INFORMATION TO USERS

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

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

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

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

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

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

UMI®
800-521-0600

Studies on Memory Consistency and Synchronization Failure Detection in Parallel Programs

Taiqi (Taichi) Fu

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

January 1998

©Taiqi (Taichi) Fu, 1998

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre reference

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-44810-X

Canada

Abstract

Studies on Memory Consistency and Synchronization
Failure Detection in Parallel Programs

Taiqi (Taichi) Fu, Ph.D.
Concordia University, 1998

We have studied two related issues in the design, execution and debugging of shared memory parallel programs: memory consistency and techniques to detect synchronization failure in parallel programs.

Overlapping execution of operations from a thread is an efficient way to effectively tolerate memory latency in shared memory multiprocessor systems. While *Sequential Consistency* provides a simple abstraction for a programmer to reason about his program, a sequentially consistent machine with full generality must execute memory operations sequentially in program order, leading to poor performance. We propose *Link Consistency* which still preserves Sequential Consistency for parallel programs that contain no data race or synchronization failure while permitting out of (program) order execution of operations. By classifying synchronization operations according to whether they satisfy some distinct conditions (*Exclusive Producer* and *Exclusive Consumer*), Link Consistency allows richer relaxation of program orders than some previous work such as Release Consistency. Compared to other work that also tries to improve upon Release Consistency such as PLpc, Link Consistency is conceptually clearer for programmer, and possesses a rigorous basis for reasoning.

We also address the following *Synchronization Failure* detection problem: Given a trace set from a *complete* (i.e. *successful*) execution, whether there exists another execution that can be derived from the same trace set and is incomplete. The answer to the problem is useful in debugging parallel programs. After showing that the problem is NP-complete, we present some techniques to tackle the problem in practice. These include Trace Reduction, Local Failure Validation, and Partial Order Checking. Trace Reduction may reduce the size of a trace set up to a small fraction of its original size while still preserving its failure property. Local Failure Validation explores a heuristic that is based on individual semaphores rather than all semaphores together. Compared to detecting strategies based on interleaving semantics, Partial Order Checking completely avoids the state explosion problem caused by the large amount of concurrence in trace sets. Our experimental result shows that for many trace sets with synchronization failures, the above techniques solve the detection problem in time linear to the trace set sizes, which are measured by N_n, the number of events or by $N_s \times N_t$, the product of the number of semaphores by the number of program threads, in a trace set.

Acknowledgments

I would like to thank my thesis supervisor, Professor Hon F. Li, for his continuous support both mentally and financially, over the years, in good times and in bad ones. I shall cherish what I have learned from him the way to think and to question. I am also very grateful to him for his patience during my studies and before my studies had begun.

I appreciate very much for the time members of my supervisory committee spent on my thesis proposal. I also appreciate very much for the comments from the examiners of my thesis to improve its quality.

I am in deep debt to my family. Yihong, my wife, did most of household work during my studies and suffered from time to time, due to my bad humors. Her help is beyond words. Minmin, our daughter, is always a source of tremendous joy to us. Without her, my studies would be much less colorful. I cannot thank my Mom and Dad enough for their trust on me whom they have missed so much, day after day. Their constant encouragement will companion me through the rest of my life.

Table of Contents

List of Figures

Chapter 1 Introduction

1.1 Motivation

1.1.1 About Memory Consistency

A natural approach to achieve higher performance for a computer system is to detect and then make use of parallelism in its programs. For sequential programs, data dependency analysis (Kuck et al. 1981) is an effective tool for such an optimization.

For a parallel program executed on a shared memory multiprocessor system, the way to relax program order is quite different. A parallel program differs from a sequential program in two important aspects. First, there are accesses to share data by different threads. Second, a new class of memory operations, called synchronization operations, is often used. Each thread now has its own program order which involves ordering among data and synchronization operations. Data dependency analysis alone is not adequate to deal with parallelism detection in parallel programs. Improper relaxation of program order in parallel program may lead to violation of sequential consistency. Two typical symptoms of such violation are unwanted data race and synchronization failure. What is the counterpart of data dependency theory for relaxing program order in parallel programs?

In trying to answer the above question, our research aims to find a set of simple and yet useful rules that tell which ordering constraints in program orders in a parallel program are important, and which are not and thus can be ignored in execution. We have explored the following relevant concepts: (i) *Data Race Freeness*, (ii) *Synchronization Failure Free-*

ness, (iii) *Exclusive Producer (Token Collision Freeness)*, and (iv) *Exclusive Consumer*. The main ideas are summarized in the following:

1. *Ordering between data accesses*: If every access to a data (called data access) are protected by some synchronization operations in every execution of a parallel program, then there is no data race and the ordering between a data access to another data access can be relaxed if there is no data dependency between them.

2. *Ordering between a data access and a synchronization operation*: In a data race free parallel program, if a synchronization operation is not used to separate (guard) a data access from another conflicting data access, then the ordering between them is immaterial and can be relaxed.

3. *Ordering between two synchronization operations*: There are special scenarios in which the relative ordering between such synchronization operations themselves is unimportant. For example if we know that a P operation *must* be enabled by some V operation, then it is possible for the P operation to be executed before the completion of some previous synchronization operations in the same thread.

The first idea is well understood and also used in the previous research on memory consistency models, first in Weak Ordering (Dubois, Scheuich, and Briggs 1986) and DRF (Adve and Hill 1990, 1993), and then in Release Consistency (Gharachorloo et al. 1990). Release Consistency has also exploited the second idea to a limited degree: since a Release operation (which is a synchronization write) does not protect shared data accesses that follow it in the same thread, there is no need to delay them until the Release is com-

pleted. A 'Release' in Release Consistency, however, is still considered to protect *all* shared data accesses that precede it. The third idea above, making use of some unique properties of some synchronization operations to further relax program orders, is exploited by us first (Li and Fu 1992, Fu 1992).

Suppose a parallel program uses binary semaphore operations for synchronization. We say a token is available for some semaphore a immediately after a V(a) is executed. We also say that there is a token collision in semaphore a if there is some execution of the program in which a V(a) is executed when a token is already available for a. A special case of token collision freeness is modeled by *Exclusive Producer*: a token is generated (by any thread) only if it is unavailable; no other thread has generated a token which is not consumed yet. Analogously, a P operation for a semaphore is an *Exclusive Consumer* if all P operations for the same semaphore occur in only one program thread. We have explored the usefulness of the notion of exclusive producer and exclusive consumer in relaxing program orders among synchronization operations. As an example, consider the classical Bounded Buffer problem (Peterson, G. L. 1981) which involves one producer and one consumer. The solution in Figure 1 uses three binary semaphores to coordinate competing data accesses to a single buffer.

```
binary semaphore mutex = 1;
binary semaphore empty = 1;
binary semaphore full = 0;
Producer()                          Consumer()
{                                   {
while (TRUE) {                      while (TRUE) {
(1) produce_item();                 (7) P(full);
(2) P(empty);                       (8) P(mutex);
(3) P(mutex);                       (9) get_item();
(4) put_item();                     (10) V(mutex);
(5) V(mutex);                       (11) V(empty);
(6) V(full);                        (12) consume_item();
}}                                  }}
```

Figure 1 An introductory example on relaxing program orders

There is no token collision for any of three semaphores. One of our rules relaxes the program order between (5) and (6) and that between (10) and (11).

P(empty) and P(full) each appears in a single thread only. This leads to another rule that relaxes some program order between P operations. For the class of parallel programs that contain no data race or synchronization failure, we propose several general rules to relax program orders among synchronization operations in Chapter 4.

Relaxing program orders among synchronization operations is more tricky than relaxing program orders between data and synchronization operations. We would also show some bad relaxation rules that have intuitive appeal but do not actually work.

1.1.2 About Synchronization Failure Detection

Data race or deadlock detection strategies can be classified as static detection or dynamic detection. Static detection includes testing (Tu, Shatz, and Murata. 1990, Masti-

cola and Ryder 1991, Stocks and Carrington 1993, Corbett 1996), and verification (Godefroid and Wolper 1991). Dynamic detection (Chandy, Misra and Haas 1983, Feitelson 1991, Rontogiannis, Pavlides and Levy 1991, Gonzalez and Garitagoitia 1993, Perkovic and Keleher 1996) reports the occurrence of deadlock or data race 'on the fly' by dynamically monitoring the states of the tasks or memory in a system.

Static detection is, in general, either conservative (reporting of false deadlocks or data races) or incomplete (failing to report true deadlocks or data races). Petri net based static program analysis (Tu, Shatz, and Murata 1990, Duri et al. 1993) transforms a program into a Petri net and the deadlock detection problem becomes a net reachability problem. An advantage of doing this is to make use of some existing analysis techniques for Petri nets. The Petri net based approach for detecting deadlocks, however, has not been made practical for large programs, mainly because of its well-known complexity. Petri net reachability problem, even though decidable (Mayr 1984), requires at least exponential space (and time) in the worst case (Lipton 1976)[1]. Dynamic detection of deadlock or data race, on the other hand, is supposed to report a deadlock or data race only when deadlock or data race actually occurs but does not rule out its existence in other runs.

We raise a novel question: Given a set of synchronization operations of a *successful* run, is it possible that due to synchronization race, there is an incomplete execution that results from the same trace set? If the answer is yes, then one may say, "Oh, you are lucky! The

1. In addition to that, the conventional Petri net is incapable of modeling systems requiring prioritized coordination of processes. Refer to (Peterson, J. L. 1981, chapter 7, p.195) for some example and discussion in detail.

program could have failed to progress!" A parallel program execution fails to progress if there is at least one program thread that is not executed to the completion. We say a parallel program contains synchronization failure if one of its execution so fails.

Like program testing or verification we locate potential synchronization failures, from a trace set. Unlike general program testing, however, we do not have to design and apply different input test data, and we have an additional piece of information about the trace set: it has at least one successful execution. We term this *Synchronization Failure Detection* problem and we show that it is NP-complete.

Even though previous work on deadlock or data race detection of a parallel program focuses on either dynamic monitoring of a program execution or static analysis of the program, post mortem trace analysis to detect safety violation such as data race in an execution has also drawn some research interests (Netzer and Miller 1990, 1991). We view our problem as a dual problem of the safety violation detection. Both belong to a bigger problem: what can we deduce from trace analysis accurately and can we do it efficiently?

1.2 Summary of Results

1.2.1 About Memory Consistency

For clarity and conciseness, we consider parallel programs that employ binary semaphores as the only synchronization mechanism. The following is a summary of our main contributions:

1. Construct simple multiprocessor reference model based on which various memory consistency models such as Release Consistency or Link Consistency, can be precisely defined, and analyzed.

Sequential processor is used as the base machine. Trivially this base machine implements sequential consistency. It actually defines sequential program order. A machine that relaxes sequential program order is a modified base machine. In general a modified machine may have more than one *next* instruction from a single thread to choose from in each step of execution. The essential difference between various memory consistency models are captured based on how these next instructions may vary. We use this modified reference machine model to capture unambiguously three fundamental concepts, namely, relaxed program orders, execution result, and sequentially consistent execution.

2. Relaxation of some unnecessary program order between data accesses and a synchronization operations.

First we formulate a set of sufficient conditions to describe when a synchronization operation does not protect a data access. It can be implemented by introducing *Scope Tag, an indicator* of ordering constraints between a synchronization operation and other operations around it, as a language construct for programmers. Alternately, defaults may be used to define each scope.

3. Relaxation of some unnecessary program order between synchronization operations.

Two special synchronization structures are exploited here: token collision freeness (exclusive producer) and exclusive consumer. Under such assumptions, we propose a new

memory consistency model, called Link Consistency. Link Consistency prescribes a set of conditions that determines the set of necessary program orders to be preserved in each program thread. It allows further relaxation of program order between some synchronization operations.

We prove that Link Consistency preserves Data Race Freeness and Synchronization Failure Freeness of a given program. We also prove that Link Consistency preserves token collision freeness. Finally, we show that for data race free and synchronization failure free parallel programs, a machine satisfying Link Consistency produces sequentially consistent results.

Compared with Release Consistency, Link Consistency further relaxes some program order among synchronization operations. Some examples including L-U decomposition have been used to compare Link Consistency with Release Consistency.

As a last note, scope tag can also be used to improve Release Consistency. However, Link Consistency without scope tag can still relax some program order that Release Consistency cannot (see Section 4.5).

1.2.2 About Synchronization Failure Detection

We require a trace set to be recorded before we analyze it to detect potential synchronization failure. As our starting point, we show that the detection problem is NP complete. Then we develop some techniques to avoid or at least reduce the size of the behavior space to be searched in answering the question.

We first study some possible short cuts to the detection problem. The first such short cut is special case trace sets for which the detection problem is solvable in *polynomial* time. The second strategy involves some *sufficient* conditions for detecting synchronization failure. For other cases, we develop some trace reduction rules and the associated reduction heuristics. The size of a trace set from an execution is usually large. So it is important and effective if one could reduce its size while still preserving its synchronization failure property. Two types of reduction rules are exploited: Local Reduction rules and Global Reduction rules. Local reduction rules reduce the size the trace of an individual program thread, while global reduction rules apply to the traces of multiple program threads.

The reduction heuristics are developed to search for synchronization failure based on some greedy strategies. They direct the detection algorithm to search in a reduced behavior space. This technique is specially useful for some trace sets where reduction rules cannot be applied. Similar to reduction heuristics, local behavior pruning tries to identify infeasible local behaviors or local behaviors that dominated by some infeasible one. Such local behaviors are then ignored.

We also investigate the use of a heuristic which is based on the observation that a property of a subsystem may also emerge as a property of the encapsulating system. Local Failure Validation represents one such strategy. First it locates a local failure that involves a single synchronization variable, when used in certain manner according to the recorded trace set. Then it checks to see if such a local failure may be incorporated in a global behavior, thus detecting a true synchronization failure.

Last, we apply partial order model checking technique in checking the behaviors that are derivable from a reduced trace set. Partial order checking significantly reduces the size of the behavior space contributed by the large amount of concurrency among program threads, as compared to the use of interleaved models. The above mentioned techniques can be applied together. Our experimental results from various applications show that a synchronization failure can be detected in time approximately linear to the size of trace set.

Our trace reduction rules can be visualized as special behavior space reduction rules. Work of (Tu, Shatz, and Murata 1990, Duri et al. 1993) uses net reduction. There are however several important differences between our work and theirs. One major difference lies in the synchronization mechanisms. Their study is for Ada programs with rendezvous synchronization, while we focus on shared memory parallel programs with semaphore-like synchronization.

Partial order model checking as an effective technique to reduce searching in the behavior space has been well established (Pratt 1986, Probst and Li 1990, 1991, 1993). Like (Godefroid and Wolper 1991, McMillan 1992), Our work is another application of this technique that yields significant reduction in search time. We have found, however, that, for our problem, the role of partial order checking is best seen when it is combined with reduction rules and/or reduction heuristics.

1.3 Thesis Organization

The remaining chapters are organized as follows: Chapter 2 summarizes the existing work on memory consistency and synchronization failure detection. Chapter 3 and Chapter 4 focus on memory consistency. Chapter 3 provides simple models for machines that execute parallel programs. It also defines parallel program executions based on these machine models. Chapter 4 defines our memory consistency model, Link Consistency using the machine models introduced in the previous chapter. Chapter 5, Chapter 6 and Chapter 7 focus on the synchronization failure detection: Chapter 5 defines derived behaviors of a trace set and illustrates the problem of potential synchronization failure detection, while Chapter 6 discusses the time complexity and various optimization techniques whose effectiveness is measured by the experiments reported in Chapter 7. Chapter 8 points out some future work and concludes the thesis.

Chapter 2 Related Work

2.1 Related Work on Memory Consistency

Multiprocessors that supports sequential consistency for arbitrary parallel programs are usually inefficient. Recent research on memory consistency (Adve and Gharachorloo 1996) turns its attention to supporting sequential consistency for a specific class of parallel programs in order to design more efficient multiprocessors.

Our research follows the same trend. We propose that if parallel programs are written with certain new constraints they could allow the hardware system to guarantee sequential consistency (for these programs) without enforcing all program orders in program threads.

2.1.1 Sequential Consistency

The semantics of sequential consistency for shared memory parallel programming was probably used earlier than the first appearance of the term in (Lamport 1979). It has been assumed that operations from the same thread will be executed both *atomically* and *sequentially* as if it were on a multi-programmed sequential processor. For example, the correctness the early two process mutual exclusion algorithm (Dekker 1965) is based on sequential consistency. Lamport coined the term sequential consistency to refer to this customary semantic model for multiprocessing. He wrote (Lamport 1979):

> *A multiprocessor is sequential consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

The essence of sequential consistency is that it enables a programmer to use (sequential) program orders as if they are sacred and to perceive every operation of a program as if it were atomic. Consequently, sequential consistency offers a simple to understand correctness criterion for designing parallel program. For example, one may apply assertional reasoning[2] to a parallel program in a similar way as to a sequential program to show its correctness (Hoare 1969, 1975).

However, there is a negative side to the design of multiprocessors to support sequential consistency for arbitrary parallel programs. Implementation of sequential consistency for arbitrary programs demands conservative execution and is inefficient, largely because of the apparent need to obey program orders. Indeed, some effective optimization techniques to relax program order based on data dependence analysis for sequential programs are no longer valid for sequential consistency.

2.1.2 Recent Research on Supporting Sequential Consistency

2.1.2.1 Weak Ordering and DRF0 Models

(Dubois, Scheuich and Briggs 1986) first proposed that if the hardware could distinguish data operations from synchronization operations then it is possible to have a more efficient system which executes data accesses between two consecutive synchronization operations in a process as long as all synchronization operations are executed strictly in program

2. "All the properties of a program and all the consequences of executing it in any given environment can, in principle, be found from the text of the program itself by means of purely deductive reasoning." (Hoare 1969)

order and local data dependency is maintained. Their model of relaxation of program order is termed Weak Ordering. (Adve and Hill 1990) refines this idea by introducing an extra condition for parallel programs: Data Race Freeness and names the refined model DRF0 (Data-Race-Free-0). DRF0 allows an Unlock operation to be issued before the data accesses (say D1) that are already issued are (globally) committed. This works because of the following: (i) data race free programs: any other thread that contains data accesses (say D2) that are in conflict with D1 will have a Lock operation that precedes these data accesses, and (ii) DRF0 will delay such Lock until D1 are globally committed. The distinction between different synchronization operations remains unclear in these works.

2.1.2.2 Release Consistency and DRF1 Models

(Gharachorloo et al. 1990) proposed Release Consistency (RC for short) by further classifying each synchronization operation as either Acquire (acqL) or Release (relL). A memory operation in a program is labeled with one of the following: acqL, relL, nsyncL, or ordinaryL. A Lock operation is typically labeled as acqL and an Unlock operation typically as relL. Based on these labels, a parallel program is Properly Labeled if it satisfies the following conditions:

> *Enough SyncL Labels:*
>
> *Pick any two accesses u on processor Pu and v on processor Pv (Pv is not the same as Pu) such that the two accesses conflict, and at least one is labeled as ordinaryL. Under any legal interleaving, if v appears after(before) u, then there needs to be at least one syncL write(read) access on Pu and*

one syncL read(write) on Pv separating u and v, such that the write appears before the read. There are enough accesses labeled as syncL if the above condition holds for all possible pairs u and v. A syncL read has to be labeled as acqL and a syncL write has to be labeled as relL.

Properly-Labeled (PL) Programs:

A program is properly-labeled if the following holds: all shared accesses are labeled as sharedL or more restricted types, all competing accesses are labeled as specialL or more restricted types, and enough specialL accesses are labeled as acqL and relL.

Even though both are very similar, a Properly-Labeled (PL) program is not exactly the same as a data race free program. A PL program has an additional type of accesses called nsyncL that is neither ordinary data access nor synchronization operation (Acquire or Release). It is noteworthy that RC cannot guarantee sequential consistency for PL programs that contain nsyncL operations. Appendix B gives such an example. Unless otherwise specified, we will not consider programs with nsyncL operation in this thesis.

Without nsyncL operations, a PL program is very close to a data race free program. A subtle difference between them remains. Labels in a PL program can be rather *artificial*: for example, a data operation in a critical section may be labeled as acqL or a Lock operation may be labeled as ordinaryL. That is, with RC, hardware uses labels of operations instead of the semantics of operations for the relaxation of unnecessary ordering constraints in program order. In a data race free program of DRF0 (and DRF1 as discussed next), there is no labeling: an operation has the generic semantics as perceived both by a programmer and hardware. Theoretically, artificial labeling approach with RC offers a

more powerful mechanism to specify necessary (or unnecessary) program orders than the fixed semantics approach. For example, by labeling two consecutive read operations in a critical section in a PL program both as rel_L, RC can enforce a strict order between these two reads. While this kind of labeling may be necessary for the hardware, which supports RC, to execute some mutual exclusion algorithms such as those in (Dekker 1965) and (Peterson, G. L. 1981), the usefulness of such labels for parallel programs that use synchronization primitives such as Test-and-Set, Barrier, or semaphore operations is rather unsubstantiated. Without nsyncL and under the "normal" labeling of synchronization operations, normal in the sense that an Acquire (Release) of DRF-1 is labeled as acqL (relL), a properly-labeled program is a data race free program and RC is identical to DRF-1 for a programmer.

RC specifies a set of hardware conditions that try to guarantee sequential consistency for PL programs (Gharachorloo et al. 1990):

> *Conditions for Release Consistency:*
>
> *1) before an ordinary Load or Store access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed,*
>
> *2) before a release access is allowed to perform with respect to any other processor, all previous ordinary Load and Store accesses must be performed, and*
>
> *3) special accesses (synchronization operations and nsync operations) are processor consistent with respect to one another.*

The first condition ensures that no ordinary data accesses will be performed before all acquire operations that precede them have been performed. The second condition ensures that no release operation can be performed before all ordinary data accesses that precede it have been performed.

The phrase '*processor consistent*' in the third condition is later replaced by '*sequentially consistent*' (Gibbons, Merritt, and Gharachorloo 1991) before an attempt is made to prove that RC appears to be sequentially consistent for properly-labeled programs. This revised condition implies that synchronization operations must be executed according to program order. In this thesis, we assume this revised definition of RC which is referred as RCsc in (Gharachorloo et al. 1992).

Compared to DRF0, RC allows further relaxation of program order in a program thread. For example, since in a PL program, a Release can only protect ordinary data accesses that precede it, those ordinary data accesses after a Release and the Release need not be executed in program order. This relaxation, however, does not come free: the hardware now must be able to distinguish between two types of synchronization labels: Acquires (acqL) and Releases (relL).

