ABSTRACT

Yu Lei. Reachability Analysis and Testing of Asynchronous Message-Passing Programs. (Under the direction of Professor Kuo-Chung Tai.)

An asynchronous message-passing program consists of concurrent processes that interact with each other by the exchange of messages. Many network protocols and distributed applications are asynchronous message-passing programs. This dissertation investigates techniques to ensure correctness of asynchronous message-passing programs.

Reachability analysis has been a successful approach to verifying concurrent programs. Existing reachability analysis techniques for asynchronous message-passing programs assume causal communication, which means that messages sent to a destination are received in the order they are sent. In the first part of this dissertation, we propose a new reachability analysis approach, called blocking-based simultaneous reachability analysis (BSRA), for asynchronous message-passing programs based on any communication scheme. We describe an algorithm for generating BSRA-based reachability graphs and show that this algorithm guarantees the detection of deadlocks. Empirical results indicate that BSRA significantly reduces the number of states in reachability graphs.

The second part of this dissertation deals with a new concept of testing concurrent programs, namely reachability testing. Let P be an asynchronous message-passing program, and X an input of P. Assume that every execution of P with X terminates. Reachability testing of P with X is to execute, in a systematic manner, all possible synchronization sequences (or SYN-sequences) of P with X such that the correctness of P with X can be determined. The main challenge of reachability testing is to derive race variants of SYN-sequences. We develop a formal approach to computing race variants of SYN-sequences consisting of send and receive events. We describe an efficient reachability testing algorithm for asynchronous message-passing programs.

In the third part of this dissertation, we propose a new test generation strategy, called In-Parameter-Order (or IPO), for pairwise testing. Pairwise testing requires that for each pair of input parameters of a system, every combination of valid values of these two parameters be covered by at least one test case. For a system with two or more input parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continues to do so for

each additional parameter. We present IPO-based test generation algorithms. We describe an IPO-based test generation tool and show some empirical results.

Reachability Analysis and Testing of Asynchronous Message-Passing Programs

by

Yu Lei

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh

2002

APPROVED BY:

Prof. S. Purush Iyer	Prof. Arne A. Nilsson
Prof. Mladen A. Vouk	Prof. Kuo-Chung Tai Chair of Advisory Committee

To those who helped me.

BIOGRAPHY

Yu Lei was born in Gongan, Hubei, China. He received his Bachelor and Master degrees from Wuhan University in 1993 and Chinese Academy of Sciences in 1996 respectively. He worked as a member of technical staff in Fujitsu Network Communications, Inc. from 1998 to 2001. He is currently pursuing his Ph.D. degree in North Carolina State University. His research interests include software engineering, concurrent software systems, and computer networks.

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Dr. Kuo-Chung Tai, for his guidance and support. A majority of this work is finished when Dr. Tai is on medical leave. Without his tremendous effort and personal sacrifice would this work never come into existence. His view, principles, and methods of research are bound to shape my academic pursuit for years to come.

I would like to extend my sincere gratitude to Dr. Munindar Singh for his support in the first year of my study in this country. It has been a great pleasure to work with him.

I want to thank Dr. S. Purush Iyer, Dr. Arne Nilsson, and Dr. Mladen Vouk for their service on my advisory committee. I have always enjoyed occasional discussions with Dr. Iyer. His advice and encouragement is very much appreciated.

I owe everything to my parents. Their struggle to provide me with the best education is not to be forgotten. I am confident that this work would make them proud.

Finally, my wife is the one who has shared the brightness and darkness of my life. Her unconditional confidence in me has been a precious gift.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

An asynchronous message-passing program consists of concurrent processes that interact with each other by the exchange of messages. It uses non-blocking send and blocking receive operations to send and receive message. *Non-blocking send* means that a process performs a send operation, and then immediately proceeds without waiting for the message to arrive at the destination; *blocking receive* means that a process performs a receive operation, and then blocks until a message arrives. Many network protocols and distributed applications are asynchronous message-passing programs.

One important aspect of asynchronous message-passing programs is that their behavior is nondeterministic [62]. Given the same input, multiple executions of an asynchronous message-passing program may exercise different synchronization sequences, and may even produce different results. Such nondeterminacy makes it difficult to ensure the correctness of asynchronous message-passing programs. This dissertation investigates techniques to ensure the correctness of asynchronous message-passing programs.

1.1.1 Reachability analysis

Reachability analysis has been a successful approach to verifying concurrent programs. It involves deriving global states of a program to detect deadlocks and other types of faults. The major limitation of reachability analysis is the state exploration problem, which refers to the fact that in the worst case, the number of global states of a concurrent program is an exponential function of the number of processes in the program [58]. A number of reachability analysis approaches have been proposed to reduce the global state space during reachability analysis while preserving certain fault detection capabilities [20] [6].

Existing reachability analysis approaches for asynchronous message-passing programs assume causal communication, which means that messages sent to the same destination be received in the same order as they are sent. However, in a fully asynchronous environment, messages may be received out of order [62]. In the first part of this dissertation, we propose a new reachability analysis approach, called blocking-based simultaneous reachability analysis (or BSRA). BSRA can be applied to asynchronous message-passing programs based on any communication scheme. The main idea of BSRA is described as follows. From a global state g, processes are allowed to proceed simultaneously until each of them terminates or is ready to execute a receive operation. Global states reached by such executions from g are called next blocking points of g. For each next blocking point of g, possible matches between waiting messages and receive operations are performed to produce immediate BSRA-based successor states of g. The intermediate global states from g to each of g's immediate BSRA-based successors are not saved. We describe an algorithm for generating BSRA-based reachability graphs and show that this algorithm guarantees the detection of deadlocks. Our empirical results indicate that BSRA significantly reduces the number of states in reachability graphs.

1.1.2 Reachability testing

Reachability testing is a new concept of testing concurrent programs [33] [64]. Let P be a concurrent program and X an input of P. An execution of P with X can be characterized by a sequence of synchronization events, referred to as a synchronization sequence (or SYN-sequence). Reachability testing of P with X is to execute, in a systematic manner, all possible synchronization sequences of P with X. By doing so, the correctness of P with X can be determined.

We assumed that P does not use nondeterministic statements. This implies that the existence of race conditions during interprocess communication accounts for the only source of nondeterminism exhibited by P. We also assume that every execution of P with X terminates. This implies that P has a finite number of possible executions. A general description of reachability testing of P with X is provided below [33].

1. Perform nondeterministic testing of P with X to collect one SYN-sequence.

2. For a collected SYN-sequence, derive its race variants by changing the outcomes of race conditions in this SYN-sequence. (These race variants are prefixes of other SYN-sequences of P with X).

3. For each new race variant derived in Step 2, force a deterministic execution of P with X according to this race variant and then let P continue execution nondeterministically to collect one SYN-sequence. (This process is referred to as prefix-based testing.)

4. For each new SYN-sequence collected in Step 3, repeat Steps 2, 3, and 4.

The main challenge of reachability testing of P with X is how to derive the race variants of an SYN-sequence. Let Q be an SYN-sequence exercised by one execution of P with X. A race variant of Q can be obtained by modifying the outcome of one or more race conditions in Q and is guaranteed to be exercised by at least one execution of P regardless of how P is implemented.

Because the details of the above steps depend upon P's concurrent programming constructs, the above description of reachability testing is abstract. In the second part of this dissertation, we present an efficient algorithm for reachability testing of asynchronous message-passing programs. An execution of an asynchronous message-passing program can be characterized by a sequence of send and receive events, which is referred to as an SR-sequence. We formally define an SR-sequence and its race variants. We develop an approach to systematically deriving the race variants of an SR-sequence.

1.1.3 Pairwise testing

Many system faults are caused by unexpected interactions between system parameters [9] [74]. For the purpose of fault discovery, it is desirable to test as many parameter interactions as possible. However, the number of possible parameter interactions grows exponentially in the number of parameters. It is often the case that due to limited resources, we cannot test all possible parameter interactions. Much research has been devoted to reduce test cost while maintaining a well-defined level of test coverage.

Pairwise testing is a specification-based testing criterion, which requires that for each pair of input parameters of a system, every combination of valid values of these two parameters be covered by at least one test case. Empirical results show that pairwise testing is practical and effective for various types of software systems [9] [13] [10]. In the third part of this dissertation, we propose a new test generation strategy, called In-Parameter-Order (or IPO), for pairwise testing. For a system with two or more input parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continues to do so for each additional parameter. The IPO strategy allows the use of local optimization techniques for test generation and the reuse of existing tests when a system is extended with new parameters or new values of existing parameters. We present practical, IPO-based test generation algorithms. We describe the implementation of an IPO-based test generation tool and show some empirical results.

1.2 Contributions

The first contribution is that we have proposed a new reachability analysis approach, that is the BSRA approach. Existing reachability analysis approaches for asynchronous message-passing programs assume causal communication, which means that messages sent to a destination are received in the order they are sent. In contrast, BSRA can be applied to asynchronous message-passing programs based on any communication scheme. We have developed an algorithm for generating BSRA-based reachability graphs and have formally established that this algorithm guarantees the detection of deadlocks. In order to evaluate the effectiveness of the BSRA approach, we have developed a reachability analysis tool based on BSRA, and have carried out an empirical study on a number of commonly used distributed algorithms. Our empirical results indicate that BSRA can significantly reduce the number of states in reachability graphs.

The second contribution is that we have developed an efficient algorithm for reachability testing of asynchronous message-passing programs. The main challenge of reachability testing of asynchronous message-passing programs is how to derive race variants of an SR-sequence. We have developed an approach to systematically deriving race variants of an SR-sequence, which is more efficient than that in [64]. Let P be an asynchronous message-passing program and X an input of P. Let Q be an SR-sequence exercised by an execution of P with X. When deriving a race variant of Q, [64] only allows the outcome of a single race condition in Q to change. In order to change the outcomes of more than one race condition in Q, [64] derives a race variant v of Q by changing the outcome of the first race condition (that needs to be changed), performs prefix-based testing according to v (i.e. forces a deterministic execution of P according to v and then let P proceed nondeterministically) to collect a new SR-sequence Q', and derives a race variant v' of Q' by changing the outcome of the second race condition (that needs to be changed), and continues to do so until the outcomes of all the race conditions (that needs to be changed) are changed. In contrast, our approach allows the outcomes of multiple race conditions to change simultaneously. By doing so, a large number of program executions can be saved. In addition, our reachability testing algorithm organizes collected SR-sequences, as well as their race variants, into a special tree structure such that they can be searched efficiently. Such a tree structure does not take much space, and can be used for regression testing, analysis of synchronization patterns, checking of assertions on sequencing of events, and other purposes.

The third contribution is that we have proposed a new test generation strategy, that is, the IPO strategy, for pairwise testing. The IPO strategy allows the use of local optimization techniques for test generation and the reuse of existing tests when a system is extended with new parameters or new values. Based on the IPO strategy, we have developed optimal and heuristic test generation algorithms. The optimal algorithms carry more theoretical significance, and the heuristic algorithms are easy to be implemented for practical use. In order to evaluate the effectiveness of the IPO strategy, we have built a test generation tool, called PairTest, which implements the heuristic algorithms. We have carried out an empirical study by generating pairwise test sets for a number of system configurations using PairTest. In comparison with a commercial (and costly) pairwise test generation tool, namely AETG [9], PairTest produces competitive results with the core algorithm having a lower asymptotic time complexity.

1.3 Organization

The remainder of this dissertation is organized as follows. The next chapter is devoted to the BSRA approach. We introduce the multi-port EFSM model. We present an algorithm to construct BSRA-based reachability graphs and formally establish that this algorithm guarantees the detection of deadlocks. We show empirical results of applying BSRA to a number of commonly used distributed algorithms. In Chapter 3, we discuss the topic of reachability testing. We provide a formal definition of race variant, and develop a formal approach to computing race variants of SYN-sequences consisting of send and receive events.

We present an efficient algorithm for reachability testing of asynchronous message-passing programs. In Chapter 4, we discuss the IPO strategy for pairwise testing. We present optimal and heuristic IPO-based test generation algorithms. We describe an IPO-based pairwise test generation tool, called PairTest. We show empirical results of applying IPO strategy to a number of system configurations. Chapter 5 concludes this dissertation and provides remarks about the future work.

Chapter 2

Blocking-based Simultaneous Reachability Analysis

2.1 Background

An asynchronous message-passing program consists of concurrent processes that interact with each other by the exchange of messages. It uses non-blocking send and blocking receive operations to send and receive messages. A non-blocking send operation allows the executing process to continue execution, and a blocking receive operation blocks the executing process until it receives a message. Many network protocols and distributed algorithms are asynchronous message-passing programs.

The communication scheme of an asynchronous message-passing program may impose certain restrictions on the relationship between the order in which messages are sent and the order in which messages are received. In [62], the following three communication schemes are identified:

- Asynchronous communication, which does not impose any restriction on the relationship between the order in which messages are sent and the order in which messages are received.

- First-In-First-Out (or FIFO) communication, which requires that messages sent from the same process to the same destination be received in the same order as they are sent. FIFO communication is a special type of asynchronous communication.

- Causal communication, which requires that messages sent to the same destination be received in the same order as they are sent. Causal communication is a special type of FIFO communication.

Various properties of asynchronous message-passing programs can be verified using reachability analysis. Let P be an asynchronous message-passing program. Reachability analysis of P involves systematically generating global states of P and verifying the satisfaction of specified properties by each global state. Since the number of global states of a concurrent program is an exponential function of the number of processes in the program [58], how to reduce the state space in reachability analysis while preserving some fault detection capabilities has been investigated for a long time [60] [25] [44] [20] [55] [6] [35].

Existing reachability analysis techniques for asynchronous message-passing programs assume causal communication. In this chapter, we propose a new reachability analysis approach, called blocking-based simultaneous reachability analysis (BSRA). BSRA can be applied to asynchronous message-passing programs based on any communication scheme. The next section discusses related work. Section 2.3 introduces the multi-port EFSM model, and discusses how to construct full reachability graphs of asynchronous message-passing programs. Section 2.4 presents the synchronization-based simultaneous reachability analysis (BSRA) approach, and proves that BSRA is capable of detecting every deadlock of an asynchronous message-passing program. Section 2.5 shows empirical results of applying BSRA to a number of commonly used distributed algorithms. Section 2.6 summaries this chapter.

2.2 Related work

2.2.1 Fair reachability analysis

In [60], Rubin and West proposed a state exploration technique to improve reachability analysis of communication protocols consisting of two processes. Their technique is based on the concept of canonical sequence. The canonical form of a sequence of execution steps of a two-process protocol can be obtained by choosing the execution steps in a pairwise manner such that each pair contains one execution step from each of the two processes. Within each pair, the execution step from one process is always chosen first, unless that step is a receive operation from an empty queue. In the latter case, the execution step from the other process is chosen first. Two execution sequences are canonically equivalent if they have the same canonical form. It is shown that the canonical equivalent of any executable sequence is an executable sequence. In order to incorporate the concept of canonical sequences into global state exploration algorithms, it is necessary to divide the protocol execution into a series of basic interactions. An interaction is a sequence of execution steps that leads the protocol from one stable state to the next, where a stable state is defined as one in which all message queues are empty. A basic interaction is an interaction that can not be divided into simpler interactions without changing the sequence of execution steps of the two processes. Their state exploration algorithm determines the basic interactions by coupling pairs of transitions in the two processes, executes each basic interaction in its canonical form, and considers intermediate states with a non-empty transmission medium when necessary. Executing only canonical sequences reduces the number of states analyzed, since only those states in which equal number of messages underway in both directions are generated. Rubin and West argued that their algorithm is capable of detecting every deadlock and unspecified receptions in two-process protocols.

The above work is extended by Gouda and Han to determine whether a two-process protocol is bounded [25]. They observed that the execution of a canonical sequence forces the two processes to progress at the same speed, and therefore, they used the name of fair reachability analysis for the state exploration technique in [60]. The reachability graph constructed by fair reachability analysis is referred to as fair reachability graph. In order to determine if a two-process protocol is bounded, the fair reachability graph is constructed and then augmented such that the augmented graph can be used to determine the node reachability of the protocol. Note that the processes are modeled as CFSMs, and a node is a local state of a finite state machine. They proved that a two-process protocol is unbounded if and only if there exists a reachable node that is in a directed cycle of all sending edges.

Liu and Miller generalized the fair reachability analysis technique to n-process cyclic protocols [44]. The processes of a cyclic protocol are connected by a unidirectional ring. They proved that for a cyclic protocol whose fair reachable state space is finite, deadlock detection is decidable with the fair reachability graph, and detection of other logic errors, such as unspecified reception, unboundedness, and non-executable transition are decidable via finite extension of the fair reachability graph.

2.2.2 Maximal progress state exploration

Gouda and Yu developed another technique of state exploration, called maximal progress state exploration, to detect every non-progress state of protocols consisting of two communicating finite state machines (or CFSMs) in [26] [27]. Let P be a protocol consisting of two CFSMs M_1 and M_2. In order to detect every non-progress state of P, maximal progress state exploration constructs two reachability graphs: one for M_1 and the other for M_2. When it constructs the reachability graph for M_1 (or M_2), M_1 (or M_2) is forced to make maximal progress in the sense that it always tries to execute transitions of M_1 if possible. Transitions of M_2 (or M_1) are executed only when M_1 can no longer make progress on its own. P has a non-progress state if and only if a non-progress state is reached during the construction of the reachability graph for M_1 and/or for M_2. Gouda and Yu proved that maximal progress state exploration is capable of detecting every deadlock state.

As mentioned earlier, Rubin and West [60] developed a technique to improve reachability analysis by forcing two machines to progress in equal speeds. However, their technique can not be used to detect overflows, and therefore can not detect non-progress states [26]. This has been the original motivation behind the development of maximal progress state exploration. It is worth to note that the work in [25] is indeed the outcome of applying the idea of maximal progress to the technique in [60].

2.2.3 Partial-order reduction

Partial-order reduction is based on the idea that exploring all the interleavings of concurrent events is not *a priori* necessary for verification [20]. Let P be a concurrent program consisting of processes P_1, P_2, $\ldots$, and P_n, where $n > 1$. Let G be a global state of P containing local states l_1, l_2, $\ldots$, and l_n, where l_i is the local state of P_i. Assume that there are n transitions t_1, t_2, $\ldots$, and t_n enabled at G, where t_i is a transition enabled at l_i. If all the interleavings ($n!$) of these n transitions reach the same state s, then considering only one of these $n!$ interleavings is sufficient to determine if s is reachable. Partial-order reduction avoids the part of combinatorial explosion due to the modeling of concurrency by interleaving.

In another perspective, partial-order reduction performs a selective search to construct a reduced reachability graph that is sufficient to verify one or more specified properties. A selective search operates as a classical state-space search except that, at each state s

reached during the search, it computes a subset T of the set of transitions that are enabled in s, and explores only the transitions in T [20]. The first technique of computing the set of transitions to explore in a selective search is called the persistent set technique [17] [20] [24]. A persistent set T is a set of transitions enabled in a state s if all transitions not in T that are enabled in s, or in a state reachable from s through transitions not in T, are independent with all transitions in T. In order to compute persistent sets, it is desirable to be able to detect independence between transitions [17]. Informally, two transitions are independent if their execution does not interact with or affect each other. In [17], it was shown that refining dependencies among transitions improves effectiveness of partial-order reduction. Note that the persistent set technique actually corresponds to a family of algorithms that were proposed independently by several researchers [18] [23] [70] .

The second technique of computing the set of transitions to consider in a selective search is called the sleep set technique [24] [20]. A sleep set associated with a state s contains a set of transitions that are enabled in s but will not be explored in s in a selective search. Used in conjunction with the persistent set technique, the use of sleep sets can further reduce the number of states explored by avoiding unnecessary exploration of multiple interleavings of independent transitions. Note that a sleep set is computed based on information about the past of the search, which is in contrast with the fact that a persistent set is inferred from the static structure of a program.

2.2.4 Simultaneous reachability analysis

Ozdemir and Ural [55] developed a relief strategy, called simultaneous reachability analysis (or SRA), for validating protocols specified as communicating finite state machines (or CFSMs). SRA is based on the idea of the execution of simultaneously executable sets of transitions at every simultaneously reachable global state of a protocol. A simultaneously executable set at a global state G is a non-empty set of transitions executable at G that contains at most one transition from each process. It is shown that every linearization of a simultaneously executable set (i.e. putting all the transitions in the set into a sequence in an arbitrary order) of a global state is an executable transition sequence from that global state and leads to the same global state. A simultaneously executable set can therefore be executed simultaneously in that the execution order of transitions is not significant. In a simultaneous reachability graph (i.e. the reachability graph constructed by SRA), each edge is associated with a simultaneously executable set, and each node is a simultaneously reachable global state. The edge generation for a simultaneously reachable state G consists of two steps. The first step is to select maximal simultaneously executable sets at G. A simultaneously executable set is maximal if it is not a proper subset of another simultaneously executable set. The second step has something to do with potentially executable transitions. A transition t is a potentially executable transition at G if t is not executable at G but may become executable at a success state reachable from G. For every selected maximal simultaneously executable set M, SRA checks if M contains any executable transition from a process which also has a potentially executable transition at G. If so, letting t' be such an executable transition, SRA selects another simultaneously executable set which includes the same set of transitions in M except t'. Ozdemir and Ural proved that SRA identifies every deadlock, every non-executable transition, every missing receiving transition causing an unspecified reception, and every channel at which a buffer overflow occurs.

Schoot and Ural developed a technique, called leaping reachability analysis (LRA), to improve the efficiency of SRA [71] [72]. The main idea of LRA is to refrain the processes that have both executable and potentially executable transitions from progressing. In case that every process has either no executable transitions or at least one potentially executable transitions, a separate edge is generated for every executable transition at G. By doing so, LRA yields extra reduction over SRA in terms of the space and time requirement. It is proved that LRA maintains the power of SRA to detect all deadlocks, all non-executable transitions, all unspecified receptions, and all buffer flows in a protocol.

Bengi and Tai further generalized the SRA approach to the multi-port extended finite state machine (EFSM) model [35] [36] [38]. The EFSM model is more expressive than the CFSM model in that the former allows that each process may contain a number of ports to receive different types of messages. SRA is first generalized for deadlock detection of a system of multi-port EFSMs [35]. The use of ports in the multi-port EFSM model gives rise to the following two problems. First, two process-independent transitions enabled at a global state G can not be executed simultaneously if they send messages to the same port. The notion of dependency needs to be modified such that two enabled transitions at a global state G are independent if and only if they are process-independent and port-independent. Second, an enabled transition t of process P_i at a global state G is said to have a race with a transition t' of process P_j ($j \neq i$) at a global state G', where G' is reachable from G, if t and t' send messages to the same port. In consideration that the order in which messages sent by t and t' may impact the system behavior, it is necessary to generate paths in which t occurs before t' and vice versa. However, t' is not enabled at G. Therefore, in order to generate paths in which t' occurs before t, the execution of t needs to be delayed until t' gets a chance to be executed. In another word, one edge needs to be generated at G such that t is not executed at G. Bengi and Tai proved that SRA with the above extension is capable of detecting every deadlock of a system of EFSMs.

SRA is later generalized for model checking of a system of EFSMs [38]. Model checking involves systematically generating all reachable states and checking them against properties specified in temporal logic formulae. Generally speaking, the simultaneous reachability graph (SRG) constructed for deadlock detection is insufficient for model checking. The notion of visible transitions is used to ensure the construction of an SRG that retains sufficient information for model checking. A transition is visible with respect to a temporal logic formula if its execution changes the values of some variables in the formula, and otherwise invisible. In order to generate all possible interleavings of visible transitions, the edge generation of SRA is modified such that each edge contains at most one visible transition.

2.3 Preliminaries

2.3.1 The multi-port EFSM model

An extended finite state machine (or EFSM) is a finite state machine in which each transition is associated with a predicate defined over a set of variables [35] [38]. A multi-port EFSM contains a set of ports for receiving different types of messages. Each port can receive messages sent from any process. It is assumed that the capacity of a port is large enough to prevent the buffer overflow problem.

A multi-port EFSM M can be formally defined as a tuple $M = (Q, q_0, V, T, I, O, F, \delta)$ (adapted from the definition of EFSM in [36]):

- Q: Set of states of M.
- q_0: Initial state of M, $q_0 \in Q$.
- V: Set of local variables of M.
- T: Set of ports of M: $\{T_1, T_2, \ldots, T_n\}$.

- I: Set of input messages for all ports of M.
- O: Set of output messages of M.
- F: Set of final states of M, $F \subseteq Q$.
- δ: Set of transitions of M.

The implementation of a port depends on the communication scheme used by M. For asynchronous communication, a port contains a pool in which messages are unordered. For FIFO communication, a port contains a separate message queue for each EFSM that communicates with M. For causal communication, a port maintains a single message queue for all the EFSMs that communicate with M.

A transition $t \in \delta$ contains the following information:

- $head(t)$: the start state of t, $head(t) \in Q$.
- $tail(t)$: the end state of t, $tail(t) \in Q$.
- $pred(t)$: a boolean expression involving variables in V, constants, and arithmetic, relational and boolean operations.
- $comp(t)$: a computation block, which is a sequence of computational statements (assignment, loop, etc) involving the received message, variables in V, and constants. The execution of $comp(t)$ terminates in a finite amount of time.
- A send operation $(!T_i.m)$ or a receive operation $(?T_i.m)$. In the former case, T_i is a port name of another EFSM, and m an output message in O. In the latter case, T_i is a port name of M, and m an input message in I. We also refer to T_i as $port(t)$ and m as $msg(t)$.

At most two of the following three components can be missing from a transition t: the predicate $pred(t)$, the computation block $comp(t)$, and the send/receive operation. M is also referred to as $proc(t)$.

