property 9.1.4, $g_{k(j-1)} \text{—}t_jt_j \rightarrow^* g_m \wedge (1) \Rightarrow$

$$t_1\mu_1t_2\mu_2...t_{j-1}\mu_{j-1}t_j\mu_j \equiv_{k,m} t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\mu_j \qquad (2)$$

Using (1), let $g_k \text{—}t_1t_2...t_{j-1} \rightarrow^* g_l \text{—}\mu_1\mu_2...\mu_{j-1} \rightarrow^* g_{k(j-1)}$. Using Lemma 9.1.2,

$t_j \in enabled(g_k) \wedge g_k \text{—}t_1t_2...t_{j-1}\rightarrow^* g_l \wedge act(t_j) \cap act(t_1t_2...t_{j-1}) = \emptyset \Rightarrow t_j \in enabled(g_l)$.

Again using Lemma 4.1.2,

$t_j \in enabled(g_l) \wedge g_l \text{—}\mu_1\mu_2...\mu_{j-1}\rightarrow^* g_{k(j-1)} \wedge act(\mu_1\mu_2...\mu_{j-1}) \cap act(t_j) = \emptyset \Rightarrow$

$\exists g_q\ (\mu_1\mu_2...\mu_{j-1}t_j \not\equiv_q t_j\mu_1\mu_2...\mu_{j-1})$.

From Property 9.1.5,

$g_k \text{—}t_1t_2...t_{j-1}\rightarrow^* g_l \wedge \mu_1\mu_2...\mu_{j-1}t_j \not\equiv_q t_j\mu_1\mu_2...\mu_{j-1} \Rightarrow$

$t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j \not\equiv_q t_1t_2...t_{j-1}t_j\mu_1\mu_2...\mu_{j-1}$. (3)

$g_k \text{—}t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\mu_j\rightarrow^* g_m$ (from (2)) $\wedge$ $g_k \text{—}t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\rightarrow^* g_q$ (from (3)) $\Rightarrow$ $g_q \text{—}\mu_j\rightarrow^* g_m$.

From Property 9.1.4,

$g_q \text{—}\mu_j\rightarrow^* g_m \wedge$ (3) $\Rightarrow$ $t_1t_2...t_{j-1}\mu_1\mu_2...\mu_{j-1}t_j\mu_j \equiv_m t_1t_2...t_j\mu_1\mu_2...\mu_{j-1}\mu_j$. (4)

From Property 9.1.6,

(2) $\wedge$ (4) $\Rightarrow$ $t_1\mu_1t_2\mu_2...t_{j-1}\mu_{j-1}t_j\mu_j \equiv_m t_1t_2...t_{j-1}t_j\mu_1\mu_2...\mu_{j-1}\mu_j$.

Let $\gamma = t_1t_2...t_j$ and $\rho = \mu_1\mu_2...\mu_j$ then $\sigma \equiv_m \gamma\rho$ and $\gamma \in linear(T)$. Finally, let $\omega = \varepsilon$ and $g_r = g_m$ then $\sigma\omega \equiv_r \gamma\rho$. $|\gamma| > 1$, $|\omega| = 0$ and $|\sigma| = |\gamma\rho|$ (from Property 9.1.1) imply $|\sigma| > |\rho|$.

Case 2: $T \not\subset First$.

We know that $\exists T\ (T \in selected(g_k) \wedge T \cap First \neq \emptyset)$. Let $T \in selected(g_k)$ such that $T \cap First$ is maximal. $act(T \setminus First) \cap act(\sigma) = \emptyset$ will be proven by contradiction.

Suppose $act(T \setminus First) \cap act(\sigma) \neq \emptyset$, i.e., $\exists P_i\ (P_i \in act(T \setminus First) \cap act(\sigma))$. Let $P_i \in act(t) \cap act(t')$, $t \in T \setminus First$ and $\mu t' \in prefix(\sigma)$ so that $act(T \setminus First) \cap act(\sigma) \neq \emptyset$. Assume that t' is the first transition on σ satisfying $P_i \in act(t) \cap act(t')$. This implies $P_i \notin act(\mu)$.

Let s_i be the state of P_i at g_k and $g_k \text{—}\mu\rightarrow^* g'$. Since $P_i \notin act(\mu)$, s_i is also the state of P_i at g'. $\mu t' \in prefix(\sigma)$ implies $t' \in enabled(g')$. $P_i \in act(t')$, $t' \in enabled(g')$ and s_i is the state of P_i at g' imply $s_i \in {}^\bullet t'$. Then ${}^\bullet t' \cap g_k \neq \emptyset$, i.e., $t' \in related(g_k)$. Let t'' be the first

occurrence of a transition in μ such that $con(t',t'',g_k)$ and $\mu't'' \in prefix(\mu)$. Note that t'' could be equal to t' because $con(t',t',g_k)$.