(Adve and Hill 1993) proposes DRF1 which improves upon DRF0 by classifying synchronization operations into Release and Acquire as in RC. Both DRF1 and RC (as revised in (Gibbons, Merritt, and Gharachorloo 1991) and without nsyncL) are the same for the programmers: both recognize special synchronization operations called release and acquire and a data race free program according to DRF1 is properly-labeled program and

vice versa. (Adve and Hill 1993) also proposes a more efficient hardware implementation of DFR1 than the implementation proposed for DRF0. Instead of delaying a subsequent LOCK operation in one process that precedes some data operations (D1) in conflict with other currently outstanding data operations (D2) from another process (see section 2.1.2.1), one can let such LOCK operation go ahead and delay D2 only. The additional requirement is a protocol between these processes: the process with D1 informs the process with D2 after it has issued D1. The same optimization can be used to implement RC too.

Even though RC has succeeded in relaxing some of program order in PL programs, each of its three hardware conditions can be further relaxed. For example, an Acquire may not delay all data operations that follow it; maybe, it only "protects" the access atomicity of a subset of these data operations. In addition, synchronization operations may not have to be executed in sequential program orders. Under some usage assumptions (Li and Fu 1992, Fu 1992), the sequential program orders of synchronization operations can also be relaxed.

2.1.2.3 PLpc Model

This is the work closest to ours. The researchers of DRF0 (Adve and Hill 1990), DRF1 (Adve and Hill 1993) and RC (Gharachorloo et al. 1990) jointly propose PLpc model (Gharachorloo et al. 1992) to improve upon both DRF1 and RC by further relaxing sequential program order among synchronization operations. PLpc identifies two classes

of synchronization operations called loop synchronization operations and nonloop synchronization operations. Loop synchronization operations are further classified as loop reads and loop writes as follows (Gharachorloo et al. 1992):

> *DEFINITION 4: LOOP AND NONLOOP READS. A competing read is a loop read if (i) it is the final read of a synchronization loop construct that terminates the construct, (ii) it computes with at most one write in any sequentially consistent execution, and (iii) whenever it competes with a write in an sequentially consistent execution, it returns the value of that write: i.e., the write is necessary to make the synchronization loop construct terminate. A competing read that is not a loop read is a nonloop read.*
>
> *DEFINITION 5: LOOP AND NONLOOP WRITES. A competing write is a loop write if it competes only with loop reads in every sequentially consistent execution. A competing write that is not a loop write is a nonloop write.*

Roughly speaking, a loop read is the critical read in a busy waiting operation which can succeed only after some unique write (loop write) is performed. The following is an example from (Gharachorloo et al. 1992) where there are two processes P1 and P2:

```
P1                                      P2

(1) A = 1;                              (5) B = 1;
(2) Flag1 = 1;(loop write)              (6) Flag2 = 1; (loop write)
(3) while (Flag2 == 0 ) (loop read)     (7) while (Flag1 == 0) (loop read)
      ; (does nothing)                        ; (does nothing)
(4) ...= B;                             (8) ... = A;
```

Figure 2 Code segment to illustrate PLpc. Statements are numbered from 1 to 8.

The write to Flag1 and the write to Flag2 are loop write and the last read in each while loop is loop read. It is easy to see that the code segment will generate an expected result

even if the statement (3) is started before the statement (2) is executed and the statement (7) is started before the statement (6) is executed.

Given Properly-Labeled programs, PLpc permits a synchronization read not to wait for the previous synchronization write unless both are non-loop synchronization operations. Figure 3 below is a graphical representation of what sequential program order PLpc relaxes more than DRF1 or RC does. The example in Figure 2 uses the relaxations PL1 and PL2.

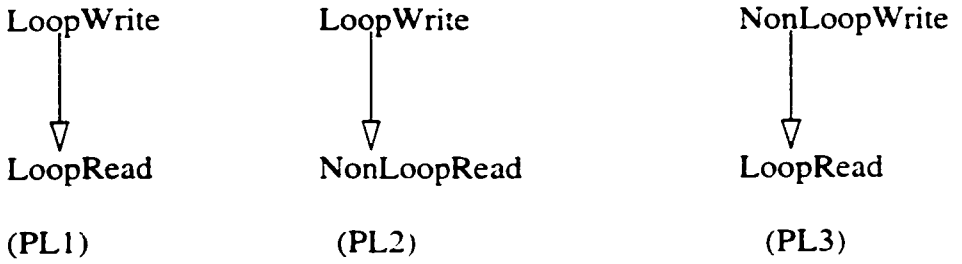


Figure 3 Relaxations of program order among synchronization operations by PLpc.

2.1.2.4 Optimizing Parallel Programs at Compilation Time

(Shasha and Snir 1988) presents a method to optimize the execution of a parallel program by some static analysis of its program orders and potential access conflicts. They assume that a parallel program consists of a fixed number of sequential program segments of memory read or write operations. Under this assumption, they construct a conflict graph of operations in a program that include uni-directional arrows specified by program orders and bi-directional arrows representing potential conflicts between operations from different program segments (between a read and a write, or two writes). Based on such conflict

graphs, they present algorithms to find a minimal subset of program orders (called a *delay set*) sufficient to ensure sequential consistency in execution.

(Krishnamurthy and Yelick 1994) points out the Shasha and Snir's approach would be NP complete if the number of threads is not fixed or unknown at compilation time. For example, a same SPMD (Single Program Multiple Data) program may be executed by an arbitrarily large number of threads. Krishnamurthy and Yelick presents an efficient algorithm to determine the minimal delay set for a given SPMD program. Since every thread is running an *identical* SPMD program, their algorithm actually analyzes two copies of the SPMD program on two separate threads only.

(Shasha and Snir 88)'s algorithm assumes that a program consists of ordinary memory read and write operations without explicit synchronization. In practice, a parallel program is often explicitly synchronized. (Krishnamurthy and Yelick 95) proposes algorithms to improve the accuracy of Shasha and Snir's algorithm utilizing the synchronization information in a program. Synchronization operations may enforce certain ordering constraints that make some conflicting data operations impossible to compete. Therefore, some delay that Shasha and Snir's algorithm will find may not be necessary.

For example, consider the following program from (Krishnamurthy and Yelick 95):

```
Initially F=X=Y=0;

Thread 1                                    Thread 2
Write(X,1);                                 Wait(F);
Write(Y,1);                                 Read(Y);
Post(F);                                    Read(X);
```

If only data operations are considered, Shsha and Snir's algorithm will detect that both the delay between two writes and the delay between two reads are required to achieve sequential consistency. But if we assume that a 'Post' waits for the previous data operations to complete and a 'Wait' delays the data operations that follow it, obviously both delays are unnecessary. The intuition behind their work is somehow similar to that in (Dubois, Scheuich and Briggs 1986) that we mention earlier.

2.1.2.5 Summary

We can view a multiprocessor consistency model as an agreement between parallel programs and the multiprocessor hardware that executes them. Programs must possesses some properties, for example data race free or properly-labeled, in order for the hardware to deliver sequentially consistent results. If a program does not possess such properties, then the hardware may not deliver correct results. The main purpose to have overlapped data accesses, as offered by these consistency models, is to tolerate potentially long memory latency of global memory accesses in multiprocessors. Such special agreements between software and hardware are called memory consistency models.

2.1.3 Other Related Work

1. Processor Consistency Model

(Goodman 1989) first defines processor consistency (processor ordering) as follows:

A multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program.

For a processor consistent multiprocessor, the order in which writes from two different processors occur need not be identical when observed by different processors.

(Gharachorloo et al. 1990) presents a condition to implement the Processor Consistency:

Condition 2.2: Condition for Processor Consistency

(A) before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed, and

(B) before a STORE is allowed to perform with respect to any other processor, all previous accesses (LOAD and STORE) must be performed.

A sequentially consistent multiprocessor is processor consistent. The reverse is not necessarily true.

2. Memory Consistency for Shared Virtual Memory Multiprocessor

For shared virtual memory (distributed shared memory) multiprocessors, in addition to memory latency, the performance of machines also greatly depend on network latency and inter-processor communication latency. To tolerate those latencies, (Bershad, Zekauskas and Sawdon 1993) proposes *Entry Consistency* in which there are two types of objects on which a parallel program operates: synchronization objects and shared data objects. Synchronization objects are used to protect data objects and is owned only by one processor (the owner) at a time as explained next.

Entry Consistency assumes that:

(i) Every synchronization object is explicitly identified, and

(ii) Every shared data objects is associated with a synchronization object and accesses to such data object must first go through the synchronization object associated with it.

Two types of accesses to a synchronization object are defined: exclusive access by which the protected data objects will be modified, or non-exclusive one by which the protected data objects are only read, not modified. The memory consistency is easy to achieve because of the explicit synchronization of data operations, as described next.

To modify a data object, a processor requests for an exclusive access of the synchronization object that is associated with the data object. When granted, the processor becomes the owner of the synchronization object and can modify the data object. If a processor need only to read the data object protected by a synchronization object, it requests for an non-exclusive access to the synchronization object and will be granted quickly. More than one processor can be granted with non-exclusive accesses to a same synchronization object. The next processor which requests and is granted an exclusive access to a synchronization object become the new owner of the synchronization object. It is assumed that a programmer will label shared data objects and synchronization objects properly. It is also assumed that a programmer will label each access to synchronization object as either read-only (non-exclusive) or read-write (exclusive). If a program is properly labeled, Entry Consistency ensures sequentially consistent result. Entry Consistency naturally relaxes the program order between a data operation and a synchronization operation that it does not protect, or between two unrelated synchronization operations. However, to provide an

explicit association between every data object and the synchronization object that protects it puts some extra burden on programmers.

To alleviate the strict requirement for a programmer to provide explicit information on the association between a data object and the synchronization operation, (Iftode, Singh and Li 1996) proposes Scope Consistency. Scope Consistency uses the implicit scope of a synchronization operation to derive the information on who protects what. For example, in a lock-based parallel program, Scope Consistency would assume that a consistency scope "naturally" consists of all critical sections protected by the same lock. The intuition is that shared memory will be accessed in separate scopes and accesses involving different scopes can occur concurrently. In many cases, such "natural" scopes coincide with the real scopes of locks and as long as such scopes are obeyed, the execution would still be sequentially consistent. In such cases, program orders between two operations of two distinct scopes can be relaxed.

However, as they have already pointed out, that is not always the case. For some programs, the straight-forward approach of identifying a scope of a lock to be a set of critical sections that appear between the acquires of the lock and the release of the lock may not work. For example, consider the following program (Iftode, Singh and Li 1996, Figure 3) where L1 is a lock:

```
Processor 0                    Processor 1
X = 1                          acquire(L1)
acquire(L1)                    a = Y
Y = 1                          b = X
release(L1)                    release(L1)
```

The natural scope of L1 is the two critical sections: one writing Y and the other reading Y and X. However, in this example, the programmer has also implicitly assumed that release(L1) of Processor 0 also protects the write to X, even if X=1 is not in any critical section. Any run of the program on a sequentially consistent machine, such as on a RC machine, will contain no data race. To relax the program order between X=1 and release(L1), according to Scope Consistency, will cause a data race between the write to X by Processor 0 and the read by Processor 1.

As a remedy for those undesirable cases, either extra acquires need to be added, or the scope of a current lock need to be expanded (by moving an acquire "higher" and an release "lower" in a program which is written on papers in a usual fashion: from top to bottom). For the above example, move acquire(L1) ahead of X=1 would work. There is still no general guidelines on when and how to add additional acquires or to move the existing acquires or releases. The notion of scope in Scope Consistency is similar to the notion of scope that we use in (Fu 1992). We have devised several simple programming constructs called scope indicators (called *Explicit Scope Annotations* in Scope Consistency) to capture, as precise as possible, the relationship between a synchronization operation and the data operations around it: what it protects and what it does *not*. Like (Bershad, Zekauskas and Sawdon 1993), we assumes that a programmer will indicate the scope explicitly. Scope Consistency is a pioneer work in trying to obtain the information about who protects what in a parallel program without much burden on programmers. Another interesting work on synchronization optimizations is (Diniz and Rinard 1997).

(Singla, Ramachandran and Hodgins 1997) introduces temporal notions into consistency for interactive applications. For example, an interactive virtual environment consisting of simulated robots can tolerate high levels of "staleness" with respect to various robots' perception of the states of each other. The proposed *Delta Consistency* is a time-based correctness criterion to govern the shared memory accesses. Sequential consistency relies on a coherent memory on which a read of a memory location returns the most recent write to that location. Delta consistency builds on a coherent memory that satisfies the following only: a read of a memory location returns the last value that was produced at most *delta* time units preceding that read operation. The value of delta represents the application's tolerance for staleness of data. Unlike release consistency or our work, their work focuses on those applications with asynchrony or "loose" synchrony in nature only.

2.2 Related Work on Synchronization Failure Detection

Even though there are recent results on detecting safety violation based on trace analysis, such as to determine the possible event ordering (Helmbold, McDowell and Wang 1993) or to detect data race in parallel program executions (Netzer and Miller 1990, 1991) as far as we know, there is no previous work done to detect potential synchronization failure in a parallel program based on analysis of trace from a successful execution. The work closest to ours is the application of trace analysis to detect data race (Netzer and Miller 1990, 1991). Our work differs from theirs in the following: we try to determine the existence or absence of any synchronization failure implied by a trace set, which itself is recorded from a successful execu-

tion; while they try to detect any data race implied by a trace set, whether it is recorded from a data race free execution or an execution with data race is unknown and irrelevant to them.

Another related work is the Verification of Sequential Consistency (VSC) (Gibbons and Korach 1992, 1994). Given a set of sequences of operations that are executed, the VSC problem is to decide whether there is some interleaving of the operations that satisfies sequential consistency. Gibbons and Korach prove that the VSC problem is NP complete and present efficient algorithms for some restricted VSC problems.

The result of dynamic detection or trace-based analysis of parallel program properties such as data race or synchronization failure is only valid with respect to that execution or the trace of that execution. Static analysis of parallel programs tries to determine whether a parallel program satisfies such properties without executing it. Unfortunately, such decision problems are in nature equivalent to the classic Turing Machine Halting Problem, which proves to be undecidable. Instead of having a parallel program and then trying to analyze it statically to determine whether it contains some unexpected properties (section 2.1.2.4), another active research area in static analysis of programs is to have compilers, with a sequential program as input, generate a parallel program that is guaranteed to be free of data race or synchronization failure.

The so-called parallelizing compilers takes some conventional (such as Fortran 77) sequential programs and transforms them into parallel ones. For example, (Hall et al.

1995, 1996) reports good performance for their SUIF (Stanford University Intermediate Format) compiler that automatically parallelizes the array-based numerical programs using scalar analysis, array analysis, intra- and inter-procedural analysis, among other techniques. (Lim and Lam 1997) is another example of parallelizing sequential programs which consist of arbitrary nestings and sequences of loops with certain restrictions. (Blume et al. 1994, August 1996, December 1996, Navarro et al. 1997) discuss Polaris, a parallelizing compiler developed at University of Illinois at Urbana-Champaign, to transform Fortran programs for target machines such as Cray T3D. Polaris incorporates compilation techniques including in-line expansion, induction variable substitution, reduction variable recognition, symbolic dependence analysis, scalar and array privatization.

The above research assumes restricted classes of sequential programs, particularly those with extensive nested loops and simple array indices. The generated parallel code usually has very regular synchronization patterns. They do not work with sequential programs in general. In addition, their focus is on the sequential code which exists such as those in large financial institutions. The parallelizing compiler does not apply to the areas where parallelism is intrinsic and cannot be derived but must be carefully accounted for and explicitly specified. One such example is the solution for the famous Dining Philosopher Problem. Another example is telecommunication or data-communication software where C programming language and semaphores or locks are still dominant and timing is critical. In contrast, both the dynamic detection approach or post-mortem trace analysis approach work for general programs.

The key difference between the research on parallelizing compilers and our research on trace set analysis is that the domain of the former is the existing sequential programs while the domain of the latter is the existing parallel programs. Assume that the parallelizing compiler itself is correct, such generated parallel programs should be free of any unexpected properties such as deadlock. Trace set analysis is, however, still useful in the following context: we do not know whether a parallel program will always behave as expected, but since we have just recorded a trace set from a successful execution, let us see whether this trace set implies some unexpected properties. The former deals with how to derive a parallel program out of a sequential one with optimal parallelism (maximum number of threads with least synchronization overhead) while the latter deals with whether an existing program contains some unexpected behaviors.

Chapter 3 Machine Model and Parallel Program Execution

3.1 Machine Models

We define two types of machines: base machine and relaxed machine. The base machine is the reference machine based on which sequential consistency is defined. A relaxed machine is an abstraction to define other consistency models.

3.1.1 Base Machine and Relaxed Machine

A parallel program is a set of sequential program threads and a set of initial values of variables in memory. Each program thread contains one and only one 'HALT' operation as its *last* statement. When a 'HALT' of a program thread has been executed, the execution of the program thread comes to an end. We assume that operations of a program have all been fetched into the instruction buffers (for various threads).

Definition 3.1. *Base Machine*. A base machine (BM) is a multi-programmed uni-processor that

(i) executes operations one at a time according to the order fetched,

(ii) randomly chooses one program thread and tries to fetch and execute its *next* operation, and

(iii) terminates only when a HALT of every thread has been executed.

A relaxed machine is similar to BM except that it does not necessarily execute operations in program orders. The 'next operation' is selected from a set of operations, called

Next Candidate Set(**NC**). Different characterization of NC leads to different memory models. The following is a general definition for relaxed machine models:

Definition 3.2. *Relaxed Machine.* A relaxed machine (RM) is a multi-programmed uni-processor that

(i) executes operations one at a time,

(ii) randomly chooses one program thread and executes *one* of its next operations in NC,

(iii) terminates only when a HALT of each thread has been executed.

Notice that (i) and (iii) above are identical to (i) and (iii) of BM respectively. It is the (ii) that makes RM differ from BM. In BM, the next operation of a thread is a singleton and unique unless HALT is executed. In RM, the next operation(s) of a thread form a set containing possibly many elements.

3.2 Parallel Program Execution and Event Ordering

3.2.1 Parallel Program and Synchronization

We consider parallel programs which are synchronized using binary semaphores only: there are only two types of shared memory operations in a parallel program: synchronization operations (P and V), and data operations. For a semaphore a, we use *P(a)* and *V(a)* as the synchronization operation. We further assume that unless otherwise specified, the initial value of semaphores are all zero.

We assume that in (ii) of the definition of BM or RM, the selection of a semaphore operation is restricted to those that could be executed successfully. Otherwise, the effect is the

same as if without being selected. We also assume that the parallel programs *terminate* and each program thread is sequential. To simplify our discussion, we will consider parallel programs without branching operations in our model.

3.2.2 Operation versus Event

We consider only shared memory and HALT operations. The memory operations can take the following forms:

(i) data read,

(ii) data write,

(iii) V(a), and

(iv) P(a).

We use *op*=<i, o, x, j> to denote an operation in the i^{th} program thread, where o is the operation type: R for Read, W for Write, P , V and H for HALT; and x is the variable involved. Since there may be more than one operation of the same type in a program thread that operates on a variable, the index j identifies each of them uniquely. In other words, the index j distinguishes between different occurrences of a same type of operation on a variable such as write(X, 5) and write(X, 6) in the third thread. The *first* such write is identified with <3, W, X, *1*>. When necessary, both P(a) and V(a) are treated as write operations.

The execution of an operation in a program code is called an event. There are many events corresponding to a same operation if the latter is executed many times. We use ev =

<op, v> to represent an event, where op is an operation and v is the value associated with an execution of op.

Definition 3.3. Associated value of an operation. The associated value of a read is the value returned by the read, and the associated value of a data write is the value written. The associated value of a P(a) or a V(a) is the value of semaphore a right after the operation is executed. For convenience, the HALT has an associated value zero.

3.2.3 Program Execution

A program execution is a sequence of events as defined next:

Definition 3.4. Program execution. An *execution* E of a program on a machine is a total ordering of the events executed by the machine.

3.2.4 Some Notations and Definitions

The following symbols are used:

a, b, c, ... : distinct semaphores.

Q, Q1, Q2, Q3: a synchronization (P or V) operation or event.

T_i : program thread *i* .

operations(Ti) : set of operations in Ti.

length(E) : number of events in E.

event(j, E) : j^{th} event in E.

events(E) : set of events in E.

sem(op) or sem(ev): the corresponding semaphore for operation op or event ev.

data(E) : data variables that appear in E.

sem(E) : semaphores that appear in E.

var(E) : data(E) $\cup$ sem(E).

Definition 3.5. *before(ev1, ev2, E)* = true iff ev1 appears (immediately or not) before ev2 in E.

after(ev1, ev2, E) = true iff ev1 appears (immediately or not) after ev2 in E.

Definition 3.6. Two semaphore operations (or events) are ***compatible*** if both operate on a same semaphore.

Definition 3.7. *Conflict operations* and *conflict events*. Two operations op1=<i1, o1, x1, j1> and op2=<i2,o2, x2, j2> are in conflict, denoted as conflict(op1, op2) if (i) x1=x2, and (ii) at least one of o1 and o2 is a write operation. Similarly, two events ev1=<op1, v1> and ev2=<op2, v2> are in conflict denoted as conflict(ev1, ev2) iff conflict(op1, op2).

Definition 3.8. If event(k, E) = ev then pos(ev, E) = k, otherwise pos(ev, E) is undefined.

Definition 3.9. subseq(i, j, E) is the sequence of events event(i, E), ... , event(j,E) if $1 \le i \le j \le$ length(E), otherwise Λ, the empty sequence. For simplicity, we also write subseq(e1, e2, E) =subseq(pos(e1), pos(e2), E). The prefix of E ending at e is given by prefix(e, E) = subseq(1, pos(e), E).

Definition 3.10. A common prefix of E1 and E2, comm(E1, E2) is a prefix common to both E1 and E2. max_comm(E1, E2) is the longest common prefix of both E1 and E2. Notice that max_comm(E1, E2)=Λ if event(1, E1) $\neq$ event(1, E2).

Definition 3.11. *next(e, E)* = event(pos(e)+1, E) if pos(e,E) < length(E). *prev(e, E)* = event(pos(e)-1, E) if pos(e,E)>1.

Definition 3.12. *E(X)*, X∈ var(E), is the sequence of events in E that operate on variable X.

Definition 3.13. An execution E1 is a *prefix* of another execution E2 if E1 = max_comm(E1,E2). E1 is a *proper prefix* of E2 if E1 is a prefix of E2 and E1≠E2.

Definition 3.14. A *maximal* execution of a parallel program is an execution that is not a proper prefix of another execution of the same program. An execution that is not maximal is called *partial* execution.

Definition 3.15. A *successful* execution of a parallel program is an execution which includes a HALT in each program thread.

A successful execution is maximal but the reverse is not necessarily true.

Definition 3.16. An execution that is maximal but not successful is a *failure*.

Definition 3.17. *Execution state*. The state of an execution E, state(E), is the set of all <X, v>-tuples where X∈ var(E) and v is the associated value of the last write to X in E.