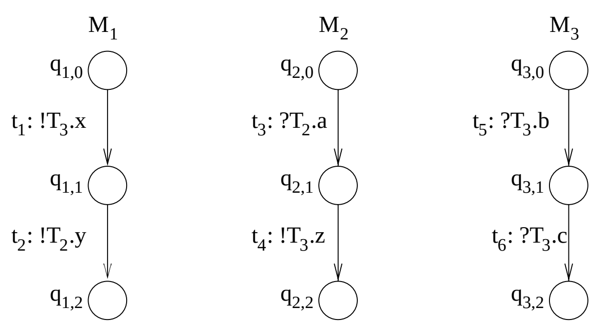
Let t be a transition of M. t is called an internal transition if t does not contain a send or receive operation, t is called a send transition if t contains a send operation, and t is called a receive transition if t contains a receive operation. When $head(t)$ is the current state of M, we can determine whether t is executable (or enabled) at $head(t)$ as follows. If

t is a send or internal transition, t is enabled at $head(t)$ provided that $pred(t)$ is not present or $pred(t)$ is $true$. If t is a receive transition, t is open at $head(t)$ provided that $pred(t)$ is not present or $pred(t)$ is $true$, and t is enabled at $head(t)$ provided that t is open at $head(t)$ and $port(t)$ is not empty.

The semantics of the execution of transition t can be described as follows. If t is an internal transition, the execution of t consists of the following actions in the given order: (1) $comp(t)$ is executed; and (2) $tail(t)$ becomes the current state of M. If t is a send transition, the execution of t consists of the following actions in the given order: (1) $msg(t)$ is saved at $port(t)$; (2) $comp(t)$ is executed; and (3) $tail(t)$ becomes the current state of M. If t is a receive transition, the execution of t consists of the following actions in the given order: (1) a message m is removed from $port(t)$; (2) $comp(t)$ is executed; and (3) $tail(t)$ becomes the current state of M.

We comment that our EFSM model is different from that in [35] [38]. In the latter model, a port contains a single message queue and a single transition may contain both send and receive operations. Figure 2.1 shows an example EFSM system.

Figure 2.1: An example EFSM system



2.3.2 Reachability analysis of asynchronous message-passing programs

Reachability analysis of a concurrent program produces a reachability graph that consists of reachable states of the program. In this section, we discuss how to construct the full reachability graph of an asynchronous message-passing programs.

Definition 1 *Let P be a system of EFSMs. A global state g of P contains the local state, the values of local variables, and the contents of the ports for each EFSM in P. The initial* *global state of P contains the initial local state, the initial values of local variables and empty ports for each EFSM in P. A final global state of P contains a final local state and empty ports for each EFSM in P.*

Definition 2 *Let g be a global state of a system P of EFSMs. g is said to be a deadlock state if g is not a final global state and g has no enabled transitions.*

Definition 3 *Let P be a system of EFSMs M_1, M_2, ..., and M_n. Let g be a global state of P. Let T be a port of an EFSM M_i, where $1 \leq i \leq n$. The frontier of T at g, denoted as $frontier(T, g)$, is the set of messages in T that can be received by an open receive transition r at g with $port(r) = T$ according to P's communication scheme.*

The computation of $frontier(T, g)$ can be described as follows. For asynchronous communication, $frontier(T, g)$ contains all messages in T at g. For FIFO communication, $frontier(T, g)$ contains the first message of each non-empty queue of T at g. For causal communication, $frontier(T, g)$ contains the first message of the only queue of T at g.

A transition between two local states is referred to as a local transition. A transition between two global states is referred to as a global transition. In our EFSM model, a local transition may involve the use of variables. A global transition can be considered as an instance of a local transition whose variables are bound according to a global state.

If necessary, we distinguish a global transition from a local transition as follows. Let t be a local transition. Let g be a global state. If t is a send transition, $[t]_g$, or $[t]$ if g is implied, denotes a global send transition at g that is an instance of t whose variables are bound according to g. If t is a receive transition, $[t]_{g,m}$, or $[t]_m$ if g is implied, where $m \in frontier(port(r), g)$, denotes a global receive transition at g that is an instance of t whose variables are bound according to g and whose execution removes m from $frontier(port(r), g)$.

Definition 4 *Let P be a system of $EFSMs$. Let g be a global state of P, and t a global transition at g. The global state g' reached by the execution of t is said to be an immediate successor state of g, denoted as $g \xrightarrow{t}_P g'$, or $g \xrightarrow{t} g'$ if P is implied.*

Definition 5 *Let P be a system of EFSMs and g a global state of P. Let $\omega = t_1 t_2 \ldots t_n$ be a sequence of global transitions. ω is an enabled (or executable) transition sequence from g if there exists a sequence of states $g = g_0$, g_1, ..., and $g_n = g'$ such that $g_{i-1} \xrightarrow{t_i} g_i$, where $1 \leq i \leq n$. Formally, $g \xrightarrow{\omega} * g'$.*

Figure 2.2: Algorithm Generate_RG

Algorithm *Generate_RG*(P: A set of multi-port EFSMs)
RG: Reachability graph, $RG = (V, E)$, V: nodes, E: edges
open: Set of nodes to be explored

$V \leftarrow \emptyset$, $E \leftarrow \emptyset$
generate the initial global state g_0
put g_0 in *open*
while $open \neq \emptyset$ **do**
  $g \leftarrow$ the most recently added global state in *open*
  remove g from *open*
  if g is a deadlock state, report deadlock
  else
    determine the set of global transitions at g
    for each global transition t at g **do**
      execute t at g, and let g' be the successor state
      if $g' \notin V$
        $open \leftarrow open \cup \{g'\}$
        $V \leftarrow V \cup \{g'\}$
      $E \leftarrow E \cup \{t\}$
return RG;

Figure 2.2 shows an algorithm to construct the full reachability graph of an asynchronous message-passing program. The set of global transitions at g is determined as follows. Let t be a local transition at g. If t is an enabled send transition at g, we generate a global transition $[t]_g$ according to t. If t is an enabled receive transition at g, we generate a global transition $[t]_{g,m}$ for each message m in $frontier(port(t), g)$.

2.4 Blocking-based simultaneous reachability analysis

In this section, we describe a new reachability analysis approach, called blocking-based simultaneous reachability analysis (or BSRA), for detecting deadlocks of asynchronous message-passing programs.

Figure 2.3 shows the framework of BSRA, namely algorithm Generate_BSRG. In essence, algorithm Generate_BSRG(P) constructs a reduce reachability graph, called blocking-based

Figure 2.3: Algorithm Generate_BSRG

```
Algorithm Generate_BSRG(P: A set of multi-port EFSMs)
RG: Reachability graph, RG = (V, E), V: nodes, E: edges
open: Set of nodes to be explored

V ← ∅, E ← ∅
generate the initial global state g0
put g0 in open
while open ≠ ∅ do
  g ← the most recently added global state in open
  remove g from open
  if g is a deadlock state, report deadlock
  else
    g_edges ← Generate_BSRGEdges(g)
    for each edge e = (b, γ) ∈ g_edges
      restore b as the current global state of P
      if γ is not empty
        execute an arbitrary permutation of γ at b
      let g′ be the current global state of P, which is the
        successor state of g reached by edge e
      if g′ ∉ V
        open ← open ∪ {g′}
        V ← V ∪ {g′}
      E ← E ∪ {e}
return RG;
```

simultaneous reachability graph (or BSRG), for a system P of EFSMs. Let g be a global state of P. An edge e at g is represented as a tuple (b, γ), where b is a successor state of g, called a next blocking point of g, and γ is an independent global receive transition set at b. In order to determine the immediate BSRA-based successor state reached by e from g, we first restore b as the current state of P, and then we execute an arbitrary permutation of γ at b.

2.4.1 Basic concepts

Definition 6 *Let P be a system of EFSMs. A global state g of P is a blocking point of P if each EFSM either has at least one open receive transition or has no enabled transitions at g.*

Definition 7 *Let P be a system of EFSMs. Let g be a global state of P. A blocking point b of P is a next blocking point of g if $g \xrightarrow{\omega} * b$, where ω is an enabled transition sequence from g, $|\omega| \geq 0$, and ω consists of send and/or internal transitions only.*

If g is a blocking point, then g is a next blocking point of its own. In this case, $|\omega| = 0$. (The reason for considering a blocking point as a next blocking point of its own is to make algorithm Generate_BSRGEdges in Section 2.4.2 concise.) A global state g may have a set of next blocking points, which is referred to as *next_bp*(g). A transition sequence consisting of send and/or internal transitions only is referred to as a send/internal transition sequence.

Let us consider the example system in Figure 2.1. A global state g containing $q_{1,2}$, $q_{2,0}$, and $q_{3,0}$ is a blocking point, because M_1 has no enabled transitions at g, and M_2 and M_3 both have an open receive transition at g. Let g_0 be the initial global state. g is a next blocking point of g_0 because $g_0 \xrightarrow{t_1 t_2} * g$, where $t_1 t_2$ is an enabled send transition sequence from g_0.

Definition 8 *Let g be a global state of a system P of EFSMs. Two global transitions t_1 and t_2 of P are said to be independent wrt g if (1) t_1 and t_2 are enabled global transitions at g; and (2) $proc(t_1) \neq proc(t_2)$.*

Let L be a set of enabled global transitions at a global state g of a system of EFSMs. If transitions in L are pairwise independent wrt g, then L is said to be an independent transition set at g.

Assume that transitions t_1 and t_2 are independent wrt global state g. If $g \xrightarrow{t_1 t_2} * g'$, then $g \xrightarrow{t_2 t_1} * g'$. Thus, the state reached from g after the execution of t_1 and t_2 is independent of the execution order of t_1 and t_2. This property can be generalized to three or more independent transitions wrt a global state, as shown below.

Lemma 2.4.1 *Let g be a global state of a system P of EFSMs. Let L be an independent transition set at g. Let σ and ω be two permutations of L. If $g \xrightarrow{\sigma} * g'$, then $g \xrightarrow{\omega} * g'$.*

2.4.2 Algorithm Generate_BSRGEdges

Let P be a system of EFSMs M_1, M_2, ..., and M_n. Let g be a global state of P. The edge generation for g consists of the following two steps. The first step is to determine *next_bp*(g), the set of next blocking points of g. The second step is to generate independent global receive transition sets at each blocking point in *next_bp*(g). An outgoing edge e of g is a tuple (b, γ), where b is a next global blocking point of g, and γ is an independent global receive transition set at g.

In Section 2.3.1, it is mentioned that a port maintains a single message queue for causal communication. This is sufficient for algorithm Generate_RG in Figure 2.2. However, we point out that a single message queue per port for causal communication cannot support BSRA. Rather, a port needs to maintain a special type of data structure, called happen-before graph, such that messages in the port can be received correctly according to causal communication. We will discuss such happen-before graphs in details in the end of this section.

Before we elaborate algorithm Generate_BSRGEdges, we define the cross product, denoted as $S_1 \otimes S_2 \otimes \ldots \otimes S_n$, of sets S_1, S_2, ..., and S_n as follows: $S_1 \otimes S_2 \otimes \ldots \otimes S_n = \{\{e_1, e_2, \ldots, e_n\} | e_i \in S_i, where\ 1 \leq i \leq n\}$. Note that unlike a Cartesian product, which contains a set of (ordered) tuples, a cross product contains a set of (unordered) sets.

Step 1 of algorithm Generate_BSRGEdges(g)

This step is to determine *next_bp*(g), the set of next blocking points of g. First, we find a set π_i of enabled send/internal transition sequences of each EFSM M_i, $1 \leq i \leq n$, at g. Let ω be an enabled send/internal transition sequence of M_i from g. Let g' be a successor state of g such that $g \xrightarrow{\omega} * g'$. Let H be the sequence of global states from g to g' according to ω. If H has no cycles, and M_i has at least one open receive transition or has no enabled transitions at g', then ω is added into π_i. (If H has a cycle, then P may repeat the cycle forever. As a result, P either makes no progress, if all transitions in ω are internal transitions, or creates the buffer overflow problem, if one or more transitions in the cycle are send transitions.) In case that M_i has at least one open receive transition or has no enabled transitions at g, an empty transition sequence ϵ is also added into π_i. Next, we compute the cross product of these transition sequence sets, letting $\pi = \pi_1 \otimes \pi_2 \otimes \ldots \otimes \pi_n$. For each set $\{\omega_1, \omega_2, \ldots, \omega_n\} \in \pi$, any interleaving of ω_1, ω_2, ..., and ω_n can be executed to reach a next blocking point of g.

Let us consider how to compute $next\_bp(g_0)$ for the example system in Figure 2.1. Let π_i, $1 \leq i \leq 3$, denote the set of enabled send/internal transition sequences of M_i at g_0. Note that $\pi_1 = \{t_1t_2\}$, $\pi_2 = \{\epsilon\}$, and $\pi_3 = \{\epsilon\}$. Thus, $\pi = \{\{t_1t_2, \epsilon, \epsilon\}\}$. So, $next\_bp(g_0)$ contains a single blocking point $b_0 = (q_{1,2}, q_{2,0}, q_{3,0}, T_1 = \{\}, T_2 = \{y\}, T_3 = \{x\})$.

Step 2 of algorithm Generate_BSRGEdges(g)

This step is to generate independent global receive transition sets for each blocking point in $next\_bp(g)$. In order to maximize the state reduction in the resulting graph, it is desirable to generate maximal independent global receive transition sets at each blocking point. Let $b \in next\_bp(g)$. Let γ be an independent global receive transition set at b. γ is a maximal independent global receive transition set at b if there does not exist a global receive transition r at g such that $\gamma \cup \{r\}$ is also an independent global receive transition set at g.

Maximal independent global receive transition sets at b can be computed as follows. First, we determine the set of global receive transitions γ_i for each EFSM M_i, where $1 \leq i \leq n$. This is accomplished by adding into γ_i a global receive transition, denoted as $[r]_{b,m}$, for each open local receive transition r of M_i at b and each message $m \in frontier(port(r), b)$. Next, we compute the cross product Γ of those non-empty global receive transition sets. Each set in Γ is a maximal independent receive transition set at b.

However, generating only maximal independent global receive transition sets is insufficient. This is due to the existence of hereafter executable receive transitions.

Definition 9 *Let P be a system of EFSMs and g a global state of P. An open local receive transition r at g is a hereafter executable transition (HET) at g if there exists an executable transition sequence $t_1t_2 \dots t_m$, $m \geq 1$, from g such that for $1 \leq i \leq m$, $proc(t_i) \neq proc(r)$, t_m is a send transition with $port(t_m) = port(r)$, $g \overset{t_1t_2 \dots t_m}{\rightarrow} * g'$, and $msg(t_m) \in frontier(port(r), g')$.*

Let r an HET at b. Let $t_1t_2 \dots t_m$ be an executable transition sequence from b according to Definition 9. Let $b \overset{t_1t_2 \dots t_m}{\rightarrow} * g'$. Then, $[r]_{g',msg(t_m)}$ is a global receive transition at g'. It is possible that the execution of $[r]_{g',msg(t_m)}$ at g' leads to a deadlock state d of P. Assume that r is a local receive transition of M_i, $1 \leq i \leq n$. In order to detect d, the execution of M_i needs to be delayed until g' is reached. Therefore, given an independent global receive transition set γ at b, if γ contains a global receive transition r' of M_i, then another independent global receive transition set γ' needs to be generated such that γ' does not include r'.

To determine whether an open local receive transition is an HET at a global state, the entire system needs to be considered. In what follows, we identify a necessary condition for an open local receive transition to be an HET at a global state, and use this condition to handle HETs.

Definition 10 *Let P be a system of EFSMs and M an EFSM of P. Let l be a local state of M. Let T be a port of P. $reachable(l, T)$ is a boolean function defined as follows: $reachable(l, T)$ is true if and only if within M, l can reach a local send transition s such that $port(s) = T$.*

Note that the computation of function *reachable* does not require system-wide analysis, and solely depends on the syntactic structure of an EFSM.

Lemma 2.4.2 *Let P be a system of EFSMs M_1, M_2, $\ldots$, and M_n. Let g be a global state of P. Let r be an open local receive transition of M_i, $1 \leq i \leq n$, at g. r is an HET at g only if there exists an EFSM M_j, where $1 \leq j \leq n$ and $i \neq j$, such that letting l be the local state of M_j at g, $reachable(l, port(r)) = true$.*

Proof

Assume that r is an HET at g. Let $\omega = t_1 t_2 \ldots t_m$ be an executable transition sequence from g according to Definition 9. Let l be the local state of $proc(t_m)$ at g. Let σ be the transition sequence obtained by removing from ω all the transitions that do not belong to $proc(t_m)$. Then, l reaches t_m via σ within $proc(t_m)$. Note that t_m is a send transition with $port(t_m) = port(r)$. Therefore, $reachable(l, port(r)) = true$. This concludes our proof.

$\square$

Let g be a global state. Lemma 2.4.2 states a necessary condition for an open local receive transition at g to be an HET at g. An open local receive transition satisfying this necessary condition at g is referred to as a potential HET at g.

In order to handle HETs at b, we modify the process of computing maximal independent global receive transition sets. Let γ_i be the set of global receive transitions of M_i at b. If

M_i has at least one potential HET at b, we add into γ_i a special transition ϵ. Then, we compute the cross product Γ of those non-empty global receive transition sets. Given an independent global receive transition set $\gamma \in \Gamma$, if γ contains only ϵ, then we ignore γ; otherwise, we delete ϵ from γ. By doing so, we implement that if an independent global receive transition set $\gamma \in \Gamma$ contains a global receive transition r from a process that has at least one potential HET at b, then $\gamma \setminus \{r\} \in \Gamma$.

For illustration, let us consider the example system in Figure 2.1 again. In Section 2.4.2, we have shown that g_0 has a next blocking point $b_0 = (q_{1,2}, q_{2,0}, q_{3,0}, T_1 = \{\}, T_2 = \{y\}, T_3 = \{x\})$. Note that $frontier(T_1, b) = \{\}$, $frontier(T_2, b) = \{y\}$, and $frontier(T_3, b) = \{x\}$. Thus, $\gamma_1 = \emptyset$, $\gamma_2 = \{[t_3]_{b,y}\}$, and $\gamma_3 = \{[t_5]_{b,x}\}$. Since $q_{2,0}$ can reach t_4, which is a send transition with $port(t_4) = T_3$, $reachable(q_{2,0}, T_3) = true$. This derives that $[t_5]_{b,x}$ is a potential HET at b. Thus, we add a special transition ϵ into γ_3, that is, $\gamma_3 = \{[t_5]_{b,x}, \epsilon\}$. Therefore, $\gamma_2 \otimes \gamma_3 = \{\{[t_3]_{b,y}\}, \{[t_3]_{b,y}, [t_5]_{b,x}\}\}$.

Figure 2.4 shows the complete algorithm to generate BSRG edges. Figure 2.5 shows two functions used by algorithm Generate_BSRGEdges: *gen_next_bp* and *gen_trans_sets*.

Figure 2.6 shows the application of algorithm Generate_BSRG to the example system in Figure 2.1, assuming that asynchronous communication is used. The left graph in Figure 2.6 contains blocking points (dashed nodes) generated during reachability analysis and the send/internal transition sequences (dashed edges) that reach those blocking points. The right graph in Figure 2.6 is the resulting BSRG graph, which does not contain dashed nodes and edges. Note that in these graphs, the path $(g_0 g_2 g_4 g_5)$ does not exist if causal communication is assumed.

Happen-before graph

As mentioned earlier, a single message queue per port for causal communication is insufficient to support BSRA. This is because in case of causal communication, more information is needed to compute the frontier of a port. Let P be an asynchronous message-passing program based on causal communication. Let T be a port of P, and g a global state of P. Since causal communication requires that messages sent to a destination be received in the same order as they are sent, the frontier of T at g, or $frontier(T, g)$, is a subset of messages in T such that for any message m in this subset, there exists no message m' in T such that m' happens before m. (Otherwise, m' happens before m but m is received before

Figure 2.4: Algorithm Generate_BSRGEdges

Algorithm Generate_BSRGEdges(g: a global state)

$g_{edges} \leftarrow \emptyset$

// 1: Generate the set of next blocking points of g
$next\_bp \leftarrow gen\_next\_bp(g)$

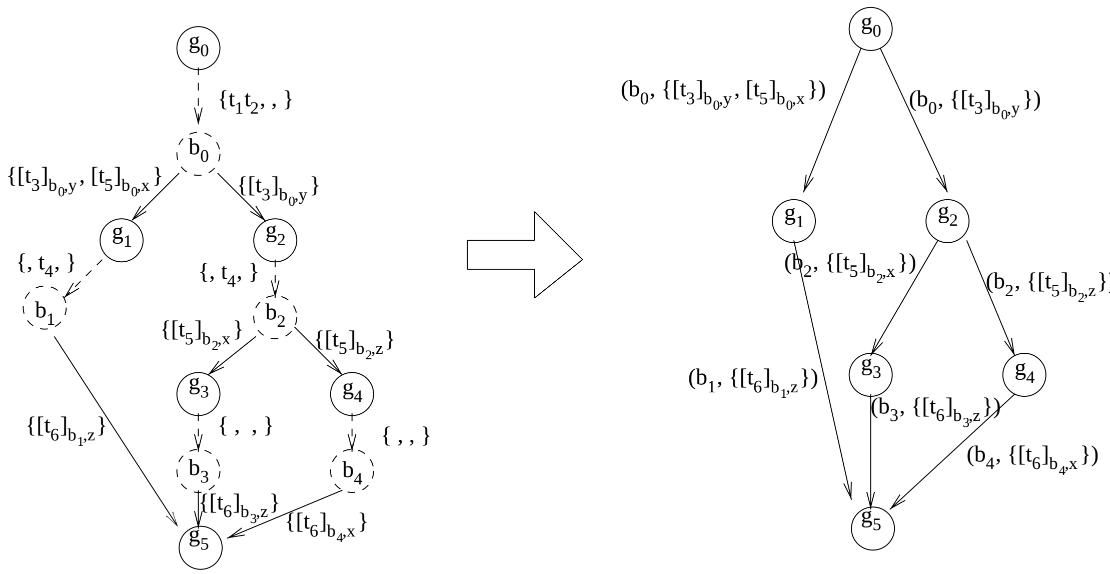
// 2: Generate edges according to each blocking point in *next-bp*
for each blocking point $b \in next\_bp$ **do**
  // 2.1: Generate independent receive transition sets at b
  $\Gamma \leftarrow gen\_trans\_sets(b)$
  // 2.2: Generate edges according to independent receive transition sets at b
  if Γ is empty
    // 2.2.1: Special case - check if b is a deadlock state
    if b has no enabled send or internal transitions
      generate a special edge $e = (b, \emptyset)$
      $g_{edges} \leftarrow g_{edges} \cup \{e\}$
    // otherwise, do nothing
  else
    // 2.2.2: Generate independent receive transition sets at b
    for each transition set γ in Γ **do**
      generate a new edge $e = (b, \gamma)$
      $g_{edges} \leftarrow g_{edges} \cup \{e\}$
return g_{edges}

Figure 2.5: Functions *gen_next_bp* and *gen_trans_sets*

```
Function gen_next_bp (g: a global state)
begin
  for 1 ≤ i ≤ n do
    if M_i either has at least one open receive transition or
      has no enabled transitions at g
      π_i ← {ε}
    else
      π_i ← ∅
    π_i ← π_i ∪ {ω|ω is a send/internal transition sequence of M_i such that
      g -ω->* g', the sequence of global states from g to g' according to ω
      has no cycles, and M_i at g' has at least one open receive transition
      or has no enabled transition}
  π ← π_1 × π_2 × ... × π_n
  next_bp ← ∅
  for τ = (ω_1, ω_2, ..., ω_n) ∈ π
    let b be the successor state reached by the execution of ω_1ω_2...ω_n from g
    next_bp ← next_bp ∪ {b}
  return next_bp
end


Function gen_trans_sets (b: a blocking point)
begin
  for 1 ≤ i ≤ n do
    γ_i ← ∅
    for each enabled receive transition r of M_i at b do
      γ_i ← γ_i ∪ {[r]_{b,m}|m ∈ frontier(port(r), b)}
    if M_i has at least one potential HET at b
      γ_i ← γ_i ∪ {ε}
  Γ ← γ_1 ⊗ γ_2 ⊗ ... ⊗ γ_n
  Γ ← Γ \ {ε}
  for each set γ ∈ Γ do
    γ ← γ \ {ε}
  return Γ
end
```

Figure 2.6: BSRG for the example system



g_0: $(q_{1,0}, q_{2,0}, q_{3,0}, T_1=\{\}, T_2=\{\}, T_3=\{\})$

b_0: $(q_{1,2}, q_{2,0}, q_{3,0}, T_1=\{\}, T_2=\{y\}, T_3=\{x\})$

g_1: $(q_{1,2}, q_{2,1}, q_{3,1}, T_1=\{\}, T_2=\{\}, T_3=\{\})$

b_1: $(q_{1,2}, q_{2,2}, q_{3,1}, T_1=\{\}, T_2=\{\}, T_3=\{z\})$

g_2: $(q_{1,2}, q_{2,1}, q_{3,0}, T_1=\{\}, T_2=\{\}, T_3=\{x\})$

b_2: $(q_{1,2}, q_{2,2}, q_{3,0}, T_1=\{\}, T_2=\{\}, T_3=\{x,z\})$

g_3: $(q_{1,2}, q_{2,2}, q_{3,1}, T_1=\{\}, T_2=\{\}, T_3=\{z\})$

b_3: $(q_{1,2}, q_{2,2}, q_{3,1}, T_1=\{\}, T_2=\{\}, T_3=\{z\})$

g_4: $(q_{1,2}, q_{2,2}, q_{3,1}, T_1=\{\}, T_2=\{\}, T_3=\{x\})$

b_4: $(q_{1,2}, q_{2,2}, q_{3,1}, T_1=\{\}, T_2=\{\}, T_3=\{x\})$

g_5: $(q_{1,2}, q_{2,2}, q_{3,2}, T_1=\{\}, T_2=\{\}, T_3=\{\})$

m', which would be inconsistent with causal communication.) Hence, in order to compute $frontier(T, g)$, we need to determine the happen-before relation among the messages in T.

In what follows, we describe a special type of data structure, called happen-before graph, in which messages in a port are stored according to their happen-before relation, and show how to compute the frontier of a port at a global state based on the happen-before graph maintained by the port.