We show that $act(t'') \cap act(\mu') = \emptyset$ by contradiction. Suppose $act(t'') \cap act(\mu') \neq \emptyset$, i.e., $\exists P_j\,(P_j \in act(t'') \cap act(\mu'))$. Let $P_j \in act(t'') \cap act(t''')$ and $\mu''t''' \in prefix(\mu')$ so that $act(t'') \cap act(\mu') \neq \emptyset$. Assume that t''' is the first transition on μ' satisfying $act(t'') \cap act(t''') \neq \emptyset$. This implies $P_j \notin act(\mu'')$. Let s_j be the state of P_j at g_k and $g_k \text{—}\mu''\rightarrow^* g''$. Since $P_j \notin act(\mu'')$, s_j is also the state of P_j at g''. $\mu''t''' \in prefix(\mu')$ implies $t''' \in enabled(g'')$. $P_j \in act(t''')$, $t''' \in enabled(g'')$ and s_j is the state of P_j at g'' imply $s_j \in {}^\bullet t'''$. Then ${}^\bullet t''' \cap g_k \neq \emptyset$, i.e., $t''' \in related(g_k)$. $con(t',t'',g_k)$, $act(t'') \cap act(t''') \neq \emptyset$ and $t''' \in related(g_k)$ imply $con(t',t''',g_k)$. Then t'' is not the first occurrence of a transition in μ such that $con(t',t'',g_k)$, a contradiction. Therefore, $act(t'') \cap act(\mu') = \emptyset$.

$\mu't'' \in prefix(\mu)$ and $act(t'') \cap act(\mu') = \emptyset$ imply that $t'' \in First$ and $t'' \in enabled(g_k)$. Using Lemma 9.4.1, $T \in selected(g_k) \wedge t \in T \wedge t'' \in enabled(g_k) \wedge con(t,t'',g_k) \Rightarrow (T \setminus \{t\}) \cup \{t''\} \in selected(g_k)$. Clearly, $|T \cap First|+1 = |((T \setminus \{t\}) \cup \{t''\}) \cap First|$. This means that $T \cap First$ is not maximal, a contradiction. Thus, $act(T \setminus First) \cap act(\sigma) = \emptyset$.

Let ω be a linearization of $T \setminus First$. Using Lemma 9.2.1, $(T \setminus First) \in ses(g_k)$ implies that there exists g_h such that $\omega \in linear(T \setminus First) \wedge g_k \text{—}\omega\rightarrow^* g_h$.

$\omega \in linear(T \setminus First) \wedge act(T \setminus First) \cap act(\sigma) = \emptyset \Rightarrow act(\sigma) \cap act(\omega) \neq \emptyset$.

$g_k \text{—}\omega\rightarrow^* g_h \wedge g_k \text{—}\sigma\rightarrow^* g_m \wedge act(\sigma) \cap act(\omega) \neq \emptyset \Rightarrow \exists g_r\,(\sigma\omega\ {}_k\!\equiv_r \omega\sigma)$ (Lemma 9.1.2).

Let $\sigma' = \sigma\omega$. Let $First'$ be the set of transitions from σ' such that transition $t \in First'$ iff $\exists \mu\,(\mu t \in prefix(\sigma') \wedge act(\mu) \cap act(t) = \emptyset)$. By the definition of $ses(g_k)$, $(T \setminus First) \in ses(g_k)$ implies

$$\forall (t,t')\,(t,t' \in (T \setminus First) \Rightarrow act(t) \cap act(t') = \emptyset). \qquad (5)$$

$act(\sigma) \cap act(\omega) \neq \emptyset \wedge \omega \in linear(T \setminus First) \wedge (5) \Rightarrow First' = First \cup (T \setminus First)$.

$First' = First \cup (T \setminus First) \wedge First \neq \emptyset \wedge T \neq \emptyset \Rightarrow T \subset First'$. Therefore, we transform Case 2 into Case 1 by letting $\sigma = \sigma'$. Then $\exists(\gamma,\omega,\rho,g_r)\,(\sigma\omega\ {}_k\!\equiv_r \gamma\rho)$. Since $(T$

$\backslash First) \subset T$, $|\omega| < |\gamma|$. Finally $|\omega| < |\gamma|$ and $|\sigma\omega| = |\gamma\rho|$ (from Property 9.1.1) imply $|\sigma| > |\rho|$. □

The following lemma is a generalization of Lemma 9.6.1. Given any sequentially reachable global state g_k and a transition sequence σ such that $g_k \text{—}\sigma\text{→}^* g_m$, the lemma states that there exists a global state g_r such that g_r is sequentially reachable from g_m and simultaneously reachable from g_k (see Figure 9.17 for a graphical illustration).