Definition 3.18. *Execution result*. Given a program execution E, the result of E, result(E), is defined to be the set {result(E,X) | X∈ data(E)} where result(E,X) is a partial order, defined as follows:

Let <events(E(X)), Rx'> be the partial order where (ev1,ev2)∈ Rx' if and only if before(ev1, ev2, E(X)) and conflict(ev1, ev2). result(E, X) = <events(E(X)), Rx> where Rx is the transitively reduced Rx'.

For example, if E(X) =W(X,1), R(X,1), R(X,1), W(X,2), R(X,2), R(X,2), then result(E,X) is the partial order given in Figure 4.

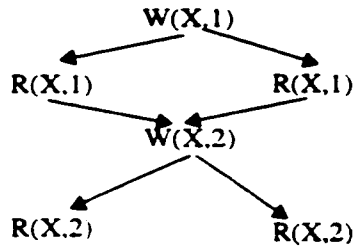


Figure 4 An example of result(E, X).

Conceptually, this definition of result(E, X) says that the observer does not care about the order in which two read accesses to X return values. It cares only about the order between conflicting accesses.

It is worthwhile to emphasize here that this definition of result is a key element in modeling relaxed machines. Since result is defined to be a partial order, not every program order needs be enforced. For example, two consecutive read operations in a program thread may be executed in either order.

Let $\mathbf{R}^*$ denote the transitive closure of the relation **R** in a partial order PO=<**G**, **R**>.

Definition 3.19. Let PO=<**G**, **R**> be a partial order and $\mathbf{G'} \subseteq \mathbf{G}$. The projection of PO on **G'** is the partial order PO'=<**G'**, **R'**> where **R'** is the transitively reduced form of {(a,b) | (a, b)$\in \mathbf{R}^*$ and a, b$\in$ **G'**}.

For example, if PO=<{a, b, c d, e}, {(a, b), (b, c), (b, e}, (c, d), (d, e)}> then the projection of PO on {b, c, d, e} is PO'=<{b, c, d, e}, {(b, c), (c, d), (d, e)}>.

3.3 Event Ordering in Parallel Program Execution

Even though events appear one after another in an execution, that does not imply that in reality a machine must follow this particular sequential order strictly to produce the same result. This is because there are many executions leading to the same result.

3.3.1 Program Order and Conflict Order

To understand relaxation of program order, we derive a partial order representation of E, denoted as *PO(E)*. PO(E) will capture *sufficient* orders among events such that as long as these orders are enforced, the execution would produce the same result(E).

The set of Next Candidates of a program thread specifies a partial order of operations for that program thread. For example, consider the example program in Figure 5. RC, under certain labeling, may have the Next Candidate sets as shown in Figure 6 below, where a number represents an operation and NC(Thread1, {1})={2, 3} indicates that after operation 1 is executed, the next candidate set for Thread 1 contains operation 2 and 3, while NC(Thread1, {1,2})={3} indicates that after operation 1 and 2 are executed, the next candidate set for Thread 1 contains operation 3 only.

Example 3.1. Initially, semaphore a=1.

```
Thread 1                Thread 2
(1) P(a);               (6) P(a);
(2) Y:=...;             (7) ...:=X;
(3) X:=...;             (8) ...:=Y;
(4) V(a);               (9) V(a);
(5) HALT;               (10)HALT;
```

Figure 5 A Simple parallel program to explain definitions

```
NC(Thread1, {}) = {1}.                NC(Thread2, {})   = {6}.
NC(Thread1, {1}) = {2, 3}.            NC(Thread2, {6}) = {7, 8}.
NC(Thread1, {1, 2})  = {3}.           NC(Thread2, {6, 7}) ={8}.
NC(Thread1, {1, 3})  = {2}.           NC(Thread2, {6, 8}) = {7}.
NC(Thread1, {1, 2, 3} = {4}.          NC(Thread2, {6, 7, 8} = {9}.
NC(Thread1, {1, 2, 3, 4) = {5}.       NC(Thread2, {6, 7, 8, 9) = {10}.
```

Figure 6 Next candidate sets for the example program

These NC sets together correspond to a partially ordered program thread as shown in Figure 7. We use PO(Ti) = <operations(Ti), POi> to denote such a partial order for the program thread *i*. PO(Ti) (POi for short) is transitively reduced.

We use $\xrightarrow{p.l}$ to represent a program order: op1 $\xrightarrow{p.l}$ op2 iff (∃Ti)(op1 and op2 ∈ operations(Ti) and (op1, op2)∈ POi). We use $\xrightarrow{p}$ to represent the transitive closure of $\xrightarrow{p.l}$.

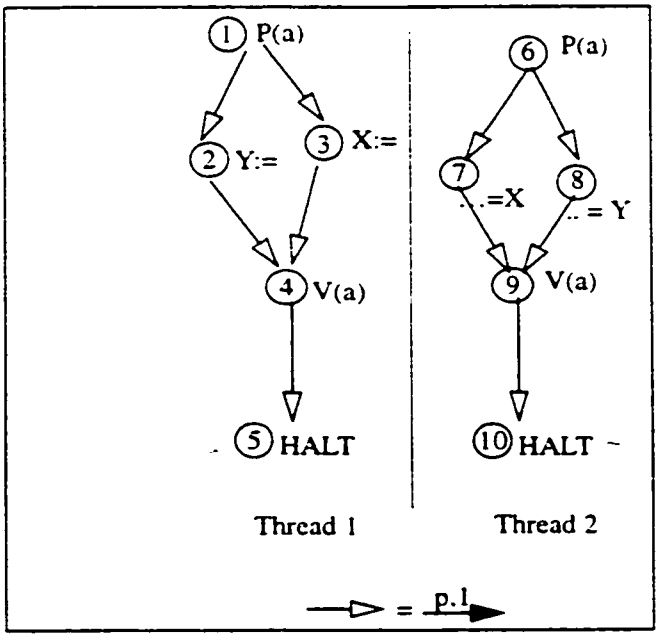


Figure 7 Possible program orders specified by RC

We will only consider those RM which specifies a program order PO(Ti) such that conflicting operations are always ordered: if op1 and op2 are two conflicting operations in operations(Ti), then either op2 $\xrightarrow{p}$ op1 or op1 $\xrightarrow{p}$ op2.

Definition 3.20. The partial order representation of an execution E , PO(E)=<Γ, **R**> is the partial order satisfying the following conditions: (i) the projection of PO(E) on to events(Ti) preserves PO(Ti), (ii) if conflict(ev1, ev2) and before(ev1, ev2, E), then (ev1,ev2)$\in$ $\mathbf{R}^*$, (iii) Γ contains every event of E, that is, Γ = events(E), and (iv) Minimality: if any ev1 or (ev1,ev2) is removed from **R**, then **R** will not satisfy (i) to (iii).

The following is a procedure to derive PO(E)=<Γ, **R**>, given E and POi which is derived from the NC set of Ti.

(1) Γ :=Φ; **R**:= Φ; i:=1;
(2) <u>while</u> i $\leq$length(E) <u>do</u>
(3) { ev1:= event(i, E); Γ := $\Gamma\cup${ev1};
(4) $\forall$ ev2$\in\Gamma$, <u>if</u> (ev2, ev1)$\in$POi <u>then</u> **R**:={(ev2, ev1)}$\cup$**R**;
(5) $\forall$ event ev2$\in\Gamma$, <u>if</u> conflict(ev1, ev2) <u>and</u> (ev2, ev1) $\notin$ $\mathbf{R}^*$ <u>then</u>
R := {(ev2, ev1)}$\cup$**R**; transitively reduce **R**;
(6) i := i+1 }
(7) $\forall$ ev1, ev2$\in\Gamma$, <u>if</u> (ev1, ev2)$\in$POi and (ev1, ev2) $\notin$ **R** <u>then</u> **R** := {(ev1, ev2)}$\cup$**R**;
(8) <u>return</u> <Γ, **R**>.

Figure 8 Derivation of PO(E) from E and POi

Here is some explanation: step (3) adds every event into Γ; step (4) accounts for every program order, according to POi, to **R**; step (5) orders every pair of conflict events; and step (7) restores any program order that is removed by transitive reduction. The correctness of the above can be reasoned: step (3) ensures that condition (iii) in Definition 3.20. is satisfied; step (4) and (7) ensure that condition (i) is satisfied, and step (5) ensures that condition (ii) is satisfied. Minimality is also achieved: if any (ev2, ev1) in step (5) is not added into **R**, condition (ii)) will be violated; (iv) is satisfied. It is noteworthy to point out that given an execution E, PO(E) is unique.

Definition 3.21. A synchronization event Q1 *immediately precedes* another synchronization event Q2 in PO(E)=<event(E), **R**>, denoted as Q1 $\xrightarrow{c,l}$ Q2, if sem(Q1) = sem(Q2), (Q1, Q2)$\in \mathbf{R}^*$ and there is no synchronization event Q3, sem(Q3) = sem(Q1), such that (Q1, Q3)$\in \mathbf{R}^*$ and (Q3, Q2)$\in \mathbf{R}^*$. We use symbol $PO_c(E)$ to denote PO(E) augmented with every $\xrightarrow{c,l}$.

We use $\xrightarrow{l}$ to denote either a $\xrightarrow{p,l}$ or a $\xrightarrow{c,l}$ that is in PO(E) (or POc(E)) and $\longrightarrow$ the transitive closure of $\xrightarrow{l}$. As an example, an execution E of the program in Figure 5 is shown in Figure 9. The derived PO(E) is the one shown in Figure 10.

```
(6) P(a)
(7) X:= ...
(8) Y:= ...
(9) V(a)
(10)HALT
(1) P(a)
(2) X:= ...
(3) Y:= ...
(4) V(a)
(5) HALT
```

Figure 9 A program execution e of the program in Figure 5

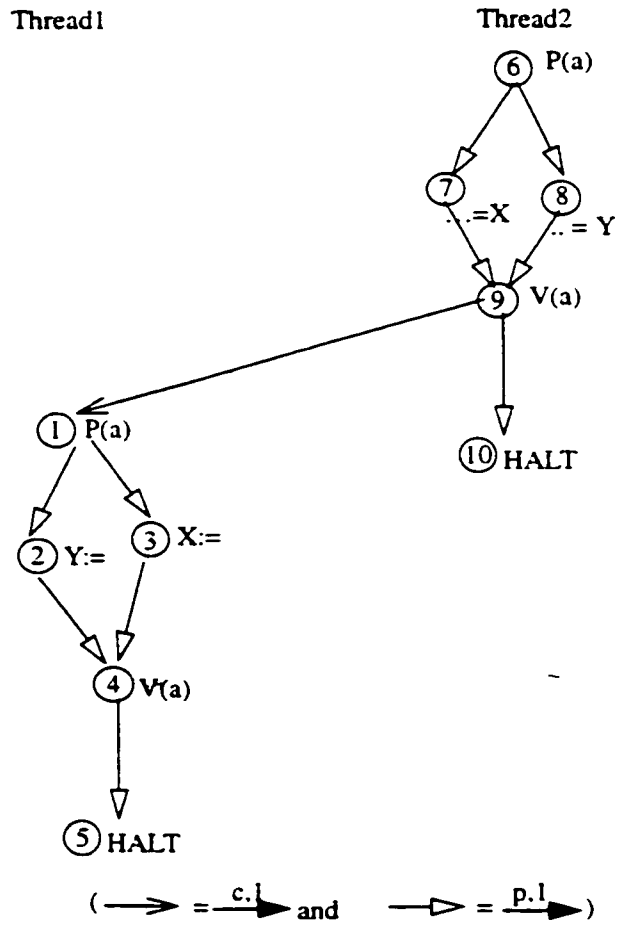


Figure 10 PO(E) from an execution of the program in Figure 5

We give some more definitions next.

Definition 3.22. A *path* from event e1 to event e2 in PO(E) (or $PO_c(E)$), denoted as path(e1,e2), is a chain of events e1=ev1, ev2, ..., ev_N=e2, such that $ev_i \xrightarrow{1} ev_{(i+1)}$ in PO(E) (or $PO_c(E)$). The path from one event to another in an execution is not necessarily unique. A path(e1,e2) is a *total-program-order-path*, denoted as *path_p(e1,e2)*, if all of its events are in a same thread.

For example, consider Figure 10. There is path_p(⑥,⑨), but not path_p(⑥,①).

Later we need to manipulate a PO(E) by removing or adding some events. Here is an outline of the procedure: Given a PO(E) in which Q $\xrightarrow{c,1}$ V. To remove Q from PO(E), we check , if there exists Q' $\xrightarrow{c,1}$ Q in $PO_c(E)$. If so and Q' and V are from two different threads then we add Q' $\xrightarrow{c,1}$ V to PO(E) after deleting Q. As a result, all events on a common semaphore will still be totally ordered in (the new) $PO_c(E)$ after such removal. Notice that after such Q is removed, the resultant partial order may not necessarily be a partial order representation of some partial execution of the *same* program.

For example, in Figure 11, if we remove Q=ev2, then ev2 $\xrightarrow{c,1}$ ev3 will be removed and Q'=ev1 $\xrightarrow{c,1}$ ev3 will be added. Thus, events on the semaphore a are still totally ordered in the resultant partial order.

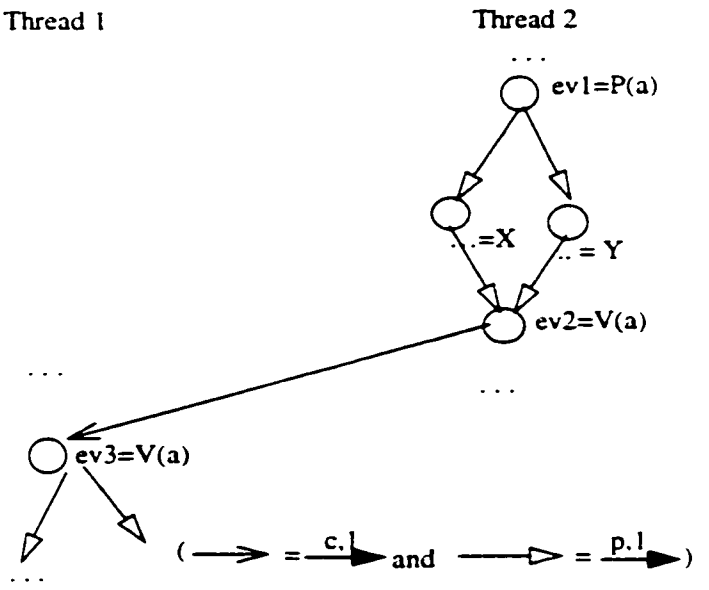


Figure 11 Manipulation of PO(E)

3.3.2 Another Example to Illustrate Definitions

To illustrate the above definitions, we present another example of parallel program in Figure 12.

Example 3.2. Consider the following parallel program whose operations are uniquely labeled as 1, 2, 3, ...

```
SHARED INTEGER X,Y;
BINARY SEMAPHORE a=FALSE;
Thread 1                          Thread 2
(1) X := 1;                       (7) P(a);
(2) Y := 2;                       (8) ... := X;
(3) V(a);                         (9) ... := Y;
(4) X := 0;                       (10) HALT
(5) ... := Y;
(6) HALT
```

Figure 12 An example parallel program with data race

Consider a RM that will not execute a V until all 'previous' data operations have been completed, and will not execute a data operation until all 'previous' P have been completed. The third constraint is that it will execute synchronization operations strictly in program order. Other than these three constraints, the RM will execute operations in each program thread in any order as long as local data dependency is maintained.

The corresponding Next Candidate Sets are:

```
NC(Thread1, {})     = {1, 1}.
NC(Thread1, {1})    = {2, 4}.
NC(Thread1, {2})    = {1, 5}.
NC(Thread1, {1, 2}) = {3, 4, 5}.
NC(Thread1, {1, 2, 3})= {4, 5}.
NC(Thread1, {1, 2, 4} = {3, 5}.
NC(Thread1, {1, 2, 5) = {3, 4}.
NC(Thread1, {1, 2, 3, 4) = {5}.
NC(Thread1, {1, 2, 3, 5) = {4}.
NC(Thread1, {1, 2, 3, 4, 5) = {6}.
```

```
NC(Thread2, {}) = {7}.
NC(Thread2, {7})= {8, 10}.
NC(Thread2, {7, 8})={9}.
NC(Thread2, {7, 9})={8}.
NC(Thread2, {7, 8, 9}) = {10}.
```

Figure 13 Next candidate sets for the example program

According to the above Next Candidate Sets, the relaxed program orders for Thread 1 and Thread 2 are given in Figure 14 next.

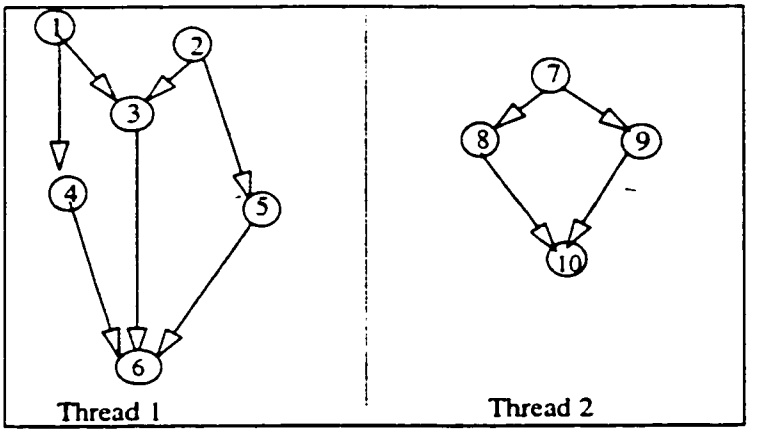


Figure 14 Example of relaxed program orders by an execution on RM

Figure 15 is a *possible* execution E1 on RM, where the first 1 in <<1, R, Y, 1>,2>> says that the event is from program *thread 1*, R and Y says that it is a read operation on variable named Y, the second 1 means that it is the *first* read operation on Y in program thread 1, the third 1 means that the event is the first instance of the operation in the execution. 2 is the return value of this event.

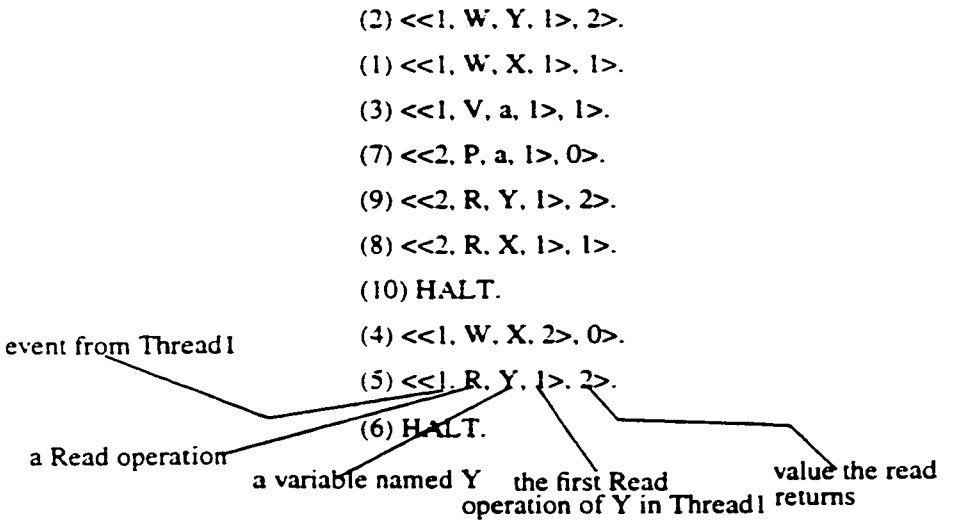


Figure 15 E1: a possible program execution on RM

The corresponding PO(E1) is the partial order in Figure 16 next.

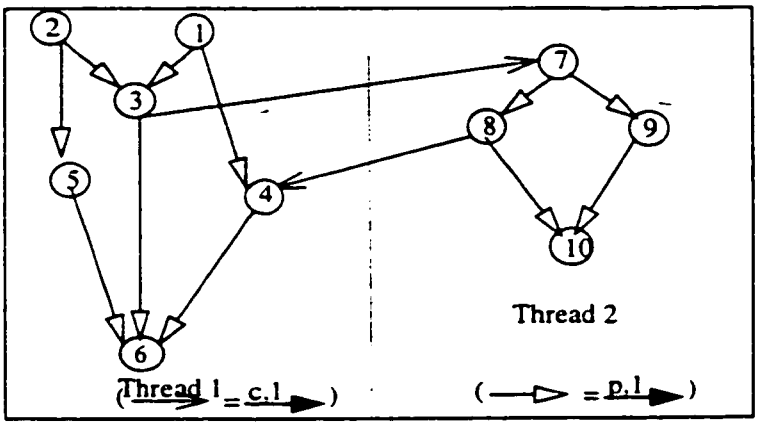


Figure 16 PO(E1) of Execution E1. Numbers correspond to those in the previous figure.

The execution result is the set of partial orders as shown in Figure 17 next.

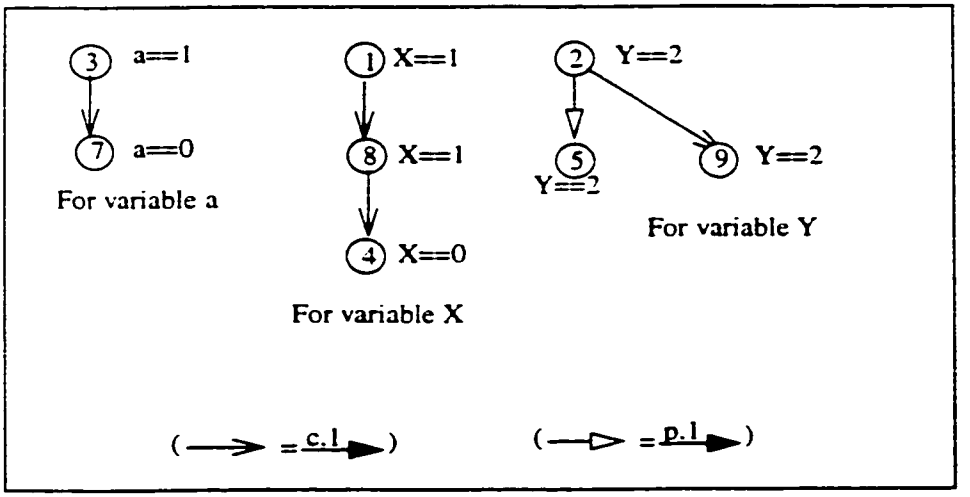


Figure 17 Execution result on RM or BM

Figure 18 shows another execution (E2) that may be generated by running the same program on the base machine BM. The corresponding PO(E2) is given in Figure 19, and sequential program orders in Figure 20. The execution result of E2 is the same as that of execution E1 as shown in Figure 17 above. We summarize various arrows representing ordering constraints in Glossary of Arrows.