Let P be an asynchronous message-passing program containing processes P_1, P_2, …, and P_n. Let T be a port of P_i, and let $hb\_graph(T)$ the happen-before graph maintained by T. $hb\_graph(T)$ contains $(n-1)$ queues $Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_n$ such that Q_j, $1 \le j \le n$ and $j \neq i$, contains the messages sent from P_j to P_i. Each queue Q_j is further divided into a sequence of sub-queues q_{j1}, q_{j2}, …, and q_{jk_j}, $k_j \ge 1$, where each sub-queue q_{jl}, $1 \le l \le k_j$, is a portion of Q_j such that $Q_j = q_{j1} \cdot q_{j2} \cdot \ldots \cdot q_{jk_j}$. Each sub-queue maintains a parent list and a child list. Every message in a parent sub-queue happens before any message in a child sub-queue.

In order to compute the frontier of T at a global state g, or $frontier(T, g)$, we check each queue Q_j, $1 \leq j \leq n$ and $j \neq i$, in $hb\_graph(T)$. If Q_j is not empty, and if the first sub-queue of Q_j (i.e. q_{j1}) has an empty parent list (i.e. q_{j1} has no parent sub-queues), then we add the first message of Q_j into $frontier(T, g)$.

The construction and maintenance of $hb\_graph(T)$ is described as follows. Initially, we create $hb\_graph(T)$ with all its queues being empty. It is natural to consider that an empty queue contains a single sub-queue which is also empty and has an empty parent and child list. Whenever a message m from P_j, $1 \leq j \leq n$ and $j \neq i$, arrives at T, m is added into Q_j. This also implies that m is added into the last sub-queue in Q_j. Whenever P_i receives (or removes) a message m from T at a global state (m must be in the frontier of T at the global state), we perform the following two tasks.

First, we check if q_{j1} becomes empty due to the removal of m (m must be in q_{j1}). If so, we find all the child sub-queues of q_{j1} based on the child list of q_{j1} and remove q_{j1} from the parent list of each child sub-queue. After that, we delete q_{j1} from Q_j.

Secondly, we need to update the happen-before graph of every port of any process other than P_i and P_j as follows. Let T' be a port of P_l, where $1 \leq l \leq n$, $l \neq i$ and $l \neq j$. Let Q'_i be the queue in T' that contains messages sent from P_i and Q'_j the queue in T' that contains messages sent from P_j. Let q'_{ik_i} be the last sub-queue in Q'_i and q'_{jk_j} the last sub-queue in Q'_j. We update $hb\_graph(T')$ as follows.

1. We divide q'_{jk_j} into two sub-queues, denoted as q'_u and q'_v, such that all the messages in q'_u are sent before m and all the messages in q'_v are sent after m.

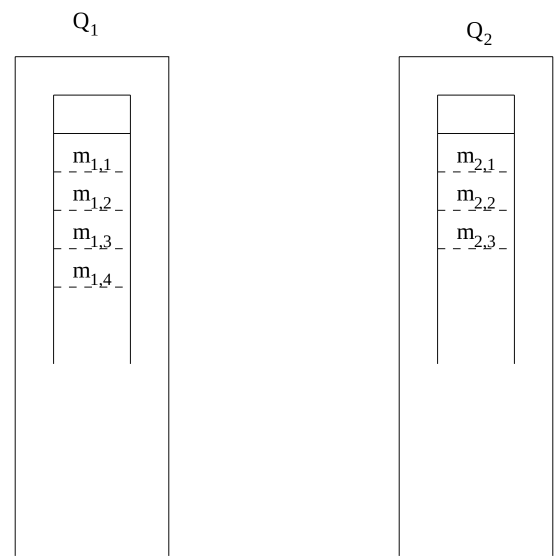
2. If q'_u is empty, then we terminate. Otherwise, we replace q'_{jk_j} in Q'_j with q'_u and q'_v. Note that we allow q'_v to be empty.

3. Let q'_w be q'_{ik_i} if q'_{ik_i} is empty. Otherwise, we create an empty sub-queue in the end of Q'_j (i.e. a new sub-queue is added next to q'_{ik_i}), and let q'_w be the new sub-queue.

4. We add q'_u into the parent list of q'_w and q'_w the child list of q'_u.

For illustration, consider a program P consisting of processes P_1, P_2, ..., and P_3, where each process P_i, $1 \leq i \leq 3$, has a port T_i. Assume that in one possible execution of P, P_1

Figure 2.7: The first snapshot of T_3's happen-before graph

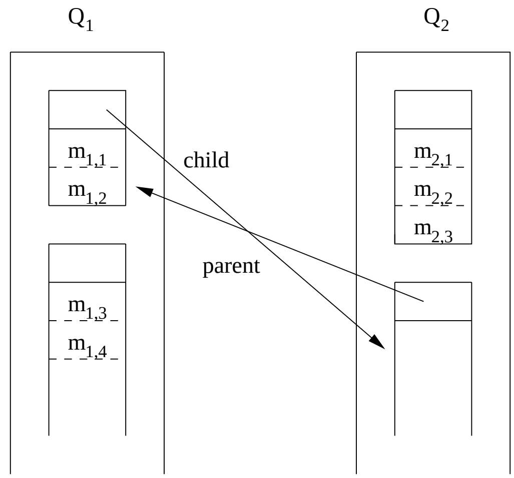


first sends messages $m_{1,1}$ and $m_{1,2}$ to P_3, which is followed by a message m to P_2, and then messages $m_{1,3}$ and $m_{1,4}$ to P_3. Also assume that in the same execution, P_2 sends messages $m_{2,1}$, $m_{2,2}$, and $m_{2,3}$ to P_3, and then receives m from P_1. Figures 2.7 and 2.8 show two snapshots of the happen-before graph of T_3. The first snapshot, as shown in Figure 2.7, is captured before m is received by P_2; the second snapshot, as shown in Figure 2.8, is captured after m is received by P_2. Note that m is not shown in both figures, because m is not a message sent to T_3.

It is very important to note that two global states of a program are equal if and only if the two states contain (1) the same local state for each process; (2) the same values for all the variables of each process; and (3) each port contains the same happen-before graph. Two happen-before graphs are the same if and only if they contain the same set of messages which are organized into the same graph structure (i.e. these messages have the same happen-before graph).

One might think that we can attach a vector timestamp to each message, and then determine the happen-before relation among the messages in a port based on the timestamps. We point out that such use of timestamp is not feasible, because timestamps are part of a global state, and since timestamps are by nature ever increasing, the equality test between two global states would become impossible.

Figure 2.8: The second snapshot of T_3's happen-before graph



2.4.3 Correctness of BSRA

In this section, we prove that BSRA is capable of detecting every deadlock of an asynchronous message-passing program.

Definition 11 *Let g be a global state of a system P of EFSMs. Let t be a transition of P and let $\omega = t_1 t_2 \ldots t_n$ be a transition sequence of P, where $g = g_1 \xrightarrow{t_1} g_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} g_n \xrightarrow{t_n} g'$. t and ω are said to be independent wrt g if t and t_i, $1 \leq i \leq n$, are independent wrt g_i.*

Lemma 2.4.3 *Let g be a global state of a system P of EFSMs. Let t be a transition of P and ω a transition sequence of P. if $g \xrightarrow{\omega t} * g'$ and t and ω are independent wrt g, then $g \xrightarrow{t\omega} * g'$.*

Definition 12 *Let P be a system of EFSMs M_1, M_2, ..., and M_n. Let $\sigma = t_1 t_2 \ldots t_m$ be a transition sequence of P. The projection $\sigma\downarrow_i$ of σ onto M_i, $1 \leq i \leq n$, is the transition sequence obtained by removing from σ all transitions not defined in M_i.*

Theorem 2.4.1 *Let P be a system of EFSMs M_1, M_2, ..., and M_n. Let g be a global state of P. Let ω_1, ω_2, ..., and ω_n be a set of enabled send/internal transition sequences from g, where ω_i, $1 \leq i \leq n$, is a transition sequence of M_i. Let α and β be two send/internal transition sequences such that $\alpha\downarrow_i = \beta\downarrow_i = \omega_i$ $(1 \leq i \leq n)$. If $g \xrightarrow{\alpha} * g'$, then $g \xrightarrow{\beta} * g'$.*

Proof

Let $\omega = \omega_1\omega_2 \ldots \omega_n$. We first show that if $g \xrightarrow{\alpha} * g'$, then $g \xrightarrow{\omega} * g'$. Let $\omega_1 = t_1t_2 \ldots t_q$. Assume that $\alpha = \mu_0t_1\mu_1t_2\mu_2 \ldots t_q\mu_q$. Note that t_1 and μ_0 are independent wrt g. Therefore, $g \overset{t_1\mu_0\mu_1t_2\mu_2\ldots t_q}{\longrightarrow} * g'$. Let $g \xrightarrow{t_1} g_1$. Note that t_2 and $\mu_0\mu_1$ are independent wrt g_1. Therefore, $g \overset{t_1t_2\mu_0\mu_1\mu_2\ldots t_q}{\longrightarrow} *g'$. By repeating this process, we can eventually establish $g \overset{t_1t_2\ldots t_q\mu_0\mu_1\ldots\mu_q}{\longrightarrow} *g'$. That is, $g \overset{\omega_1\nu_1}{\longrightarrow} * g'$, where ν_1 is the transition sequence obtained by removing from α all the transitions in ω_1.

Similarly, we can show that $g \overset{\omega_1\omega_2\nu_2}{\longrightarrow} * g''$, where ν_2 is the transition sequence obtained by removing from α all the transitions in ω_1 and ω_2. By repeating this process, we can establish $g \overset{\omega_1\omega_2\ldots\omega_n}{\longrightarrow} * g'$. That is, $g \xrightarrow{\omega} * g'$

Assume that $g \xrightarrow{\beta} * g''$. Similarly, we can derive that $g \xrightarrow{\omega} * g''$. Therefore, $g'' = g'$. This concludes our proof.

$\square$

Theorem 2.4.1 allows multiple processes to advance simultaneously from a global state g to reach a next blocking point of g.

Definition 13 *Let P be a system of EFSMs and g a global state of P. Let $e = (b, \gamma)$ be an edge generated algorithm Generate_BSRGEdges(g). The global state g' reached after the execution of e at g is said to be an immediate BSRG-based successor of g, denoted as $g \overset{(b,\gamma)}{\rightsquigarrow} g'$, where g' is reached from b by the execution of any permutation of transitions in γ.*

Lemma 2.4.4 *Let P be a system of EFSMs M_1, M_2, ..., and M_n. Let g be a global state of P. Let d be a deadlock state of P. Assume that $g \xrightarrow{\sigma} * d$, where $|\sigma| > 0$. Algorithm Generate_BSRGEdges(g) generates an edge $e = (b, \gamma)$, where b is a next blocking point of g and γ an independent global receive transition set at b, such that $g \overset{(b,\gamma)}{\rightsquigarrow} g' \xrightarrow{\alpha} * d$ and $|\alpha| < |\sigma|$.*

Proof

Let $\sigma \downarrow_i = \omega_{i,0}r_{i,1}\omega_{i,1}r_{i,2} \ldots \omega_{i,m_i-1}r_{i,m_i}\omega_{i,m_i}$, where $1 \le i \le n$, $0 \le m_i \le |\sigma|$, $\omega_{i,j}$, $0 \le j \le m_i$, is a send transition sequence with $|\omega_{i,j}| \ge 0$, and $r_{i,k}$, $1 \le k \le m_i$, is a global receive transition. Let $\omega = \omega_{1,0}\omega_{2,0} \ldots \omega_{n,0}$. Let $g \xrightarrow{\omega} * b$, and $\gamma = \{r_{i1} | 1 \le i \le n, m_i > 0, r_{i1}$ *is a global receive transition at* $b\}$. In what follows, we show that $b \in$ *next_bp(g)* is generated by Step 1 of algorithm Generate_BSRGEdges, and γ is generated by Step 2.1

or Step 2.2.1 of algorithm Generate_BSRGEdges. Therefore, $e = (b, \gamma)$ is generated by algorithm Generated_BSRGEdges.

In order to show $b \in next\_bp(g)$, we first show that b is a blocking point. For $1 \le i \le n$, if $m_i = 0$, then $\sigma\downarrow_i = \omega_{i,0}$. Thus, the local state of M_i at b is the same as that at d. Since d is a deadlock state, M_i has no enabled transitions at b. If $m_i > 0$, then $r_{i,1}$ is an instance of a local receive transition of M_i that is open at b. Therefore, each EFSM M_i, $1 \le i \le n$, either has an open receive transition or has no enabled transitions at b. Hence, b is a blocking point. Since ω is an enabled send/internal transition sequence from g, $b \in next\_bp(g)$.

Now let us show that γ is generated by algorithm Generate_BSRGEdges. If $\gamma = \emptyset$, then $g \xrightarrow{\omega} * d$. That is, b and d are the same deadlock state. Since b has no enabled transitions, γ is generated by Step 2.2.1 of algorithm Generate_BSRGEdges. Now we assume that $\gamma \neq \emptyset$. Let $\gamma' = \{r_{i,1} | 1 \le i \le n,\ m_i > 0,\ r_{i,1}$ is not a global receive transition at $b\}$. Let $P_1 = \{M_i | 1 \le i \le n, m_i > 0,\ r_{i,1} \in \gamma\}$, $P_2 = \{M_i | 1 \le i \le n, m_i > 0,\ r_{i,1} \in \gamma'\}$, $P_3 = \{M_i | 1 \le i \le n, m_i = 0$ (*i.e.* $r_{i,1}$ does not exist)$\}$. Note that P_1, P_2, and P_3 are pairwise disjoint and $P_1 \cup P_2 \cup P_3 = \{M_1, M_2, \ldots, M_n\}$. When function *gen_trans_sets* is executed, γ_i for each M_i, $1 \le i \le n$, is constructed with the following properties: (1) If M_i is in P_1, γ_i contains $r_{i,1}$; (2) If M_i is in P_2, then the local receive transition of which $r_{i,1}$ is an instance is an HET at b. Thus, γ_i contains ϵ; (3) If M_i is in P_4, the local state of M_i at b is the same as that at d. Since d is a deadlock state, M_i has no enabled transitions at b. Thus, $\gamma_i = \emptyset$. Therefore, the cross product of γ_1, γ_2, ..., and γ_n contains an element, say ϕ, that consists of $r_{i,1}$ for each M_i in P_1, and ϵ for each M_i in P_2, and nothing for each M_i in P_3. Since ϵ is deleted from ϕ before function *gen_trans_sets* returns, γ is generated by Step 2.1 of algorithm Generate_BSRGEdges.

In order to show $|\alpha| < |\sigma|$, we first show either $|\omega| > 0$ or $|\gamma| > 0$. Assume that $|\omega| = 0$. Since g is not a deadlock state, $|\gamma| > 0$. Note that α is the transition sequence obtained by removing from σ all the transitions in ω and γ. Therefore, $|\alpha| < |\sigma|$. This concludes our proof.

$\square$

Theorem 2.4.2 *Let P be a system of EFSMs M_1, M_2, ..., and M_n. Let BSRG(P) be the BSRG graph constructed by algorithm Generate_BSRG. P has a deadlock state if and only if BSRG(P) has a deadlock state.*

Proof

We first show that if BSRG(P) has a deadlock state d, then d is a reachable global state of P. Let g_0 be the initial global state of P. Let $H = e_1 e_2 \ldots e_m$ be a path that leads to d from g_0 in BSRG(P), where $e_i = (b_i, \gamma_i)$ is an edge of BSRG(P) such that b_i is a blocking point and γ_i is a set of independent global receive transitions. Let $\pi_i = \{\omega_{i1}, \omega_{i2}, \ldots, \omega_{in}\}$ be the set of send/internal transition sequences that can be executed to reach b_i, where $1 \leq i \leq m$, and letting $1 \leq j \leq n$, ω_{ij} is an enabled send/internal transition sequence of M_j at b_i. Let $\omega_i = \omega_{i1}\omega_{i2}\ldots\omega_{in}$ and υ_i an arbitrary permutation of γ_i. Let $\sigma = \omega_1\upsilon_1\omega_2\upsilon_2\ldots\omega_m\upsilon_m$. Then, $g_0 \xrightarrow{\sigma} * \, d$.

Now we show that algorithm Generate_BSRG generates every deadlock state of P. Let g_0 be the initial global state of P. Let d be a deadlock state of P such that $g_0 \xrightarrow{\sigma} * \, d$, where $|\sigma| \geq 0$. If $|\sigma| = 0$, it is trivial that d (or g_0) is generated by algorithm Generate_BSRG. In the rest of the proof, we assume $|\sigma| > 0$.

According to Lemma 2.4.4, $g_0 \overset{(b_1,\gamma_1)}{\rightsquigarrow} g_1 \xrightarrow{\alpha_1} * \, d$, where $b_1 \in \mathit{next\_bp}(g_0)$, γ_1 is an independent receive transition set at b_1, and $|\alpha_1| < |\sigma|$. If $|\alpha_1| = 0$, then $g_1 = d$. Therefore, d is generated by algorithm Generate_BSRG. Otherwise, we continue to apply Lemma 2.4.4. Eventually, we can establish that $g_0 \overset{(b_1,\gamma_1)}{\rightsquigarrow} g_1 \overset{(b_2,\gamma_2)}{\rightsquigarrow} g_2 \ldots \overset{(b_n,\gamma_n)}{\rightsquigarrow} g_n = d$. Therefore, d is generated by algorithm Generate_BSRG. This concludes our proof.

□

2.5 Empirical results

Our empirical study has two goals. First, we want to measure the state reduction rates achieved by BSRA. Second, we want to compare, if applicable, the state reduction rates achieved by BSRA with those achieved by existing reachability analysis approaches.

We measure the state reduction rates of BSRA as follows. Given a system P of EFSMs, we apply algorithms Generate_RG and Generate_BSRG to generate RG(P) and BSRG(P) respectively. Let N_{RG} the number of states in RG(P), and N_{BSRG} the number of states in BSRG(P). The state reduction rate achieved by BSRA for P is calculated as $1 - N_{BSRG}/N_{RG}$.

In order to accomplish the second goal, we compare BSRA to two major reachability analysis approaches: partial order reduction (or POR) and simultaneous reachability analysis (or SRA). It has been shown that POR [20] [56] and SRA [38] [55] can achieve significant state reduction rates. Since previous work on POR and SRA assumes causal communication, our comparison between BSRA, POR and SRA is limited to causal communication.

In [36], Karacali applied SRA to the following distributed algorithms and compared their reachability graphs produced by SRA and POR (according to SPIN [56]): (1) Leader election algorithm of Dolev, Klawe and Rodeh (LE) [59], which determines the leader among a set of processes organized in a unidirectional ring structure; (2) Token passing algorithm of Misra (TP) [59], which ensures mutual exclusion among a set of processes organized in a unidirectional ring structure; (3) Mutual exclusion algorithm of Ricart and Agrawala/Suzuki/Kasami (ME) [59], which ensures mutual exclusion among a set of processes on a fully connected network; (4) Flooding protocol (FL) [69], which is used to to construct a minimum spanning tree in a randomly connected network; (5) Sliding window protocol (SW) [69], which exercises flow control for reliable data transfer between two processes; (6) Readers and writers protocol with concurrent reading (RW) [37], which ensures consistency of shared data among concurrent readers and writers. For details of these algorithms, the reader is referred to the PROMELA programs listed in [36]. (PROMELA is a protocol specification language used by SPIN [31].)

To ensure a fair comparison, we apply BSRA to the same algorithms and then compare the results from BSRA to those from POR and SRA. As in [36], each algorithm takes as a parameter the number of processes involved in the algorithm except that for SW, the window size is used as a parameter.

For each reachability analysis approach, we show the number of states V, the number of edges E, and the state reduction rates %. Note that data for RA, POR, and SRA come from [36]. Due to minor differences in the graph construction, the sizes of the full reachability graphs are different between the FSM model used by POR and the multi-port EFSM model used by SRA and BSRA. Therefore, the state reduction rates of POR are measured relative to the sizes of the full reachability graphs based on the FSM model, and the state reduction rates of SRA and BSRA are measured relative to the sizes of the full reachability graphs based on the EFSM model. This is consistent with [36].

Tables 2.1 and 2.2 show empirical results for fully asynchronous communication and FIFO communication respectively. In the first column, we show the name and size of each algorithm in the form of *algorithm - size*. Since previous work on POR and SRA assumes causal communication, there are no data for these two approaches in Tables 2.1 and 2.2. Note that we have less data for fully asynchronous and FIFO communication. This is because the state space of a program increases dramatically with these two communication schemes and we have very limited computing power. Also note that algorithms TP, ME, and RW have the same state space independent of their communication schemes. To avoid duplication, empirical results for these three algorithms are only reported for causal communication.

The remaining tables show empirical results for causal communication. Since we have more data available for causal communication, we show empirical results for each algorithm in a separate table. The first column of each table shows the size of the corresponding algorithm. We observe that when causal communication is assumed, BSRA performs better than POR and SRA for all the cases we consider except that for algorithm FL with 3 processes, SRA performs slightly better than BSRA. In many cases, BSRA performs significantly better than POR and SRA. For instance, for the RW problem with 4 readers and 1 writer (the last row of Table 2.8), BSRG generates 1375 states with the state reduction rate up to 98%, while POR generates 39001 states with the state reduction rate as 52%, and SRA generates 31217 states with the state reduction rate as 54%.

Table 2.1: Empirical results for fully asynchronous communication

	RG		BSRG		
Algo. - size	V	E	V	E	%
LE - 3	1015	1698	410	734	70
FL - 3	90	149	31	40	66
FL - 4	173	340	44	63	75
FL - 5	340	803	60	95	82
SW - 2	14287	47692	645	2151	95

Table 2.2: Empirical results for FIFO communication

	RG		BSRG		
Algo. - size	V	E	V	E	%
LE - 3	379	834	9	8	98
FL - 3	90	149	31	40	66
FL - 4	173	340	44	63	75
FL - 5	340	803	60	95	82
SW - 2	2939	5300	219	585	94

Table 2.3: Empirical results for causal communication - LE

	POR					SRA							
	RG		PRG			RG		SRG				BSRG	
size	V	E	V	E	%	V	E	V	E	%	V	E	%
3	369	808	59	59	84	379	834	37	36	90	9	8	98
4	2180	6465	77	77	96	2228	6594	45	44	98	11	10	99
5	14048	52288	95	95	99	14194	52707	53	52	100	13	12	100
6	92895	414808	113	113	100	93194	415807	61	60	100	15	14	100

Table 2.4: Empirical results for causal communication - TP

	FSM					EFSM							
	RG		PRG			RG		SRG			BSRG		
size	V	E	V	E	%	V	E	V	E	%	V	E	%
3	104	119	92	94	12	111	125	97	97	13	27	27	76
4	159	215	120	122	25	165	219	124	124	25	39	39	77
5	248	420	148	150	40	252	419	151	151	40	53	53	79
6	405	877	176	178	57	405	864	178	178	56	69	69	83
7	698	1906	204	206	71	690	1865	205	205	70	87	87	88
8	1263	4215	232	234	82	1239	4110	232	232	81	107	107	92
9	2372	9356	260	262	89	2316	9107	259	259	89	129	129	95
10	4569	20705	288	290	94	4449	20136	286	286	94	153	153	97
11	8942	45558	316	318	96	8694	44285	313	313	96	179	179	98
12	17667	99595	344	346	98	17163	96786	340	340	98	207	207	99
13	35096	216352	372	374	99	34080	210215	367	367	99	237	237	99

Table 2.5: Empirical results for causal communication - FL

	FSM					EFSM							
	RG		PRG			RG		SRG			BSRG		
size	V	E	V	E	%	V	E	V	E	%	V	E	%
3	69	118	48	56	30	68	116	13	15	81	21	28	69
4	209	418	135	166	35	208	416	36	49	83	34	51	84
5	497	1166	310	408	38	496	1164	84	118	83	50	83	90
6	1081	2958	661	934	39	1080	2956	157	222	85	69	124	94
7	2257	7134	1364	2068	40	2256	7132	255	361	89	91	174	96
8	4617	16670	2771	4498	40	4616	16668	378	535	92	116	233	98
9	9345	38110	5586	9680	40	9344	38108	526	744	94	144	301	98
10	18809	85726	11217	20686	40	18808	85724	699	988	96	175	378	99

Table 2.6: Empirical results for causal communication - ME

	FSM					EFSM							
	RG		PRG			RG		SRG			BSRG		
size	V	E	V	E	%	V	E	V	E	%	V	E	%
2	62	104	27	30	56	62	103	15	15	76	3	2	96
3	2194	5733	528	705	76	2198	5807	450	857	80	19	21	99

Table 2.7: Empirical results for causal communication - SW

	FSM					EFSM							
	RG		PRG			RG		SRG			BSRG		
size	V	E	V	E	%	V	E	V	E	%	V	E	%
1	94	117	81	89	14	89	108	71	76	20	12	13	87
2	2946	5313	1177	1488	60	2939	5300	1243	1685	58	219	585	94
3	41344	82829	15116	21431	63	41335	82812	16141	21524	61	2296	9869	96

Table 2.8: Empirical results for causal communication - RW

	FSM					EFSM							
	RG		PRG			RG		SRG			BSRG		
size	V	E	V	E	%	V	E	V	E	%	V	E	%
3	314	546	250	344	20	394	685	298	540	24	43	78	90
4	4827	11154	2909	4502	40	4926	11390	2910	7440	41	277	673	95
5	81155	227934	39001	66477	52	67962	192108	31272	102553	54	1375	4447	98

2.6 Summary

In this chapter, we have presented a new reachability analysis approach, namely BSRA, for asynchronous message-passing programs. BSRA can be applied programs based on any communication scheme and guarantees the detection of deadlocks. Empirical results indicate that BSRA can significantly reduce the number of states generated during reachability analysis.