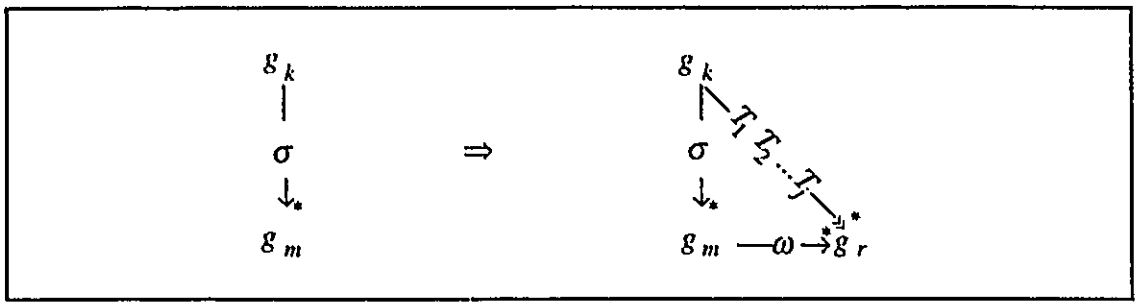


Figure 9.17 Illustration of Lemma 9.6.2.

Lemma 9.6.2

$\forall(g_k,\sigma)\ (g_0 \text{→}^* g_k \wedge g_k \text{—}\sigma\text{→}^* g_m \wedge \sigma \neq \varepsilon \Rightarrow$

$\exists(g_r,\omega j)\ (g_k \text{—}T_1\text{—»}g_{k(1)} \text{—}T_2\text{—»}g_{k(2)} \wedge \ldots \wedge g_{k(j-1)} \text{—}T_j\text{—»}g_r \wedge$

$\sigma\omega_{\,k} \equiv_r \gamma_1\gamma_2\ldots\gamma_j \wedge \gamma_i \in linear(T_i)\ (i = 1,2,\ldots j))$.

Proof: Using Lemma 9.6.1, $\exists(T_i,\gamma_i,\omega_i,\rho_i,g_{r(i)}\ (i = 1,2,\ldots j))$

$((\gamma_1 \in linear(T_1) \wedge \sigma\omega_{1\,k} \equiv_{r(1)} \gamma_1\rho_1 \wedge |\sigma| > |\rho_1|) \wedge$

$(\gamma_2 \in linear(T_2) \wedge \rho_1\omega_{2\,k(1)} \equiv_{r(2)} \gamma_2\rho_2 \wedge |\rho_1| > |\rho_2|) \wedge \ldots \wedge$

$(\gamma_j \in linear(T_j) \wedge \rho_{j-1}\omega_{j\,k(j-1)} \equiv_{r(j)} \gamma_j\rho_j \wedge |\rho_{j-1}| > |\rho_j|))$.

Clearly, $g_{r(i)} \text{—}\omega_{\,i+1} \text{→}^* g_{r(i+1)}$ for $i = 1,2, \ldots j\text{-}1$.

$\sigma\omega_1\omega_2\ldots\omega_{j\,k} \equiv_{r(j)} \gamma_1\gamma_2\ldots\gamma_j\rho_j$ will be proven by induction on j.

Basis: Let $j = 1$ then $\sigma\omega_{1\,k} \equiv_{r(1)} \gamma_1\rho_1$.

Induction: For the induction step suppose the claim holds for j-1, i.e.,

$$\sigma\omega_1\omega_2...\omega_{j-1}\ {}_k{\equiv}_{r(j-1)}\ \gamma_1\gamma_2\cdots\gamma_{j-1}\rho_{j-1} \quad (1)$$

It will be proven that it also holds for j. (1) $\wedge$ $g_{r(j-1)}$ —ω_j→* $g_{r(j)}$ $\Rightarrow$

$$\sigma\omega_1\omega_2...\omega_{j-1}\ \omega_j\ {}_k{\equiv}_{r(j)}\ \gamma_1\gamma_2\cdots\gamma_{j-1}\rho_{j-1}\omega_j. \quad (2)$$

(2) $\wedge$ $\rho_{j-1}\omega_j\ {}_{k(j-1)}{\equiv}_{r(j)}\ \gamma_j\rho_j$ $\Rightarrow$ $\sigma\omega_1\omega_2...\omega_j\ {}_k{\equiv}_{r(j)}\ \gamma_1\gamma_2...\gamma_j\rho_j$.