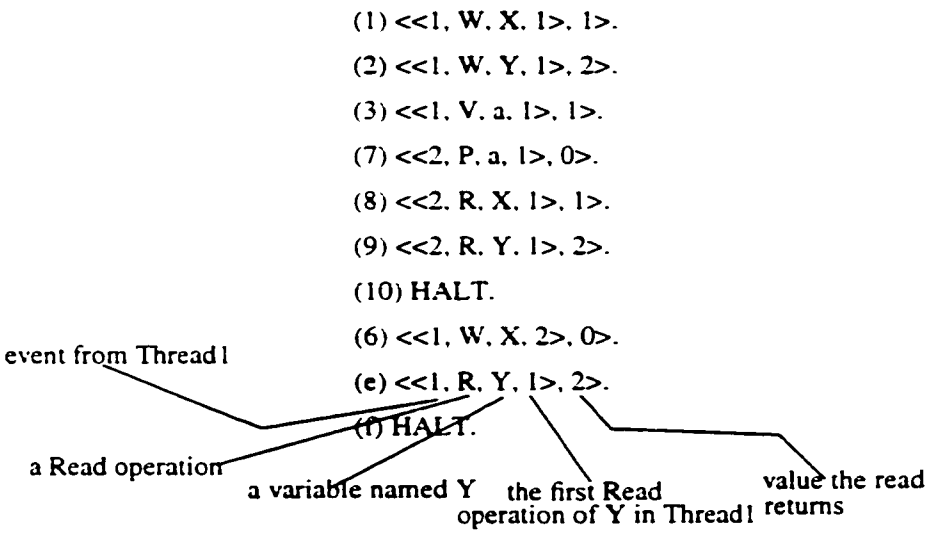


Figure 18 E2: a possible program execution on BM

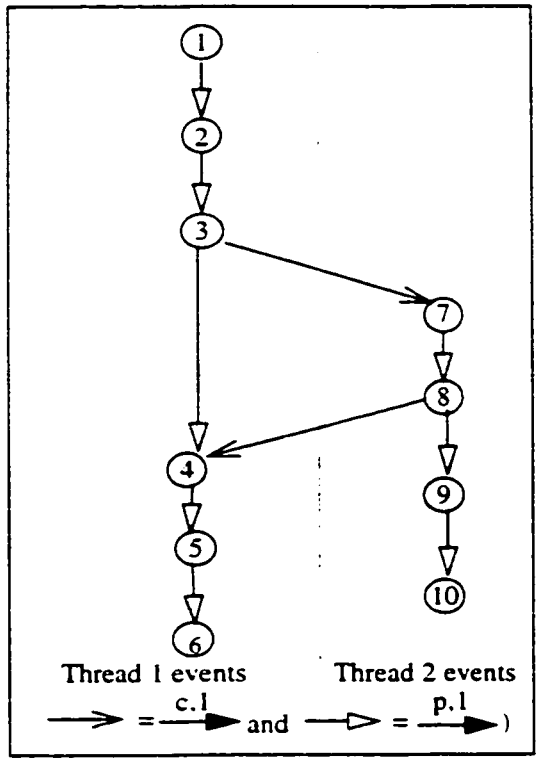


Figure 19 PO(E2) for a program execution on BM

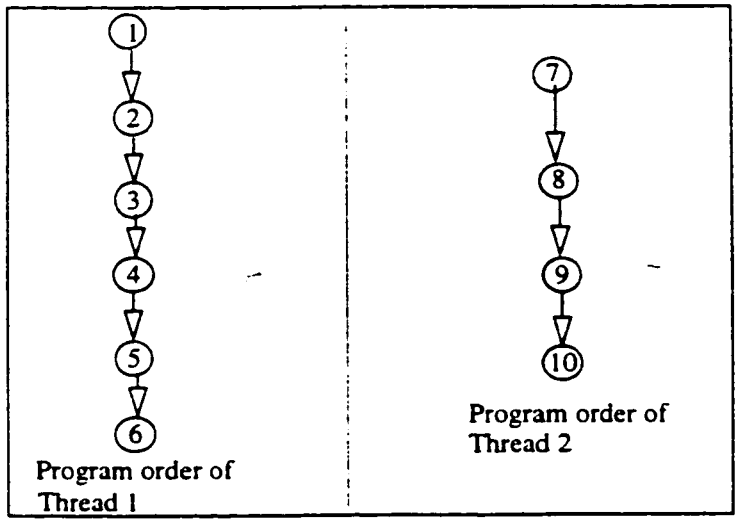


Figure 20 Sequential program orders of an execution on BM

3.4 Expected and Unexpected Executions

3.4.1 Sequentially Consistent Executions

Definition 3.23. Sequentially consistent execution. A program execution of a parallel program on an RM is sequentially consistent if and only if its result is obtainable from some execution of that program on BM. The result of a sequentially consistent execution is called sequentially consistent result.

According to the above definition, the execution E1 in Figure 15 is sequentially consistent since there is an execution E2 by BM in Figure 18 that generates the same result shown in Figure 17.

Definition 3.24. An RM machine is sequentially consistent (or it satisfies sequential consistency) with respect to a class of parallel programs iff for every parallel program in that class, the machine generates sequentially consistent results only.

Trivially, BM is, by definition, a sequentially consistent machine.

3.4.2 Data Race Free and Synchronization Failure Free

We assume that synchronous parallel programs are supposed to be data race free and contain no synchronization failure. Due to errors in a parallel program, a program execution may contain data race or synchronization failure. We define such executions next.

Definition 3.25. A parallel program execution E is *data race free* on a machine (BM or RM) if for any pair of conflicting data events, ev1 and ev2 from two different program threads, there always exists at least one path(ev1,ev2) in PO(E), which includes a

set of paired V and P events: <V1, P1>, <V2, P2>, ... , <V_n, P_n> with every V_i and V_j ($i \neq j$) from two different threads, in the form of (path_p(ev1,V1), V1 $\xrightarrow{c.l}$ P1, path_p(P1,V2), V2 $\xrightarrow{c.l}$ P2, ... , and path_p(P_n, ev2)). Figure 21 illustrates the above definition graphically. We use *path_s(ev1, ev2)* to denote such a path. We say that there is a data race between conflicting data events ev1 and ev2 from two different program threads in an execution E, denoted as *race(ev1, ev2, E)*, if there is neither path_s(ev1, ev2) nor path_s(ev2,ev1) in PO(E).

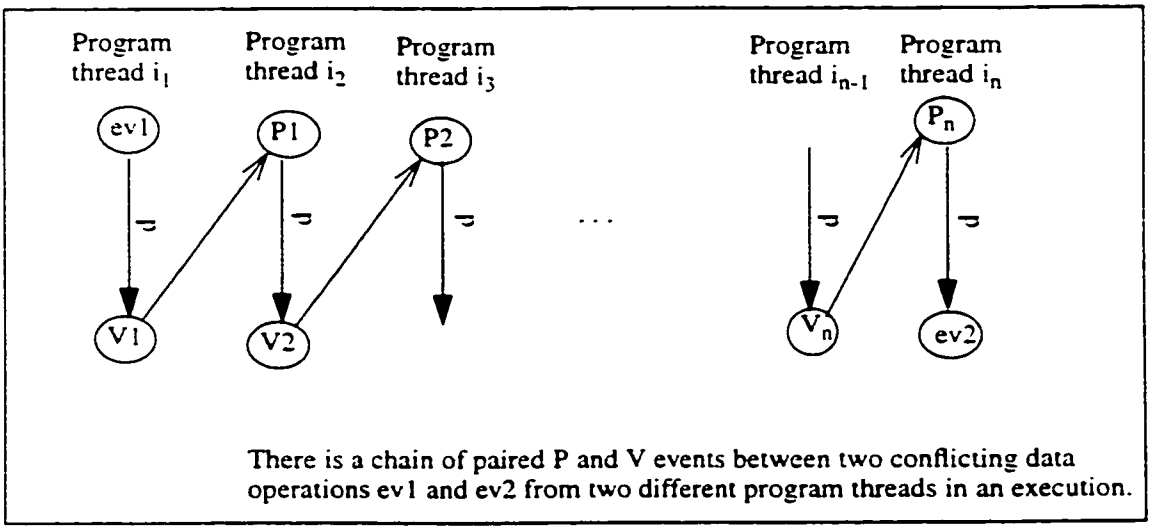


Figure 21 Illustration of data race freeness

Definition 3.26. A parallel program is *data race free* on a machine (BM or RM) if every execution of the program on the machine is data race free. Otherwise, we say a program contains data race.

As an example, the execution in Figure 16 (or Figure 19) of the program in Figure 12 contains data race since there are two conflicting events ⓓ and ⓗ from two different program threads and there is no path_s(h,d) in Figure 16 (or Figure 19).

Assumption 3.1. We assume that unless otherwise specified, parallel programs are data race free in this thesis.

Other than data race, it is also possible that there is synchronization failure in some parallel program.

Assumption 3.2. We assume that in the first part (up to Chapter 5) of this thesis, unless otherwise specified, parallel programs do not contain synchronization failure.

Definition 3.27. *Pdsf* is a class of parallel programs that are data race free and contain no synchronization failure on BM.

3.4.3 Sequentially Consistent Machines and Pdsf

We focus our discussion on how to relax sequential program orders in Pdsf from now on. That is, we will only consider sequentially consistent RM that executes Pdsf. Since a program in Pdsf is data race free on a BM, one expects that it must be so on such an RM too. Similarly, a program in Pdsf must contain no synchronization failure on such an RM. In next chapter, we will present an RM that is sequentially consistent for programs in Pdsf. We also show that the RM is data race free and synchronization failure free for programs in Pdsf.

Chapter 4 Link Consistency

In the previous chapter we have introduced RM, a general relaxed machine to capture the essence of memory consistency models that relax some unnecessary program orders. RM selects its next operation for execution randomly from a Next Candidate Set. In this chapter we introduce a new instance of RM, called RMlc. The 'next operations' of RMlc are defined by the Conditions of Link Consistency. We show that RMlc is a sequentially consistent machine with respect to Pdsf.

The key characteristic of RM is the multiplicity of 'next operations' in a program thread in execution. For example, we can model RC by a special RM, called RMrc whose 'next operations' are exactly defined by the three conditions of RC. RMrc is supposed to be a sequentially consistent machine with respect to properly-labeled parallel programs.

At the end of this chapter, we will compare the capability of RMrc and RMlc in relaxing sequential program orders through some examples.

4.1 Data Race versus Synchronization Race

4.1.1 Ordering Among Synchronization Operations

Synchronization operations are used to ensure some proper ordering or atomicity of data operations from various threads. For example, one thread should start to consume data after another thread has produced it; or two threads should not modify the same piece of data at the same time. Data race is considered abnormal and bad for synchronous programs. Extensive research (Netzer and Miller 1990, 1991) has been conducted recently to

detect its occurrence in a given execution. Synchronization race refers to race conditions among synchronization operations, such as an available 'token' generated by a V operation which may be consumed by one of the compatible P operations. Unlike data race, synchronization race is often considered to be a prime source of desirable nondeterminism.

Not all conflicting synchronization operations can race. It depends on the roles played by these synchronization operations in the program. By assuming that a parallel program is data race free and synchronization failure free, one could relax some of the sequential program orders. Similarly, by assuming that some synchronization operations in a parallel program race only in some special patterns, additional program orders can be relaxed. We next present a motivating example to illustrate when and how program orders among some synchronization operations can be relaxed.

4.1.2 A Motivating Example For Link Consistency

Example 4.1. A motivating example in which program order among synchronization operations can be relaxed. Consider the following toy parallel program that computes the formula U:=X+Y and W:=X+Z in Figure 22, where X, Y and Z represent the values that are computed by individual processors (P2, P3, and P4), U and W are computed by another processor (P1). For simplicity, assignment statements are not broken down into individual read, write and arithmetic operations.

```
BINARY     SEMAPHORE   flag1=flag2=flag3=FALSE;
SHARED     INTEGER     X, Y, Z;
LOCAL      INTEGER     U, V;

P1                P2                P3                P4

(1)P(flag1);(Pe)  (6) X:=...;       (8) Y:=...;       (10)Z:=...;
(2)P(flag2);(Pe)  (7) V(flag1);(Ve) (9) V(flag2);(Ve) (11)V(flag3);(Ve)
(3) U:=X+Y;       (13) HALT.        (14) HALT.        (15)HALT.
(4) P(flag3);(Pe)
(5) W:=X+Z;
(12) HALT.
```

Figure 22 An example for relaxing program order among synchronization operations

If we assume that a P operation will delay any data operation after it until the P is completed, and a V will be deferred until all data operations before it has been executed, then it is easy to see that in an execution, there is no need to order the execution of the three P operations: P(flag1), P(flag2) and P(flag3) can be executed in any order. One of the reasons that make such relaxation possible is that P(flag1) (as well as P(flag2), or P(flag3)) occurs in one program thread only. We will call such P operations *Exclusive Consumers*. The minimal ordering that need to be maintained is the inter-processors data dependency as shown in Figure 23.

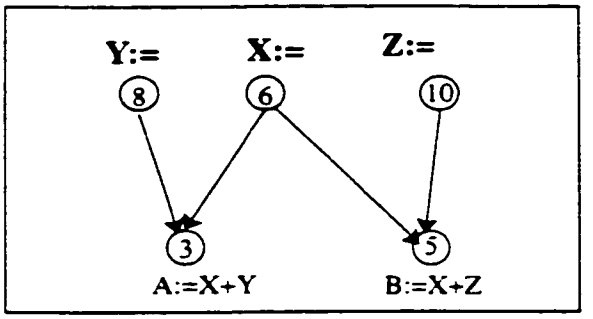


Figure 23 An Inter-processor Data Dependency Graph

4.2 Link Consistency

What makes a synchronization operation an exclusive consumer or an exclusive producer? How is it useful in defining a weaker memory model? These are the questions addressed in this section.

4.2.1 Exclusive Producer and Exclusive Consumer

Definition 4.1. Given a parallel program and a machine RM, there is *token collision* on a given semaphore *a* if there exists an execution Ei of the program on the machine such that there are two different events ev1 and ev2 on semaphore *a*, ev1,ev2∈ events(Ei), both ev1 and ev2 are V events, and ev1 $\xrightarrow{c.l}$ ev2 in PO(Ei). In such a case we also say that semaphore *a* has a token collision in Ei and that the parallel program contains token collision.

Definition 4.2. A given semaphore *a* in a program is *token collision free* if *a* has no token collision in any execution of that program. That is, the occurrence of any V(*a*) operation excludes the occurrence of another V until some P(a) operation is executed.

Therefore, we call such V(a) operations (and the semaphore a) *Exclusive Producer.* A program is said to be *token collision free* if every semaphore it contains is token collision free.

Definition 4.3. For a given semaphore a in a program, if a is an Exclusive Producer and all P(a) operations occur in one program thread only, then we call such P(a) operations (and the semaphore a) *Exclusive Consumer.*

By definition, an Exclusive Consumer semaphore must also be an Exclusive Producer. In the rest of this chapter, an exclusive producer for semaphore a will be denoted by Ve(a) and an exclusive consumer for semaphore a by Pe(a). In the context where no confusion will be caused, we also use Ve or Pe for short.

4.2.2 Definition of Link Consistency

We define Link Consistency (LC) and RMlc. To define a specific RM, one has to first define NC(i), NC for i^{th} program thread Ti of a parallel program. NC(i) depends on what is already executed and what is left in Ti, which is denoted as M(i). Operations in M(i) form in sequence. In the absence of branching, if an operation op1 appears before another operations op2 then we say op1 precedes op2. Next we define NC(i) $\subseteq$ M(i) for RMlc.

Definition 4.4. The current *next candidate set* for Ti for RMlc, NC(i), is defined to be a subset of M(i) that satisfies all of the following conditions:

Conditions of LC:

(i) Data operation D

$D \in NC(i)$ iff $\neg\exists y \in M(i) \wedge y$ precedes D in Ti such that

(LC1) conflict(D.y), or

(LC2) y is a P.

(ii) Synchronization V

$V \in NC(i)$ iff $\neg\exists y \in M(i) \wedge$ y precedes V in Ti such that

(LC3) y is a D (R or W), or

(LC4.1) y is a $Q \neq Ve$.

(iii) Synchronization P

$P \in NC(i)$ iff

(LC4.2) P is an exclusive consumer, or

(LC4.3) $\neg\exists y \in M(i) \wedge$ y precedes P in Ti such that y is a $Q \neq Ve$.

Condition LC1 ensures local data dependency: NC(i) does not contain more than one conflicting operation from a same thread at any time. Condition LC2 is for safe consumption: a D is not put into NC(i) if there is any P in current NC(i) that precedes it in M(i). Condition LC3 is for safe production: a V is not put into NC(i) if there is any D in current NC(i) that precedes it in M(i). Condition LC4 together say that there is at most one synchronization operation in NC(i) at any time with the exceptions of (1) multiple Pe and possibly some Q that precedes them (LC4.2) or (2) multiple Q and possibly some Ve that precedes them (LC4.1 and LC4.3) can be in the same NC(i).

Definition 4.5. An *RMlc* (Relaxed Machine define by LC) is a RM whose next operations are as defined above.

Informally, LC conditions capture the following relaxations (R0 to R4) of sequential program order as shown in Figure 24 next. Notice that the relaxations R0, R1 and R2 are very similar to those relaxed by RMrc.

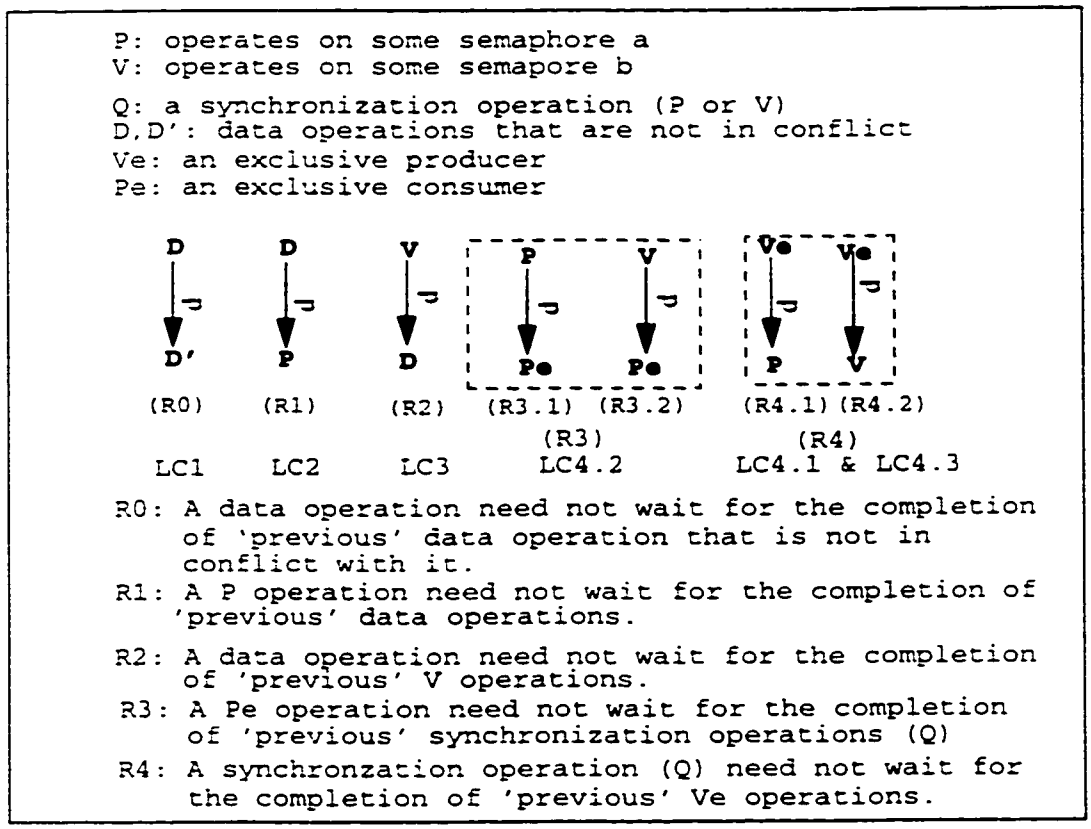


Figure 24 Informal representation of relaxations of program orders by LC (Rule R0 to R4)

There is a corresponding mapping from the above Conditions of LC and Relaxations of LC as follows. Altogether, L1 through L4 specifies NC and PO(Ti), the partial order for each program thread Ti.

(i) R0 corresponds to LC1: two data operations not in conflict can be executed in any order; R1 corresponds to the LC2: The execution of a data operation that

follows a P can only be started after the completion of the P, but a data operation and a P that follows it can be executed in any order;

(ii) R2 to LC3: the execution of a V can only be started after the completion of all data operations that precedes it, but a V and a data operation that follows it can be executed in any order;

(iii) R3 to LC4.2: a synchronization operation and a Pe that follows it can be executed in any order; and

(iv) R4 to LC4.1 and LC4.3: a Ve and a V that follows it can be executed in any order (LC4.1) and a Ve and a P that follows it can be executed in any order (LC4.3).

4.2.3 An Example for LC

Next we use an example program of two threads in Figure 26 to show some examples of Pe and Ve.

The program contains mutual exclusive accesses to variable X and producer/consumer type of synchronization for accessing variable Y. Operations Ve or Pe are indicated using (Ve) or (Pe). For Thread 1, the NC sets are given in Figure 25 and the partially ordered program order that LC enforces is given in Figure 27, in which the operation 3 (Ve(a)) and the operation 5 (Ve(b)) are not ordered because of the relaxation R4.2; while the operation 3 (Ve(a)) and the operation 4 (a data operation) are not ordered because of the relaxation R2.

```
NC(Thread1, {})={1};
NC(Thread1, {1}) = {2,4};
NC(Thread1, {1,2}) = {3,4};
NC(Thread1, {1,4}) = {2};
NC(Thread1, {1,2,4}) = {3,5};
NC(Thread1, {1,2,3}) = {4};
NC(Thread1, {1,2,4,5}) = {3};
NC(Thread1, {1,2,3,4}) = {5};
```

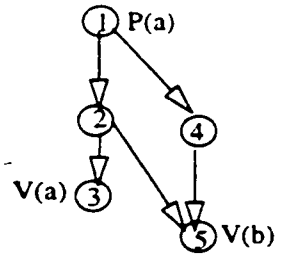
Figure 25 NC sets for thread 1 for Example 4.2

```
BINARY SEMAPHORE a, b;
SHARED INTEGER X, Y;
LOCAL INTEGER W, Z;
Initially X=Y=0; a=1; b=0;

Thread 1                        Thread 2

1 P(a);                         6 P(b); (Pe)
2 X:=1;                         7 Z:=Y;
3 V(a); (Ve)                    8 P(a);
4 Y:=1;                         9 W:=X;
5 V(b); (Ve)                    10 V(a); (Ve)
11 HALT.                        12 HALT.
```

Figure 26 Example 4.2 with both Mutual Exclusion and Producer Consumer synchronization



2 and 4: data operations
Halt operations are not shown.