The idea of simultaneous execution used in BSRA is inspired by SRA [55] [35]. However, the two approaches significantly differ from each other. SRA allows multiple processes to proceed simultaneously, but each process can execute at most one transition at each step. In contrast, BSRA allows multiple processes to proceed simultaneously until each process either terminates or is ready to execute a receive transition. It is possible that a process executes a sequence of transitions at each step. In addition, SRA assumes causal communication, while BSRA can handle any communication scheme.

As mentioned in Section 2.2, various partial order reduction techniques have been developed for reachability analysis. These techniques can be used to reduce the size of reachability graphs generated by BSRA. To illustrate this, we briefly discuss how to apply the sleep set technique [20] to the BSRG graph in Figure 2.6. In this BSRG graph, the two paths $g_0g_1g_5$ and $g_0g_2g_3g_5$ represent the same partial order, and thus one of them is redundant. According to the sleep set technique, each node is associated with a set, called a sleep set, to contain transitions that need to be suppressed. When node g_2 is generated, its sleep set contains the match between t_5 and x. As a result, the transition from g_2 to g_3 would be suppressed.

One major application of reachability analysis is model checking, which involves checking global states of a program to verify properties specified as temporal logic formulae. BSRGs are sufficient for deadlock detection, but not for model checking. Future work on how to extend BSRA for model checking is discussed in Chapter 5.

Chapter 3

Reachability Testing

3.1 Background

Due to variations in message latencies and process speeds, the behavior of a concurrent program is nondeterministic. Let P be a concurrent program and X an input of P. Multiple executions of P with X may produce different results. Even if P with X has been executed successfully many times, it is possible that a future execution of P with X will produce an incorrect result.

Reachability testing of P with X is to test, in a systematic manner, all possible executions of P with X by deriving sequences of synchronization events (or SYN-events). An execution of a concurrent programming construct generates one or more SYN-events. An actual or expected execution of P with X can be characterized by its sequence of synchronization events, referred to as a synchronization sequence (or SYN-sequence), in that the result of an execution of P with X can be determined by P, X, and the SYN-sequence of this execution. The definitions of SYN-events and SYN-sequences of P are based on P's concurrent programming constructs [63]. An SYN-sequence can be a total or partial ordering of SYN-events. Different serializations of a partially ordered SYN-sequence generate different totally ordered SYN-sequences, which are equivalent and produce the same result. In the following discussion, SYN-sequences are assumed to be partially ordered unless otherwise specified.

We assumed that P does not use nondeterministic statements. This implies that the existence of race conditions during interprocess communication accounts for the only source of nondeterminism exhibited by P. We also assume that every execution of P with X

terminates. This implies that P has a finite number of possible executions. A general description of reachability testing of P with X is provided below [33]:

1. Perform nondeterministic testing of P with X to collect one SYN-sequence.

2. For a collected SYN-sequence, derive its race variants by changing the outcomes of race conditions in this SYN-sequence. (These race variants are prefixes of other SYN-sequences of P with X).

3. For each new race variant derived in Step 2, force a deterministic execution of P with input X according to this race variant and then let P continue execution nondeterministically to collect one SYN-sequence. (This process is referred to as prefix-based testing.)

4. For each new SYN-sequence collected in Step 3, repeat Steps 2, 3, and 4.

The above procedure eventually terminates since the number of different SYN-sequences of P with X is finite. Since reachability testing executes every possible SYN-sequence of P with X at least once, it can determine the correctness of P with X. Note that the above description of reachability testing is abstract. This is because the details of the above steps depend upon P's concurrent programming constructs. A discussion about how to build tools for supporting reachability testing of concurrent programs can be found in [63].

In [33], Hwang, Tai and Huang showed how to perform reachability testing of concurrent programs that use read and write operations and do not contain nondeterministic statements. Let P be such a program. Each shared variable in P is assigned a version number, which is initialized to zero and increased by one after each write operation on this variable. A read event is denoted as $R(U, V)$, which refers to a read operation on variable U with its version number being V. A write event is denoted as $W(U, V)$, which refers to a write operation on variable U with the resulting version number being V. A sequence of read and write events is called an RW-sequence. Let Q be a collected RW-sequence of P with input X. The race variants of Q are derived by generating totally ordered sequences of read and write operations (not events) in Q. When a sequence of read and write operations in Q results in a read or write event with its version number different from that of the corresponding event in Q, this sequence is a race variant of Q.

In [64] Tai presented an algorithm for reachability testing of asynchronous message-passing programs. The execution of an asynchronous message-passing program exercises a sequence of send and receive events, or SR-sequence. Tai showed how to construct race variants of an SR-sequence by changing the outcome of a single race condition. However, in a concurrent environment, the outcomes of multiple race conditions are likely to change simultaneously. In this case, those outcomes have to be considered one by one, and each step requires one program execution. Such generation of race variants is time-consuming. An important issue is how to generate all race variants of an SR-sequence directly (i.e., without using prefix-based testing). Another important issue not discussed in [64] is how to organize collected SR-sequences and their race variants in order to allow efficient search. (Similar issues exist in [33] for RW-sequences.)

In this chapter, we extend the work in [64] by addressing the two issues mentioned above. The next section discusses related work. Section 3.3 provides preliminary information. Section 3.4 describes an approach to computing the race variants of an SR-sequence. Section 3.5 shows an efficient algorithm, based on the computation of race variants, for reachability testing of asynchronous message-passing programs. Section 3.6 summarizes this chapter.

3.2 Related work

In [11] the controlled execution technique was proposed for testing asynchronous message-passing programs. A message race at a receive operation refers to the existence of two or more messages that could be accepted at the receive operation. During the controlled execution of an asynchronous message-passing program, the following procedure is applied once to each process. At each receive in a process, the execution of this process is suspended until the set of all messages that could be accepted at the receive operation has been collected (this happens when all other processes are blocking at a receive operation). Each acceptable message at the receive operation is chosen to be accepted by the receive operation, and then the controlled execution continues. The suspension or resumption of execution of each process is accomplished by modifying the implementation of the underlying message-passing system. This technique has two major problems. First, it does not consider how to reduce the chance of executing a partially-ordered path more than once. Second, for a program that terminates on any input, this technique does not guarantee to execute every partially-ordered path at least once [62]. Therefore, this technique does not guarantee the detection of deadlocks and assertion violations.

In [21] Godefroid developed a stateless search technique to explore the state space of a concurrent program. His technique is also considered as a testing technique for concurrent programs. Godefroid's technique uses a stack to perform a selective depth-first search in the state space of a concurrent program. The stack contains the sequence of transitions (equivalent to SYN-events) that leads from the initial global state to the current global state being explored. Global states associated with this sequence of transitions are implied by these transitions, but they are not saved. When a global state is being explored, its set of executable transitions is saved. To continue the search, one element of the set is removed, added to the stack, and executed to produce the global state to be explored next. When the use of a transition for state search is complete, this transition is removed from the stack, and the global state corresponding to the path in the stack is restored by restarting the program and forcing a deterministic execution of the program according to the path in the stack. If the set of remaining executable transitions for the restored state is not empty, then the search continues. Otherwise, the use of the top transition in the stack for state search is complete. Godefroid's technique applies partial order reduction techniques [20] to reduce the chance of executing two or more different totally-ordered paths with the same partial order. Based on this stateless search technique, Godefroid developed a system, called VeriSoft, for testing concurrent programs written in a sequential language plus concurrent programming constructs provided by Unix [21] [22]. VerifSoft creates a new process, called the scheduler, that uses Unix commands to delay and resume the execution of processes.

For a concurrent program that terminates on any input, Godefroid's technique executes each partially ordered SYN-sequence at least once and thus guarantees the detection of deadlocks and assertion violations. In contrast, reachability testing executes every partially-ordered SYN-sequence exactly once and thus provides better performance. Although reachability testing needs to maintain a tree structure, which is discussed in Section 3.5, to keep all generated SR-sequences, this tree structure does not take much space and can be used for regression testing, analysis of synchronization patterns, checking of assertions on sequencing of events, and other purposes.

One common problem for stateless search techniques is the inability of detecting cycles. In [66] Tai and Karacali proposed how to extend Godefroid's technique by adding a simple yet effective method for predicting the existence of cycles in the stack used for storing paths. They also proposed an extension to Godefroid's technique based on simultaneous reachability analysis [35] [38], which allows concurrent executions of multiple independent transitions.

3.3 Preliminaries

3.3.1 SR-sequence

In this section, we formally define an SR-sequence. We assume that an asynchronous message-passing program uses the following types of send and receive operations [62]:

- non-blocking send, which sends a message to a designated process and does not wait for the completion of message delivery; and
- blocking receive, which waits to receive a message sent by any process.

As discussed Chapter 2, the behavior of an asynchronous message-passing program depends on its underlying communication scheme [62]. In this chapter, we assume that communication is fully asynchronous, meaning that messages sent from one process to another process may not necessarily be received in the same order as they are sent. Our work, however, is not tied to any communication scheme.

Let P be an asynchronous message-passing program containing processes P_1, P_2, $\ldots$, and P_n. An execution of P with input X exercises a sequence of send and receive events. A send event refers to the exercise of a send operation, and a receive event refers to the exercise of a receive operation. We assign each event e a descriptor $\hat{e}$ as follows. If e is a send event, $\hat{e} = (i, j, k)$, indicating that e is the jth event of P_i and has P_k as its destination, where $1 \leq i, k \leq n$ and $j \geq 1$. If e is a receive event, $\hat{e} = (u, v)$, indicating that e is the vth event of P_u, where $1 \leq u \leq n$ and $v \geq 1$. If the message sent by a send event s is received by a receive event r, then it is said that a synchronization, denoted as (s, r), occurs between s and r. We also assign each synchronization $\gamma = (s, r)$ a descriptor $\hat{\gamma} = (i, j, u, v)$, indicating s is the jth event of P_i, and r the vth event of P_u, where $j \geq 1$, $v \geq 1$, and $1 \leq i, u \leq n$.

An SR-sequence consists of a sequence of send and receive events that are related by a set of synchronizations. Below we formally define an SR-sequence.

Definition 14 *An SR-sequence Q is defined as a triple $(\mathcal{S}, \mathcal{R}, \mathcal{Y})$, where $\mathcal{S}$ is a set of send events, $\mathcal{R}$ a set of receive events, and $\mathcal{Y} \subseteq \mathcal{S} \times \mathcal{R}$ a partial one-to-one function. $\mathcal{Y}$ is also referred to the synchronization relation of Q.*

Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence. A send event $s \in \mathcal{S}$ is said to be a received send event if there exists a receive event $r \in \mathcal{R}$ such that $(s, r) \in \mathcal{Y}$. Otherwise, s is said to be a non-received send event. A receive event $r \in \mathcal{R}$ is said to be a completed receive event if there exists a send event $s \in \mathcal{S}$ such that $(s, r) \in \mathcal{Y}$. Otherwise, r is said to be a pending receive event. Given two events e and e' in Q, e happens before e', denoted as $e \rightarrow_Q e'$ or $e \rightarrow e'$ if Q is implied, if either of the following conditions is held [40]: (1) e and e' are events of the same process and e is prior to e'; (2) $(e, e') \in \mathcal{Y}$; or (3) there exists an event e'' in Q such that $e \rightarrow e''$ and $e'' \rightarrow e'$.

An SR-sequence is said to be well-formatted if it represents a possible execution of an asynchronous message-passing program. Formally, an SR-sequence $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ is well-formatted if the following conditions are met: (1) A process is not allowed to send a message to itself. Formally, $\gamma = (s, r) \in \mathcal{Y} \wedge \hat{\gamma} = (i, j, u, v) \Longrightarrow i \neq u$; (2) No events exist after a pending receive event. Formally, $e \in \mathcal{S} \cup \mathcal{R} \wedge r \in \mathcal{R} \wedge r \rightarrow e \Longrightarrow \exists s \in \mathcal{S} : (s, r) \in \mathcal{Y}$; and (3) Within a process, there are no missing events based on indices, and each index is used only for one event. In the remainder of this chapter, we assume that an SR-sequence is well-formatted.

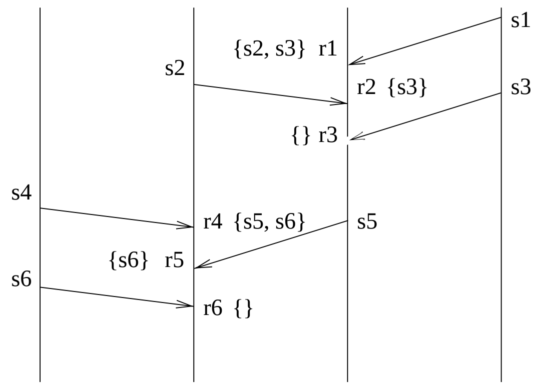
To provide an intuitive view, an execution of an asynchronous message-passing program is often depicted using a time-space diagram. In a time-space diagram, a process is represented as a vertical line, which contains the sequence of events happened in the process. A solid-line arrow is drawn from a send event s to a receive event r if the message sent by s is received by r. A non-received send event is associated with a dangling arrow pointing to the destination. A pending receive event is not associated with any arrow.

Figure 3.1 shows an example time-space diagram, which can be written as an SR-sequence $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$, where $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5, s_6\}$, $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, and $\mathcal{Y} = \{(s_1, r_1), (s_2, r_2), (s_3, r_3), (s_4, r_4), (s_5, r_5), (s_6, r_6)\}$.

3.3.2 Race variant

Let P be an asynchronous message-passing program and X an input of P. A race variant of an SR-sequence of P with X can be obtained by modifying the outcome of one or more race conditions in the SR-sequence and is guaranteed to be exercised by another execution of P with X regardless of how P is implemented. In this section, we formally define the concept of race variant.

Figure 3.1: An example time-space diagram



Definition 15 *Let P be an asynchronous message-passing program and X an input of P. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ and $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}')$ be two SR-sequences of P with X. Q and Q' are equivalent, denoted as $Q \equiv Q'$, if the following conditions are met:*

- *For each event $e \in \mathcal{S} \cup \mathcal{R}$, there exists an event $e' \in \mathcal{S}' \cup \mathcal{R}'$ such that $\hat{e} = \hat{e'}$, and vice versa; and*

- *For each synchronization $\gamma \in \mathcal{Y}$, there exists a synchronization $\gamma' \in \mathcal{Y}$ such that $\hat{\gamma} = \hat{\gamma'}$, and vice versa.*

Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence. Let $e \in \mathcal{S} \cup \mathcal{R}$. $before_Q(e)$, or $before(e)$ when Q is implied, is defined as follows: $before_Q(e) = \{e' \in \mathcal{S} \cup \mathcal{R} | e' \rightarrow_Q e\}$. $prefix_Q(e)$, or $prefix(e)$ when Q is implied, is defined as follows: $prefix_Q(e) = (\mathcal{S} \cap before(e), \mathcal{R} \cap before(e), \{(s, r) \in \mathcal{Y} | s, r \in before(e)\})$.

Definition 16 *Let P be an asynchronous message-passing program and X an input of P. Let Q and Q' be two SR-sequences of P with X. Let e be an event in Q, and e' an event in Q'. e and e' are equivalent, denoted as $e \equiv e'$, if $prefix_Q(e) \equiv prefix_{Q'}(e')$ and $\hat{e} = \hat{e'}$.*

It is worth noting that the notion of event is with respect to a single program execution, or an SR-sequence. However, in this chapter, we treat equivalent events in different SR-sequences as the same events. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ and $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}')$ be two SR-sequences of P with X. Let $e \in \mathcal{S} \cup \mathcal{R}$. If there exists an event $e' \in \mathcal{S}' \cup \mathcal{R}'$ such that $e \equiv e'$, then it is said that $e \in \mathcal{S}' \cup \mathcal{R}'$, and vice versa. In such cases, it is implied that $prefix_Q(e) \equiv prefix_{Q'}(e')$.

Definition 17 *Let P be an asynchronous message-passing program and X an input of P. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence of P with X. Let $r \in \mathcal{R}$ and $(s, r) \in \mathcal{Y}$. The race set of r in Q, denoted as $race(Q, r)$ or $race(r)$ if Q is implied, is defined as follows: A send event s' in $\mathcal{S}$, other than s, is in $race(r)$ if there exists another SR-sequence $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}')$ of P with X such that $s' \in \mathcal{S}'$, $r \in \mathcal{R}'$, and $(s', r) \in \mathcal{Y}'$.*

Theorem 3.3.1 (adapted from [62]) shows how to compute the race set of a receive event in an SR-sequence.

Theorem 3.3.1 *Let P be an asynchronous message-passing program based on fully asynchronous communication. Assume that P contains processes P_1, P_2, $\ldots$, and P_n. Let X an input of P. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence of P with X. Let $r \in \mathcal{R}$ and $(s, r) \in \mathcal{Y}$. Then, $race(Q, r) = \{s' \in \mathcal{S} | s'\ has\ r's\ process\ as\ the\ destination,\ r \not\rightarrow s', and\ if\ (s', r') \in \mathcal{Y}, then\ r \rightarrow r'\}$.*

In Figure 3.1, the race set of each receive event in the example SR-sequence is shown next to the event. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence. Let r be a receive event in Q. Assume that $(s, r) \in \mathcal{Y}$. We denote $Race(r) = race(r) \cup \{s\}$.

Definition 18 *Let P be an asynchronous message-passing program and X an input of P. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ and $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}')$ be two SR-sequences of P with X. Q' is a race variant of Q if the following conditions are met:*

- $\mathcal{S}' \subseteq \mathcal{S} \wedge \mathcal{R}' \subseteq \mathcal{R} \wedge \mathcal{Y} \nsubseteq \mathcal{Y}' \wedge \mathcal{Y}' \nsubseteq \mathcal{Y}$; *and*
- $\forall s \in \mathcal{S}, s' \in \mathcal{S}', r \in \mathcal{R}' : (s, r) \in \mathcal{Y} \wedge (s', r) \in \mathcal{Y}' \wedge s \not\equiv s' \Longrightarrow s' \in race(Q, r)$.

We explain the two conditions in Definition 18 as follows. The first condition says that every event in Q' is also in Q or can find an equivalent counterpart in Q. It also says that Q and Q' have at least one different synchronization. The second conditions says that if a receive event r in Q' receives the message m that is different from the message received by r in Q, then the send event that sends m must belong to the race set of r in Q. The reader should notice that, in this case, no events happen after r in Q'. This is because if r receives a different message in Q', it would be impossible for any event happening after r to find an equivalent counterpart in Q. We use $variant(Q)$ to denote all the race variants of Q.

3.4 An approach to computing race variants

In this section, we describe a formal approach to computing the race variants of an SR-sequence. We start with basic definitions and theorems, and then present an algorithm for computation of race variants.

3.4.1 Basic definitions and theorems

Definition 19 *Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence. The tail set of Q, denoted as $tail(Q)$, is defined as $tail(Q) = \{r \in \mathcal{R} | \nexists r' \in \mathcal{R} : r \rightarrow r'\}$.*

Intuitively, a receive event r is in $tail(Q)$ if no receive events happen after r in Q. Note that all the receive events in $tail(Q)$ are concurrent with each other. Otherwise, there must exist two receive events $r_1, r_2 \in tail(Q)$ such that $r_1 \rightarrow r_2$. This is self-contradictory, since $r_1 \rightarrow r_2$ implies $r_1 \notin tail(Q)$ due to the fact that r_2 happens after r_1.

Definition 20 *Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$. The set of race variants related to $tail(Q)$, denoted as $variant\_tail(Q)$, is defined as $variant\_tail(Q) = \{(\mathcal{S}', \mathcal{R}', \mathcal{Y}') \in variant(Q) | \mathcal{R}' \cap tail(Q) \neq \emptyset\}$.*

In another word, a race variant Q' of an SR-sequence Q is in $variant\_tail(Q)$ if the receive event set of Q' contains at least one receive event in $tail(Q)$.

Definition 21 *Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$, and $R \subseteq \mathcal{R}$. $after(R) = \{e \in \mathcal{S} \cup \mathcal{R} | \exists e' \in R : e' \rightarrow e\}$.*

Informally, $after(R)$ is the set of events in Q that happen after at least one receive event in R. For the ease of reference, $after(R)$ is also referred to as the set of events that happen after R.

Definition 22 *Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$, and $R \subseteq \mathcal{R}$. A trim operator, denoted as $\setminus_t$, between Q and R is defined such that $Q \setminus_t R = (\mathcal{S}', \mathcal{R}', \mathcal{Y}')$, where $\mathcal{S}' = \mathcal{S} \setminus after(R)$, $\mathcal{R}' = \mathcal{R} \setminus (R \cup after(R))$, and $\mathcal{Y}' = \{(s, r) \in \mathcal{Y} | s \in \mathcal{S}' \wedge r \in \mathcal{R}'\}$.*

In the above definition, "$\setminus$" is the ordinary set difference operator. Informally, $Q \setminus_t R$ is an SR-sequence obtained by removing from Q all the events in R and all the events happening after R. Note that a synchronization (s, r) needs to be removed from Q if either s or r is removed from Q.

Theorem 3.4.1 *Let* $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$. $variant(Q) = variant(Q') \cup variant\_tail(Q)$, *where* $Q' = Q \setminus_t tail(Q)$.

Proof

First we show $variant(Q) \subseteq variant(Q') \cup variant\_tail(Q)$. Let $Q'' = (\mathcal{S}'', \mathcal{R}'', \mathcal{Y}'') \in variant(Q)$. If $\mathcal{R}'' \cap tail(Q) \neq \emptyset$, then $Q'' \in variant\_tail(Q)$. Otherwise, $\mathcal{R}'' \subseteq \mathcal{R} \setminus tail(Q)$, that is, $\mathcal{R}'' \subseteq \mathcal{R}'$. If we prove that $\mathcal{S}'' \subseteq \mathcal{S}'$, then $Q'' \in variant(Q')$.

Assume that $\mathcal{S}'' \nsubseteq \mathcal{S}'$. There must exist a send event $s \in \mathcal{S}''$, but $s \notin \mathcal{S}'$. This is possible only if s happens after one or more (receive) events in $tail(Q)$. However, if $s \in \mathcal{S}''$, all the events happening before s must be in $\mathcal{S}'' \cup \mathcal{R}''$. This means that there exists at least one receive event $r \in tail(Q)$ such that $r \in \mathcal{R}''$. This contradicts to the assumption that $\mathcal{R}'' \cap tail(Q) = \emptyset$. Therefore, $\mathcal{S}'' \subseteq \mathcal{S}'$. This derives that $variant(Q) \subseteq variant(Q') \cup variant\_tail(Q)$.

Obviously, $variant(Q') \subseteq variant(Q)$ and $variant\_tail(Q) \subseteq variant(Q)$. Therefore, $variant(Q') \cup variant\_tail(Q) \subseteq variant(Q)$. This concludes our proof.

$\square$

The above theorem can be intuitively understood as follows. Given $Q' \in variant(Q)$, Q' either contains one or more (receive) events in $tail(Q)$, or contains no (receive) events in $tail(Q)$. In the first case, $Q' \in variant\_tail(Q)$. In the latter case, Q' is a race variant of the SR-sequence $Q \setminus_t tail(Q)$.

Note that $Q' \setminus_t tail(Q)$ is an SR-sequence smaller than Q. Theorem 3.4.1 suggests that the problem of how to compute $variant(Q)$ can be reduced to the smaller problem of how to compute $variant(Q \setminus_t tail(Q))$. This makes it possible to compute $variant(Q)$ in a recursive manner. The challenge is, however, how to compute $variant\_tail(Q)$. Below we address this issue.

Definition 23 *Let* Q *be an SR-sequence and* $R \subseteq tail(Q)$. $variant\_intersect(Q, R)$ *is defined as follows:* $variant\_intersect(Q, R) = \{(\mathcal{S}', \mathcal{R}', \mathcal{Y}') \in variant(Q) | \mathcal{R}' \cap tail(Q) = R\}$.

We point out that $variant\_intersect(Q, tail(Q))$ is different from $variant\_tail(Q)$. A race variant Q' is in $variant\_intersect(Q, tail(Q))$ if Q' contains all the events in $tail(Q)$; a race variant Q' is in $variant\_tail(Q)$ as long as Q' contains at least one event in $tail(Q)$.

Theorem 3.4.2 *Let* $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ *be an SR-sequence.*

$$variant\_tail(Q) = \bigcup_{R \subseteq tail(Q) \wedge R \neq \emptyset} variant\_intersect(Q, R)$$

Proof

For the ease of reference, let $\mathcal{V} = \bigcup_{R \subseteq tail(Q) \wedge R \neq \emptyset} variant\_intersect(Q, R)$.

We first show $variant\_tail(Q) \subseteq \mathcal{V}$. Let $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}') \in variant\_tail(Q)$. Let $R' = \mathcal{R}' \cap tail(Q)$. By definition, $R' \neq \emptyset$. Since $R' \subseteq tail(Q)$, $Q' \in variant\_intersect(Q, R')$. Therefore, $variant\_tail(Q) \subseteq \mathcal{V}$.

Now we show $\mathcal{V} \subseteq variant\_tail(Q)$. Let $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}') \in variant\_tail(Q)$. Assume $Q' \in variant\_intersect(Q, R')$, where $R' \subseteq tail(Q) \wedge R' \neq \emptyset$. Since $\mathcal{R}' \cap tail(Q) = R' \neq \emptyset$, $Q' \in variant\_tail(Q)$. Therefore, $\mathcal{V} \subseteq variant\_tail(Q)$.

$\square$

Theorem 3.4.2 suggests that the problem of how to compute $variant\_tail(Q)$ can be transformed to the problem of how to compute $variant\_intersect(Q, R)$, where R is a non-empty subset of $tail(Q)$.

Theorem 3.4.3 *Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$, and $R \subseteq tail(Q)$. Let $Q' = Q \setminus_t (tail(Q) \setminus R)$.*

$$variant\_intersect(Q, R) = \bigcup_{R' \subseteq (tail(Q') \setminus R)} variant\_intersect(Q', R \cup R')$$

Proof

For the ease of reference, let $\mathcal{V} = \bigcup_{R' \subseteq (tail(Q') \setminus R)} variant\_intersect(Q', R \cup R')$. Let $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}')$.