Since $|\sigma|$ is finite, $|\sigma| > |\rho_1|$, for i = 1,2, ...j-1, $|\rho_i| > |\rho_{i+1}|$, and Lemma 9.6.1 can be applied to ρ_i as long as $|\rho_i| > 0$, it can be assumed that $|\rho_j| = 0$. Then one can write $\sigma\omega\ {}_k{\equiv}_r$ $\gamma_1\gamma_2...\gamma_j$ where $\omega = \omega_1\omega_2...\omega_j$ and $g_{r(j)} = g_r$. ||

Theorem 9.6.1 Let g_d is a deadlock state. g_0 →* g_d iff g_0 —»* g_d.

Proof: *If Part.* Straightforward from Theorem 9.5.1. Since every simultaneously reachable global state is sequentially reachable, every deadlock state in the simultaneous automaton of a program is also a deadlock state in the sequential automaton of that program.

Only If Part. Let g_0 —σ →* g_d. From Lemma 9.6.2, $\sigma\omega\ {}_0{\equiv}_r\ \gamma_1\gamma_2...\gamma_j$ where g_0 —$T_1T_2...T_j$—»* g_r and $\gamma_i \in linear(T_i)$ for i = 1,2,...j. Since g_d is a deadlock state, ω must be empty. Therefore, $\sigma\ {}_0{\equiv}_d\ \gamma_1\gamma_2...\gamma_j$. ||

9.7 Verifying Safety Properties

In section 7.5, we have showed that the verification a safety property of a concurrent system can be reduced to the reachability of a local state. In this section, we prove the local reachability preserving property of simultaneous product so that any safety property can be verified using simultaneous product in place of sequential product.

Theorem 9.7.1 A local state is sequentially reachable iff it is simultaneously reachable.

Proof: *if Part.* Straightforward from Theorem 9.5.1. Since every simultaneously reachable global state is sequentially reachable, every simultaneously reachable local state is sequentially reachable.

Only If Part. Let s be a sequentially reachable local state. We consider two complementary cases :

Case 1: $s \in g_0$.

Since g_0 is simultaneously reachable, s is also simultaneously reachable.

Case 2: $s \notin g_0$.

By definition, there exists a sequentially reachable global state g such that $s \in g$.

Let $g_0 \text{—}\sigma\text{→}^* g$. Since $s \notin g_0$ and $s \in g$, there must be a transition t in σ such that $s \in {}^\bullet t$. From Lemma 9.6.2, $\sigma\omega_0 \equiv_r \gamma_1\gamma_2...\gamma_j$ where $g_0 \text{—}T_1T_2...T_j\text{—»}^* g_r$ and $\gamma_i \in linear(T_i)$ for $i = 1,2,...,j$. From Property 9.2.2, t is also in $\gamma_1\gamma_2...\gamma_j$. This implies that there exists a global state g' such that $g_0 \text{—}\gamma_1\gamma_2...\gamma_i \text{→}^* g'$, $1 \le i \le j$ and $s \in g'$. Since g' is simultaneously reachable, s is also simultaneously reachable. □

From the result of Theorem 9.7.1, we can formulate an automata-theoretic model checking method using simultaneous product as follows: Given a program P described as a set of processes P_i (i = 1,2,...,n) represented by finite automata on finite words and a safety property f represented by prefix-closed finite automaton on finite words A_f, check whether P satisfies f by using the following steps:

1. Build the automaton $A_{\neg f}$ corresponding to the complement of A_f. Since A_f is prefix-closed, $A_{\neg f}$ is naturally an automaton with only one accepting state (denoted X).

2. Check if the local sate X is sequentially reachable in the concurrent system obtained by taking the simultaneous product of processes in the program and $A_{\neg f}$: $P_1 \oplus P_2 \oplus ... \oplus P_n \oplus A_{\neg f}$.

Chapter 10
CONCLUDING REMARKS FOR PART II

A new product called simultaneous product is described for automata-theoretic model-checking. When we take the simultaneous product of the processes of a concurrent program, we obtain a subset of the joint behavior of the processes, or in other words, the reduced state space of the program. The reduced state space is represented by an automaton called simultaneous automaton which is defined over a set of sets of actions (rather than a set of actions) where the set of actions corresponds to the simultaneous execution of these actions. It is proven that every reachable deadlock state and local state of the program are also reachable in the reduced state space obtained by simultaneous product. It is shown that the local reachability preserving property of the simultaneous product leads to an on-the-fly model-checking method for verifying any safety property, which is performed by taking simultaneous product of the processes of the program and the process representing the negation of the property.