Figure 27 Partially ordered program order for Thread 1 under LC.

4.3 Relating Link Consistency to Sequential Consistency

4.3.1 Main Results Concerning Link Consistency

In this section, we show our main result concerning LC with the following theorem.

Theorem 4.1 RMlc is a sequentially consistent machine for programs in Pdsf.

By this theorem, RMlc can use the relaxations R1 to R4 to execute a program from Pdsf and guarantee sequentially consistent result. R1 through R4 specify a partially ordered program order for each program thread. If RMlc has multiple processors, operations that are in such partial order can be executed concurrently and as a result, a program may be executed in a shorter time.

Other related results that are established include the following:

Theorem 4.2 Any parallel program from Pdsf is data race free on RMlc.

Theorem 4.3 If a semaphore in a program from Pdsf is token collision free on BM, then it is token collision free on RMlc too.

Theorem 4.4 Any parallel program from Pdsf is synchronization failure free on RMlc.

4.3.2 Proof Strategy

As we show in Figure 24, LC allows four kinds of relaxations of sequential program orders. The relaxation R0 is the trivial one and we can immediately see, from the definition of the execution result (Definition 3.19.) and the assumption that a parallel program is data race free (section 3.4.2, Assumption 3.1), that it is valid. The execution result, defined

to be a partial order, does not care the relative ordering of execution of two data operations from the same thread if they do not conflict.

The relaxations R1 and R2 are related to ordering between data operations and synchronization operations, while R3 and R4 relax sequential program orders among synchronization operations themselves.

Our approach to prove a machine M1 is sequentially consistent for programs in Pdsf is to show that for any maximal execution E1 on machine M1, we can *construct* another execution E2 on a machine M2, known to be sequentially consistent for programs in Pdsf, such that result(E1) = result(E2). Then, according to Definition 3.23. M1 is sequentially consistent.

Knowing that BM is sequentially consistent, we use BM to prove that RMlc_1, which is the same as BM except it uses relaxation R1 and R2, is sequentially consistent for programs in Pdsf. Then we use RMlc_1 to prove that RMlc_2, which uses relaxation R1, R2 and R4, is a sequentially consistent machine for programs in Pdsf.

Afterwards, we use RMlc_2 to prove that RMlc_3, which uses relaxation R1, R2, R3.1 and R4, is a sequentially consistent machine for programs in Pdsf. Finally, we use RMlc_3 to prove that RMlc is a sequentially consistent machine for programs in Pdsf. Details of the proofs are given in Appendix A.

We have attempted to further relax the rule R3 as shown in Figure 24. Unfortunately, our intuitive extension may lead to violation of sequential consistency. For example, consider

the program in Figure 28 below. Operations are indexed with number 1, 2, ... The relationship between operation 1 and operation 2, at first glance, resembles the relationship between two operations of R3.1 in Figure 24: both semaphore a and semaphore b are token collision free on BM and operation 1 is a Pe (But operation 2 is *not* a Pe since P(a) occurs in both threads). If we relax the program order between operation 1 and 2, we can construct a deadlocked execution of the program (by having operation 3 enable operation 2) even though the program really contains no synchronization failure on any sequentially consistent machine.

```
BINARY SEMAPHORE a, b;
Initially a=b=0;

Thread 1                    Thread 2
1 P(b);                     3 V(a);
2 P(a);                     4 P(a);
7 Halt.                     5 V(b);
                            6 V(a);
                            8 Halt.
```

Figure 28 An example calling for caution in relaxation of program orders

4.4 Refining Link Consistency

In this section, we try to improve LC further. The program order P(a) $\xrightarrow{p}$ D1 is enforced in RMlc. So is the program order D2 $\xrightarrow{p}$ V(a). The normal perception for the necessity of such an ordering constraint is that a synchronization operation V(P) 'protects' *all* data operations before (after) it to achieve desirable property of a program such as data race freeness. It does not have to be always so, however.

Definition 4.5. We say that P(a) *does not protect* D1 if, after P(a) $\xrightarrow{p}$ D1 is removed from

$PO(Ti)^*$, RMlc still produces sequentially consistent result. Similarly we say that V(a) *does not protect* D2 if, after D2 $\xrightarrow{P}$ V(a) is removed from $PO(Ti)^*$, RMlc still produces sequentially consistent result. Otherwise we say that the P(a) *protects* D1 or V(a) *protects* D2.

4.4.1 Determination of 'Protect' and 'Not Protect' Relation

To automatically determine the 'not protect' relation of a parallel program without running it is in general impossible. However, the program designer may have such a knowledge at design time.

For example in Figure 22 , the programmer knows that P(flag2) in Thread 1 does not protect W:=X+Z in the same thread. This is because W:=X+Z is not in conflict with Y:= ... in Thread 3 which contains the only V(flag2) that can enable P(flag2). Thus P(flag2) cannot be in any path (which consists of paired V and P as shown in Figure 21) separating W:=X+Z from X:= ... in Thread 2 or X:=... in Thread 4. Therefore operation P(flag2) in Thread 1 does not protect data operation W:=X+Z, and the latter can be executed even before P(flag2) is started.

4.4.2 Implementation of 'Protect' and 'Not Protect' Relation

The 'protect' or 'not protect' information has to be conveyed by a programmer to RMlc so that proper program orders can be enforced or relaxed. RMlc, however, has to provide a means for the programmer to express such information easily.

It is possible to devise some language constructs to specify *exactly* such 'protect' or 'not protect' information in a program. However, it may not be cost effective in practice. Compromises have to be made between the accuracy in capturing such relations and its cost of implementation.

(Fu 1992) proposes a simple construct called *Scope Tag* to convey the ordering constraints between synchronization operations and data operations, as well as ordering constraints between synchronization operations themselves. Each synchronization operation is associated with a scope tag, a three element structure, <*type, direction, affected_operations*>. The first element indicates the type of operations: being data operation or synchronization operations; the second indicates the direction of the ordering constraints: below or above; and the third specifies a subset of operations of the type being affected. Direction 'below' is associated with a P operation and direction 'above' is for a V operation. We refer the scope of a synchronization operation to be the *set* of the affected operations according to its scope tag.

Consider Example 4.1 in Figure 22 again. P(flag1) in Thread 1 will carry the scope tag <'data operations', 'below', 'all'> which say that P(flag1) will delay all data operations that follows it until it is executed; its scope is the set of all data operations that follows it. P(flag2) will carry the scope tag <'data operations', 'below', 'up to next P'> which says that P(flag2) will delay all data operations that are between P(flag2) and P(flag3) until P(flag2) is executed; its scope is the set of all data operations that are between P(flag2) and P(flag3). V(flag1) in thread 2 will carry the scope tag <'data operations', 'above', 'all'>

which says that before V(flag1) can be executed, all data operations before it must have been executed already; its scope is the set of all data operations that precede it.

We use LCs (RMls) to denote LC (RMlc) that uses scope tags. Conditions of LCs are same to those of LC except for conditions listed below:

Conditions of LCs:

(i) Data operation D

$D \in NC(i)$ iff $\neg\exists y \in M(i) \wedge$ y precedes D in Ti such that

(LC1) conflict(D,y), or

(LC2) y is a P and D is in the scope of P.

(ii) Synchronization V

$V \in NC(i)$ iff $\neg\exists y \in M(i) \wedge$ y precedes V in Ti such that

(LC3) y is a D (R or W) and D is in the scope of V, or

(LC4.1) y is a $Q \neq Ve$.

4.4.3 Compare LC and LCs with an Example

We assume that hardware recognizes and enforces ordering constraints expressed in scope tags. For Example 4.1 in Figure 22 with the scope tags in section 4.4.2, RMls will only enforce the ordering constraints as shown by solid arrows in Figure 29, the minimal one according to Figure 23. Without using any scope tag, RMlc will also enforce the dashed arrow, which is an unnecessary; RMlc enforces it simply because

P(flag2) $\xrightarrow{p}$ D1=(W:=X+Z). Notice that RMrc also will enforce this dashed arrow, regardless of how labeling is applied.

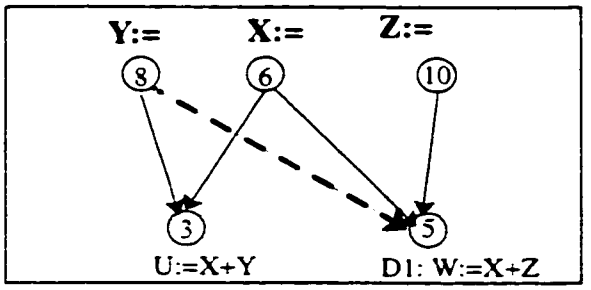


Figure 29 Ordering enforced by RMlc among data operations.

4.5 Comparison with Other Relaxed Memory Consistency Models

Next we give three examples to show the differences between RC and LCs as well as the differences between LC (LCs) and PLpc in relaxing unnecessary program orders.

4.5.1 Some Explanations on Comparison

RC allows operations to be labeled arbitrarily as long as the program is Properly-Labeled. For a given program (that uses semaphores only for synchronization), before we can compare LCs with RC, the program has to be labeled with *acqL*, *relL*, or *ordinaryL*. There are many possible proper labeling of a given program. In our comparison, we adopt a *natural* way of labeling. We label a P(a) with acqL, a V(a) with relL, and a data operation ordinaryL.

It is easy to show that under this natural labeling, RMrc is more restrictive than RMlc (and RMls) in relaxing program orders. The reasoning is as follows: (1) RMlc can treat every P(a) and V(a) as non-exclusive operations and only R0 to R2 in Figure 24 would

apply. Since R0 to R2 relax the same as RMrc does, RMlc relax at least as much as RMrc does: (2) There are programs in which some P is an exclusive consumer, some V is an exclusive producer, and RMlc relaxes program order involving such P and V using R3 and R4 to while RMrc does not.

Because of the liberal way of labeling offered by RC, it is not our intent to use the following examples to conclude that LC (or LCs) is strictly or theoretically more flexible. What we do know is, however, that for the following examples LC (or LCs) is truly more flexible than RC. That is, there *cannot* exist a possible labeling by RC for any of these examples such that the RMrc will enforce the same or fewer program order than RMlc (or RMls). We believe this is also true for many other applications.

4.5.2 Example 4.1 Revisited

Consider the program in Figure 22 again. The ordering constraints graph among operations of program thread 1 that RC enforces is depicted in Figure 30 (i). In this graph, some unnecessary delays are imposed among some operations, such as the one between operation 1 and operation 2. LC enforces another ordering constraints graph as shown in Figure 30 (ii), which only enforces the necessary ordering constraints as dictated by Figure 23.

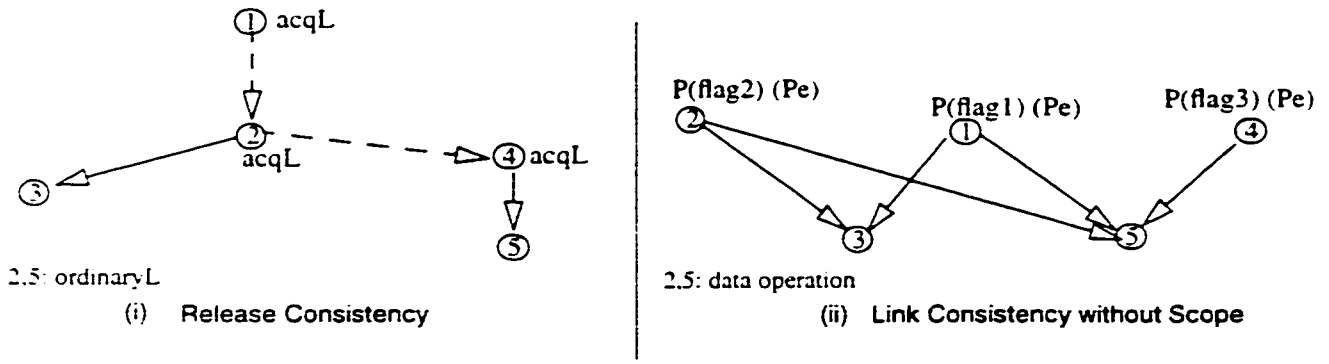


Figure 30 Ordering constraints graphs of operations in Thread 1 of Example 4.1. The example is in Figure 22. (i) uses RC and (ii) uses LC. Dashed arrows indicate unnecessary delays which are absent if LC is used.

It is easy to verify that no matter how one labels operations in thread 1 in Figure 22 using RC, one cannot get the partial orders in Figure 30 (ii). Consider all possible labeling of operation (5) by RC: relL, acqL, or ordinaryL. We show each labeling cannot lead to Figure 30 (ii):

If operation (5) is labeled as relL, then no matter how operation (3) is labeled, RC will enforce (3) $\xrightarrow{p}$ (5) which is absent in Figure 30 (ii). If operation (5) is labeled as acqL, then operation (3) must be labeled as ordinaryL (any other labeling will lead to (3) $\xrightarrow{p}$ (5) by RC, which is absent Figure 30 (ii)), which implies that both operation (1) and (2) must be labeled as acqL (to have (1) $\xrightarrow{p}$ (3) and (2) $\xrightarrow{p}$ (3)). But if operation (1) and (2) are so labeled, RC will enforce (1) $\xrightarrow{p}$ (2), which is absent in Figure 30 (ii).

At last, if operation ⑤ is labeled as ordinaryL, then both operation ① and ② must be labeled as acqL (to have ① $\xrightarrow{p}$ ③ and ② $\xrightarrow{p}$ ③), which, same as the above case, implies that RC will enforce ① $\xrightarrow{p}$ ②, which is absent in Figure 30 (ii).

Therefore, The labeling of RC cannot achieve the partial order in Figure 30 (ii).

Notice that in the above comparison, LC, i.e., Link Consistency without scope, is used.

4.5.3 Example 4.2 Revisited

We use the example program in Figure 26 again. We assume that scope tags indicate that in both program threads of the program, a P operation protects the data operation that immediately follows it only and a V operation protects the data operation that immediately precedes it only. The ordering constraints graph among operations of program thread 1 that RC enforces is depicted in Figure 31 (i). In this graph, some unnecessary delays are imposed among some operations of Thread 1, such as the one between operation 3 and operation 5.

It is easy to verify that no matter how one labels operations in thread 1 in Figure 26 using RC, one cannot get the partial orders in Figure 31 (ii). Figure 31 (ii) does not enforce an ordering constraint between P(a) and operation ④. For RC to do the same, at least one of them must be labeled as ordinaryL. But this is impossible since it will imply that operation ⑤ must be labeled as relL (to have ① $\xrightarrow{p}$ ⑤ or ③ $\xrightarrow{p}$ ⑤), which will further imply ② $\xrightarrow{p}$ ⑤ since a relL operation in RMrc will wait until all previous operations have been completed. ② $\xrightarrow{p}$ ⑤ is an ordering constraint absent

in Figure 31 (ii). Therefore the labeling of RC cannot achieve the partial order of thread 1 in Figure 31 (ii).

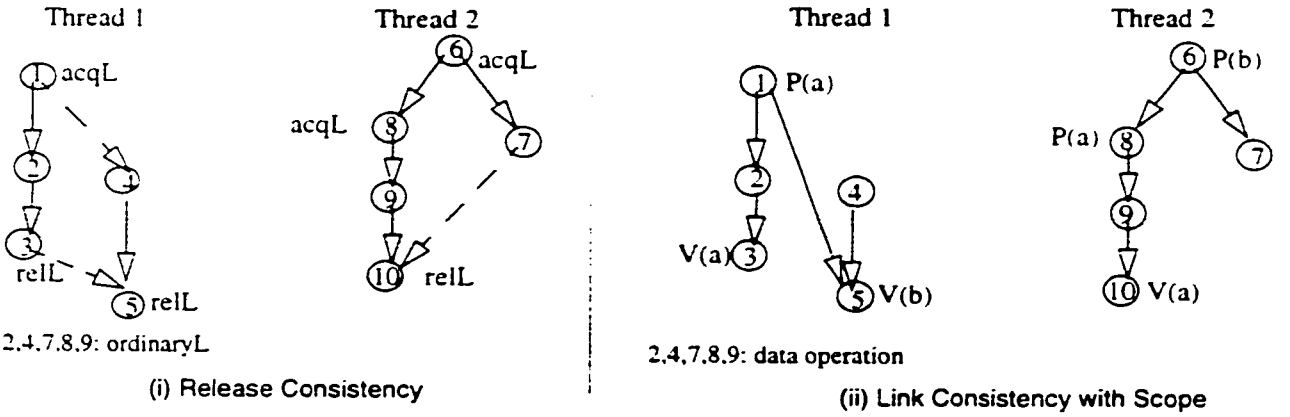


Figure 31 Ordering constraints graphs of Example 4.2. (i) is enforced by RC and (ii) is by LCs. Dashed arrows indicate unnecessary delays which are absent with LCs.

Notice that that labeling of RC can achieve the partial order of thread 2 in Figure 31 (ii). One way is to label operation (7) as ordinaryL and the rest of operations in thread 2 as acqL. Also notice that if LC instead LCs is used, then the partial order for Thread 1 in Figure 31 (ii) will be the one in Figure 32 next.

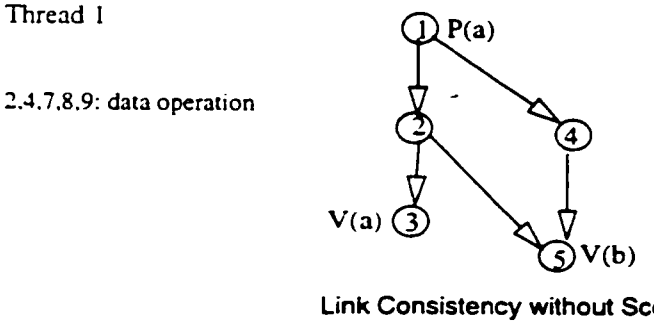


Figure 32 Ordering constraints graphs of Example 4.2 as enforced by LC

We show next that no matter how one labels operations in thread 1 in Figure 26 using RC, one cannot get the same partial order using RC. This partial order does not enforce an ordering constraint between operation (3) and operation (5). For RC to do the same, at least one of them must be labeled as ordinaryL. If (5) is labeled as ordinaryL, then, both (2) and (4) must be labeled as acqL to have (2) $\xrightarrow{P}$ (5) and (4) $\xrightarrow{P}$ (5). But this is impossible since it will imply that (2) $\xrightarrow{P}$ (4) which is absent in Figure 32. So (5) must be labeled as acqL or relL and (3) labeled as ordinaryL (and (2) labeled as acqL to have (2) $\xrightarrow{P}$ (3)). We show such a labeling is also impossible next.

If (5) is labeled as relL, then RC will enforce (3) $\xrightarrow{P}$ (5) which is absent in Figure 32. But if (5) is labeled as acqL, then (4) must be labeled as acqL or relL, which is impossible since it will imply (2) $\xrightarrow{P}$ (4) which is absent in Figure 32.

Therefore the labeling of RC cannot achieve the partial order of thread 1 in Figure 32. Notice that that RC can achieve the partial order of thread 2 in Figure 31 (b). One way is to label operation (7) as ordinaryL and the rest of operations in thread2 as acqL. Also notice that for this example, some critical ordering constraints such as the one between operation (6) and (8) in Thread 2 have to be enforced on both RC and LCs. It is rather easy to see the relaxation of the arrow between operation (6) and operation (8) (both are P operations) will lead to some result that is not sequentially consistent.

4.5.4 L-U Decomposition, a Larger Example (Example 4.3)

A concrete example based on L-U decomposition is presented next. The problem definition and the algorithm design are discussed in detail in (Fu 1992). Suppose L-U decomposition of 8×8 matrices is computed using three processors. A sub-task is performed by two processors. The sub-task is based on the data dependency graph shown in Figure 33 (a). Specifically Processor 1 computes A_i, $1 \le i \le 4$, and Processor 2 computes A_i, $5 \le i \le 8$. A_i stands for some computation involving two distinct elements (such as a_{34} and a_{35}) of the matrix to be decomposed. Using binary semaphores, a program for this sub-task can be written and it preserves exactly the original data dependency with LCs. The ordering constraints enforced by LCs is given in Figure 33 (c). Operations 1, 3, 5 and 7 are *P operations* on individual semaphores, and 10, 12, 14 and 16 are V operations on individual semaphores. With RC, the augmented data dependency graph is shown in Figure 33 (b), and the ordering constraints enforced in each processor are shown in Figure 33 (d).

To compare these machines, we assume the same hardware parameters as those in (Gharachorloo et al. 1990): the latency of a cache miss is 40 cycles, and the service time (the shortest time delay between the issue of two consecutive accesses that miss in a cache) is 10 cycles. We assume also that after each write access (including V) the caches have to be brought to a coherent state, that is, every write is treated as a miss. We further assume that other conditions are ideal, such as computations of all A_i's of a processor are pipelined, instructions are issued per cycle if there is no cache miss. Consequently it takes

150 cycles (100 cycles for local computation and 50 cycles for two independent writes) to compute A_i.

We focus on the execution on Processor 2. Figure 34 shows the executions on RC and LCs respectively. RC allows the pipelined execution of A_i's (9, 11, 13, and 15), but the execution of the P operations (10, 12, 14, and 16) is sequential. Altogether it takes 310 (150+4*40) cycles for Processor 2 to complete its computation. LCs takes 220 (150+3*10+40) cycles only.