We first show that $\mathcal{V} \subseteq variant\_intersect(Q, R)$. Let $Q'' = (\mathcal{S}'', \mathcal{R}'', \mathcal{Y}'') \in \mathcal{V}$, and assume that $Q'' \in variant\_intersect(Q', R \cup R'')$, where $R'' \subseteq tail(Q') \setminus R$. Since $\mathcal{R}' = \mathcal{R} \setminus (tail(Q) \setminus R)$, $\mathcal{R}' \cap tail(Q) = R$. Because $\mathcal{R}'' \subseteq \mathcal{R}'$, $\mathcal{R}'' \cap tail(Q) \subseteq \mathcal{R}' \cap tail(Q) = R$. Obviously, $R \subseteq \mathcal{R}''$. Therefore, $R \subseteq \mathcal{R}'' \cap tail(Q)$. That is, $\mathcal{R}'' \cap tail(Q) = R$. So, $Q'' \in variant\_intersect(Q, R)$. This derives $\mathcal{V} \subseteq variant\_intersect(Q, R)$.

Now we need to show that $variant\_intersect(Q, R) \subseteq \mathcal{V}$. Let $Q'' = (\mathcal{S}'', \mathcal{R}'', \mathcal{Y}'') \in variant\_intersect(Q, R)$. Because $\mathcal{R}'' \cap tail(Q) = R$, $\mathcal{R}'' \subseteq \mathcal{R} \setminus (tail(Q) \setminus R)$. That is, $\mathcal{R}'' \subseteq \mathcal{R}'$. If $\mathcal{S}'' \subseteq \mathcal{S}'$, $Q'' \in variant(Q')$. Assume that $\mathcal{S}'' \not\subseteq \mathcal{S}'$. There must exist a send event $s \in \mathcal{S}''$ but $s \notin \mathcal{S}'$. This is possible only if s happens after one event r in $tail(Q) \setminus R$.

However, $r \to s$ implies $r \in \mathcal{R}''$. This contradicts to the fact that $\mathcal{R}'' \cap tail(Q) = R$. Therefore, $Q'' \in variant(Q')$. Note that $R \subseteq tail(Q')$ and $R \subseteq \mathcal{R}''$. Thus, $R \subseteq \mathcal{R}'' \cap tail(Q')$. Therefore, $Q'' \in \mathcal{V}$. This concludes our proof.

□

Note that $variant\_intersect(Q, tail(Q))$ is a special case of Theorem 3.4.3. In what follows, we describe how to directly compute $variant\_intersect(Q, tail(Q))$.

Definition 24 *An event vector $v = (e_1, e_2, \ldots, e_n)$ is an ordered set of events where $\forall 1 \leq i, j \leq n : e_i \neq e_j$.*

Let $v = (e_1, e_2, \ldots, e_n)$. v is a send event vector if e_i is a send event, where $1 \leq i \leq n$; v is a receive event vector if e_i is a receive event, where $1 \leq i \leq n$.

Definition 25 *A synchronize operator $\bowtie$ between a send event vector $v_s = (s_1, s_2, \ldots, s_n)$ and a receive event vector $v_r = (r_1, r_2, \ldots, r_n)$ is defined such that $v_s \bowtie v_r = \{(s_1, r_1), (s_2, r_2), \ldots, (s_n, r_n)\}$.*

Definition 26 *An extract operator $\downarrow$ between a set $\mathcal{Y}$ of synchronizations and a set $\mathcal{R}$ of receive events is defined such that $\mathcal{Y} \downarrow \mathcal{R} = \{(s, r) \in \mathcal{Y} | r \in \mathcal{R}\}$.*

Intuitively, given a set of synchronizations $\mathcal{Y}$ and a set of receive events $\mathcal{R}$, $\mathcal{Y} \downarrow \mathcal{R}$ extracts synchronizations involving receive events in $\mathcal{R}$ from $\mathcal{Y}$.

Definition 27 *Let $\mathcal{Y}$ and $\mathcal{Y}'$ be two synchronization sets. A replace operator $\sharp$ between $\mathcal{Y}$ and $\mathcal{Y}'$ can be defined such that $\mathcal{Y} \sharp \mathcal{Y}' = \{(s, r) \in \mathcal{Y} | \forall (s', r') \in \mathcal{Y}' : r \neq r'\} \cup \{(s, r) \in \mathcal{Y}' | \exists (s', r') \in \mathcal{Y} : r = r'\}$.*

Intuitively, the replace operator between $\mathcal{Y}$ and $\mathcal{Y}'$ replaces a synchronization (s, r) in $\mathcal{Y}$ with a synchronization (s', r') in $\mathcal{Y}'$ if $r = r'$.

Theorem 3.4.4 *Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$. Assume an arbitrary order for receive events in $tail(Q)$ such that $tail(Q) = (r_1, r_2, \ldots, r_n)$. Let $\mathcal{V} = (Race(r_1) \times Race(r_2) \times \ldots \times Race(r_n)) \setminus \{(s_1, s_2, \ldots, s_n)\}$, where $(s_i, r_i) \in \mathcal{Y}$ and $1 \leq i \leq n$. Let $\Gamma = \{v \bowtie tail(Q) | v \in \mathcal{V}\}$. $variant\_intersect(Q, tail(Q)) = \{(\mathcal{S}, \mathcal{R}, \mathcal{Y} \sharp \mathcal{Y}^*) | \mathcal{Y}^* \in \Gamma\}$.*

Proof

For the ease of reference, we denote $\{(\mathcal{S}, \mathcal{R}, \mathcal{Y}\sharp\mathcal{Y}')\}$ as Λ. First, we need to show that $variant\_intersect(Q,$
$tail(Q)) \subseteq \Lambda$. Let $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}') \in variant\_intersect(Q, tail(Q))$. We need to prove $\exists \mathcal{Y}^* \in \Gamma : \mathcal{Y}' = \mathcal{Y}\sharp\mathcal{Y}^*$. In another word, we need to prove $\mathcal{Y}' \downarrow tail(Q) \in \Gamma$. Let $\mathcal{Y}' \downarrow tail(Q) = \{(s'_1, r_1), (s'_2, r_2), \ldots, (s'_n, r_n)\}$. By Definition 18, $s'_i \in Race(r_i)$, where $1 \leq i \leq n$. Therefore, $(s'_1, s'_2, \ldots, s'_n) \in \mathcal{V}$. So, $\mathcal{Y}' \downarrow tail(Q) \in \Gamma$.

Obviously, $\Lambda \subseteq variant\_intersect(Q, tail(Q))$. This concludes our proof.

□

3.4.2 A race variant computation algorithm

In this section, we present an algorithm, namely *Generate-Variants*, to compute the race variants of an SR-sequence. As shown in Figure 3.2, algorithm *Generate-Variants* is a trivial algorithmic translation of those theorems established in the previous section. The framework of algorithm *Generate-Variants* is based on Theorem 3.4.1; function *Generate-Tail-Variants* is based on Theorem 3.4.2; function *Generate-Intersect-Variants* is based on Theorem 3.4.3.

Note that both algorithm *Generate-Variants* and function *Generate-Intersect-Variants* are recursive. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence. Algorithm *Generate-Variants* terminates because we obtain a smaller SR-sequence by trimming the tail set at each recursive step. Function *Generate-Intersect-Variants* terminates for the following reason. For each recursion, the first parameter Q is reduced by the trim operation $Q \setminus_t (tail(Q) \setminus R)$, while the second parameter R remains at least the same size. Eventually, $tail(Q)$ will be equal to R, and then the function exits.

Below is a description of an example program M, which consists of processes M_1 through M_6.

- M_1: send a message to M_2;
- M_2: receive …; receive …;
- M_3: send a message to M_2;
- M_4: send a message to M_5;

Figure 3.2: Algorithm Generate-Variants

Algorithm *Generate-Variants*(Q: an SR-sequence)
begin
  let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$
  if $\mathcal{R} = \emptyset$
    return $\emptyset$;
  else
    return *Generate-Variants*$(Q \setminus_t tail(Q)) \cup$ *Generate-Tail-Variants*(Q);
end

function *Generate-Tail-Variants*(Q: an SR-sequence)
begin
  tail-variants $= \emptyset$;
  for each non-empty subset R of $tail(Q)$ **do**
  begin
    tail-variants $=$ *tail-variants* $\cup$ *Generate-Intersect-Variants*(Q, R);
  end
  return *tail-variants*;
end

function *Generate-Intersect-Variants*(Q: an SR-sequence, R: a set of receive events)
begin
  intersect-variants $= \emptyset$;
  if $R == tail(Q)$
    let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$
    assume an arbitrary order among receive events in $tail(Q)$ such that $tail(Q) = (r_1, r_2, \ldots, r_n)$
    assume that $\mathcal{Y} \downarrow tail(Q) = \{(s_1, r_1), (s_2, r_2), \ldots, (s_n, r_n)\}$
    $\mathcal{V} = (Race(r_1) \times Race(r_2) \times \ldots \times Race(r_n)) \setminus \{(s_1, s_2, \ldots, s_n)\}$
    $\Gamma = \{v \bowtie R | v \in \mathcal{V}\}$
    intersect-variants $= \{(\mathcal{S}, \mathcal{R}, \mathcal{Y} \sharp \mathcal{Y}^*) | \mathcal{Y}^* \in \Gamma\}$
  else
    $Q' = Q \setminus_t (tail(Q) \setminus R)$;
    for each subset R' of $tail(Q') \setminus R$ **do**
    begin
      intersect-variants $=$ *intersect-variants* $\cup$ *Generate-Intersect-Variants*$(Q', R \cup R')$;
    end
  return *intersect-variants*;
end

- M_5: receive ...; receive ...;
- M_6: send a message to M_5;

Assume that a nondeterministic execution of M produces SR-sequence Q_1 in Figure 3.3. Now we use algorithm *Generate-Variants* to compute $variant(Q_1)$.

- Note that $tail(Q_1) = \{r_2, r_4\}$. $variant\_tail(Q_1) = variant\_intersect(Q_1, tail(Q_1)) \cup variant\_intersect(Q_1, \{r_2\}) \cup variant\_intersect(Q_1, \{r_4\})$. Since $Race(r_2) = \{s_2\}$ and $Race(r_4) = \{s_4\}$, $variant\_intersect(Q_1, tail(Q_1)) = \emptyset$.

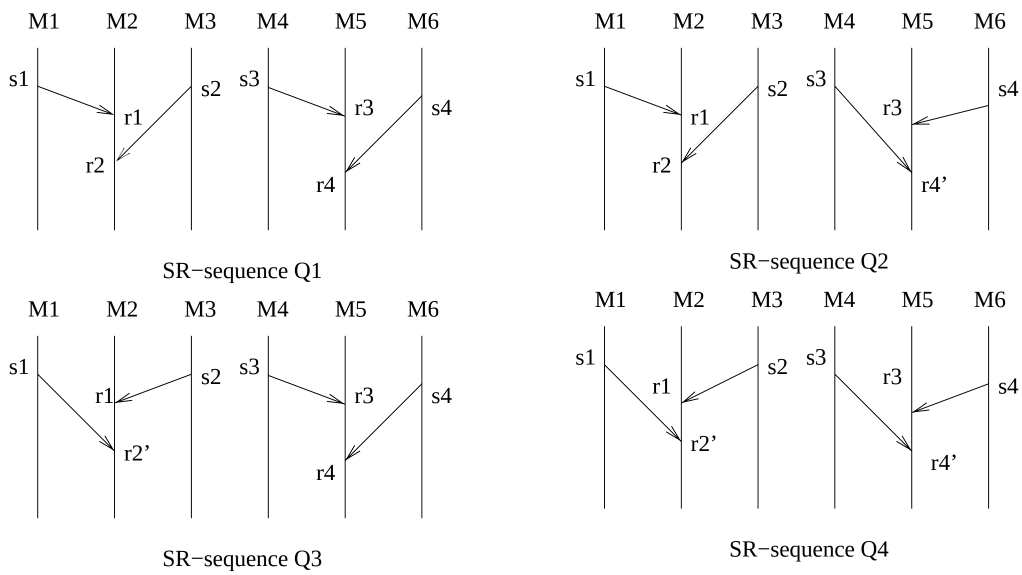
- In order to compute $variant\_intersect(Q_1, \{r_2\})$, we trim r_4 off from Q_1, which produces Q'_1 in Figure 3.4. $tail(Q'_1) = \{r_2, r_3\}$. Thus, $variant\_intersect(Q_1, \{r_2\}) = variant\_intersect(Q'_1, \{r_2\}) \cup variant\_intersect(Q'_1, \{r_2, r_3\})$. Note that $Race(r_2) = \{s_2\}$, and $Race(r_3) = \{s_3, s_4\}$. Hence, $variant\_intersect(Q'_1, \{r_2, r_3\})$ contains the only race variant V_1 in Figure 3.5. In order to compute $variant\_intersect(Q'_1, \{r_2\})$, we trim r_3 off from Q'_1, which produces Q'_2 in Figure 3.4. Since $tail(Q'_2) = \{r_2\}$ and $Race(r_2) = \{s_2\}$,, $variant\_intersect(Q'_2, \{r_2\}) = \emptyset$. Hence, $variant\_intersect(Q_1, \{r_2\})$ contains the only race variant V_1.

- Similarly, $variant\_intersect(Q_1, \{r_4\})$ contains the only race variant V_2 shown in Figure 3.5. So, we conclude the computation of $variant\_tail(Q_1)$, which contains two race variants: V_1 and V_2.

- Now we trim $tail(Q_1)$ off from Q_1, which produces Q'_3 in Figure 3.4. Note that $tail(Q'_3) = \{r_1, r_3\}$. Thus, $variant\_tail(Q'_3) = variant\_intersect(Q'_3, tail(Q'_3)) \cup variant\_intersect(Q'_3, \{r_1\}) \cup variant\_intersect(Q'_3, \{r_3\})$. Since $Race(r_1) = \{s_1, s_2\}$ and $Race(r_3) = \{s_3, s_4\}$, $variant\_intersect(Q'_3, tail(Q'_3))$ contains race variants V_3, V_4 and V_5, which are shown in Figure 3.5. In order to compute $variant\_intersect(Q'_3, \{r_1\})$, we need to trim r_3 off from Q'_3, which produces Q'_4 as shown in Figure 3.5. Note that $tail(Q'_4) = \{r_1\}$ and $Race(r_1) = \{s_1, s_2\}$. Thus, $variant\_intersect(Q'_3, \{r_1\})$ contains race variant V_6. Similarly, $variant\_intersect(Q'_3, \{r_3\})$ contains the only race variant V_7. At this point, we conclude the computation of $variant\_tail(Q'_3)$, which contains five race variants: $V_3 - V_7$.

- Now we need to trim r_1 and r_3 off from Q'_3, which produces Q'_5 as shown in Figure 3.4. Note that Q'_5 does not have any receive events. Thus, the algorithm terminates. At

Figure 3.3: All possible SR-sequences of program M



this point, we conclude the computation of $variant(Q_1)$, which contains seven race variants: $V_1 - V_7$.

3.5 An algorithm for reachability testing

Figure 3.6 shows an algorithm, namely *Reachability-Test*, for reachability testing of asynchronous message-passing programs. All the SR-sequences generated during reachability testing are kept in a tree structure, called SR_Tree, which is discussed in details later. A race variant V is used for prefix-based testing only if V is not a prefix of any SR-sequence in the SR_Tree. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ and $Q' = (\mathcal{S}', \mathcal{R}', \mathcal{Y}')$ be two SR-sequences. Q' is a prefix of Q if $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{R}' \subseteq \mathcal{R}$, and $\mathcal{Y}' \subseteq \mathcal{Y}$.

To organize a set of SR-sequences into a tree structure, we define a canonical form of an SR-sequence. Let P be an asynchronous message-passing program containing processes P_1, P_2, …, P_n. Let X an input of P. Let $Q = (\mathcal{S}, \mathcal{R}, \mathcal{Y})$ be an SR-sequence of P with X. The canonical form $[Q]$ of Q is a totally-ordered event sequence such that $[Q] = \sigma_0\gamma_1\sigma_1\gamma_2 \ldots \gamma_q\sigma_q$, where σ_i, $0 \leq i \leq q$, is a send event sequence, $|\sigma_i| \geq 0$, and γ_j, $1 \leq j \leq q$, a receive event sequence, $|\gamma_j| > 0$.

Figure 3.4: SR-sequences of program M generated during race variant computation

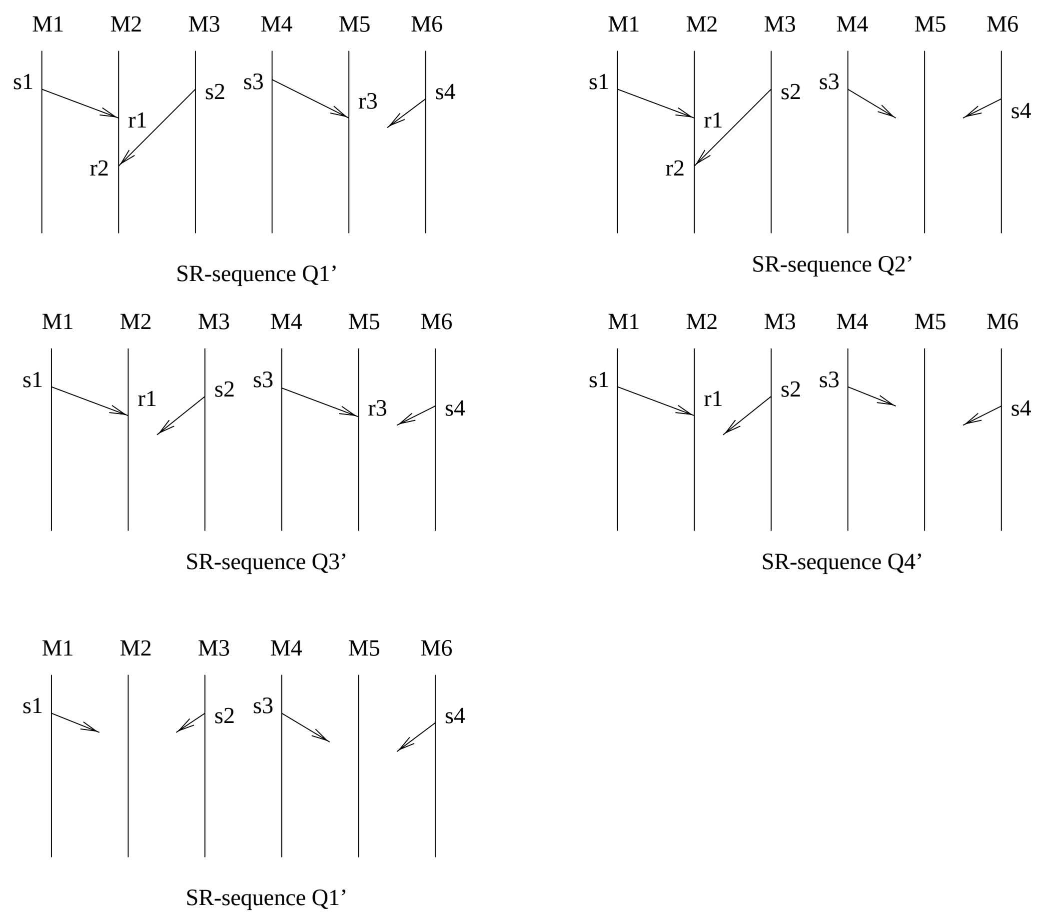
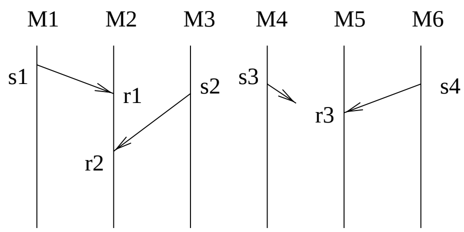
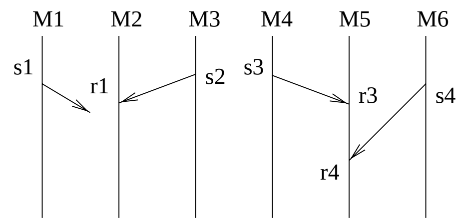


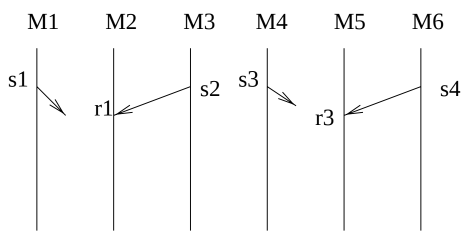
Figure 3.5: Race variants of SR-sequence Q_1



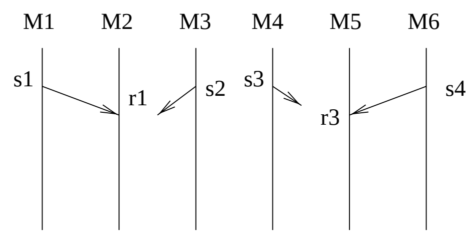
Race Variant V1



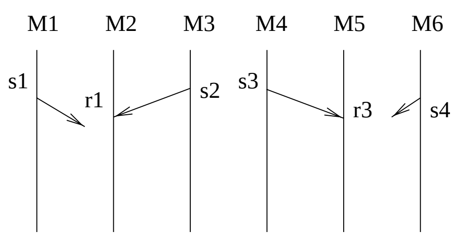
Race Variant V2



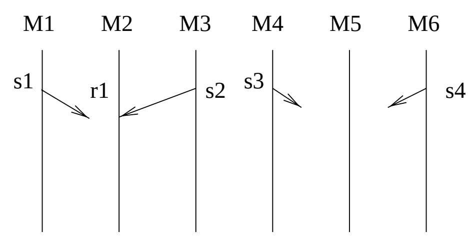
Race Variant V3



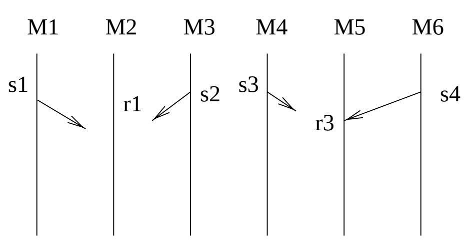
Race Variant V4



Race Variant V5



Race Variant V6



Race Variant V7

Figure 3.6: Algorithm Reachability-Test

```
Algorithm Reachability-Test(P: a program, X: an input)
begin
  execute P with X nondeterministically,
    and let Q0 be the SR-sequence exercised by such an execution
  add Q0 into SR_Tree
  Reachability-Explore(P, X, Q0)
end

function Reachability-Explore(P: a program, X: an input, Q: an SR-sequence)
begin
  race-variants ← Generate-Variants(Q)
  for each race variant V ∈ race-variants do
  begin
    if V is not a prefix of any SR-sequence in SR_Tree do
      perform prefix-based testing of P with X according to V, and let Q′
        be the SR-sequence exercised by such an execution
      add Q′ into SR_Tree
      Reachability-Explore(P, X, Q′)
  end
end
```

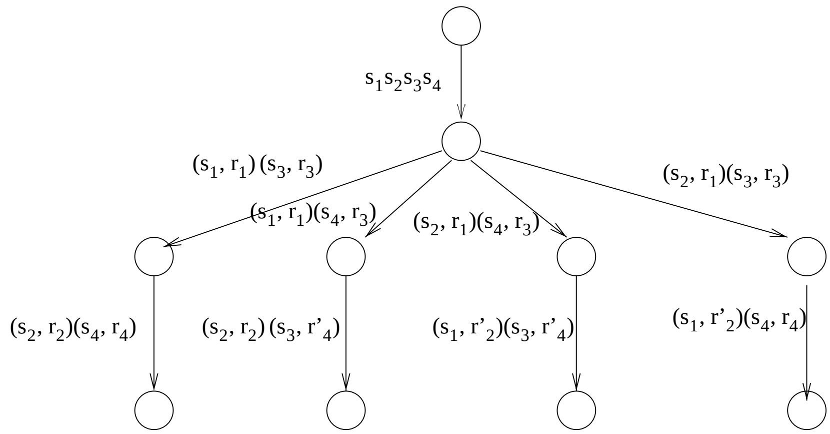
The construction of $[Q]$ is described as follows. $[Q]$ is initialized to be empty. If Q contains no receive events, we construct $\sigma = \omega_1\omega_2 \ldots \omega_n$, where ω_i, $1 \leq i \leq n$, is the sequence of (send) events of process P_i in Q, $|\omega_i| \geq 0$. σ is then appended to $[Q]$. Otherwise, we let $\omega_{i,0}r_{i,1}\omega_{i,1} \ldots r_{i,m_i}\omega_{i,m_i}$ be the event sequence of P_i in Q, where $1 \leq i \leq n$, $m_i \geq 0$, $\omega_{i,k}$, $0 \leq k \leq m_i$, is a send event sequence, $|\omega_{i,k}| \geq 0$, and $r_{i,j}$, $1 \leq j \leq m_i$, is a receive event. We construct $\sigma = \omega_{1,0}\omega_{2,0} \ldots \omega_{n,0}$ and $\gamma = r_{i_1,1}r_{i_2,1} \ldots r_{i_l,1}$, where $1 \leq l \leq n$, letting $0 < h < l$, $1 \leq i_h < i_{h+1} \leq n$, every receive event in γ is concurrent with each other and receives a message sent by a send event in σ. σ and γ are then appended to $[Q]$. Next, we remove from Q all the events in σ, and γ if it exists. If Q becomes empty after such removal, the construction of $[Q]$ is completed and we stop. Otherwise, we repeat the above process. It is easy to see that the canonical form constructed above is unique for a given SR-sequence.