In the simultaneous product of processes of a concurrent program, the global states are generated by simultaneously executing several (global) transitions, i.e., by allowing several synchronizations to occur in the concurrent program at a time. When several transitions are simultaneously executed, the interleaving sequences of these transitions are not generated, that is, the states that are supposed to be generated during the execution of these sequences are not generated. Therefore, replacing single (or sequential) executions by simultaneous executions reduces the number of states to be generated.

The proposed method has some advantages over the other methods. When compared to partial order methods, it has a better best case space complexity. When k transitions are simultaneously executed, at the best case the simultaneous product generates one global state while the partial order methods generate one interleaving of these transitions, i.e., k global states. Both methods do not generate more global states than the conventional method does, thus their worst cases are equal. The partial order methods proposed for deadlock detection has ignoring problem, that is, they may ignore the execution of some transitions in a concurrent program. It is well-known that the verification of any safety property can be reduced to checking the sequential reachability of a local state. The partial order methods are not sufficient to check the sequential reachability of a local state due to their ignoring problem. Therefore, they require to be augmented to provide local reachability. On the other hand, the local reachability preserving property is another advantage of simultaneous product.

Simultaneous product and optimal simulation [JaKo90a] use the same idea to reduce the state space. Optimal simulation aims to obtain the smallest possible state space required for verification while simultaneous product does not. An efficient algorithm is described for simultaneous product but the existence of an efficient algorithm for optimal simulation is still an open problem.

Most of the reduction methods applied to sequential automata can also be applied to simultaneous automata. For example Holzmann's supertrace and state space caching algorithms can be directly combined with simultaneous automaton. Even the partial order methods can be applied to simultaneous automaton to obtain further reduction.

The verification of general properties can be reduced to checking reachable cycles containing specific local states [VaWo86, Wolp89, JaJe91, CoVa92]. In [Pele94], Peled claims that it is often the case that a property is insensitive to reordering some of the concurrent transitions of the program. Partial order methods have been already extended for

the verification of such properties [GoWo94, Valm93, Pele94]. We claim that the extension of simultaneous product for the verification of such properties is also possible but require more research.

Chapter 11
CONCLUDING REMARKS FOR THESIS

When a concurrent system allows concurrent (or independent) execution of transitions, the conventional verification method interleaves them in many possible orders. Any algorithm which is based on exhaustively generating all possible interleavings of concurrent transitions requires a large number system states to be stored in the memory of a computer. This is called state explosion problem to which concurrency is known as the major contributor.

In order to avoid state explosion, this thesis intensively studies an alternative approach which is based on simultaneous execution of transitions. When a concurrent system allows concurrent execution of transitions, the approach simultaneously executes them rather than generating all possible interleavings, therefore, eliminates the state explosion due to concurrency. The idea of simultaneous execution is successfully formulated for verification of the logical correctness of protocols and the safety properties of concurrent programs in this thesis. Through the thesis, protocols are represented as set of finite state machines communicating over simplex, error free and bounded FIFO queues while concurrent programs are represented as a set of finite automata communicating by executing common actions. Even though both protocol and concurrent program verification methods proposed in this thesis are based on the idea of simultaneous execution, the differences between their models, backgrounds and applications require them to be studied separately. Therefore,

this thesis is presented in two parts. The reader is referred to the separate conclusion chapters (chapter 6 and chapter 10).

REFERENCES

[AlSc85] B. Alpern and F. B. Schneider, "Defining liveness", *Information Processing Letters*, vol. 21, pp. 181-185, 1985.

[AlSc87] B. Alpern and F. B. Schneider, "Recognizing safety and liveness", *Distributed Computing*, vol. 2, pp. 117-126, 1987.

[BaSc69] K. A. Bartlett, R. A. Scantlebury and P. T. Wilkonson, "A note on reliable full-duplex transmission over half-duplex links", *Commun. ACM*, vol. 12, no. 5, pp. 260-261, 1969.

[Boch78] G. V. Bochmann, "Finite state description of communication protocols", *Computer Networks*, vol. 2, pp. 361-372, 1978.

[BrZa81] D. Brand and P. Zafiropulo ,"On communicating finite state machines", Technical Report RZ 1053, IBM Zurich Research Lab., Rüschlikon, Switzerland, 1981.

[BrZa83] D. Brand and P. Zafiropulo ,"On communicating finite state machines", *J. ACM*, vol. 30, no. 2, A, pp. 323-342, 1983.

[Büch62] J. R. Büchi, "On a decision method in restricted second order arithmetic", *Int. Congr. Logic, Method and Philos. Sci.*, pp. 1-12, 1962.