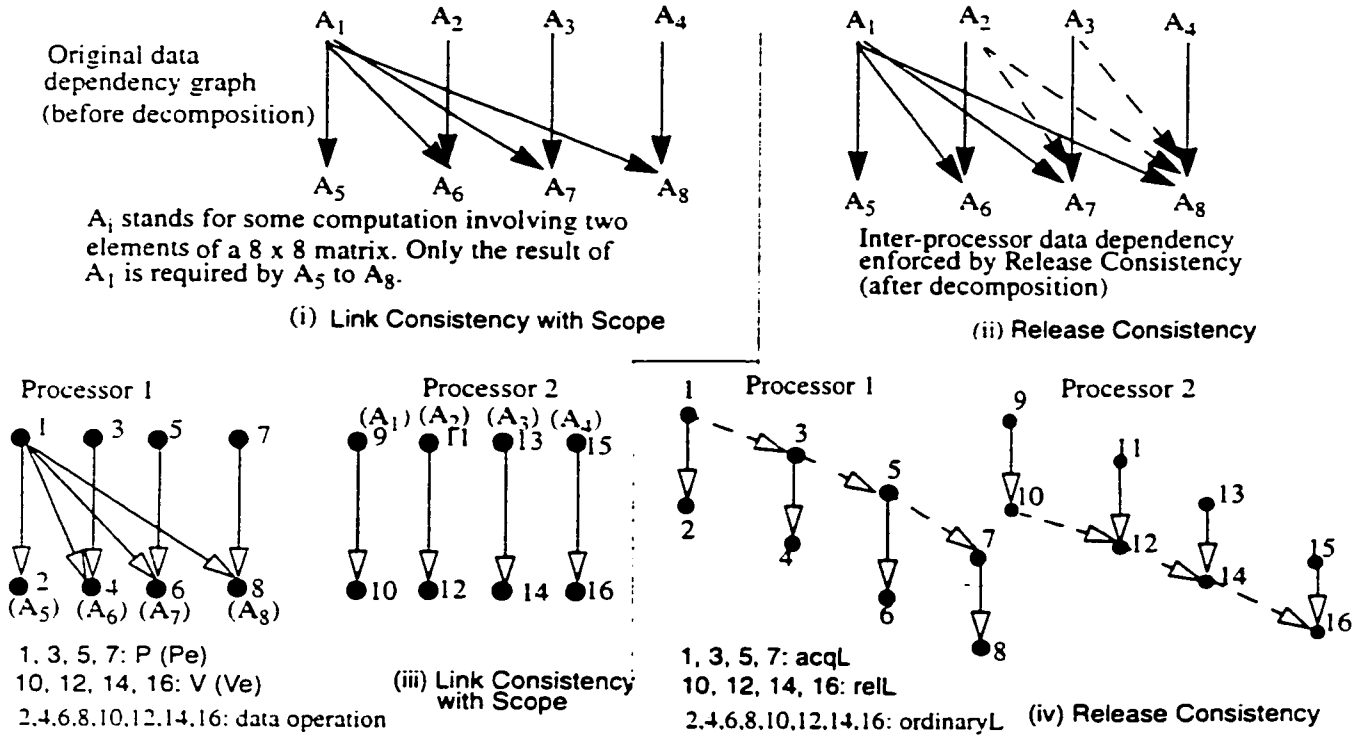


Figure 33 Data dependency graphs and ordering constraints graphs of L-U decomposition. (ii) and (iv) are for RC and (i) and (iii) are for LCs. Dashed arrows indicate unnecessary delays which are absent with LCs.

It is easy to verify that no matter how one labels operations that are executed on processor 1 in using RC, one cannot get the partial orders in Figure 33 (iii). Figure 33 (iii) does not enforce an ordering between operation ① and operation ③. For RC to do the same, at least one of them must be labeled as ordinaryL (if both labeled as syncL, then an ordering will be enforced between them by RMrc). In either case, operation ⑧ must be labeled as relL (to have ① $\xrightarrow{P}$ ⑧ or ③ $\xrightarrow{P}$ ⑧), which will imply ② $\xrightarrow{P}$ ⑧ since a relL operation on RMrc will wait until all previous operations have been completed. ③ $\xrightarrow{P}$ ⑧ is an ordering constraint absent in Figure 33 (iii). Therefore, the labeling of RC cannot achieve the partial order in Figure 33 (iii).

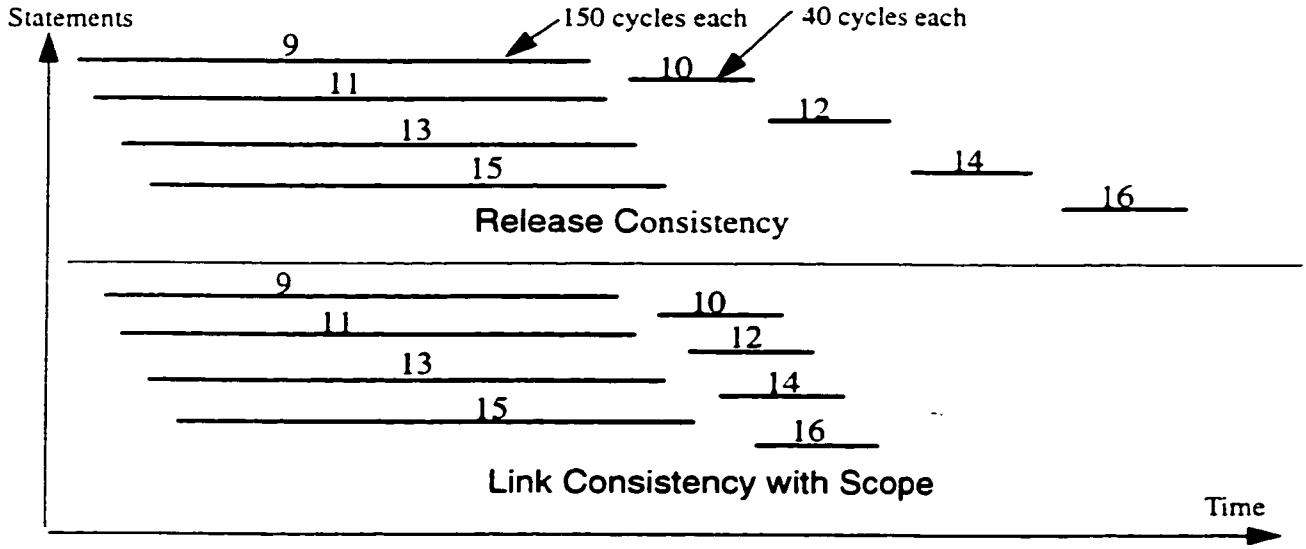


Figure 34 Comparison of cycles of two possible executions on RC and on LCs

Notice that if LC instead LCs is used, then the partial order for Processor 1 in Figure 33 (iii) will be the one in Figure 35.

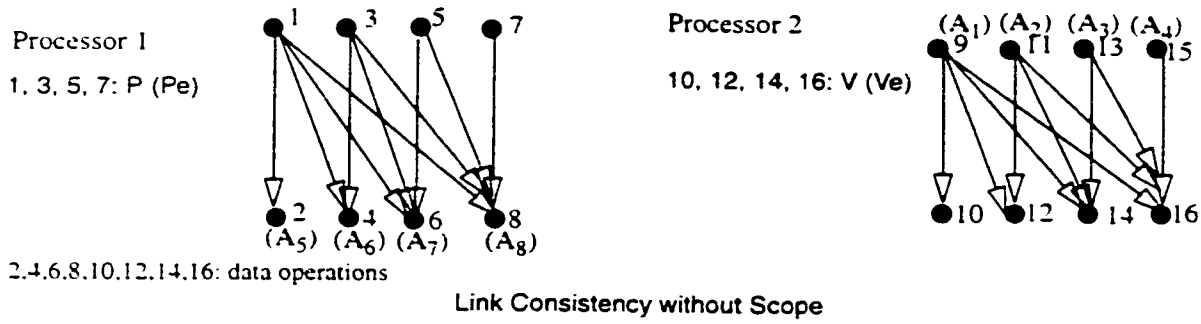


Figure 35 Ordering constraints graphs of L-U decomposition as enforced by LC.

We show that no matter how one labels operations in Processor 1 or Processor 2 using RC, one cannot get the same partial order using RC as that in Figure 35. We show the case for Processor 1 first.

If operation 5 is labeled as ordinaryL then both operation 6 and operation 8 must be labeled as relL to have (5) $\xrightarrow{p}$ (6) and (5) $\xrightarrow{p}$ (8). But this is impossible since it will imply that (6) $\xrightarrow{p}$ (8), which is absent in Figure 35. Otherwise, if operation 5 is labeled as relL, then both operation 6 and operation 8 must be labeled as acqL or relL to have (5) $\xrightarrow{p}$ (6) and (5) $\xrightarrow{p}$ (8), which will imply (6) $\xrightarrow{p}$ (8) again and thus impossible. Finally if operation 5 is labeled as acqL then no matter how operation 7 is labeled, RC enforces (5) $\xrightarrow{p}$ (7), which is absent in Figure 35. Therefore, RC cannot achieve the partial order for Processor 1 in Figure 35. In a very similar way, one can show the case for Processor 2.

4.5.5 About PLpc Memory Consistency Model

PLpc consistency model (Gharachorloo et al. 1992) is an attempt to further improve on RC and DRF1. It proposes to relax the program order between a synchronization write and a synchronization read operation if at least one of them is so-called loop operation.

A Loop read competes with at most one write (called a loop write) in every sequentially consistent execution, and a Loop write competes only with loop reads in every sequentially consistent execution. A loop write does not competes with another write that is in conflict with it.

For example, consider the program in Figure 36 from (Gharachorloo et al. 1992). The write operation 'Flag=1' is a loop write and the last read operation in 'while(Flag==0)', the one which returns 1, is a loop read .

```
Thread 1                    Thread 2

A = 1;
B = 1;
Flag = 1;

                            while (Flag == 0) does nothing;
                            ... = B;
                            ... = A;
```

Figure 36 An Example to illustrate loop operations of PLpc

Consider parallel programs that are synchronized using semaphores. A Loop read bears some analogy with an Exclusive Consumer (Pe) in LC: the former competes with at most one write while the latter will only be enabled by one V. A Loop write bears some analogy with an Exclusive Producer (Ve) in LC: (i) it enables a loop read to succeed while a Ve

enables a Pe, and (ii) it does not competes with another write while a Ve is an operation on a token collision free semaphore, that is, without a P on the same semaphore being executed in between, two Ve on a common semaphore will not be executed consecutively in any sequentially consistent execution.

Both PLpc and LC try to relax program orders between synchronization operations. Figure 3 (section 2.1.2.3) summarizes what PLpc relaxes. Figure 24 summarizes what LC relaxes, in which we also indicate those relaxations that can be deduced from Figure 3 under the hypothetical implementations of P and V, as we will show next.

LC differs from PLpc first in the level of granularity of synchronization operations. We consider a P and a V as the primtive synchronization operations, while PLpc considers the single-access based read and write used in synchronization. Therefore LC and PLpc relax program order between different types of operations. For the comparison purpose, we assume (surely too simplistic) that the implementation of a P consists of several competing reads including one loop read and one non-competing write, and the implementation of a V consists of a loop write only as given in Figure 37. In our discussion above, we assume that all operations for an implementation of P or a V are executed atomically.

```
P(a)                                        V(a)
{ read until a=1;(loop read[3])             a:=1.(loop write)
a:=0. }
```

Figure 37 Hypothetical implementations of P and V

3. More accurately, only the last *successful* read is a loop read. Other reads are ignored.

Assuming the above implementation, the following approximate correspondence can be drawn:

PLpc		LC
PL1 plus PL2	==	R4.1
PL3	==	R3.2

(There is no correspondence from PLpc for R3.1 and R4.2 from LC.)

PL1 and PL2 together says that a Loop write need not delay other synchronization reads that follow, which is analogous to R4.1 saying that a Ve need not delay other P operations that follow. PL3 which says that a synchronization write need not delay other Loop reads that follow is analogous to R3.2 saying that a V need not delay other Pe operations that follow.

What LC offers in addition is the relaxation of program orders between two P and between two V operations. Also our relaxation rules for program order between V and P (R4.1 and R3.2) do not rely on the implementation of P and V. On the other hand, for PLpc one has to label each individual operation in an implementation of P and V to use its relaxation rules: a different implementation of P and V may have their operations labeled differently so that its relaxation rules may not be applicable any more.

Moreover, PLpc or RC have focused on relaxation of program order that will not cause data race execution for a data race free program. LC also considers relaxation that will not cause synchronization failure for programs free of it on BM. Some relaxation of program order may cause a synchronization failure for a program that is otherwise synchronization failure free, such as Example MC2 in Appendix C.

Finally, as a minor note, there is a slight difference between the relationship of Loop write and Loop read and the relationship of Ve and Pe. As we see from its definition (see section 2.1.2.3), a Loop write can be read by (can compete with) more than one Loop read since a read does not change the value of a variable, but a Ve can only enable (can only compete with) at most one Pe.

Chapter 5 Trace Set and Derived Behaviors

In this chapter, we define the trace set of synchronization operations of an execution and the behaviors that are derived from a trace set. We assume that a parallel program consists of program threads which synchronize using binary semaphores only. We further assume that it has at least one successful execution, and for a successful execution, every synchronization operation that is executed in each thread is recorded to form a *thread trace*. The set of thread traces in an execution is called a *trace set*. Finally we assume that there is no access to control variables in every thread trace: when an operation is completed, the next operation that may be started is the one that follows immediately. Next we give some examples to illustrate our problem.

5.1 Some Illustrating Examples

The following symbols are used in our discussion:

$ev_{a1}, ev_{a2}, \ldots$: different occurrences of ev_a. We use ev_a or even *ev* if there is no confusion.

For example, *Va1*, *Va2*, ... (*Pa1*, *Pa2*, ...) represent distinct V (P) events on semaphore *a*.

ti: *thread trace i* which is a sequence of synchronization events on semaphores performed by the i^{th} program thread. ti specifies a linear order of events and we assume that the meaning of ev is (occurs) before ev' in ti or ev is (occurs) after ev' in ti is self-explained. Let ${}^{0}ti$ (ti^{0}) denote the first (last) event of ti.

T: a *trace set* of an execution consisting of a set of recorded thread traces {t1, t2, ...}

and I, a set of initial values of semaphores. The initial value of a semaphore (1 or 0) is represented by the presence or absence of a V event on the semaphore in I. In the subsequent discussion, I is considered as a set of events: each in a separate pseudo-thread that terminates before any other thread starts. I can be an empty set {}.

Example 5.1: T=<I, {t1, t2}> = <{Va, Vb},{Pa1 Pb1 Va1 Pc1 Vb3 Vc1, Pb2 Vb2 Vc2 Pc2 Pa2}>.

In Example 5. 1, two thread traces are recorded from some successful execution involving three semaphores. One thread has recorded Pa1Pb1Va1Vb3Pc1Vc1 and the other has recorded Pb2Vb2Vc2Pc2Pa2. The initial state of semaphores I={Va, Vb} means that semaphores a and b are in the true state. The two threads execute the two linearly ordered traces t1 and t2 respectively.

In modeling the execution of a trace set, a partially ordered behavior, called a *derived behavior*, is used to capture the order imposed by the linear thread traces and semantics of semaphores. Before presenting formal definitions, we first highlight the idea with some examples.

Example 5.2: A behavior of T of Example 5.1:

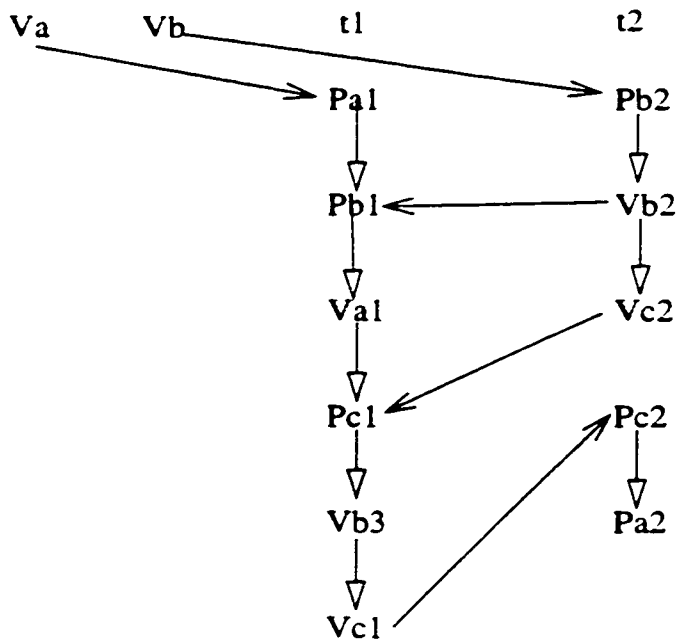


Figure 38 Behavior B: a partial order based on T in Example 5.1

In this example, B, a behavior of T, is presented as a partial order. Some observation deserves being mentioned here. An arrow $\longrightarrow$ is only from an event to its immediate successor in a same ti. An arrow crossing two threads ($\longrightarrow$) always connects a V event to a compatible P event or vice versa, such as Vb2 to Pb1. Moreover, the partial order is consistent with each linear trace: an event precedes another in ti iff the same holds in B. In the above, the precedence from Vc2 to Pc2 in t2 is established by the indirect sequence Vc2Pc1Vb3Vc1Pc2 in B.

Example 5.3: Another behavior B' of T of Example 5.1:

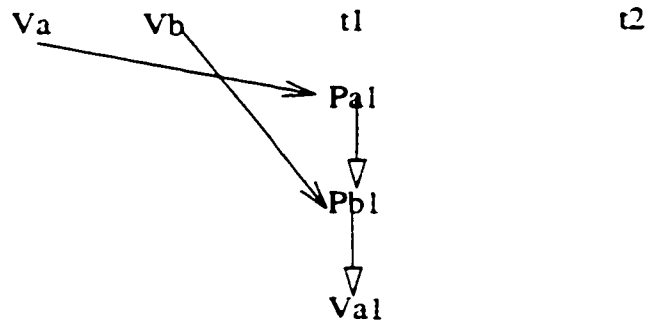


Figure 39 B': another partial order based on T in Example 5.1

In this example, B' also satisfies the observations we made about B in Example 5.2. One additional observation deserves being mentioned about B': it contains only three events from t1 and none from t2. B' cannot be extended without violating semaphore semantics: both the event that immediately follows Va1 and the first event in t2 are P (on semaphore b or c) but only semaphore a is currently set. We call such a partial order *failure* behavior.

Example 5.4: T = <I, {t1, t2}> = <{Va1, Vb1}, {Pa1Pb1Va2Vb2, Pb2Pa2Va3Pb3}>.

In some of the derived behaviors of T, Va1 could enable Pa1 and Vb1 could enable Pb2. As a result, both Pb1 and Pa2 will wait infinitely. This is the classical *deadlock* situation.

Example 5.5: T = <I, {t1, t2}> = <{Va1}, {Pa1, Va2Pa2}>

In some of the derived behaviors of T, both Va1 and Va2 could be executed (and collided) before either of Pa is executed and one of the threads could wait indefinitely afterwards.

5.2 Symbols and Definitions

Next, we give some more symbols and definitions. Since each ti is also a sequence of events, some symbols and definitions we use in Chapter 3 for an execution E is still valid for ti. Notice that the context is, however, changed: in the preceding chapters E represents a total order of events from all program threads, each of which is a partial order, while ti represents *one* linear thread trace from some execution. From this chapter onwards ti is a linearly ordered trace.

5.2.1 Symbols and Definitions about Thread Trace and Trace Set

The following symbols are used:

S={a,b,c, ...}: a set of distinct semaphores of a trace set T.

V (P*)*: zero or more V (P) events on a same semaphore in a sequence.

$V^+(P^+)$: one or more V (P) events on a same semaphore in a sequence.

prefix(ti): any subsequence of events (possibly empty) from ti starting from the first event of ti. *prefix(ev, ti)*: a prefix(ti) ending with event ev.

suffix(ti): any subsequence of events (possibly empty) from ti ending with the last event of ti. *suffix(ev, ti)*: a suffix(ti) starting with event ev. Obviously [prefix(ev, ti)]^0 = ^{0}[suffix(ev, ti)] = ev if ev is in ti.

^0po: set of events in po (a partial order of events) that are not preceded by any other event in po.

po^0: set of events in po (a partial order of events) that do not precede any other event in

po.

Trace set prefix, prefix(T): <I, {prefix(ti) | ti in T}> = <I, {pt1, pt2, ... , pt_n}>.

Trace set suffix, suffix(T): {suffix(ti) | ti in T} = {st1, st2, ... , st_n}.

Suffix frontier, ^{0}*suffix(T)*: {^{0}st1, ^{0}st2, ... , ^0st_n}.

A, A_1, A_2, ... : arbitrary subsequences (including Λ, the empty one) of thread traces.

For example, if T = <I, {t1, t2}> = <{Vc1}, {Va1Pb2Pc1Pa1Vb2, Vb1Pb1}> then suffix(Pb2, t1) = Pb2Pc1Pa1Vb2, suffix(Pb1,t2) = Pb1, prefix(Pb2, t1) = Pa1Pb2, and prefix(Vb1, t2) = Vb1. A possible prefix(T) is <I, {prefix(Pb2, t1), prefix(Vb1, t2)}>, a possible suffix(T) is {suffix(Pb2, t1), suffix(Pb1, t2)} = {Pb2Pc1Pa1Vb2, Pb1} and ^{0}suffix(T)={Pb2, Pb1}. A=Pc1Pa1 is a subsequence of t1.

Definition 5.1 Let po1=<E, R> and po2=<E', R'> are two partial orders. po1 is a *subset (proper subset)* of po2 iff E is a subset of E' and R* is a subset (proper subset) of (R')* where * denotes transitive closure.

5.2.2 Definitions of Derived Behaviors from Trace Set

Definition 5.2 Let pT = prefix(T) = <I, {pt1, pt2, ...}> and e_{ij} denote the j^{th} event in ti. A *(derived) behavior* B of T is a partial order involving pT, written as <Ep, O> where Ep is a set of events in pT and O is transitively reduced such that:

Condition (i) For every ti, j<k ⇒ (e_{ij}, e_{ik})∈ O* (the transitive closure of O).

Condition (ii) For every Pa, there is at least one Va satisfying (Va,Pa)∈ O* and (Va,Pa')∈ O* ⇒(Pa,Pa')∈ O*.

Condition (iii) For each pair of Pa and Va. either (Va,Pa) or (Pa,Va) $\in O^*$.

Condition (iv) $(e_{ik}, e_{jr}) \in O$ and $i \neq j \Rightarrow e_{ik}$=Va/Pa and e_{jr}=Pa'/Va' for some a.

It is worthwhile to discuss briefly the intuition behind the above definition. It is immediate that condition (i) ensures that B is consistent with each pti. Therefore, if an event precedes another in the program order of a thread trace, so will it in B. Condition (ii) ensures that every Pa is enabled by at least one Va exclusively, and no other compatible Pa event should come between them in B. Condition (iii) ensures that any pair of Va and Pa events are always serialized. Notice that we allow concurrent V's in B. Condition (iv) eliminates unnecessary ordering among events. These properties can be stated and proved formally.

Theorem 5.1 The projection of B onto events involving semaphore a is a partial order of the form [{V};P]*;[{V}]* = {V};P;{V};P; ... , where each {V} is a nonempty set of partially ordered V events and an event appearing before some ';' precedes all events that appear after that ';'.

Proof: Without loss of generality, let us identify the i^{th} subset of V events in the sequence as **Si**, and the P events in the sequence as **P1**, **P2**, etc.We wish to show the projection of B on a semaphore must be of the form: **S1**;**P1**;**S2**;**P2**; ... .

The assertion holds because:

(a) Each V must be in some **Sj**, otherwise condition (iii) will be violated.

(b) Each **Sj** must not be empty, otherwise condition (ii) will be violated.QED.

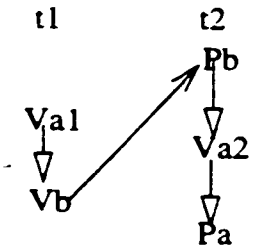
Notice that any trace set T has at least one behavior, the behavior B_0, which consists of events in I only. If I is empty, then B_0 is empty too. In the future, we refer to the sequence

{V};P;{V};P; ... for a semaphore as the *conflict sequence* of the semaphore. We call the ordering introduced by condition (iii) VP synchronization. From Theorem 5.1, each set of conflict sequences correspond to a unique set of VP synchronizations. To make the model more concise and elegant, we purposely do not order two V events in a subset **Si** unless such an ordering is transitively induced by some other VP synchronization (of other semaphores) and program order. We call a set of such conflict sequences for different threads a *canonical* set. To illustrate this, consider the following example.