An edge e of SR_Tree represents either a send event sequence or a receive event sequence. We use $seq(e)$ to denote the event sequence represented by an edge e. A path p of SR_Tree consisting of edges e_1, e_2, …, and e_n represents the concatenation of the event sequence represented by each edge e_i, $1 \leq i \leq n$, which is denoted as $seq(p) = seq(e_1) \cdot seq(e_2) \cdot \ldots \cdot seq(e_n)$. In order to add an SR-sequence Q into SR_Tree, we first construct its canonical form $[Q]$. Let $[Q] = \sigma_0\gamma_1\sigma_1\gamma_2 \ldots \gamma_q\sigma_q$, where σ_i, $0 \leq i \leq q$, is a send event sequence, $|\sigma_i| \geq 0$, and γ_j, $1 \leq j \leq q$, a receive event sequence, $|\gamma_j| > 0$. If there exists a path p in SR_Tree such that $seq(p) = [Q]$, then Q is already in SR_Tree and nothing needs to be done. Otherwise, we find the longest path p in SR_Tree such that $seq(p)$ is a prefix of Q. Then, we augment p to create a new path p' such that $seq(p') = [Q]$.

The application of algorithm *Reachability-Test* produces four SR-sequences $Q_1 - Q_4$, as shown in Figure 3.3. It is worth noting that according to Definition 16, r_2 and r_2' are considered as different events, even though they correspond to the same operation. Similarly, r_4 and r_4' are considered as different events. Figure 3.7 shows the SR_Tree generated by algorithm *Reachability-Test* for M.

We comment that the efficiency of algorithm *Reachability-Test* can be improved as follows. In algorithm *Generate-Variants*, we check if, given an SR-sequence Q, $Q \setminus_t tail(Q)$ is a prefix of any SR-sequence in the SR_Tree maintained by algorithm *Reachability-Test*. If so, we only need to generate the race variants related to $tail(Q)$, that is, $variant\_tail(Q)$. (This is because all the race variants of $Q \setminus_t tail(Q)$ must have been generated before.) Otherwise, we need to generate all the race variants of Q, that is, $variant(Q)$.

Figure 3.7: SR_Tree for the example system



3.6 Summary

In this chapter, we have presented an efficient algorithm for reachability testing of asynchronous message-passing programs. We have shown how to derive all race variants of an SR-sequence and how to organize the generated SR-sequences, as well as their race variants, for efficient search and insertion. For an asynchronous message-passing program that terminates on any input, our algorithm executes every partially-ordered path exactly once and thus guarantees the detection of deadlocks and assertion violations.

For an asynchronous message-passing program P that may run forever on some input, we can still apply reachability testing (and other stateless search techniques) except that we need to force P to terminate. One solution is to replace each infinite loop in P with a loop having a finite number of iterations. A different solution is to set a limit on the execution time of P or on the length of generated SR-sequences. Another solution is to apply a cycle prediction method and stops extending an SR-sequence if a predicted cycle is found.

As shown in Section 3.5, the tree structure SR_Tree is used to keep collected SR-sequences and their race variants. The goal of reachability testing is to collect all possible SR-sequences. However, this goal may be impractical for large, complicated message-passing programs. In this case, SR_Tree can be used to select race variants, according to some criteria, for generating new SR-sequences.

In this chapter, we consider asynchronous message-passing programs that do not contain nondeterministic statements. To extend our reachability testing algorithm to programs containing nondeterministic statements, the definition of an SR-sequence needs to be modified to specify the selection of a nondeterministic statement. The control for performing a deterministic execution according to an SR-sequence also becomes more complicated. One example of a nondeterministic statement is the selective wait statement in Ada. How to define and replay a SYN-sequence of an Ada program was shown in [65].

Finally, we address the issue about the construction of tools for supporting reachability testing. As mentioned in Section 3.2, both the controlled execution technique and Godefroid's technique modify the implementation of the underlying operating system or message-passing system. Another approach is to transform a program written in a concurrent language into a slightly different program in the same language such that the modified program supports the collection or replay of a SYN-sequence. Tools based on this approach have been constructed for semaphores, monitors and the Ada language [5] [65]. We are constructing similar tools for asynchronous message-passing programs.

Chapter 4

A Test Generation Strategy for Pairwise Testing

4.1 Background

Pairwise testing (or 2-way testing) is a specification-based testing criterion, which requires that for each pair of input parameters of a system, every combination of valid values of these two parameters be covered by at least one test case. Empirical results show that pairwise testing is practical and effective for various types of software systems [9] [10] [13].

To illustrate the concept of pairwise testing, consider a system with parameters and values as shown below:

- parameter A has values A_1 and A_2,
- parameter B has values B_1 and B_2, and
- parameter C has values C_1, C_2 and C_3.

For parameters A and B, $\{(A_1, B_1)$, (A_1, B_2), (A_2, B_1), $(A_2, B_2)\}$ is the only pairwise test set. For parameters A, B and C, a large number of pairwise test sets exist. Below are three of them with the numbers of tests being 6, 7 and 8 respectively:

- $\{(A_1, B_1, C_1)$, (A_1, B_2, C_2), (A_2, B_1, C_3), (A_2, B_2, C_1), (A_2, B_1, C_2), $(A_1, B_2, C_3)\}$,
- $\{(A_1, B_1, C_1)$, (A_1, B_2, C_1), (A_2, B_1, C_2), (A_2, B_2, C_3), (A_2, B_1, C_1), (A_1, B_2, C_2), $(A_1, B_1, C_3)\}$

- $\{(A_1, B_1, C_1)$, (A_1, B_2, C_1), (A_2, B_1, C_2), (A_2, B_2, C_2), (A_2, B_1, C_1), (A_1, B_1, C_2), (A_1, B_1, C_3), $(A_2, B_2, C_3)\}$.

In this chapter, we propose a new test generation strategy, called in-parameter-order (or IPO), for pairwise testing. For a system with two or more input parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continues to do so for each additional parameter. The extension of an existing pairwise test set for an additional parameter contains the following two steps: (a) horizontal growth, which extends each existing test by adding one value of the new parameter, and (b) vertical growth, which adds new tests, if necessary, after the completion of horizontal growth.

The remainder of this chapter is organized as follows. Section 4.2 discusses related work. Section 4.3 describes the IPO strategy and its advantages. It also shows an optimal algorithm for vertical growth, which can be used in conjunction with any algorithm for horizontal growth. Section 4.4 presents two algorithms for horizontal growth. Section 4.6 describes an IPO-based test generation tool and shows some empirical results. Section 4.7 concludes this chapter.

4.2 Related work

Different test generation strategies have been published for pairwise testing. The strategy proposed in [9] starts with an empty test set and adds one test at a time. To generate a new test, the strategy produces a number of candidate tests according to a greedy algorithm and then selects one that covers the most uncovered pairs. The greedy algorithm to produce a candidate test is described as follows. Assume that a system has n parameters. We first choose a parameter p and then a value l of p such that the parameter value appears in the greatest number of uncovered pairs. Next, we choose a random order for the remaining parameters. Therefore, we have an order for all n parameters $p = p_1$, p_2, $\ldots$, and p_n. Note that the value of p_1 is already selected. In order to select a value for parameter p_i, where $1 < i \leq n$, we consider all possible values of p_i in conjunction with the $i-1$ values already chosen for parameter p_1, $\ldots$, p_{i-1} and choose one that covers the greatest number of new pairs. Since the candidate tests depend on the parameter order that is randomly selected, using a different random seed can produce a different test set. One useful optimization is to generate a number of different test sets using different random seeds, and then choose the best among them. A pairwise test generation tool, called AETG, is developed based on this test generation strategy. We have carried out an empirical study to compare our test generation tool, called PairTest, with AETG. Empirical results indicate that PairTest produces competitive results with the core algorithm of a lower asymptotic complexity. Note that it is shown in [9] that the number of tests for pairwise coverage grows logarithmically in the number of parameters and quadratically in the number of values of each parameter.

Another approach to generating a pairwise test set is to use orthogonal arrays [28] [47]. An orthogonal array is defined as a tuple (c, k, n, t), where c is the number of rows in the array (or the number of tests), k is the number of columns (or the number of parameters), n indicates that each parameter would have at most n values, and t is the strength of the array [73]. An orthogonal array has strength t if in any t columns of the array, all t-way combinations of the values of the corresponding parameters appear at least once and for the same number of times. Therefore, for pairwise testing, we need to generate orthogonal arrays with strengths equal to 2. The method of orthogonal arrays requires that all parameters have the same number of values and that each pair of values be covered the same number of times [47]. The first requirement can be relaxed by adding *don't care* values for missing values. But the use of *don't care* values creates extra tests [74]. The second requirement is considered unnecessary for software testing and also creates extra tests for pairwise testing [8]. The number of tests generated by the method of orthogonal arrays is proportional to the square of the number of parameters [73].

In [73], the method of covering arrays is described, which aims to improve the method of orthogonal arrays. A covering array is defined in the same way as an orthogonal array, except for how the strength is defined. A covering array has strength t if in any t columns of the array, all t-way combinations of the values of the correspond parameters appear at least once. Such a definition effectively removes the second requirement with the method of orthogonal arrays. This method introduces four elementary types of arrays, including B-array, R-array, I-array, and N-array, and uses them as building blocks to construct larger covering arrays by concatenating them vertically and horizontally. Note that B-array and R-array are derived based on orthogonal arrays, I-array is an array containing all ones and N-array is an array that contains a block of twos, vertically concatenated with a block of threes and so on up to n. Empirical results indicate that the method of covering arrays reduces the number of tests to grow logarithmically in the number of parameters. However, as pointed out in [73], further work needs to be done with the method of covering arrays to deal with the situation where there are varying number of values for each parameter.

4.3 In-Parameter-Order (IPO) strategy for generating pairwise test sets

In this section, we describe a test generation strategy, called in-parameter-order (or IPO), for pairwise testing. For a system with two or more input parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continues to do so for each additional parameter. The extension of a test set for the addition of a new parameter includes the following two steps: (a) horizontal growth, which extends each existing test by adding one value of the new parameter, and (b) vertical growth, which adds new tests, if necessary, after the completion of horizontal growth. Section 4.3.1 describes the IPO strategy and its advantages. Section 4.3.2 shows an optimal algorithm for vertical growth, which can be used in conjunction with any algorithm for horizontal growth. Section 4.4 presents two algorithms for horizontal growth.

4.3.1 Framework of the IPO Strategy

Assume that system $\mathcal{S}$ has parameters $p_1, p_2, \ldots$, and p_n, $n \geq 2$. Below is a general description of the IPO strategy for generating a pairwise test set $\mathcal{T}$ for $\mathcal{S}$.

The IPO test generation strategy for pairwise testing provides the following advantages:

- The IPO strategy allows the use of "local" optimization techniques for horizontal and vertical growth in order to develop practical test generation algorithms. (We will show how to do so in sections 4.3.2 and 4.4.)

- Assume that $\mathcal{T}$ is a pairwise test set for a system $\mathcal{S}$. Suppose that a new version of $\mathcal{S}$ has one or more new parameters. By applying the IPO strategy, we can easily extend $\mathcal{T}$ to produce a new pairwise test set $\mathcal{T}'$ for the new version of $\mathcal{S}$. Furthermore, since $\mathcal{T}'$ reuses tests in $\mathcal{T}$, the effort for test preparation according to $\mathcal{T}'$ is less than that according to a new pairwise test set without reusing tests in $\mathcal{T}$.

Figure 4.1: The IPO strategy

```
Strategy In-Parameter-Order
begin
    /* for the first two parameters p1 and p2 */
    T := {(v1, v2) | v1 and v2 are values of p1 and p2 respectively}
    if n = 2 then stop;

    /* for the remaining parameters */
    for parameter pi, i = 3, 4, ..., n do
    begin
        /* horizontal growth */
        for each test (v1, v2, ..., vi-1) in T do
            replace it with (v1, v2, ..., vi-1, vi), where vi is a value of pi;
        /* vertical growth */
        while T does not cover all pairs between pi
            and each of p1, p2, ..., pi-1 do
            add a new test for p1, p2, ..., pi to T;
    end
end
```

- Assume that $\mathcal{T}$ is a pairwise test set for a system $\mathcal{S}$. Suppose that $\mathcal{S}$ has a new version in which some parameters have new values. By applying the vertical growth in the IPO strategy, we can easily extend $\mathcal{T}$ by adding new tests to produce a new pairwise test set $\mathcal{T}'$ for the new version of $\mathcal{S}$. $\mathcal{T}'$ may be larger than a pairwise test set generated by applying the IPO strategy without reusing $\mathcal{T}$. However, $\mathcal{T}'$ can save effort for test preparation due to the reuse of tests in $\mathcal{T}$.

4.3.2 An optimal algorithm for vertical growth

We first use the example system in Section 4.1 to illustrate our algorithm for vertical growth. As mentioned earlier, $\{(A_1, B_1), (A_1, B_2), (A_2, B_1), (A_2, B_2)\}$ is a pairwise test set for parameter A and B. Assume that the horizontal growth for parameter C produces the following tests: (A_1, B_1, C_1), (A_1, B_2, C_1), (A_2, B_1, C_2), and (A_2, B_2, C_2). These four tests do not cover pairs $(A_1, C_2), (A_1, C_3), (A_2, C_1), (A_2, C_3), (B_1, C_3)$ and (B_2, C_3). So we need to generate new tests to cover these pairs in vertical growth. (A_1, C_2) is covered by test $(A_1, -, C_2)$, where "$-$" denotes *don't care*, meaning that the value does not need to be specified for covering any pair. Similarly, (A_1, C_3), (A_2, C_1), and (A_2, C_3) are covered by $(A_1, -, C_3)$, $(A_2, -, C_1)$ and $(A_2, -, C_3)$ respectively. To cover (B_1, C_3), we change $(A_1, -, C_3)$ to (A_1, B_1, C_3) without adding a new test. Similarly, we cover (B_2, C_3) by changing $(A_2, -, C_3)$ to (A_2, B_2, C_3). Thus, we generate four new tests to cover the six missing pairs. (Note that it is impossible to use less than four tests to cover the six missing pairs.) So the generated pairwise test set has a total of eight tests.

Assume that the horizontal growth for parameter p_i has produced a test set $\mathcal{T}$ for $p_1, p_2, \ldots$, and p_i. (Thus, $\mathcal{T}$ is resulted from the extension of a pairwise test set for $p_1, p_2, \ldots$ and p_{i-1}.) Let π be the set of missing pairs according to $\mathcal{T}$. Each pair in π contains a value of p_i and a value of one of $p_1, p_2, \ldots, p_{i-1}$. Assume that $|\pi| > 0$. Let d_j, $1 \leq j \leq i$, be the domain size of parameter p_j and $m_j = max\{d_k | 1 \leq k \leq j\}$. The following algorithm for vertical growth shows how to construct a minimum test set $\mathcal{T}'$ such that $\mathcal{T} \cup \mathcal{T}'$ is a pairwise test set for $p_1, p_2, \ldots$ and p_i.

In the above algorithm, m_i tests are first initialized with "$-$"s for each value of p_i. At most but often less than m_i tests are needed to cover all the missing pairs associated with a particular value of p_i. θ is therefore maintained to keep track of actual numbers of tests generated to cover all the missing pairs associated for each value of p_i.

After the completion of algorithm IPO_V, $\mathcal{T}'$ may contain "$-$" values. If p_i is the last parameter, each "$-$" value for p_k, $1 \leq k \leq i$, is replaced by any value of p_k. Otherwise, these "$-$" values are replaced by parameter values in the horizontal growth for p_{i+1} as described in Section 4.4.

Assume that p_i contains values $u_1, u_2, \ldots$ and u_q. For $1 \leq j \leq q$, let π_j be the set of pairs in π that contain u_j as the value of p_i. If $|\pi_j| > 0$, let π_{jm}, $1 \leq m \leq i-1$, be the set of pairs of π_j that contain a value of p_m. The number of new tests according to π_j is $max\{|\pi_{jm}| | m = 1, 2, \ldots, i-1\}$. The set of new tests according to π_j is minimum, since it is impossible to cover pairs in π_j with less number of tests. Furthermore, the set of new tests according to π_j cannot cover pairs that are in π, but not in π_j. Thus, the above algorithm constructs a minimum set of new tests according to π. Therefore, we have the following theorem.

Theorem 4.3.1 *Let S be a system with parameters $p_1, p_2, \ldots,$ and p_i. Assume that according to the IPO strategy, the vertical growth for parameter p_i has produced a test set $\mathcal{T}$ for $p_1, p_2, \ldots,$ and p_i. Let π be the set of missing pairs according to $\mathcal{T}$. Algorithm IPO_V*

Figure 4.2: Algorithm IPO_V

```
algorithm IPO_V(T, π)
begin
    let A be a two-dimension (d_i × m_i) array of tests initialized with tests
        that have all the values as "–";
    let θ be an array of integers of size d_i initialized with 0;
    for each pair (p_k.w, p_i.u) in π do
    begin
        let u be the jth value of p_i;
        /* check if this pair can be covered by an existing test */
        for l = 1 to θ[j] do
        begin
            if A[j][l] has "–" as the value of p_k and "u" as the value of p_i
            then
            begin
                modify A[j][l] by replacing the "–" with w;
                break;
            end
        end
        /* add a new test to cover this pair */
        if l > θ[j] then
        begin
            θ[j] + +;
            modify A[j][θ[j]] such that w is the value of p_k and u is the value of p_i;
        end
    end;
    /* collect generated tests into the result set */
    let T′ be an empty set;
    for r = 0 to d_i do
    begin
        for s = 0 to θ[r] do
        begin
            add A[r][s] into T′;
        end
    end
    return T′;
end;
```

constructs a minimum set $\mathcal{T}'$ of new tests according to π such that $\mathcal{T} \cup \mathcal{T}'$ is a pairwise test set for S.

The time complexity of algorithm IPO_V is dominated by the first *for* loop, which has a nested loop. The size of π is at most $d_i * \sum_{j=1}^{i-1} d_j$, which is less than or equal to $m_i^2 * (i-1)$. For each pair $(p_k.u, p_i.v)$ in π, the nested loop searches the tests with v as the value of p_i to determine if there exists one with "$-$" as the value of p_k among those tests. There are at most m_i tests with v as the value of p_i, each of which can be checked in $O(1)$. So the nested loop takes $O(m_i)$. The time complexity of algorithm IPO_V is therefore $O(m_i^3 i)$. For a system with n parameters, algorithm IPO_V is executed for $i = 3$ to n. Let $d = max\{d_i | 1 \leq i \leq n\}$. Thus, the total execution time for algorithm IPO_V is $O(d^3 n^2)$.

Theorem 4.3.2 *Let S be a system with n parameters and let d be the maximum domain size of these parameters. According to the IPO strategy, the size of the pairwise test set generated for S is at most $d^2 * (n-1)$.*

Proof For $n = 2$, the size of the generated pairwise test set is at most d^2. For $n = 3$, horizontal growth does not increase the number of tests generated for $n = 2$, and vertical growth adds at most d^2 tests. So the size of the generated test set for $n = 3$ is at most $d^2 * 2$. By using the same argument, the size of the generated pairwise test set for any system with n parameters is at most $d^2 * (n-1)$. □

In practice, as shown in Section 4.6, the sizes of test sets generated by PairTest are $O(d^2 \log(n))$.

4.4 Two algorithms for horizontal growth

Assume that $\mathcal{T}$ is a pairwise test set for parameters $p_1, p_2, \ldots,$ and p_{i-1}. As mentioned earlier, the horizontal growth for parameter p_i is to replace each test $(v_1, v_2, \ldots, v_{i-1})$ in $\mathcal{T}$ with $(v_1, v_2, \ldots, v_{i-1}, v_i)$, where v_i is a value of p_i. Thus, the horizontal growth for p_i is to construct a list of $|\mathcal{T}|$ values of parameter p_i such that the jth value of the list, $1 \leq j \leq |\mathcal{T}|$, is the value added to the jth test in $\mathcal{T}$. Such a list is referred to as a permutation of parameter p_i for $\mathcal{T}$. There are $|\mathcal{D}(p_i)|^{|\mathcal{T}|}$ possible permutations of parameter p_i for $\mathcal{T}$. For a permutation ζ of parameter p_i for $\mathcal{T}$, let $(\mathcal{T}, \zeta)$ denote the test set resulted from adding the jth element of ζ, $1 \leq j \leq |\mathcal{T}|$, to the jth test in $\mathcal{T}$.

Ideally, we hope to find a permutation ζ of parameter p_i for $\mathcal{T}$ such that $(\mathcal{T}, \zeta)$ is a minimum pairwise test set for $p_1, p_2, \ldots,$ and p_i. If such a permutation does not exist, we need to choose a permutation ζ of parameter p_i for $\mathcal{T}$ such that the set of new tests constructed by algorithm IPO_V (shown in Section 4.3.2) according to the set of missing pairs based on $(\mathcal{T}, \zeta)$ is minimum. Since there are $|\mathcal{D}(p_i)|^{|\mathcal{T}|}$ possible permutations of parameter p_i for $\mathcal{T}$, finding an "optimal" permutation by exhaustive search is very time-consuming. In section 4.1, we show an algorithm for horizontal growth, called IPO_H_EC ("EC" stands for "equivalence classes"). This algorithm finds an optimal permutation of parameter p_i for $\mathcal{T}$ without using exhaustive search. However, algorithm IPO_H_EC still has exponential time complexity. In section 4.2, we show algorithm IPO_H_IV ("IV" stands for "individual values"), which is practical to use.

4.4.1 Algorithm IPO_H_EC

Definition 28 *Let $\mathcal{T}$ be a pairwise test set for parameters $p_1, p_2, \ldots,$ and p_{i-1}. For a permutation ζ of parameter p_i for $\mathcal{T}$, let the weight of ζ with respect to $\mathcal{T}$, or weight$(\zeta, \mathcal{T})$, be defined as the number of new tests constructed by algorithm IPO_V according to the set of missing pairs based on $(\mathcal{T}, \zeta)$.*

The motivation for algorithm for IPO_H_EC is that the set of permutations of parameter p_i with size $|\mathcal{T}|$ can be divided into equivalent classes such that permutations in the same class have the same weight with respect to $\mathcal{T}$. Thus, finding an optimal permutation of parameter p_i for $\mathcal{T}$ requires only the use of one permutation from each equivalent class. Below we show several definitions and theorems that are necessary for understanding algorithm IPO_H_EC.

Definition 29 *Let $\zeta = (u_1, u_2, \ldots, u_n)$ and $\xi = (v_1, v_2, \ldots, v_n)$ be two permutations of a parameter α. An* onto mapping *from ζ to ξ is a 1-to-1 function ψ from ζ to ξ such that $v_i = \psi(u_i)$.*

Definition 30 *Two different permutations ζ and ξ of a parameter are said to be equivalent, noted as $\zeta \equiv \xi$, provided that there exists an onto mapping from ζ to ξ or vice versa.*

For the example system in Section 4.1, consider the following two permutations of parameter C: $C' = (C_1, C_1, C_2, C_3)$ and $C'' = (C_2, C_2, C_1, C_3)$. According to the above definitions, the two permutations are equivalent due to the existence of the following 1-to-1 function ψ from C' to C'':

$$\psi(u) = \begin{cases} C_2 & if\ u = C_1 \\ C_1 & if\ u = C_2 \\ C_3 & if\ u = C_3 \end{cases}$$

Theorem 4.4.1 *Let $\mathcal{T}$ be a pairwise test set for parameters $p_1, p_2, \ldots,$ and p_{k-1}. For two permutations ζ and ξ of parameter p_i with respect to $\mathcal{T}$, if ζ and ξ are equivalent, then $weight(\zeta, \mathcal{T}) = weight(\xi, \mathcal{T})$.*

Proof

Assume p_i has d valid values $v_1, v_2, \ldots, v_d$. Let π be the set of missing pairs if we add ζ into $\mathcal{T}$. Let π' be the set of missing pairs if we add ξ into $\mathcal{T}$. We make a distribution of π and get φ_{ij}, where $1 \leq i \leq d$ and $1 \leq j \leq k$, to represent the set of missing pairs between p_i and p_j with the value v_i for p_i. We also make similar distribution of π' and get φ'_{ij}. Since $\zeta \equiv \xi$, there exists an onto mapping ψ from ζ to ξ. Let $v_l = \psi(v_i)$, then $|\varphi_{ij}| = |\varphi'_{lj}|$. Further, $\sum_{i=1}^{d} max_{j=1}^{k}\{|\varphi_{ij}|)\} = \sum_{l=1}^{d} max_{j=1}^{k}\{|\varphi'_{lj}|)\}$. We thus conclude that $weight(\zeta, \mathcal{T}) = weight(\xi, \mathcal{T})$.

□

To illustrate the above theorem, consider again the example system in Section 4.1. $\mathcal{T} = \{(A_1, B_1),\ (A_1, B_2),\ (A_2, B_1),\ (A_2, B_2)\}$ is a pairwise test set for parameters A and B. As shown earlier, $C' = (C_1, C_1, C_2, C_3)$ and $C'' = (C_2, C_2, C_1, C_3)$ are equivalent. To calculate $weight(C', \mathcal{T})$, we construct $(\mathcal{T}, C')$: $\{(A_1, B_1, C_1), (A_1, B_2, C_1), (A_2, B_1, C_2), (A_2, B_2, C_3)\}$. The set of missing pairs is $\{(A_2, C_1), (A_1, C_2), (B_2, C_2), (A_1, C_3), (B_1, C_3)\}$, which is covered by the new test set $\mathcal{T}' = \{(A_2, -, C_1), (A_1, B_2, C_2), (A_1, B_1, C_3)\}$. Thus, $weight(C', \mathcal{T}) = 3$. To calculate $weight(C'', \mathcal{T})$, we construct $(\mathcal{T}, C'')$, which is $\{(A_1, B_1, C_2),\ (A_1, B_2, C_2),\ (A_2, B_1, C_1),\ (A_2, B_2, C_3)\}$. The set of missing pairs is $\{(A_2, C_2), (A_1, C_1), (B_2, C_1), (A_1, C_3), (B_1, C_3)\}$, which is covered by the new test set $\mathcal{T}'' = \{(A_2, -, C_2), (A_1, B_2, C_1), (A_1, B_1, C_3)\}$. Thus, $weight(C'', \mathcal{T}) = 3$. Note that although $\mathcal{T}' \neq \mathcal{T}''$, $|\mathcal{T}'| = |\mathcal{T}''| = 3$.