[ChMi83] T. Y. Choi and R. E. Miller, "A decomposition method for the analysis and design of finite state protocols", *ACM SIGOMM'83*, pp. 167-176, 1983.

[ChMi86] T. Y. Choi and R. E. Miller, "Protocol analysis and synthesis by structured partitions", *Computer Networks and ISDN Systems*, vol. 11, pp. 367-381, 1986.

[ClEm81] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic", *Workshop on Logics of Programs, Lecture Notes in Computer Science*, vol. 131, pp. 52-71, 1981.

[ClEm86] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications", *ACM TOPLAS*, vol.8, no. 2, pp. 244-263, 1986.

[ClGr87] E. M. Clarke and O. Grumberg, "Research on automatic verification of finite-state concurrent systems", *Ann. Rev. Comput. Sci.*, vol . 2, pp. 269-290, 1987.

[CoVa92] C. Courcoubetis, M. Vardi, P. Wolper and M. Yannakakis, "Memory-efficient algorithms for the verification of temporal properties", *Formal Methods in System Design*, vol. 1, pp. 275-288, 1992.

[Emer90] E. A. Emerson, "Temporal and Modal Logic", *Handbook of Theoretical Computer Science*, pp. 997-1072, 1990.

[Fink88] A. Finkel, "A new class of analyzable cfsms with unbounded fifo channels", *Protocol Specification, Testing and Verification, VIII*, pp. 283-294, 1988.

[GoHa85] M. G. Gouda and J. Y. Han, "Protocol validation by fair progress state exploration", *Computer Networks*, vol. 9, pp. 353-61, 1985.

[GoHo92] P. Godefroid, G. J. Holzmann and D. Pirottin, "State space caching revisited", *4th Workshop on Computer Aided Verification*, pp. 178-191, 1992.

[GoWo93] P. Godefroid and P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties", *Formal Methods in System Design*, vol 2. pp. 149-164, 1993.

[GoWo94] P. Godefroid and P. Wolper, "A partial approach to model checking", *Information and Computation*, vol. 110, pp. 305-326, 1994.

[GoYu84] M. G. Gouda and Y. T. Yu , "Protocol validation by maximal progress state exploration", *IEEE Trans. on Commun.*, vol. COM-32, pp. 94-97, 1984.

[Haje78] J. Hajek, "Automatically verified data transfer protocols ", *4th Int. Conf. on Computer Communications*, pp. 749-756, 1978.

[Hoar69] C. A. R. Hoare, "An axiomatic basis for computer programming", *Commun. ACM*, vol. 12, no. 10, pp. 576-580, 1969.

[Hoar78] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666-677, Aug. 1978.

[HoGo92] G. J. Holzmann, P. Godefroid and D. Pirottin, "Coverage preserving reduction strategies for reachability analysis", *Protocol Specification, Testing and Verification, XII*, pp. 349-363, 1992.

[Holz85a] G. J. Holzmann, "Backward symbolic execution of protocols", *Protocol Specification, Testing and Verification, V*, pp. 19-30, 1985.

[Holz85b] G. J. Holzmann, "Tracing protocols", *AT&T Technical Journal*, vol. 64, no. 10, pp. 2413-2433, 1985.

[Holz87a] G. J. Holzmann, "On limits and possibilities of automated protocol analysis", *Protocol Specification, Testing and Verification, VII* ,pp. 339-344, 1987.

[Holz87b] G. J. Holzmann, "Automated protocol validation in Argos: assertion proving and scatter searching," *IEEE Trans. on Soft. Eng.*, vol. SE-13, no. 6, pp. 683-696, 1987.

[Holz88] G. J. Holzmann, "An improved protocol reachability analysis technique", *Software-Practice and Experience*, vol. 18, no. 2, pp. 137-161, 1988.

[Holz90] G. J. Holzmann, "Algorithms for automated protocol verification", *AT&T Technical Journal*, vol. 69, no. 1, pp. 32-44, 1990.

[Holz92] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[HuCr77] G. E. Hughes and M. J. Creswell, *An Introduction to Modal Logic*, London, Methuen, 1977.

[HuCh93] Y. C. Hung and G. H. Chen, "Reverse reachability analysis: a new technique for deadlock detection on communicating finite state machines", *Software-Practice and Experience*, vol. 23, no. 9, pp. 965-979, 1993.

[ItIc83] M. Itoh and H. Ichikawa, "Protocol verification using reduced reachability analysis", *The Trans. on the IECE of Japan*, vol. E66, no. 2, pp. 88-93, 1983.