Example 5.6:

The partial order on the LHS has a canonical conflict sequence given by {Va1,Va2};Pa while that on the RHS has a non-canonical conflict sequence {Va1;Va2};Pa.

Example 5.7:



The above has a canonical conflict sequence despite the fact {Va1;Va2};Pa is the conflict sequence for semaphore a. This is because the ordering between Va1 and Va2 is induced by Vb $\longrightarrow$ Pb and t2.

Theorem 5.2 Among all partial orders of T which satisfy conditions (i)-(iii) and which give rise to a same set of canonical conflict sequences, the partial order that satisfies condition (iv) is a proper subset of any other.

Proof: Let Os be the set of partial orders, each representing the conflict sequence of a distinct semaphore. Construct a partial order of T, say B', using $O'^* = (t1 \cup t2 \cup \ldots \cup t_n \cup Os)^*$. Obviously O'* is a proper subset of O* and is therefore unique.

What remains to be shown is that B' and only B' satisfies (iv).

B' satisfies (iv): trivially none of the inter-thread orderings of the form (V,V), (Va,Pb) and (Pa, Vb) is needed to preserve ti and the canonocal conflict sequences. In particular, inclusion of (V,V) will violate canonical conflict sequence, while inclusion of (Va,Pb) and (Pa,b) will make O' not minimal.

Only B' satisfies (iv): this follows from the uniquenes of B'. QED.

The use of canonical conflict sequence set reduces the number of partial orders to be analyzed. Specifically, we do not care to distinguish between behaviors which differ only in the ordering of some V events but otherwise have identical VP synchronizations. In Example 5.6, the behavior with unordered Va1 and Va2 covers also the cases where they are serialized in either order: the collision of Va1 with Va2 leads to the same consequence. In Example 5.7, Va1 is ordered before Va2 due to the synchronization of threads. Thus our choice of behavior model is elegant and minimal, giving us an efficient tool for conceptulization and analysis.

B in the left of Figure 40 is a behavior of T= <{ }, {Va1Vb1Pb1Va2Pa1Va3, Va4Pa2}>. Two kinds of arrows ($\xrightarrow{p.1}$ and $\xrightarrow{c.1}$) are used: the former between two events from a same trace and the latter between two events on a same semaphore. We will define these two arrows exactly later on. We will also explain the partial order in the right later.

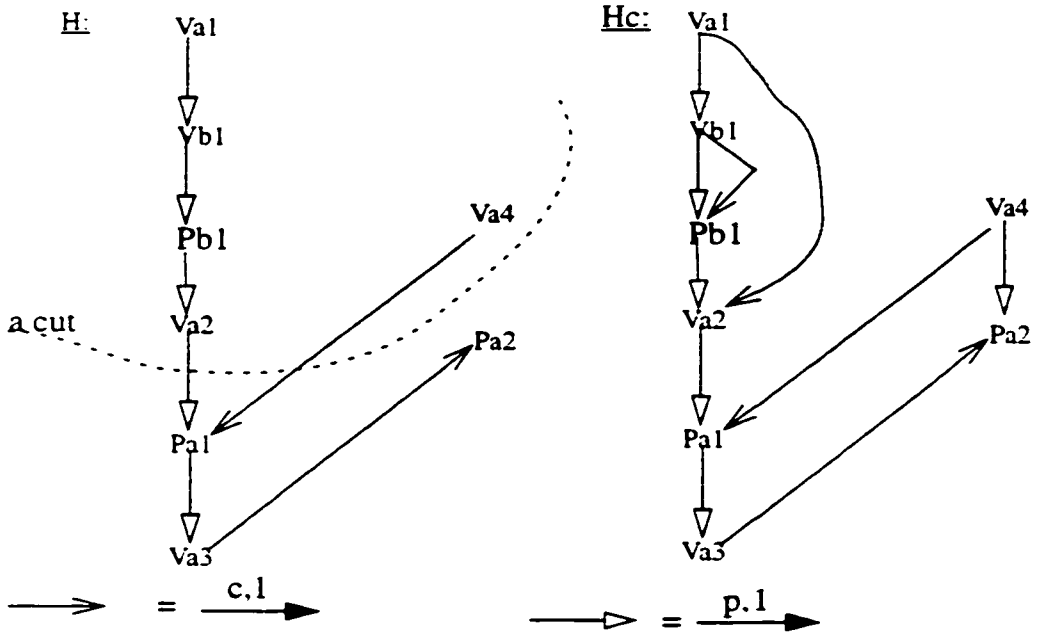


Figure 40 Examples of behavior and augmented behavior of T

Given a behavior B and the conflict sequence of some semaphore: **S1**;**P1**;**S2**;**P2**; ...; **Sk**, we introduce the following terms:

Definition 5.3 *Immediately Precedes*: Given a behavior B, **Pi** immediately precedes every event in ${}^{0}\mathbf{S}_{i+1}$, and every event in $\mathbf{Si}^{0}$ immediately precedes **Pi**. Moreover, two V events in **Si** could also be ordered. We also say V1 immedaitely precedes V2 in **Si** iff V1 precedes V2 and there is no V3 in **Si** that precedes V2

and is preceded by V1. We use $\xrightarrow{c.1}$ to represent such immediately preceding relation.

For example, in Figure 40, Va1, Va2 and Va4 are in a collision set. Va1 $\xrightarrow{c.1}$ Va2, Va2 $\xrightarrow{c.1}$ Pa1 and Va4 $\xrightarrow{c.1}$ Pa1.

Definition 5.4 *Token Collision*: The V events in each **Si** are said to have collided (they are followed by a same **Pi**). We call **Si** a *collision set.*

Definition 5.5 *Hidden V Event*: A V event in some **Si** is hidden in B iff it immediately precedes another V event in **Si**.

Definition 5.6 *Availability*: The semaphore is available in B iff **Sk** is non-empty, i.e, the conflict sequence ends with non-empty **Sk** rather than some **Pk**. A V event is available in B iff it is in **Sk** and not hidden in B.

For example, Va2 in the partial behavior above the 'cut' is available but not Va1 since it is hidden by Va2.

By our definition, B=<Ev, O> is transitively reduced. For convenience, we also use Bc = <Ev, Oc> , an augmented behavior of B as defined next:

Definition 5.7 An augmented behavior Bc of B = <Ev,O> is given by <Ev, Oc> where Oc= $O \cup O'$ and $O' = \{(ev, ev') \mid ev \text{ immediately precedes } ev' \text{ in } B\}$.

In the right of Figure 40 is an example of Bc. Notice that in drawing B (Bc), we use two different arrows to represent two different relationships between two events in O (Oc): $\xrightarrow{p.1}$ to represent linear order of events in each trace and $\xrightarrow{c.1}$ to represent the

"immediately precedes" relation. all events from a same thread trace are totally ordered by $\xrightarrow{p.l}$ and all events on a same semaphore are totally ordered by $\xrightarrow{c.l}$ in Bc. We also say that $\xrightarrow{p.l}$ captures the program order of ti and $\xrightarrow{c.l}$ captures the conflict order of a behavior. In addition, we use $\xrightarrow{l}$ to denote $\xrightarrow{p.l}$, $\xrightarrow{c.l}$ or both, and $\xrightarrow{p}$ for the transitive closure of $\xrightarrow{p.l}$.

Notice that it is possible to have both ev1 $\xrightarrow{p.l}$ ev2 and ev1 $\xrightarrow{c.l}$ ev2 in a Bc. For example, there are both Vb1 $\xrightarrow{p.l}$ Pb1 and Vb1 $\xrightarrow{c.l}$ Pb1 in Bc in Figure 40. To distinguish, we say Q1 $\xrightarrow{a.l}$ Q2 in a behavior iff Q1 $\xrightarrow{c.l}$ Q2 but not Q1 $\xrightarrow{p.l}$ Q2. For example, Va4 $\xrightarrow{a.l}$ Pa1 but not Va4 $\xrightarrow{a.l}$ Pa2 in Figure 40.

Definition 5.8. Given a trace ti, event Qa1 is the *successor* (P or V) event of Qa2 in ti iff (i) Qa2 occurs before Qa1 in ti and (ii) there is no Qa3 (Qa3≠Qa1 and Qa3≠Qa2) such that Qa1 occurs before Qa3 and Qa3 occurs before Qa1. Qa2 is said to be the *predecessor* (P or V) event of such Qa1.

For example, in Example 5.1, Vb2 is the successor V event of Pb2 and Vc2 is the predecessor V event of Pc2.

Definition 5.9 The *state of a behavior* B, *state(B)*, is the set of available semaphores. If there is no semaphore available in B, then state(B) is an empty set, { }.

Definition 5.10 A behavior B=<Ev,O> of T is *complete* if Ev=events(T). A behavior B of T is *maximal* if there is no other behavior B' of T, such that B is a proper prefix of B', otherwise, it is *partial*. A *failure behavior* is a maximal but not complete behavior.

Definition 5.11 A *behavior suffix* of B=<Ev,O> of T is a suffix(T) consisting of what remains in T after every event in Ev is removed. If B is a failure, then its suffix frontier ^{0}Bs contains P events only and is called a *failure suffix*.

For example, in Figure 40, the behavior above the cut (the dotted curve) is a proper prefix of B and for this prefix behavior, Bs= {Pa1Va3, Pa2} and ^{0}Bs={Pa1, Pa2}.

The above behavior-related discussion is for a trace set T involving multiple semaphores. It also applies to a trace set involving one semaphore only. A reduced trace set involving one semaphore only is a local trace set as defined next.

Definition 5.12 Given T=<I, {t1, t2, ...}>, the local trace set on semaphore a is *Ta = <Ia, {ta1, ta2, ...}>* where Ia (tai) is the projection of I (ti) onto events involving semaphore a only. tai is called *local (thread) trace*.

For example, for T in Example 5.4 above, Ta= <{Va1},{Pa1Va2, Pa2Va3}>.

We call derived behaviors of Ta *local behaviors* and call those local behaviors that are failures *local failures*. In contrast, we also call derived behaviors of a general T *global behavior* or *global failures* if they are failures. A local behavior of Ta is nothing but a canonical conflict sequence of semaphore a. According to condition (iv), each subset of {Va} in a canonical conflict sequence for Ta is ordered only if they appear in a same thread. We say that a global behavior B *contains* a local behavior B_l of semaphore a iff Ba = B_l, where Ba is the projection of B onto events on semaphore a only. Obviously, such Ba

is a behavior of Ta. We say that a local behavior B_l is *feasible* iff there is a global behavior B that contains it, otherwise, B_l is *infeasible*.

5.2.3 Behavior Generation

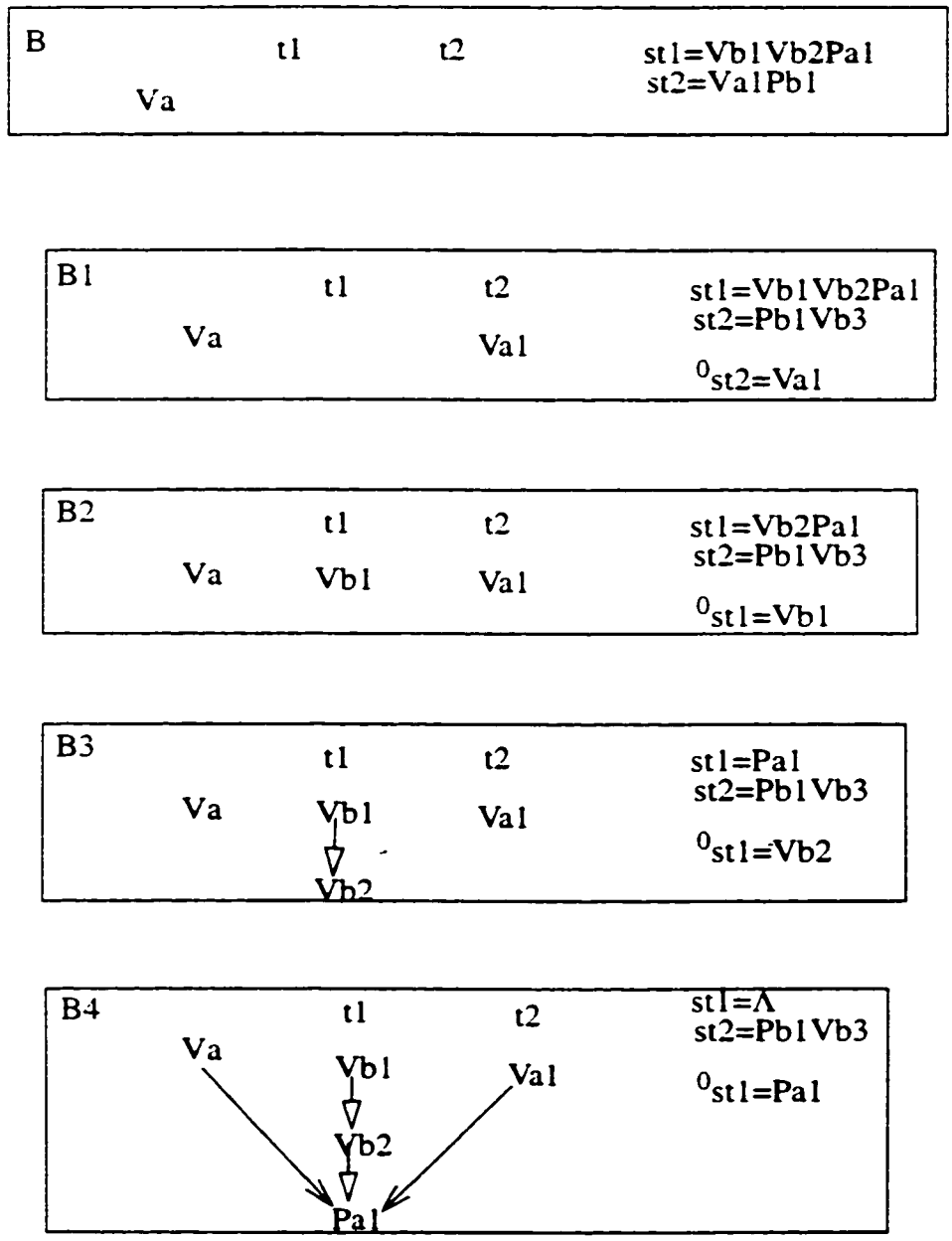
The behaviors of a trace set can be generated by a simple extension algorithm introduced in this section. Intuitively, consider a (partial) behavior B of a prefix pT of T. We use {st1,st2, ...} to represent the suffix of T formed by removing pT={pt1, pt2, ...} from T sti=ti-pti. For simplicity, we use '-' here to denote prefix removal. The extension of B to B' can now be presented.

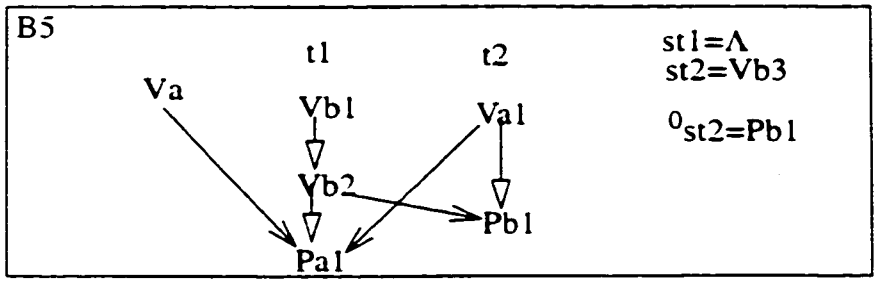
Input: A partial order B=<Ev, O> and the associated suffix {st1, st2, ... stn}. If the conflict sequence of semaphore b in B ends with a Pb event, the latter will be identified as Pb^0, and similarly if it ends with {Vb}, the latter will be identified as $\{Vb\}^0$.

Output: B', a behavior extended from B by one more event from the suffix of B.

Procedure BBE: basic_behavior_extension(B, {st1, st2, ... , stn});
B' := <Ev', O'> := B;
if possible, choose a ^{0}sti on some semaphore b such that ^{0}sti=Pb $\Rightarrow$ Vb is available in the current B';
case (1) ^{0}sti := Vb and Pb^0 exists:
O' := O$\cup${(pti^0, ^{0}sti), (Pb^0,^{0}sti)};
Ev' := Ev$\cup${^{0}sti};
case (2) ^{0}sti=Vb and no Pb^0 exists (there may exist $\{Vb\}^0$):
if no $\{Vb\}^0$ exists then Ev' := Ev$\cup${^{0}sti};
else O' := O$\cup${(pti^0, ^{0}sti)}; Ev' := Ev$\cup${^{0}sti};
case (3) ^{0}sti=Pb:
O' := O$\cup${(pti^0, ^{0}sti)}$\cup${(Vb, ^{0}sti) | Vb$\in \{Vb\}^0$};
Ev' := Ev$\cup${^{0}sti};
if B' is a complete behavior, then return B' and indicate its completeness;
else return "Continute with B' ";
else return "B' is a failure behavior";

In the above, it is assumed that O' is transitively reduced after each augmentation. An illustration of the procedure is shown below. Let T = <I, {t1,t2}> = <{Va}, Vb1Vb2Pa1, Va1Vb1> and B contains events in I only as shown below. On the left are the new st1 and st2 after each augmentation, and the event selected for the augmentation.





The extension from B to B1, the one from B1 to B2, and the one from B2 to B3 all use case (2). Both the extension from B3 to B4 and the one from B4 to B5 use case (3). Obviously, to further extend B5, case (1) has to be used. Notice that extension of a partial behavior may not be unique. For example, B could be extended to a different behavior than B1.

Theorem 5.3 B' that is returned by the BBE is a valid behavior of T.

Proof: B' is a partial order. This follows immediately from the fact that the augmentation of O to O' includes only precedences leading to ^{0}sti and none leading to ^{0}sti that originally exists in O*. Conditions (i)-(iii) follow trivially from the construction. Moreover, the procedure only adds to O direct precedences of (pti^0, ^{0}sti) and (i) (Pb^0,^{0}sti) or (ii) {(Vb, ^{0}sti)|Vb∈{Vb}^{0}}. The former case involves events in the thread trace ti only, while the latter case concerns VP synchronization of a same semaphore. Thus B' also satisfies Condition (iv). QED.

5.3 Problem Definition and Its Complexity

Given a trace set T from a successful execution, there can be more than one derived behaviors of synchronization operations, some of which may be synchronization failures due to token collision and synchronization race. Our problem can be stated as: *Given a trace set T, decide if there is any synchronization failure among its derived behaviors.*

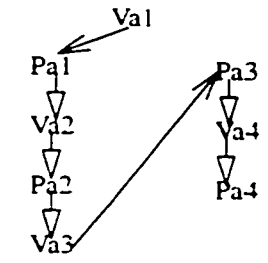
To have a better understanding of the problem, we examine various types of synchronization races and their contribution to the complexity of our problem.

5.3.1 Two Key Notions: Synchronization Race and Token Collision

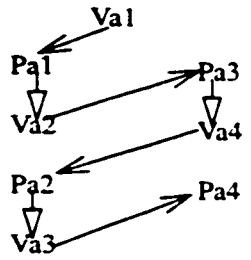
1. Synchronization Race

A synchronization operation ev1 and another synchronization operation ev2 race if there is a partial behavior which can be extended by the execution of either ev1 or ev2.

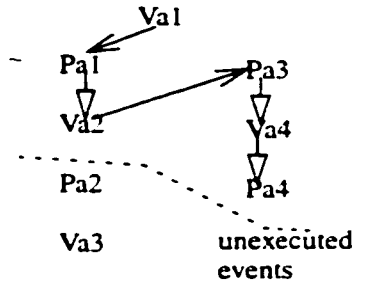
We use T= <{Va1}, {Pa1Va2Pa2Va3, Pa3Va4Pa4}> to illustrate the notion of synchronization race and our problem. This trace set has the following six derived (maximal) behaviors, not all of which are complete:



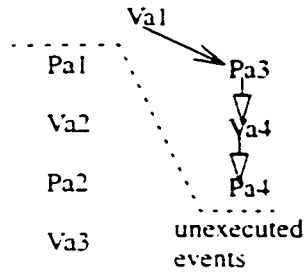
Behavior 1, Failure



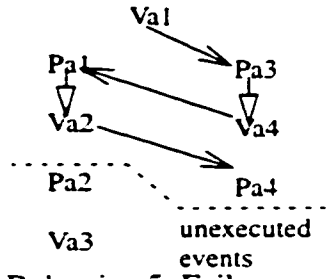
Behavior 2, Failure



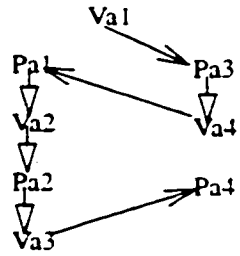
Behavior 3, Failure



Behavior 4. Failure



Behavior 5. Failure



Behavior 6. Failure

In the above, Va1 can enable either Pa1 or Pa3, leading to totally different behaviors. This is one form of synchronization race: race between two P. There can also be race between two V, between a V and a P, or even between multiple P's and V's.

Definition 5.13 A trace set T contains *synchronization race* iff it has more than one maximal behavior.

2. Token Collision

As we have already discussed in chapter 3, one important aspect of a binary semaphore is that two V operations can be executed concurrently and the net result of that is as if only *one* V were executed and thus only *one* compatible P is enabled subsequently. We visualize a V operation to carry a token which can enable a compatible P. A semaphore in a trace set T is token-collision free (TCF) iff in every behavior of T, its conflict sequence contains only a singleton in each subset {V}, i.e., the conflict sequence must be of the form: Va;Pa;Va; ... .

Notice that a trace set containing no synchronization race does not imply that the trace set is TCF. A simple example is T=<{}, {VaVaPa}>.

5.3.2 Complexity of the Problem

Because the finiteness of the size of T, we can always solve the problem by enumerating every possible behavior that can be derived from T as in Example 5.1. Unfortunately, as the size of T grows, the time to perform such enumeration grows exponentially with the number of races unless NP=P. We show that the problem is NP complete in Theorem 5.4.

Theorem 5.4 The synchronization failure detection problem is NP complete.

We prove that the problem is NP complete by constructing a reduction from 3SAT problem (Garey and Johnson 1979). Let C = {C1, C2, ... , Cn} be a set of clauses. Ci = {c1, c2, c3}, where c1 c2, or c3 is an instance of some variable or its negation, x or x', $x \in U$ which is a set of variables, $\{u_1, u_2, ..., u_m\}$.