Theorem 4.4.2 *Let ζ and ξ be different permutations of parameter α such that $\zeta = (u_1, u_2, \ldots, u_n)$, $\xi = (v_1, v_2, \ldots, v_n)$, and ζ and ξ are equivalent. For each w in the domain $\mathcal{D}$ of parameter α, there are no i and j $(1 \leq i, j \leq n)$, such that $w = u_i = v_i \wedge w = u_j \neq v_j$.*

Proof

Since $\zeta \equiv \xi$, there exists an 1-to-1 function ψ from ζ to ξ. Since $u_i = u_j = w$, then $\psi(u_i) = \psi(u_j)$, that is, $v_i = v_j$. We thus complete the proof.

□

Definition 31 *Assume that parameter α has domain $\mathcal{D}$. Let $\mathcal{M}$ be the set of all α's permutations with size m. Assume that a dummy parameter has a domain of boolean values: 0 and 1. Let $\mathcal{B}$ be the set of all the dummy parameter's permutations with size m. A project function $\varphi : \mathcal{M} \times \mathcal{D} \to \mathcal{B}$ is defined as follows. Let $\zeta = (u_1, u_2, \ldots, u_m) \in \mathcal{M}$ and $v \in \mathcal{D}$. Let $\varrho = \varphi(\zeta, v)$. Then, $\varrho = (b_1, b_2, \ldots, b_m) \in \mathcal{B}$, where*

$$b_i = \begin{cases} 1 & u_i = v \\ 0 & u_i \neq v \end{cases}$$

Definition 32 *Given a project function $\varphi : \mathcal{M} \times \mathcal{D} \to \mathcal{B}$. Let $v \in \mathcal{D}$. The (appearance) pattern set of v for all α's permutations with size m is defined as $[\mathcal{B}]_v^m = \{\varrho \in \mathcal{B} | \varrho = \varphi(\zeta, v) \wedge \zeta \in \mathcal{M}\}$. ϱ is also referred as an appearance pattern, or a pattern, of v.*

Definition 33 *Suppose a parameter α has the (appearance) pattern set $[\mathcal{B}]_v^m$ for a valid value v for all permutations with size m. Let $\varrho = (b_1, b_2, \ldots, b_m) \in [\mathcal{B}]_v^m$. The weight of ϱ is $|\varrho| = \sum_{i=0}^{m} b_i$. We also define $[\mathcal{B}]_v^{(m,i)} = \{\varrho | \varrho \in \mathcal{B} \wedge |\varrho| = i\}$, $[\mathcal{B}]_v^{(m,i)}|_{b_j = v_k} = \{\varrho | \varrho \in \mathcal{B} \wedge |\varrho| = i \wedge b_j = v_k\}$.*

By Theorem 4.4.1, finding an optimal permutation of parameter p_i for $\mathcal{T}$ requires only the use of one permutation from each equivalent class of permutations of size $|\mathcal{T}|$ of p_i. Thus, the number of permutations considered for choosing an optimal permutation can be significantly reduced. The following theorem shows how to determine the number of equivalent classes of permutations of size $|\mathcal{T}|$ of p_i .

Theorem 4.4.3 *Assume that parameter α has domain $\mathcal{D}$, where $|\mathcal{D}| = d$. Let $\mathcal{M}$ be the set of all α's permutations with size m. Let $\mathcal{N}(m, d)$ be the number of equivalence classes in $\mathcal{M}$.*

$$\mathcal{N}(m,d) = \begin{cases} 1 & if\ d{=}1,\ m{=}1\ or\ 0 \\ \sum_{i=1}^{m} (C_{m-1}^{i-1} * \mathcal{N}(m-i, d-1)) & otherwise \end{cases}$$

Proof

Let us consider the boundary cases first. When $m = 1$, each permutation consists of a single value of α. Obviously, all single-value permutations are equivalent. Then, $\mathcal{N}(1, d) = 1$. When $d = 1$, the parameter has the only value, say v, to take. Regardless of m, the only permutation is formed by filling in all of the positions by v. Thus, $\mathcal{N}(m, 1) = 1$.

Now we consider the cases where $m > 1$ and $d > 1$. We prove by induction. Assume that we have derived $\mathcal{N}(x, y)$'s for $x < m$ and $y < d$. We are going to derive $\mathcal{N}(m, d)$. Before we proceed, we define two functions: (1) $\sigma : \mathcal{M} \rightarrow \mathcal{D}$, where $\sigma(\zeta) = v_1$ for $\zeta = (v_1, v_2, \ldots, v_m) \in \mathcal{M}$; (2) $\psi_{u \rightarrow v} : \mathcal{M} \rightarrow \mathcal{M}$, where $\psi_{u\rightarrow v}(u_1, u_2, \ldots, u_m) = (v_1, v_2, \ldots, v_m)$ with $v_i = v$ if $u_i = u$, $v_i = u$ if $u_i = v$ and $v_i = u_i$ otherwise.

Suppose α has d values $v_1, v_2, \ldots, v_d$. We divide $\mathcal{M}$ into d groups $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_d$, where $\mathcal{M}_i = \{\zeta \in \mathcal{M} | \sigma(\zeta) = v_i\}$. We show that $\forall i, j \forall \zeta \exists \xi (\zeta \in \mathcal{M}_j \wedge \xi \in \mathcal{M}_i \wedge \zeta \equiv \xi)$. Given $\zeta \in \mathcal{M}_j$, let $\xi = \psi_{v_j \rightarrow v_i}(\zeta)$. Obviously, $\xi \equiv \zeta$. Since $\sigma(\xi) = \sigma(\psi_{v_j\rightarrow v_i}(\zeta)) = v_i$, $\xi \in \mathcal{M}_i$. Intuitively, if we pick up one group $\mathcal{M}_i$, then $\forall \zeta \in \mathcal{M}_j (j \neq i)$, we can always locate a permutation $\xi \in \mathcal{M}_i$, which makes $\xi \equiv \zeta$. By Theorem 4.4.1, we conclude that it is sufficient for us to examine only one group of permutations.

Without loss of generality, we pick up $\mathcal{M}_1$. Let $|\zeta_{v_1}|$ be the number of v_1's in ζ, where $\zeta \in \mathcal{M}_1$. Then, $1 \leq |\zeta_{v_1}| \leq m$. We make a distribution of $\mathcal{M}_1$ and get $\mathcal{M}_{1i}$ where $\mathcal{M}_{1i} = \{\zeta \in \mathcal{M}_1 || \zeta_{v_1}| = i\}$. Let us consider $\mathcal{M}_{1i}$. Then, $\forall \zeta(\zeta \in \mathcal{M}_{1i} \wedge \sigma(\zeta) = v_1 \wedge |\zeta_{v_1}| = i)$. By the theory of combination, $|[\mathcal{B}]_{v_1}^{(m,i)}|_{b_1=v_1}| = C_{m-1}^{i-1}$ considered that the first position is always occupied by v_1. Obviously, for any pattern $\varrho \in [\mathcal{B}]_{v_1}^{(m,i)}|_{b_1=v_1}$, ϱ fixes i positions by v_1, but leaves $m-i$ positions unfixed. Note that v_1 does not appear in the unfixed positions. Thus, ϱ has $\mathcal{N}(m-i, d-1)$ permutations by our notation.

Note that all the permutations in $\mathcal{M}_1$ has different (appearance) patterns. Consider any two permutations $(u_1, u_2, \ldots, u_3)$ and $(v_1, v_2, \ldots, v_3)$. Since v_1 always shows up in the first position, there must be some position i such that $u_i = v_1$ (or $v_i = v_1$) but $u_i \neq v_i$. By Theorem 4.4.2, no permutations in $\mathcal{M}_1$ are equivalent. Therefore, the number of equivalence classes in $\mathcal{M}_1$ is $|M|_{equiv} = \mathcal{N}(m, d) = \sum_{i=1}^{m} \mathcal{M}_{1i} = \sum_{i=1}^{m}(C_{m-1}^{i-1} * \mathcal{N}(m-i, d-1))$.

$\square$

As an example, for a parameter having 4 values, the number of different permutations of size 10 is $4^{10} = 1048576$, but the number of equivalence classes of these permutations is $\mathcal{N}(10, 4) = 43947$. Since $\mathcal{N}(m, d) \geq 2^{m-1}$, $\mathcal{N}(m, d)$ is an exponential function of m. For a given parameter and permutation size, the following theorem shows how to construct a set of permutations such that this set contains exactly one permutation from each equivalence class.

Definition 34 *Assume a pattern $\varrho = (b_1, b_2, \ldots, b_m)$ of a value of w. Let $|\varrho|_i = \sum_{j=0}^{i} b_j$. (Obviously, $|\varrho|_m = |\varrho|$.) Let $\mathcal{B}$ be a set of patterns of size m. Let $\mathcal{M}$ be a set of permutations of size $m - |\varrho|$ and $\mathcal{M}'$ a set of permutations of size m. We define an operator $* : \mathcal{B} \times \mathcal{M} \to \mathcal{M}'$ as follows. Let $\zeta \in \mathcal{M}$, $\xi \in \mathcal{M}'$, and $\xi = \varrho * \zeta$, Then, $\xi = (v_1, v_2, \ldots, v_m)$, where*

$$v_i = \begin{cases} w & b_i = 1 \\ u_{i-|\varrho|_i} & b_i = 0 \end{cases}$$

Theorem 4.4.4 *Assume that parameter α has domain $\mathcal{D}$, where $|\mathcal{D}| = d$. Let $\mathcal{D} = \{v_1, v_2, \ldots, v_d\}$ and $\mathcal{D}_i = \{v_{d-i+1}, v_{d-i+2}, \ldots, v_d\}$, where $1 \le i \le d$. Let $\mathcal{E}(m, \mathcal{D})$ be defined as follows:*

$$\mathcal{E}(m, \mathcal{D}) = \begin{cases} \{\} & if\ m = 0 \\ \{(v_1)\} & if\ m = 1 \\ \{\overbrace{(v_1, v_1, \ldots, v_1)}^{m}\} & if\ |\mathcal{D}| = 1 \\ \cup_{i=1}^{m}([\mathcal{B}]_{v_1}^{(m,i)}|_{b_1=v_1} * \mathcal{E}(m-i, \mathcal{D}_{d-1})) & otherwise \end{cases}$$

$\mathcal{E}(m, \mathcal{D})$ contains exactly one permutation of α with size m from each equivalence class of permutations of α with size m.

Proof

The conclusion can be easily drawn with being viewed as a constructive restatement of Theorem 4.4.3.

□

For the example system in Section 4.1, since the only pairwise test set for parameters A and B has four tests, we need to consider permutations of size 4 of parameter C. There are a total of 81 (3^4) such permutations, but they can be divided into 14 equivalent classes according to Theorem 4.4.3. Below are 14 permutations of size 4 of parameter C, one from each equivalent class: $\{(C_1, C_1, C_1, C_1)$, (C_1, C_1, C_1, C_2),(C_1, C_1, C_2, C_1), (C_1, C_2, C_1, C_1), (C_1, C_1, C_2, C_2), (C_1, C_1, C_2, C_3), (C_1, C_2, C_1, C_2), (C_1, C_2, C_1, C_3), (C_1, C_2, C_2, C_1), (C_1, C_2, C_3, C_1), (C_1, C_2, C_2, C_2), (C_1, C_2, C_2, C_3), (C_1, C_2, C_3, C_2), $(C_1, C_2, C_3, C_3)\}$. We choose one of these 14 permutations that has the minimum weight, and this permutation has the minimum weight among all possible 81 permutations. Note that for the example in Section 4.3.2, the horizontal growth for parameter C uses the permutation (C_1, C_1, C_2, C_2), which has 4 as its weight. It was shown earlier that the weight for (C_1, C_1, C_2, C_3) is 3. Later we will show that the weight for (C_1, C_2, C_3, C_1) is 2, which is the minimum weight.

Below we show algorithm IPO_H_EC for horizontal growth. Assume that $\mathcal{T}$ is a pairwise test set for $p_1, p_2, \ldots$, and p_{i-1}. The main loop of IPO_H_EC finds a permutation with the minimum weight, which is used to perform the horizontal growth for parameter p_i. In the end of algorithm IPO_H_EC, $\mathcal{T}$ is replaced with $(\mathcal{T}, \zeta)$. Note that $Partition(p, m)$ returns a set of permutations of parameter p_i with size m according to Theorem 4.4.4.

4.4.2 Algorithm IPO_H_IV

Since algorithm IPO_H_EC has exponential time complexity, it is not practical for systems with a large number of parameters. In this section, we show a horizontal growth algorithm that is practical to use, but may produce more tests than algorithm IPO_H_EC. Assume that $\mathcal{T}$ is a pairwise test set for parameters $p_1, p_2, \ldots$, and p_{i-1}. Initially, the set π of missing pairs contains all possible pairs between values of p_i and $p_1, p_2, \ldots$, and p_{i-1}. For each test in $\mathcal{T}$, we consider all values of p_i, choose one that covers the most number of pairs in π, extend the test by adding the chosen value of p_i, and remove from π pairs covered by the extended test. Thus, we incrementally construct a permutation of p_i with size $|\mathcal{T}|$ by choosing one value of p_i for each test in T.

Assume that the domain of p_i contains values $v_1, v_2, \ldots$, and v_q. If $|\mathcal{T}| \leq q$, we can simply choose v_j, $1 \leq j \leq |\mathcal{T}|$, as the new value for the jth test in $\mathcal{T}$, since the resulting test covers exactly $i-1$ missing pairs. (In fact, we can choose any permutation of p_i of size $|\mathcal{T}|$ such that all elements of this permutation are distinct.) If $|\mathcal{T}| > q$, for $1 \leq j \leq q$, we choose v_j as the new value for the jth test in T, and for $j > q$, we consider all values of p_i and choose one that covers the most number of missing pairs.

Now we apply the above discussion to the example system in the beginning of this chapter. $\{(A_1, B_1), (A_1, B_2), (A_2, B_1), (A_2, B_2)\}$ is the only pairwise test set for A and B. Since C has three values C_1, C_2 and C_3, we extend (A_1, B_1), (A_1, B_2) and (A_2, B_1) by adding C_1, C_2 and C_3, respectively. The extended tests are (A_1, B_1, C_1), (A_1, B_2, C_2) and (A_2, B_1, C_3), and the resulting set of missing pairs is $\{(A_2, C_1)$, (B_2, C_1), (A_2, C_2), (B_1, C_2),

Figure 4.3: Algorithm IPO_H_EC

algorithm *IPO_H_EC*($\mathcal{T}$, p_i)
begin
    /* Initialization */
    $\zeta = null$;
    $\omega = \infty$;
    /* Main Loop */
    $\mathcal{M} := Partition(p_i, |\mathcal{T}|)$;
    for each permutation ζ' in $\mathcal{M}$ **do**
    begin
        $\omega' := weight(\zeta', \mathcal{T})$;
        if $\omega' \leq \omega$ **then**
            $\zeta = \zeta'$;
            $\omega = \omega'$;
    end
    replace $\mathcal{T}$ with $(\mathcal{T}, \zeta)$;
end

function *Partition*(p, m)
begin
  let $\mathcal{D} = \{v_1, v_2, \ldots, v_{|\mathcal{D}|}\}$ be the domain of p;
  for $i = 1$ **to** m
    $\mathcal{E}(i, \mathcal{D}_1) = \{(v_1)\}$
  for $i = 2$ **to** $|\mathcal{D}|$
    for $j = 1$ **to** m
      $\mathcal{E}(i, \mathcal{D}_j) = \{\}$;
      for $k = 1$ **to** j
        $\mathcal{E}(i, \mathcal{D}_j) = \mathcal{E}(i, \mathcal{D}_j) \cup ([\mathcal{B}]_{v_i}^{(j,k)}|_{b_1 = v_i} * \mathcal{E}(j - k, \mathcal{D}_{j-1}))$;
end

(A_1, C_3), $(B_2, C_3)\}$. Now we need to choose one of C_1, C_2, and C_3 for (A_2, B_2). If we add C_1 to (A_2, B_2), the extended test (A_2, B_2, C_1) covers two missing pairs (A_2, C_1) and (B_2, C_1). If we add C_2 to (A_2, B_2), the extended test (A_2, B_2, C_2) covers only one missing pair (A_2, C_2). If we add C_3 to (A_2, B_2), the extended test (A_2, B_2, C_3) covers only one missing pair (B_2, C_3). Thus we choose (A_2, B_2, C_1) as the fourth test. As a result, we have four missing pairs: (A_2, C_2), (B_1, C_2), (A_1, C_3), and (B_2, C_3). By applying algorithm IPO_V, we generate tests (A_2, B_1, C_2) and (A_1, B_2, C_3) to cover these missing pairs.

Assume that $\mathcal{T}$ is a pairwise test set for parameters $p_1, p_2, \ldots,$ and p_{i-1}. Figure 4.4 shows the complete algorithm, namely IPO_H_IV, which extends $\mathcal{T}$ to generate a pairwise test set for parameters $p_1, p_2, \ldots,$ and p_i.

In algorithm IPO_H_IV, $\mathcal{T}$ is also treated as a list with elements in arbitrary order. For $1 \leq j \leq i$, d_j is the domain size of parameter p_j and $m_j = max\{d_k | 1 \leq k \leq j\}$. For the reasons of efficiency, a coverage matrix λ and a pointer matrix θ are maintained. The former is used to determine whether a pair is covered or not in $O(1)$. Given a pair $(p_k.u, p_i.v)$, where assume that u is the ith value of parameter p_k and v the jth value of parameter p_i, $(p_k.u, p_i.v)$ is covered if and only if $\lambda[j][k][i] = true$. The latter is used to deal with "$-$" values. $\theta[j][k] = l$ means that the pair $(p_k.u, p_i.v)$, where u is the lth value of parameter p_k and v the jth value of parameter p_i, has not been covered yet. Therefore, a "$-$" value of parameter p_k can be replaced with the lth value of parameter p_k to cover a new pair. Note that function *mark* updates λ and θ when the jth test in $\mathcal{T}$ is extended, and function *eval* determines how many pairs would be covered if the jth test in $\mathcal{T}$ is extended by v.

Now we consider the time complexity of algorithm IPO_H_IV. The *mark* function is used to maintain λ and θ whenever a test is extended by adding one value of p_i. For each pair, *mark* takes $O(1)$ to update λ, and $O(m_i)$ to update θ. Considering that an extended test has $i-1$ pairs, *mark* has the time complexity of $O(m_i * i)$. The *eval* function determines the number of new pairs covered by a test if the test were extended by a given value of p_i. It takes $O(1)$ for *eval* to check one pair, and $O(i)$ for all the $i-1$ pairs covered by a test. The time complexity of algorithm IPO_H_IV is dominated by the second *for* loop, which has a nested *for* loop. The nested loop takes $O(i * m_i)$, since d_i values of p_i are evaluated, each of which takes $O(1)$ to evaluate. The invocation of *mark* takes $O(i * m_i)$ as mentioned earlier. Assume that the sizes of test sets generated by IPO are $O(m_i^2 * \log(i))$, the time complexity of algorithm IPO_H_IV is $O(m_i^2 * \log(i) * m_i * i) = O(m_i^3 * \log(i) * i)$. For a system with n parameters, algorithm IPO_H_IV is executed for $i = 3$ to n. Let $d = max\{d_i | 1 \leq i \leq n\}$.

Figure 4.4: Algorithm IPO_H_IV

```
algorithm IPO_H_IV(T, p_i)
begin
   /* initialization */
   let λ be a three-dimension d_i × (i − 1) × m_i array of booleans initialized with false
   let θ be a two-dimension d_i × (i − 1) array of integers initialized with 0
   s = min(|D(p_i)|, |T|);
   /* no need to select values of p_i */
   for j = 1 to s do
   begin
      extend the jth test in T by adding the jth value of p_i;
      mark(T, p_i, j, λ, θ);
   end;
   if s = |T| then return;
   /* necessary to select values of p_i */
   for j = s + 1 to |T| do
   begin
      w' := 0; v' := 0;
      for each value v in p_i do
      begin
         w = eval(T, p_i, v, j, λ, θ);
         if w' < w
         begin w' := w; v' := v; end;
      end;
      extend the jth test in T by adding value v';
      mark(T, p_i, j, λ, θ);
   end;
end;
```

Figure 4.5: Function *mark*

```
function mark(T, p_i, j, λ, θ)
begin
   let τ be the jth test in T;
   let v be the value of p_i in τ, assuming that v is the kth value of p_i;
   for l = 1 to i − 1 do
   begin
      let w be the value of p_l in τ, assuming that w is the nth value of p_l;
      if w ≠ "−"
      then λ[k][l][n] = true;
      else
         q = θ[k][l];
         if q ≠ −1
            replace w as the qth value of p_l;
            for t = q + 1 to d_l do
               if !λ[k][l][t] break;
            if t ≤ d_l
            then update θ[k][l] with the tth value of p_l;
            else θ[k][l] := −1;
   end
end
```

Figure 4.6: Function *eval*

```
function eval(T, p_i, v, j, λ, θ)
begin
   r = 0;
   let τ be the jth test in T;
   assume v be the kth value of p_i;
   for l = 1 to i − 1 do
   begin
      let w be the value of p_l in τ, and assume w be the nth value of p_l;
      if w ≠ "−"
      then
         if λ[k][l][n] = false then r++;
      else
         if θ[k][l] ≠ −1 then r++;
   end
   return r;
end
```

Thus, the total execution time for algorithm IPO_H_IV is $O(d^3 * n^2 * \log(n))$.

4.5 Test generation with relations and constraints

In practice, relations and constraints are often used to allow variation of pairwise testing according to the user's needs [9]. A relation specifies a set of parameters and a set of valid values for each parameter. A constraint for a relation specifies a set of invalid tests for that relation. Let $\mathcal{R}$ be a relation. Let $\mathcal{C}$ be a constraint of $\mathcal{R}$. A test set $\mathcal{T}$ of $\mathcal{R}$ is a pairwise test set of $\mathcal{R}$ being consistent with $\mathcal{C}$ if $\mathcal{T}$ covers all the pairs of $\mathcal{R}$ that are valid according to $\mathcal{C}$ and $\mathcal{T}$ does not contain any invalid test specified by $\mathcal{C}$. A pair of $\mathcal{R}$ is valid according to $\mathcal{C}$ if there exists at least one valid test that contains this pair.

To illustrate the concept of relation and constraint, let us consider the following example [9]:

Table 4.1: An example relation

Call Type	Billing	Access	Status
Local	Caller	Loop	Success
Long Distance	Collect	ISDN	Busy
International	800	PBX	Blocked

Table 4.2: A constraint for the example relation

Call Type	Billing	Access	Status
International	800	*	*

Table 4.1 shows an example relation that contains four parameters involved with placing a phone call: *Call Type*, *Billing*, *Access*, and *Status*. Table 4.2 shows a constraint for the example relation ($*$ is a wildcard). The constraint specifies any test with $Call\ Type = International$ and $Billing = 800$ is an invalid test, independent of the values of *Access* and *Status*. In another word, there are no valid tests that contain pair $(International, 800)$. Therefore, $(International, 800)$ is invalid according to the constraint in Table 4.2.

Now we discuss how to revise algorithms IPO_H_IV and IPO_V to support test generation with constraints. Let $\mathcal{R}$ be a relation consisting of parameters p_1, p_2, $\ldots$, and p_n.

Let $\mathcal{C}$ be a constraint of $\mathcal{R}$. Assume that $\mathcal{T}$ is a pairwise test set for parameters p_1, p_2, ..., and p_{i-1} $(1 < i \leq n)$ and $\mathcal{T}$ is consistent with $\mathcal{C}$. In order to extend $\mathcal{T}$ to generate a pairwise test set $\mathcal{T}'$ that is consistent with $\mathcal{C}$ for parameters p_1, p_2, ..., and p_i $(1 < i \leq n)$, we need to use algorithms IPO_H_IV and IPO_V to grow $\mathcal{T}$ in both horizontal and vertical directions respectively.

At each step of algorithm IPO_H_IV, a value v of parameter p_i that covers the greatest number of missing pairs between p_i and p_j $(1 \leq j \leq i-1)$ is selected to extend a test τ in $\mathcal{T}$. Let τ' be the extended test. It is possible that τ' is an invalid test according to $\mathcal{C}$. Therefore, algorithm IPO_H_IV needs to be revised such that v is selected to extend τ if and only if τ' covers the greatest number of missing pairs between p_i and p_j $(1 \leq j \leq i-1)$ and τ' is a valid test according to $\mathcal{C}$. This revision can be accomplished by modifying function *eval* to return a valid weight for a value of p_i if and only if adding the value to the current test produces a valid test according to $\mathcal{C}$.

In order to cover a missing pair $(p_i.u, p_j.v)$, where $1 \leq j \leq i-1$, algorithm IPO_V tries to find a test τ that has u (or v) as the value of p_i (or p_j) and "$-$" as the value of p_j (or p_i). If τ exists, algorithm IPO_V modifies τ to cover $(p_i.u, p_j.v)$ by replacing "$-$" with v (or u) for p_j (or p_i). Otherwise, a new test is introduced with u being the value of p_i and v the value of p_j to cover $(p_i.u, p_j.v)$. Note that in the former case, it is possible that the test after such a replacement is invalid according to $\mathcal{C}$. Therefore, algorithm IPO_V needs to be revised such that a replacement takes place if and only if the test after the replacement is a valid test according to $\mathcal{C}$. This revision can be accomplished by trying to find another test τ'' that has u (or v) as the value of p_i (or p_j) and "$-$" as the value of p_j (or p_i) such that τ'' can be modified to cover $(p_i.u, p_j.v)$ and the modified test is valid according to $\mathcal{C}$. If τ'' does not exist, we can always add a new test to cover $(p_i.u, p_j.v)$.

The above approach fails in the following two cases:

1. During horizontal growth, when we try to select a value of p_i to extend a test τ in $\mathcal{T}$, it is possible that we can not find any value of p_i such that the extended test, τ', is valid according to $\mathcal{C}$;

2. At the end of vertical growth, a test τ may contain some "$-$" values for certain parameters. When we try to replace a "$-$" value with a valid parameter value in τ, it is possible that replacing the "$-$" value with any valid parameter value produces an invalid test according to $\mathcal{C}$.