[JaJe89] C. Jard and T. Jeron, "On-line model-checking for finite linear temporal logic specifications", *Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science*, vol. 407, pp. 189-196, 1989.

[JaJe91] C. Jard and T. Jeron, "Bounded-memory algorithms for verification On-the-fly", *3rd Workshop on Computer Aided Verification, Lecture Notes in Computer Science*, vol. 575, pp. 192-202, 1991.

[JaKo89] R. Janicki and M. Koutny, "Towards a theory of simulation for verification of concurrent systems", *Parallel Architectures and Languages Europe 1989, Lecture Notes in Computer Science*, vol. 366, pp. 73-88, 1989.

[JaKo90a] R. Janicki and M. Koutny, "On some implementation of optimal simulations", Technical Report-90-07, McMaster University, Hamilton, Ontario, 1990.

[JaKo90b] R. Janicki and M. Koutny, "Using optimal simulations to reduce reachability graphs", *2nd Workshop on Computer Aided Verification, Lecture Notes in Computer Science*, vol. 531, pp. 166-175, 1990.

[JaLa86] R. Janicki, P. E. Lauer, M. Koutny and R. Devillers, "Concurrent and maximally concurrent evolution of non-sequential systems", *Theoretical Computer Science*, vol. 43, pp. 213-238, 1986.

[KaPe92] S. Katz and D. Peled, "Verification of distributed programs using representative interleaving sequences", *Distributed Computing*, vol. 6, pp. 107-120, 1992.

[KaWa88] Y. Kakuda, Y. Wakahara, and M. Norigoe, "An acyclic expansion algorithm for protocol validation", *IEEE Trans. on Soft. Eng.*, vol. SE-14, no. 8, pp. 1059-1067, 1988.

[Lamp77] L. Lamport, "Proving the correctness of multiprocess programs", *IEEE Tran. Soft. Eng.*, vol SE-3, no. 2, pp. 125-143, 1977.

[Lamp80] L. Lamport, "'Sometime' is sometimes 'not never'", *7th Ann. ACM Symp. on Principles of Programming Languages*, pp. 174-185, 1980.

[Lamp83] L. Lamport, "What good is temporal logic?", *IFIP Congr. on Information Processing*, pp. 657-668, 1983.

[LiCh87] F. J. Lin, P.M. Chu. and M. T. Liu, "Protocol verification using reachability analysis: the state explosion problem and relief strategies", *Computer Communication Review*, vol. 17, no. 5, 1987.

[LiMi94] H. Liu and R.E. Miller, "Generalized fair reachability analysis for cyclic protocols: part 1", *Protocol Specification, Testing and Verification, XIV*, pp. 258-273, 1994

[LiPn84] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification", *12th Ann. ACM Symp. on Principles of Programming Languages*, pp. 97-107, 1984.

[MaSa87] N. F. Maxemchuk and K. Sabnani, "Probabilistic verification of communication protocols", *Protocol Specification, Testing and Verification, VII*, pp. 307-320, 1987.

[Mazu86] A. Mazurkiewicz, "Trace theory", *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, Lecture Notes in Computer Science*, vol. 255, pp. 279-323, 1986.

[Miln80] R. Milner, "A calculus for communicating systems", *Lecture Notes in Computer Science*, vol. 92, 1980.

[Pach87] J. Pachl, "Protocol description and analysis based on a state transition model with channel expression", *Protocol Specification, Testing and Verification, VII*, pp. 307-320, 1987.

[Pele93] D. Peled, "All from one, one for all: on model checking using representatives", *5th Workshop on Computer Aided Verification*, pp. 409-423, 1993.

[Pele94] D. Peled, "Combining partial order reductions with on-the-fly model checking", *6th Workshop on Computer Aided Verification*, pp. 377-390, 1994.

[PePu90] W. X. Peng and S. Purushothaman, "A unified approach to the deadlock detection problem in networks of communicating finite state machines", *2nd Workshop on Computer Aided Verification, Lecture Notes in Computer Science*, vol. 531, pp. 243-252, 1990.

[Pete77] J. L. Peterson, "Petri nets", *Comput. Serv.*, vol. 9, no. 3, pp. 233-251, 1977.

[Pnue77] A. Pnueli, "The temporal logic of programs", *18th Ann. IEEE Symp. on Foundations of Computer Science*, pp. 46-57, 1977.

[Pnue80] A. Pnueli, "The temporal semantic of concurrent programs", *Theoretical Computer Science*, vol. 13, pp. 45-60, 1980.

[Pnue86] A. Pnueli, "Application of temporal logic to the specification and verification of reactive systems: A survey of current trends", *Lecture Notes in Computer Science*, vol. 224, pp. 510-584, 1986.