In both cases, we can not complete τ to obtain a valid test according to $\mathcal{C}$. Therefore, τ needs to be removed from the current test set.

Special care should be taken to remove τ from the current test set. This is because those pairs that are covered by τ would be left uncovered by the removal of τ. If τ is removed during horizontal growth, we need to start over the test generation process to add new tests, using algorithm IPO_H_IV and/or algorithm IPO_V if necessary, to cover missing pairs between p_1 and p_2, and then missing pairs between p_1, p_2 and p_3, and continue to do so until all the missing pairs between p_1, p_2, $\ldots$, and p_{i-1} are covered. If τ is removed during vertical growth, we only need to use algorithm IPO_V to add new tests to cover those pairs between p_i and p_j $(1 < j \leq n)$.

In summary, our approach to supporting test generation with constraints can be described as follows.

- During horizontal growth, we modify algorithm IPO_H_IV such that a parameter value is selected to extend an existing test if and only if the extended test covers the greatest number of missing pairs and is valid according to the constraint.
- At the end of horizontal growth, we remove all the tests that can not be extended.
- During vertical growth, we modify algorithm IPO_V such that whenever it replaces a "$-$" value with a parameter value, it ensures that the test after the replacement is a valid test according to the constraint.
- At the end of vertical growth, we remove all the tests that contain "$-$" values that can not be replaced with any parameter value.
- After we finish horizontal and vertical growth for all the parameters, we check whether the resultant test set is a pairwise test set for the relation. If so, we terminate. Otherwise, we continue to the next step.
- Let π be the set of missing pairs. We start over the whole test generation process to cover all the missing pairs in π.
- In the end of the new test generation process, we check whether the outcome is a pairwise test set for the relation. If so, we terminate. Otherwise, letting π' be the set of missing pairs. If $|\pi'| \geq |\pi|$, we terminate. Otherwise, we start over the whole test generation process again.

We point out that the above solution does not guarantee the pairwise coverage in terms that a pair α is valid according to $\mathcal{C}$, but α may not be covered by the test set generated by our solution. We leave the optimal solution for the future work.

4.6 PairTest: An IPO-based test generation tool

We have implemented an IPO-based test generation tool, called PairTest, that includes algorithm IPO_H_IV for horizontal growth and algorithm IPO_V for vertical growth. We have also implemented algorithm IPO_H_EC, which is not included in PairTest due to its exponential time complexity.

Major features of PairTest include the following:

- PairTest supports the generation of pairwise test sets for systems with or without existing test sets and for systems modified due to changes of input parameters and/or values.
- PairTest provides information for planning the effort of testing and the order of applying test cases.
- PairTest provides a graphical user interface (GUI) to make the tool easy to use.
- PairTest was written in Java and thus can run on different platforms.

PairTest supports the following types of test generation:

- For a system without an existing test set, PairTest generates a pairwise test set.
- For a system with an existing test set, PairTest allows two options. One is to generate a new pairwise test set, without reusing the existing test set. The other is to keep the existing test set and generate additional tests, if necessary, such that the combined test set is a pairwise test set. The combined test set produced by the second option may be larger than the new test set produced by the first option. However, the second option allows the reuse of tests and thus can save the effort for test preparation.
- For a system that has an existing test set and is being modified due to changes of parameters, values, relations and constraints, PairTest allows two options. One is to generate a new pairwise test set, without reusing the existing test set. The other is

to modify the existing test set and then generate additional tests, if necessary, such that the combined test set is a pairwise test set.

Another test generation tool for pairwise testing is AETG (Automatic Efficient Test Generator)[3] [8] [9]. We used AETG[4] to produce pairwise test sets for the six systems mentioned in [9]. We also used PairTest to generate pairwise test sets for the same six systems. Table 4.3 shows the size information produced by AETG and PairTest for these six systems. As shown in Table 4.3, each of AETG and PairTest produces smaller test tests than the other for some systems. Later we will show that Pairwise has lower time complexity than AETG.

Note that in Table 4.3, S1 has 4 3-value parameters, S2 has 13 3-value parameters, S3 has 15 4-value parameters, 17 3-value parameters, and 29 2-value parameters, S4 has 1 4-value parameter, 39 3-value parameters, and 35 2-value parameters, S5 has 100 2-value parameters, and S6 has 20 10-value parameters.

Table 4.3: Sizes of pairwise test sets generated by AETG and PairTest

System	S1	S2	S3	S4	S5	S6
AETG	11	17	35	25	12	193
PairTest	9	17	34	26	15	212

It was shown that for a system with n parameters, each having d values, the size of a minimum pairwise test set grows at most logarithmically in n and quadratically in d [9]. Empirical results based on AETG indicates that when the number of candidate test cases for a new test case is 50, the number of test cases grows logarithmically in n [9]. We have carried out empirical studies to determine the growth function for the size of a pairwise test set generated by PairTest in terms of n and d. Table 4.4 shows the sizes of test sets generated by PairTest for systems with $d = 4$ and different values of n. Table 4.5 shows the sizes of test sets generated by PairTest for systems with $n = 10$ and different values of d. According to statistical analysis[5], the values of s (number of tests) in Tables 4.4 and 4.5 grow in $O(\log(n))$ and $O(d^2)$ respectively. These empirical results match the theoretical results mentioned earlier.

[3]AETG is a trademark of Telcordia Technologies Inc. and is covered by United States Patent 5,542,043.

[4]Telcordia allows free use of AETG for two weeks over the web. AETG does not provide the length of time used for test generation.

[5]Curve fitting using the SAS package was performed on data in Tables 4.4 and 4.5.

Tables 4.4 and 4.5 also show time information for test sets generated by PairTest. The execution time information was collected when PairTest was compiled and run on a PC with Intel 450MHZ Pentium II processor, Windows 98 and JDK 1.2.2. According to statistical analysis, the values of t (time for test generation) in Tables 4.4 and 4.5 grow in $O(n^2 \log(n))$ and $O(d^3)$ respectively. Based on the observation that the size of a test set generated by PairTest is $O(d^2 \log(n))$, we have shown that the time complexity of the IPO strategy is $O(d^3 n^2 \log(n))$. Based on the same observation for AETG, we shown that the time complexity of the AETG heuristic algorithm in [9] is $O(d^4 n^2 \log(n))$ in Theorem 4.6.1.

Theorem 4.6.1 *Assume that a system S has n parameters, and each parameter has at most d valid values. The AETG algorithm presented in [9] generates a pairwise test set for S in $O(d^4 n^2 \log(n))$.*

Proof

According to [9], the AETG algorithm generates M (e.g. $M = 50$) candidate tests and selects one that covers the most number of uncovered pairs for each test. Assume that a system S has n parameters, and the ith parameter has d_i parameters. Also assume that we have selected r test cases in the resultant test set $\mathcal{T}$. In Figure 4.7, we restate the algorithm presented in [9], which is used by AETG to generate a candidate test.

Now we consider the time complexity of algorithm *AETG_Select*. Let $d = max\{d_i | 1 \leq i \leq n\}$. The number of uncovered pairs is $O(d^2 n^2)$. In order to find f and l, statement (*) needs to go through the set of uncovered pairs, which takes $O(d^2 n^2)$. For statement (**), it takes $O(n)$ to generate a random order of n parameters. Assume that it takes $O(1)$ to determine if a pair is covered (this can be done by maintaining a coverage matrix as in algorithm IPO_H_IV). The size of τ is in $O(n)$. The inner *for* loop takes $O(dn)$ to calculate the number of uncovered pairs for each value of f_j and derive the one which covers the greatest number of uncovered pairs. Considering that the outer *for* loop iterates through $n - 1$ parameters, the nested *for* loop together takes $O(n^2 d)$. Therefore, the time complexity of *AETG_Select* is $O(n^2 d^2)$. Assume that the sizes of test sets generated by AETG are $O(d^2 \log(n)$, the time complexity of the AETG algorithm is $O(d^4 n^2 \log(n))$.

□

Figure 4.7: The AETG test generation algorithm

```
algorithm AETG_Select (S, T)
begin
   choose a parameter f and a value l for f such that l appears
      in the greatest number of uncovered pairs; ...(*)
   let f_1, ..., f_n be a random order of n parameters with f_1 = f; ...(**)
   for j = 2 to n do
   begin
      w' = 0; v' = 0;
      for each value v of f_j do
      begin
         τ = {(u, v)|u is the value selected for f_i, 1 ≤ i < j };
         let w be the number of uncovered pairs in τ;
         if w > w' then
         begin w' := w; v' := v; end
      end
      let v' be the value selected for f_j;
   end
end
```

Table 4.4: Results of PairTest for systems with n 4-value parameters

n (# of parameters)	10	20	30	40	50	60	70	80	90	100
s (# of tests)	31	34	41	42	48	48	51	51	51	53
t (time in seconds)	0.11	0.16	0.22	0.44	0.77	0.99	1.37	1.81	2.23	2.96

4.7 Summary

In this chapter, we have proposed the IPO test generation strategy for pairwise testing. The IPO strategy allows the use of local optimization techniques for test generation and the reuse of existing tests when a system is extended with new parameters or new values of existing parameters. We have presented three IPO-based test generation algorithms. Two of them, algorithms IPO_H_IV and IPO_V, have polynomial time complexities and thus are practical to use. We have constructed a test generation tool, called PairTest, which implements algorithms IPO_H_IV and IPO_V, supports the reuse of existing test sets, and provides graphical user interface. Our empirical results indicate that (1) the size of a pairwise test set generated by PairTest grows logarithmically in the number of parameters

Table 4.5: Results of PairTest for systems with 10 parameters, each having d values

d (# of values)	5	10	15	20	25	30
s (# of Tests)	47	169	361	618	956	1355
t (time in seconds)	0.05	0.28	0.72	1.54	2.96	5.16

and quadratically in the number of values in each parameter, and (2) PairTest is very efficient for generating pairwise test sets. As mentioned earlier, pairwise testing (or 2-way testing) is a special case of n-way testing. The IPO strategy and the algorithms presented in the chapter can be easily extended for n-way testing.

One problem with pairwise testing is that if the domains of input parameters are large, the number of generated tests is huge. For a system with each parameter having d values, the number of tests required for pairwise testing is at least d^2. Thus, if each parameter has 1,000 values, at least 1 million tests are required for pairwise testing. To alleviate this test explosion problem, one solution is to divide each input domain into partitions, select one representative value from each partition, and generate tests according to representative values for input parameters. By controlling the number of partitions for each input parameter, we can determine the number of tests needed for pairwise testing.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In Chapter 2, we have proposed a new reachability analysis approach, namely BSRA. BSRA can be applied to asynchronous message-passing programs based on any communication scheme, including fully asynchronous, FIFO, and causal communication. We have presented an algorithm, namely Generate_BSRG, to construct an BSRA-based reachability graph, or BSRG. Algorithm Generate_BSRGEdges is used by algorithm Generate_BSRG for edge generation, and takes a global state g as input and generates the set of outgoing edges of g as output. An outgoing edge e of g consists of a next blocking point b of g and a set γ of independent global receive transitions at b, that is, $e = (b, \gamma)$. The BSRG-based successor state g' of g reached by e is derived by advancing the current global state to b and then executing γ simultaneously (i.e. executing an arbitrary permutation of γ at b). We have formally established that reachability graphs constructed by algorithm Generate_BSRG retain sufficient information for the detection of deadlocks. In order to evaluate the effectiveness of BSRA, we have developed a prototype based on BSRA and have carried out an empirical study by applying BSRA to a number of commonly used distributed algorithms. We have measured state reduction rates achieved by BSRA, and have compared empirical results from BSRA to those from POR and SRA. Empirical results have indicated that BSRA significantly reduces the number of states generated during reachability analysis.

In Chapter 3, we have developed an efficient reachability testing algorithm, namely *Reachability-Test*, for asynchronous message-passing programs. The main challenge of reachability testing of asynchronous message-passing programs is how to derive the race variants of an SR-sequence. We have presented an algorithm, namely *Generate-Variants*, for the derivation of race variants. Algorithm *Generate-Variants* allows simultaneous modification of multiple race conditions and works as follows. Given an SR-sequence Q, it derives the race variants of Q related to the tail set $tail(Q)$ of Q, removes $tail(Q)$ from Q to obtain a smaller SR-sequence Q', and then continues to do so for Q' until all the race variants of Q are derived. In order to search SR-sequences and their race variants efficiently, algorithm *Reachability-Test* uses the special tree structure SR_Tree to keep all SR_sequences and their race variants collected during reachability testing. SR_Tree does not take much space, and has many useful applications such as regression testing, analysis of synchronization patterns, and checking of assertions on sequencing of events. For an asynchronous message-passing program that terminates on any input, algorithm *Reachability-Test* executes every partially-ordered path exactly once and thus guarantees the detection of deadlocks and assertion violations.

In Chapter 4, we have proposed a new test generation strategy, called In-Parameter-Order (or IPO), for pairwise testing. For a system with two or more input parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continues to do so for each additional parameter. The extension of a test set for the addition of a new parameter includes the following two steps: (a) horizontal growth, which extends each existing test by adding one value of the new parameter, and (b) vertical growth, which adds new tests, if necessary, after the completion of horizontal growth. We have presented IPO-based test generation algorithms: IPO_H_EC and IPO_H_IV for horizontal growth and IPO_V for vertical growth. Algorithm IPO_H_EC has exponential time complexity, but it can be used in conjunction with IPO_V to construct an optimal algorithm, which is theoretically significant in that it reveals what the IPO strategy can achieve at best. We have implemented algorithms IPO_H_IV and IPO_V in a pairwise testing tool, called PairTest, and have carried out an empirical study on a number of system configurations using PairTest. In comparison with a commercial pairwise testing tool, namely AETG, PairTest produces competitive results with the core algorithm of a lower asymptotic time complexity.

5.2 Future work

5.2.1 Model checking using BSRA

Model checking is a more general verification problem than deadlock detection [6]. It can be used to verify various properties specified as temporal logic formulae. The straightforward solution to model checking involves systematically generating all the reachable states of a system and checking each of them to see if the properties to be verified are satisfied. Like deadlock detection, model checking suffers from the state explosion problem. In this dissertation, we have shown how to use the BSRA approach for deadlock detection of asynchronous message-passing programs. We believe that it is possible to extend the BSRA approach for model checking of asynchronous message-passing programs.

Two major issues need to be resolved before BSRA can be extended to model checking. The first issue has something to do with the use of global variables in temporal logic formulae that specify properties to be verified. Since our multi-port EFSM model does not actually have global variables, we need to introduce global variables into the BSRA framework and address implications caused by doing so. The concept of visible transition is used to deal with the implications of introducing global variables into the SRA framework [36] [38]. A transition t is visible with respect to a logic formula f if its execution changes the values of some variables in f. Otherwise, t is invisible with respect to f. We believe it is possible to use the same concept to deal with the implications of introducing global variables into the BSRA framework.

The second issue is related to the fact if the value of any variable involved in a property formula is changed by the execution of a (visible) transition, the successor state reached by such an execution should be checked. This implies that no two visible transitions can be executed concurrently. To address this issue, the definition of a blocking point needs to be modified such that a blocking point is a global state in which each process either has no enabled transitions or has at least one open receive transition or has at least one visible transition. In addition, when we generate independent transition sets at a blocking point, we should ensure that each transition set contains at most one enabled visible transition at the blocking point.

There are more issues to be taken care of in order to use BSRA for model checking for asynchronous message-passing programs. We leave such a full extension for the future work.

5.2.2 Further improvement on reachability testing

In Chapter 3, we have developed an efficient algorithm for reachability testing of asynchronous message-passing programs. We believe that it is possible to further improve the performance of our reachability testing algorithm.

First, in function *Reachability-Explore*, we compute all the race variants of an SR-sequence at each recursive step. During execution, those race variants would have to be stored on a run-time stack. When the number of race variants is large, the stack may take a significant amount of space. A possible enhancement is that at each recursive step, we generate one race variant of an SR-sequence, and only when we return from that recursive step, we generate the next race variant of the SR-sequence. This is feasible because our approach to computing race variants is essentially systematic.

Second, we use the special tree structure SR_Tree to keep all SR-sequences, as well as their race variants, collected during reachability testing. As mentioned in Chapter 3, the SR_Tree can be used for regression testing, analysis of synchronization patterns, checking of assertions on sequencing of events, and other purposes. However, such use of SR_Tree may not be desirable in some context, or may be impractical, e.g. when a complicated message-passing program is involved (due to the potential large number of SR_sequences). We believe that it is possible to revise algorithm *Reachability-Test* such that the revised algorithm does not keep any SR-sequences collected during reachability testing. We point out that the challenge of doing so is how to guarantee, without the use of SR_Tree, that the algorithm executes every partially-ordered SR-sequence exactly once.

Bibliography

[1] Bowen Alpern and Fred B. Schneider. Recognizing saftey and liveness. *Distributed Computing*, 2:117–126, 1987.

[2] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 15:307–309, 1986.

[3] D. Brand and P. Zafirpulo. On communicating finite-state machines. *Journal of the ACM*, 31(2):323–342, 1984.

[4] L. Cacciari and O. Rafiq. On improving reduced reachability analysis. *Formal Description Techniques*, pages 137–152, 1993.

[5] R. H. Carver and K. C. Tai. Replay and testing for distributed programs. *IEEE Software*, pages 66–74, March 1991.

[6] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[7] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator. *Proc. IEEE Int. Symp. Software Reliability Engineering*, pages 303–309, 1994.

[8] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The combinatorial desgin approach to automatic test generation. *IEEE Software*, 13(5):83–89, September 1996.

[9] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–443, July 1997.

[10] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, and C. M. Lott. Model-based testing of a highly programmable system. *Proc. IEEE Int. Symp. Software Reliability Engineering*, pages 174–178, 1998.

[11] S. K. Damodaran-Kamal and J. Francioni. Nondeterminacy: Testing and debugging message passing parallel programs. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 118–128, May 1993.

[12] Suresh K. Damodaran-Kamal and Joan M. Francioni. Testing races in parallel programs with otot strategy. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 216–227, August 1994.

[13] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. *Proceedings of the Nineteenth International Conference on Software Engineering*, pages 205–215, May 1997.

[14] Matthew B. Dwyer and Lori A. Clarke. Dataflow analysis for verifying properties of concurrent programs. In David Wile, editor, *SIGSOFT'94: Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75. ACM Press, 1994.

[15] P. A. Emrath, S. Ghosh, and D. A. Padua. Detecting non-determinacy in parallel programs. *IEEE Software*, 9(1):69–77, January 1992.

[16] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. of 4th Computer Aided Verification Workshop*, pages 1–15, 1992.

[17] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. CAV'93*, volume 697, pages 438–449, 1993.

[18] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, Amsterdam, July 1991.

[19] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):149–164, May 1994.

[20] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems.* Springer, 1996.

[21] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[22] Patrice Godefroid, Bob Hanmer, and Lalita Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In *Proc. ACM SIGSOFT Intl. Symposium on Software Testing and Analysis*, pages 124–133, 1998.

[23] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575, pages 332–342, Aalborg, July 1992.

[24] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[25] M.G. Gouda and Ji-Yun Han. Protocol validation by fair progress state exploration. *Computer Networks and ISDN Systems*, 9(5):353–361, 1985.

[26] M.G. Gouda and Y.T. Yu. Maximal progress state exploration. In *Proc. ACM SIGCOMM*, pages 68–75, 1983.

[27] M.G. Gouda and Y.T. Yu. Protocol validation by maximal progress state exploration. *IEEE Trans. on Communications*, 32(1):94–97, 1984.

[28] E. Heller. Using design of experiment structures to generate software test cases. In *12th Int. Conf. Testing Computer Software*, pages 33–41, June 1995.

[29] Gerad J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[30] Gerard J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18(2):137–161, Februray 1988.

[31] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[32] Chung-Ming Huang, Jenq-Muh Hsu, and Shiun-Wei Lee. ECFSM-based probabilistic protocol verification. *Information Processing Letters*, 55(1):1–9, July 1995.

[33] Gwan-Hwan Hwang, Kuo-Chung Tai, and Ting-Lu Huang. Reachability testing: An approach to testing concurrent software. *Int. Journal on Software Engineering and Knowledge Engineering*, 5(4):493–510, 1995.

[34] M. Itoh and H. Ichikawa. Protocol verification using reduced reachability analysis. *Transactions on the IECE of Japan*, E66(2):88–93, 1983.

[35] B. Karacali, K.C. Tai, and M.A. Vouk. Deadlock detection of efsms using simultaneous reachability analysis. In *Proc. IEEE Dependable Systems and Networks*, pages 315–324, June 2000.

[36] Bengi Karacali. *Simultaneous Reachability Analysis of Concurrent Systems*. PhD thesis, Department of Computer Science, North Carolina State University, 2000.

[37] Bengi Karacali and K.C. Tai. Automated test sequence generation using sequencing constraints for concurrent programs. In *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 97–108, 2000.

[38] Bengi Karacali and K.C. Tai. Model checking based on simultaneous reachability analysis. In *Lecture Notes in Computer Science*, volume 1885, pages 34–53. Springer, 2000.

[39] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[40] Leslie Lamport. Time, clock, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[41] Y. Lei and K. C. Tai. In-parameter-order: A test generation strategy for pairwise testing. *Proc. 3rd IEEE High-Assurance Systems Engineering Symposium*, pages 254–261, Nov 1998.

[42] Yu Lei and Kuo chung Tai. Blocking-based simultaneous reachability analysis of asynchronous message-passing programs. In *submitted to IEEE Int. Symposium on Software Reliabibility Engineering*, 2002.

[43] Yu Lei and K. C. Tai. In-parameter-order: A test generation strategy for pairwise testing. *Technical Report TR-2001-03, Dept. of Computer Science, North Carolina State University, Raleigh, North Carolina*, March 2001.

[44] Hong Liu and Raymond E. Miller. Generalized fair reachability analysis for cyclic protocols. *IEEE/ACM Transaction on Networking*, 4(2):192–204, 1996.

[45] Hong Liu and Raymond E. Miller. Partial-order valdiation for multi-process protocols modeled as communicating finite state machines. In *Proceedings of International Conference on Network Protocols*, pages 76–83, 1996.

[46] Douglas Long and Lori A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *ACM*, pages 21–35, 1991.

[47] R. Mandl. Orthogonal latin squares: An application of experimental design to compiler testing. *Comm. ACM*, 28(10):1054–1058, October 1985.

[48] P.M. Merlin. Specification and validation of protocols. *IEEE Transactions on Communications*, COM-27(11):1671–1680, 1979.

[49] H. Miller and S. Katz. Saving space by fully exploiting invisible transitions. *Lecture Notes in Computer Science*, 1102:336–347, 1996.

[50] Neeraj Mittal and Vijay K. Garg. Debugging distributed programs using controlled re-execution. In *Symposium on Principles of Distributed Computing*, pages 239–248, 2000.

[51] Gleb N. Naumovich, Lori A. Clarke, and Leon J. Osterweil. Verification of communication protocols using data flow analysis. *ACM Software Engineering Notes*, 21(6):93–105, 1996.

[52] Robert H. B. Netzer, Timothy W. Brennan, and Suresh K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 31–40, 1996.

[53] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, 1992.

[54] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Leters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[55] Kadir Ozdemir and Hasan Ural. Protocol validation by simultaneous reachability analysis. *Computer Communications*, pages 772–788, 1997.

[56] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer-Aided Verification*, pages 377–390, June 1994.

[57] Doron Peled. Ten years of partial order reduction. *Computer Aided Verification*, 1427:17–28, 1998.

[58] M. Pezze, R. N. Taylor, and M. Young. Graph models for reachability of concurrent programs. *ACM Trans. on Software Engineering and Methodology*, 4(2), 1995.

[59] Michael Raynal. *Distributed Algorithms and Protocols.* John Wiley & Sons, 1988.

[60] J. Rubin and Colin H. West. An improved protocol validation technique. *Computer Networks: The International Journal of Distributed Informatique*, 6(2):65–73, May 1982.

[61] Carl Sunshine. Formal techniques for protocol specification and verification. *The Computer*, pages 20–27, 1979.

[62] K.C. Tai. Race analysis of traces of asynchronous message-passing programs. In *Proc. 17th International Conference on Distributed Computing Systems*, pages 261–268, May 1997.

[63] K.C. Tai and R.H. Carver. Testing of distributed programs. *Handbook of Parallel and Distributed Computing*, pages 955–978, 1996.

[64] Kuo-Chung Tai. Reachability testing of asychronous message-passing programs. In *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 50–61, May 1997.

[65] Kuo-Chung Tai, Rihard H. Carver, and Evelyn E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):54–63, January 1991.

[66] Kuo-Chung Tai and Bengi Karacali. On godefroid's state-less search technique for testing concurrent programs. In *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems*, pages 77–84, 2001.

[67] Kuo-Chung Tai and Bengi Karacali. On godefroid's state-less search technique for testing concurrent programs. In *Proc. IEEE 5th Intl. Symposium on Autonomous Decentralized Systems*, pages 77–84, 2001.

[68] Kuo-Chung Tai and Yu Lei. A test generation strategy for pairwise testing. *IEEE Transaction on Software Engineering*, 28(1):109–111, 2002.

[69] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[70] A. Valmari. Stubborn sets for reduced state space generation. *Lecture Notes in Computer Science*, 483:491–515, 1991.

[71] Hans van der Schoot and Hasan Ural. An improvement of partial-order verification. *Software Testing, Verification and Relibility*, 8:83–102, 1998.

[72] Hans van der Schoot and Hasan Ural. On improving reachability analysis for verifying progress properties of networks of cfsms. In *Proc. 18th Intl. Distributed Computing Systems*, pages 130–137, 1998.

[73] Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the 13th International Conference on the Testing of Communicating Systems*, pages 59–74, August 2000.

[74] Alan W. Williams and Robert L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. *Proc. IEEE Int. Symp. Software Reliability Engineering*, pages 246–254, 1996.

[75] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proc. CONCUR'93, volume 715 of Lecture Notes in Computer Science*, pages 233–246, August 1993.