[PrSa91] R. Probert and K. Saleh, "Synthesis of communicating protocols: survey and assessment", *IEEE Trans. on Computers*, vol. 40, no. 4, April 1991.

[Rudi88] H. Rudin, "Protocol engineering: a critical assessment." *Protocol Specification, Testing and Verification, VIII* ,pp. 3-16, 1988.

[RuWe82] J. Rubin and C. H. West, "An improved protocol validation technique", *Computer Networks*, vol. 6, Apr. 1982.

[Sajk85] M. Sajkowski, "Protocol verification techniques: status quo and perspectives", *Protocol Specification, Testing and Verification, IV, pp.* 697-720, 1985.

[Sidh82] D. P. Sidhu, "Protocol design rules", *Protocol Specification, Testing and Verification*, pp. 283-300, 1982.

[TaHo91] K. C. Tai, H. F. Ho and G. H. Chen, "Protocol validation using a pumping-based approach", COMPSAC'91, pp. 339-344, 1991.

[Valm91] A. Valmari, "Stubborn sets for reduced state space generation", *Advanced in Petri Nets 1990, Lecture Notes in Computer Science*, vol. 483, pp. 491-515, 1991.

[Valm92] A. Valmari, "A stubborn attack on state explosion", *Formal Methods in System Design*, vol. 1, pp. 297-322, 1992.

[Valm93] A. Valmari, "On-the-fly verification with stubborn sets", *5th Workshop on Computer Aided Verification, Lecture Notes in Computer Science*, vol. 697, pp. 397-408, 1993.

[Vard87] M. Y. Vardi, "Verification of concurrent programs: the automata-theoretic framework", *IEEE Symp. on Logic in Computer Science*, pp. 167-176, 1987.

[VaWo86] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification", *1th Symp. on Logic in Computer Science*, pp. 332-344, 1986.

[VuCo82] S. T. Vuong and D. D. Cowan, "A decomposition method for the validation of structured protocols", *INFOCOM*, pp. 209-220, 1982.

[West78a] C. H. West, "General technique for communications protocol validation", *IBM J. Res. Develop.*, vol. 22, no. 1, pp. 393-404, 1978.

[West78b] C. H. West, "An automated technique for communications protocols", *IEEE Trans. on Commun.*, vol. COM-26, no. 8, pp. 1271-1275, 1978.

[West86] C. H. West, "Protocol verification by random state exploration", *Protocol Specification, Testing and Verification, VI* ,pp. 233-242, 1986.

[WeZa78] C. H. West and P. Zafiropulo, "Automated validation of a communication protocol: the CCITT X.21 Recommendation", *IBM J. Res. Develop.*, vol. 22, no. 1, pp. 60-71, 1978.

[WoGo94] P. Wolper and P. Godefroid, "Partial-order methods for temporal verification", *6th Workshop on Computer Aided Verification,* pp. 233-246, 1994.

[Wolp89] P. Wolper, "On the relation of programs and computations to models of temporal logic", *Temporal Logic in Specification, Lecture Notes in Computer Science*, vol. 389, pp. 75-123, 1989.

[WoVa83] P. Wolper, M. Y. Vardi and A. P. Sistla, "Reasoning about infinite computation paths", *24th IEEE Symp. on Foundations of Computer Science*, pp. 185-194, 1983.

[Yuan88] M. C. Yuang, "Survey of protocol verification techniques based on finite state machine models", *NBS Computer Networking Symposium*, pp. 164-172, 1988.

[YuKe90] M. C. Yuang and A. Kershenbaum, "Parallel protocol verification: the two-phase algorithm," *Protocol Specification, Testing and Verification, IX* ,pp. 339-353, 1990.

[YuGo82] Y. T. Yu and M. G. Gouda, "Deadlock detection for a class of communicating finite state machines", *IEEE Trans. on Commun.*, vol. COM-30, no. 12, pp. 2514-2518, 1982.

[Zafi78] P. Zafiropulo, "Protocol validation by duologue-matrix analysis", *IEEE Trans. on Commun.*, vol. COM-26, no. 8, pp. 1187-1194, 1978.

[ZaTa88] Y. X. Zang, K. Takahashi, N. Shiratori and S. Noguchi, "An interactive synthesis algorithm using a global state transition graph", *IEEE Trans. on Soft. Eng.*, vol. 14, no. 3, pp. 394-404, 1988.

[ZhBo86] J. Zhao and G. V. Bochmann, "Reduced reachability analysis of communication protocols: A new approach", *Protocol Specification, Testing and Verification, VI*, pp. 243-254, 1986.