1. Reduction from 3SAT to the synchronization failure detection problem

The following four construction rules specify how to construct a trace set from a set of clauses by the 3SAT problem:

(1) For each variable, u_i, define six semaphores named as: u_i^0, u_i^1, u_i^2,u_i^3,u_i^4,u_i^5.
(2) For three items in a clause Ci, define three semaphores named as: $z^{i,1}$, $z^{i,2}$, and $z^{i,3}$ respectively.
(3) For each variable, ui, define six threads $\{Tu_i\}$:

Thread (i): Vu_i^0

Thread (ii): $Pu_i^0Vu_i^1$...

Thread (iii): $Pu_i^0Vu_i^2$...

Thread (iv): $Pu_i^1Pu_i^3Vu_i^5$

Thread (v): $Pu_i^2Pu_i^4Vu_i^5$

Thread (vi): $Pu_i^5Vu_i^0$

In Threads (ii) , for any ui' that occurs in clause C_k as the jth item, add a $V_z^{k,j}$ at the end, and in Threads (iii) , for any ui that occurs in clause C_k as the jth item, add a

$V_z^{k.j}$ at the end.

(4) for each clause, C_j, define one thread:

$P_z^{j.1}P_z^{j.2}P_z^{j.3}Vu_1^{3}Vu_1^{4}Vu_2^{3}Vu_2^{4}...Vu_m^{3}Vu_m^{4}$

In general for m variable and n clauses, the construction requires (6m + 3n) semaphores and (6m + n) threads.

As an example, consider C={C1, C2} where C1 = (a + b + c') and C2 = (b+c+d'), we have the following construction:

From (1):
a: $a^0, a^1, a^2, a^3, a^4, a^5$,
b: $b^0, b^1, b^2, b^3, b^4, b^5$,
c: $c^0, c^1, c^2, c^3, c^4, c^5$,
d: $d^0, d^1, d^2, d^3, d^4, d^5$,

From (2):
For C1: a: $z^{1.1}$, b: $z^{1.2}$, c': $z^{1.3}$
For C2: b: $z^{2.1}$, c: $z^{2.2}$, d': $z^{2.3}$

From (3):

a:
T1: Va^0
T2: Pa^0Va^1
T3: $Pa^0Va^2V_z^{1.1}$
T4: $Pa^1Pa^3Va^5$
T5: $Pa^2Pa^4Va^5$
T6:Pa^5Va^0

b:
T7: Vb^0
T8: Pb^0Vb^1
T9: $Pb^0Vb^2V_z^{1.2}V_z^{2.1}$
T10: $Pb^1Pb^3Vb^5$
T11: $Pb^2Pb^4Vb^5$
T12: Pb^5Vb^0

c:
T13: Vc^0
T14: $Pc^0Vc^1V_z^{1.3}$
T15: $Pc^0Vc^2V_z^{2.2}$
T16: $Pc^1Pc^3Vc^5$
T17: $Pc^2Pc^4Vc^5$
T18: Pc^5Vc^0

d:
T19: Vd^0
T20: $Pd^0Vd^1V_z^{2.3}$
T21: Pd^0Vd^2
T22: $Pd^1Pd^3Vd^5$
T23: $Pd^2Pd^4Vd^5$
T24: Pd^5Vd^0

From (4):
C1=(a +b + c'): T25: $P_z^{1.1}P_z^{1.2}P_z^{1.3}V_a^{3}V_a^{4}V_b^{3}V_b^{4}V_c^{3}V_c^{4}V_d^{3}V_d^{4}$
C2=(b + c + d'): T26: $P_z^{2.1}P_z^{2.2}P_z^{2.3}V_a^{3}V_a^{4}V_b^{3}V_b^{4}V_c^{3}V_c^{4}V_d^{3}V_d^{4}$

So the trace set T= <{ }, {ti|1≤i≤26}>.

2. The proof using the above reduction

(1) The problem is in NP.

Since it takes only polynomial time to construct an interleaving of events in trace set T and to verify whether the interleaving corresponds a synchronization failure, the problem is in NP.

(2) The construction takes polynomial time.

This is easy to see from construction rules (1) to (4). What remains to be shown is the following:

(3) The trace set from above construction contains synchronization failure iff C is satisfiable.

'if part': C is satisfiable with some assignment of true and false for every variable, which implies that for every Ci at least one cj (1≤j≤3) is true. We show a schedule that leads to a synchronization failure for the trace set in such a case.

For every Vu_i^0 (1≤i≤m), let Vu_i^0 be paired with Pu_i^0 in Thread (ii) in $\{Tu_i\}$ if a is assigned a true value, otherwise, let Vu_i^0 be paired with Pu_i^0 in Thread (iii) in $\{Tu_i\}$. According to the construction rules, the schedule cannot have all three $P_z^{j.1}$, $P_z^{j.1}$, $P_z^{j.1}$, in the thread constructed using rule (4) for Cj (1≤j≤n), be paired since at least one compatible V_z($V_z^{j.1}$, $V_z^{j.}$ or $V_z^{j.3}$) cannot be executed. As a result, neither Vu_i^3 nor Vu_i^4 can be executed, and subsequently, no Vu_i^5 and hence no Vu_i^0 can be executed, for 1≤i≤m. From the

construction, however, we see that there is another Pu_i^0 event. This indicates a situation where there is no V for compatible P that is ready. Thus T fails for this schedule.

'only if' part: C is not satisfiable with any assignment of true and false for every variable, which implies that there is at least one Ci, none of whose three items, c1, c2 and c3, is assigned a true value. We show next that T cannot fail.

For every V_{ui}^0 ($1 \le i \le m$), let V_{ui}^0 be paired with either P_{ui}^0. If Vu_i^0 is paired with P_{ui}^0 in Thread (ii) in $\{Tu_i\}$, then assign ui to be true, otherwise false. This actually generates an assignment A for every variable. Since C is not satisfiable, there exists at least one clause each of whose c_k ($1 \le k \le 3$) is assigned a false value by A. According to our construction, this means that for at least one j ($1 \le j \le n$), all $V_z^{j.1}$, $V_z^{j.2}$ and $V_z^{j.3}$ will be executed and therefore, all three $P_z^{j.1}$, $P_z^{j.2}$ and $P_z^{j.3}$ in the thread constructed for Cj using rule (4) can be paired. Consequently, every P in the constructed trace set will be paired due to those V events following the $P_z^{j.1}$, $P_z^{j.2}$, $P_z^{j.3}$. T cannot fail. QED.

Chapter 6 Synchronization Failure Detection for Binary Trace Sets

In this chapter, we discuss some techniques to solve the synchronization failure detection problem effectively despite its NP completeness.

6.1 Some Definitions

A trace set may contain different patterns, for example, the repeated occurrences of a same sequence of events. We give some definitions to capture special patterns.

Definition 6.1 An *all-P sequence* is an non-empty sequence of P events on *distinct* semaphores. Similarly, an *all-V sequence* is an non-empty sequence of V events on distinct semaphores.

Example 6.1 Pa1Pb1Pc2 is an all-P sequence and Va1Vb3Vc5 is an all-V sequence.

Definition 6.2 An all-V sequence is a *matching* V sequence of an all-P sequence if both have the same length and for every P event in the latter there is a compatible V event in the former. An all-P sequence is a *matching* sequence of another iff they are identical up to the labeling of semaphores but not event numbering.

Example 6.2 Va1Vb3Vc5 or Vc5Va1Vb3 is a matching V sequence of Pa1Pb1Pc2 and Pa2Pb3Pc4 is a matching P event of Pa1Pb1Pc2.

6.2 Outline of Techniques to Harness Computational Complexity

Several approaches will be introduced to reduce the detection complexity as much as possible. They are briefly outlined below.

1. Special Cases

For some special trace sets, there is some efficient algorithm to verify whether they contain synchronization failure. For example, if a trace set contains no synchronization race, then we know it cannot fail. As another example, if a trace set contains only one semaphore or it can be partitioned into a disjoint set of smaller trace sets, each containing one semaphore only, then there is some polynomial time algorithm to solve the problem.

2. Reductions of the Original Trace Set

We may reduce the size of a trace set by removing some events while preserving its failure property; the reduced trace set contains synchronization failure iff the original one does.

To reduce the length of a thread trace, there are two types of reduction techniques, local reductions and global reductions. A local reduction examines individual thread trace and tries to delete some events from it, while a global reduction considers many thread traces together before eliminating some events from them.

3. Reduction Heuristics

Unlike reduction rules which preserve failure property, a reduction heuristic works optimistically: if a reduced trace set contains synchronization failure, then so does the original. However, the reverse is not necessarily true.

4. Local Failure Validation

A failure behavior of a trace set must be the result of synchronization races among many events on different semaphores. Such a synchronization failure is a global failure. When we project a trace set on to individual semaphore, we will usually get a smaller trace set, which may contain local failures. As we will describe later, there is an efficient algorithm to determine whether a trace set with one semaphore contains synchronization failure. Local failure validation is based on the optimistic hypothesis that the existence of a local failure may sometimes reflect the existence of a global one.

5. Partial Order Checking

We apply checking on the partially ordered behaviors directly. This is still more efficient than the use of interleaving models (Probst and Li 1990, 1991, 1993).

6. Pruning Rules

A global behavior of a trace set T consists of a set of local behaviors, one per thread trace. However, not all combinations of local behaviors can form a global behavior. Indeed, some local behavior may never be included in a global behavior. We call a local behavior *infeasible* if there is no global behavior that includes it. There is no need to check a combination of local behaviors if one of local behavior is infeasible.

Given a local behavior, we can check its infeasibility first. But in some cases, we can determine the infeasibility of a local behavior L1 by relating it to another infeasible local

behavior L2. We term such relationship *dominance* between local behaviors: L1 is dominated by L2.

6.3 Special Failure Rules

We present some theorems to quickly determine whether certain trace sets contain synchronization failure. The first result gives a sufficient condition for T=<I,{t1, t2, ...}> to have a failure behavior. In particular, suppose one of the traces contains two consecutive occurrences of Pa, then the trace set contains synchronization failure. We state this as Theorem 6.1.

Theorem 6.1 ti=...Pa1Pa2... $\Rightarrow$ T contains a synchronization failure behavior.

Proof: We choose to obtain a complete behavior B of T by extending from the initial state in such a way that Pa1 is chosen only if no other event can be chosen in the suffix. This is allowed in the behavior extension procedure outlined earlier without affecting the correctness of the behavior extension. If such a complete behavior does not exist, then we have arrived at a failure behavior and the theorem holds trivially. Thus in extending some prefix of B with Pa1 to become B', it must be that each trace suffix is either empty or started with some Pb (b≠a) and the state of the prefix does not contain an available Vb. But after the extension, the state of B' does not contain any needed V that is available to extend B' further. B' is partial as its suffix contains at least Pa2. So B' is a failure behavior. QED.

The theorem is very useful. A static scan of the trace set could detect consecutive occurrences of Pa for some semaphore a and the decision can be made in time linear to the size of the trace set.

Two consecutive P events to distinct semaphores in a trace may not cause synchronization failure. However, if these P events appear in another trace in the reverse order while other traces all start with P events, then we could deduce synchronization failure in T according to the following theorem.

Theorem 6.2 T = <I, {t1, t2, ...}> = <{Va,Vb}, {PaPb ..., PbPa ..., P ..., P ... , P ..., ...}> has a failure behavior.

Proof: A failure behavior can be constructed by extending B_0 with Pa from t1 and then with Pb from t2, which leads to a state in which no V is available. The resulting behavior suffixes all start with P and no further extension is possible. QED.

Example 6.3 T = <{Va, Vb}, {PaPbVaVbVc, PbPaVaVb, PcVcVaVb}> contains synchronization failure.

Theorem 6.3 Let #Va be the number of Va events and #Pa be the number of Pa events in a trace set T. Then #Va < #Pa ⇒ T has a failure behavior.

Proof: This result is immediate from the definition of semaphore conflict sequence: every Pa must be immediately preceded by a set of non-empty {Va}. QED.

The above theorem is also useful for dynamic checking during behavior extension. Suppose the extension has led to a state I and some suffix Bs, and the conditions of the theorem hold for the trace set <I, Bs>, then the extension could immediately terminate.

Theorem 6.4 ti = ... Pa1VbPbPa2 ... ⇒ T has a failure behavior.

Proof: We use a similar proof strategy as that used in Theorem 6.2. We extend the

empty prefix with a bias against Pa1 until any other trace suffix is either empty or starts with some P whose compatible V is not in the current state St. Then we continue with the extension with Pa1VbPb leading to a state St-{Va}. At this new state, no further extension is possible and we have arrived at a failure behavior. QED.

Notice that using a reasoning similar to the above one, one can prove that a trace set including PaA_1A_2Pa contains synchronization failure, where A_2 is an all-P sequence and A_1 is a matching V sequence of A_2 such that A_1 and A_2 do not involve semaphore a.

Example 6.4 ti=... PaVbVcPbPcPa ... $\Rightarrow$ T contains synchronization failure.

A thread trace is trivial if it contains V events only. We assume that each trace set contains non trivial thread traces only. In reality, if we are given a trace set with trivial thread traces, we could remove them from the trace set and include the corresponding V's to I.

Theorem 6.5 Suppose a trace set T involves a single semaphore. Then T does not contain a failure behavior iff

(i) I={V} and all ti must be of the form $V^*(PV^+)^+$;

(ii) I={}, all ti must be of the form $V^*(PV^+)^+$, and one of them starts with V; or

(iii) I={V} and there is only one trace in T, t1=V^*P; or

(iv) I={} and there is only one trace in T, t1=V^+P.

Proof: $\Rightarrow$

(iii) and (iv) are obvious. We show that both (i) and (ii) cannot fail.

(i) In any partial behavior extension, if a V is used, then the new state ={V}. In such a case, a further extension is possible.If a P is used, then again the behavior can always be extended with the next V in the same trace. This repeats until a complete behavior is obtained.

(ii) Since at least one trace starts with a V, in the worst case, the initial behavior can be extended with this V. The rest then follows as in (i).

⇐

To not satisfy (i) to (iv), there are only three possibilities for the traces of T:

(a) there is some ti= ... PP ... ,

(b) there is no PP in T but there is more than one trace and at least one (say, t1) ends with a P, or

(c) I={ } and every ti starts with a P.

(c) is obvious. (a) is immediately from Theorem 6.1. We show (b) can fail next. The synchronization failure can be constructed by starting to extend the behavior using all of the preceding V events plus those in I in every trace followed by only events in t1.

This will lead to a behavior whose state is empty and no further extension of any trace is possible. Since the traces are non trivial, the suffix of every other traces is not empty. Thus we have arrived at a failure behavior. QED.

The above theorem can be used for both static checking of a given trace set T or dynamic checking of the state and suffix associated with a partial behavior during behav-

ior extension. The checking of conditions can be performed in time linearly to the size of T. A simple algorithm can be designed which scans each trace once and maintains the satisfaction of the conditions while doing so.

Example 6.5 Suppose (a) T=<{}, {V, VPV, VVPP}, (b) T=<{}, {V, VPV, VVPV}, (c) T=<{}, {V, VPVPVP}, (d) T=<{}, {V, VPV, VVPV, VP}. The trace set in (a) contains synchronization failure since it contains PP. The trace sets in (b) and in (c) do not contain synchronization failure since every P has a V successor. The trace set in (d) contains synchronization failure since there are two thread trace with P and one of them ends with P.

Theorem 6.6 The detection of synchronization race can be performed in time polynomial to the size of the trace set T.

Proof: From Theorem 5.1 and Theorem 5.2, each behavior has a dinstinct canonical conflict sequence. Hence the assertion. QED.

6.4 Trace Reduction Rules and Heuristics

The following results attempt to reduce the size of a trace. The complexity of synchronization failure analysis is dependent on the size of a given trace set. Thus a reduction of the complexity of its synchronization failure analysis could be obtained if we could eliminate some subsequences of events of a trace. For simplicity, we introduce a new notation: we use $t1 \stackrel{f}{=} t2$ to denote 't1⇔t2 with respect to the failure property'. If the relation is not symmetric, but T (in the LHS) has a synchronization failure implies T' (in the RHS) has a synchronization failure, then we write it as $t1 \stackrel{f}{\Rightarrow} t2$. When t2 is a subsequence of t1, these are

called local reductions as each trace is considered independent of the others. However, their validity requires the semaphores to be TCF.

6.4.1 Reduction Rules

Theorem 6.7 Suppose semaphore a is TCF. Pa1VaPa2 $\stackrel{L}{=}$Pa1.

Proof: $\Rightarrow$: Consider the original trace set T that contains the LHS in some trace, say t1.

Case (i) A failure behavior B of T does not contain Pa1.

Trivially, B is also a failure behavior of the trace set T' obtained by substituing t1=Pa for t1=Pa1Va1Pa2.

Case (ii) A failure behavior B of T contains Pa1.

Semaphore a is TCF $\Rightarrow$ the conflict sequence of a must be of the form: ...Pa1;Va1; ... , else Va1 will conflict with the Va done after Pa1.

This further means that we could remove Pa1Va1 from B by projecting B onto all other events except Pa1 and Va1. The resulting partial order is trivially a failure behavior of T', the trace set obtained by substituting t2 for t1.

$\Leftarrow$:The reverse claim is straightforward. Here we consider replacing t2 in T' by t1. As before, suppose a failure behavior B' of T' does not include Pa, then trivially, it is also a failure behavior of T obtained by replacing t2 by t1. If B' contains Pa, we could augment B' by inserting Pa1Va1 before the extension using Pa (which is now rewritten as Pa2) and arrive at a failure behavior B of T. QED.

Theorem 6.8 Suppose semaphore a and b are TCF. VaVb $\stackrel{f}{=}$ VbVa.

Proof: We show that VaVb $\stackrel{f}{=>}$VbVa. The reverse holds by symmetry.

As before, semaphore a is TCF implies that the conflict sequence of semaphore a must be of the form Va:Pa:Va: ... .

Case (i) A failure behavior B of LSH does not contain VaVb:

Trivially, B is also a behavior of T with VaVb replaced by VbVa.

Case (ii) A failure behavior B of LSH always contains VaVb:

Consider the precedence graph formed by B. From condition (iv) of a behavior, we know that any indirect path (path of length greater than two) from Va to Vb must contain a Pa and a Pb in some other thread(s). However, this immediately indicates a violation of TCF; Vb could collide with some other Vb used in enabling the Pb in the above indirect path. Thus, there is no other path from Va to Vb (other than the program order) in B. Then we can reverse the program order to get Vb $\dashrightarrow$ Va from B and the resulting behavior B' is a partial and a failure behavior of T' which is obtained from T by substituing VaVb with VbVa. QED.

Theorem 6.9 Suppose both semaphore a and b are TCF. PaPbVaVbPaPb $\stackrel{f}{=}$PaPb.

Proof: $\Rightarrow$: Consider a failure behavior B of T (LHS) that contains PaPb (thus VaVb as well). As before, the case where B does not contain PaPb is trivial, and it will be omitted here.

Semaphore b is TCF $\Rightarrow$ PbVb appears in both program order and conflict sequence order, i.e., Pb precedes Vb immediately in the latter as well. Thus we could reduce B by deleting Pb and Vb to obtain a new behavior B' which is a failure behavior of T' obtained by deleting PbVb from T. This is repeated for PaVa and we arrive at a failure behavior of the RHS, i.e., the trace set obtained by deleting PaPbVbVa from T.

$\Leftarrow$: The case of a failure behavior B's of RHS containing PaPb can be similarly proved. B' is augmented by inserting VbVaPaPb right after the original sequence of PaPb, leading to a new behavior which is also a failure behavior of the LHS. QED.

The results of Theorem 6.7 and Theorem 6.9 can be generalized as the following:

$A_1A_2A_3 \stackrel{f}{=} A_1$ if sem(A_1) are TCF, where A_3 is a matching all-P sequence of A_1, and A_2 is a matching V sequence of A_1. This general form of local reduction can be proved using a similar reasoning for Theorem 6.7 and Theorem 6.9.

Theorem 6.10 If Va$\in$I and PaVa is the only form in which the synchronization operations on semaphore a occur in a trace set T, then T' $\stackrel{f}{=}$ T, where T' is obtained by deleting all occurrences of PaVa from T.

Proof: T$\Rightarrow$T':

The validity of this theorem rests upon the correspondence between a canonical conflict sequence set and its resulting behavior. Since Pa and Va always appear

together as a consecutive pair in T, semaphore a must be TCF. Once a Pa is executed, no one else could operate on semaphore a until that thread executes the Va. In fact, every such PaVa pair must appear successively in the conflict sequence to form a new conflict sequence and thus a new behavior, which preserves the failure property of the original. Hence the claim.

T'⇒T:

Since Va is in I and every Pa is followed by a Va in T, given any failure behavior B' of T' , we could use the behavior extension algorithm to generate a behavior B of T, such that both B and B' will have the same state and suffix frontier. Thus, B is a failure behavior of T. QED.

Example 6.6 T=<{Va,Vb, Vc},{PbPaVaPc, PcPaVaPbVbPaVa, PaVaPbVcPaVa}> $\stackrel{f}{=}$ T'=<{Va, Vb, Vc}, {PbPc, PcPbVb, PbVc}.

6.4.2 Reduction Heuristics

Unlike reduction rules that preserve the failure property of a trace set, reduction heuristics provides a weaker guarantee: the reduced trace set contains synchronization failure ⇒ the original one also contains synchronization failure. The reverse is not necessarily true.

Theorem 6.11 Suppose t2 is obtained from t1 by inserting the subsequence VaPa somewhere inside t1. t1 $\stackrel{f}{\Rightarrow}$t2 .

Proof: Let B be a failure behavior of the LHS. As before, if B does not contain any event after Pa (in t2) , then B is also a failure behavior of the RHS and we are done. Consider the case where B includes some event ev after Pa (in t2). Let B' be

the maximal behavior prefix of B that excludes the ev. According to the behavior extension procedure, we could augment B' by VaPa and the resulting partial behavior has the same state as B'. The suffix of the resulting behavior for the RHS is the same as the suffix of B' for the LHS. Thus, the resulting behavior could also be extended, by including ev and some other events in the suffix, to some failure behavior of the RHS. QED.

Notice that the reverse $A_1\underline{VaPa}A_2 \stackrel{f}{=>} A_1A_2$ is not necessarily true.

Example 6.7 T=<{Va}, {VaPa, Pa}> can fail but T'=<{Va}, {Pa}> cannot.

Theorem 6.12 $A_1VaA_2 \stackrel{f}{=>} A_1Va\underline{Va}A_2$.

Proof: As before, the case where a failure behavior B does not contain Va is immediate. So we consider the case where B must contain V. We could augment B by expanding V into VV, resulting in a new behavior B' which is a failure behavior of the RHS. QED.

Notice that the reverse $A_1Va1\underline{Va2}A_2 \stackrel{f}{=>} A_1Va1A_2$ is not necessarily true, as shown by the Example FD2 in Appendix C.

Theorem 6.13 Suppose t1 and t2 are two thread treces, ${}^0t1$= Va1, and ${}^0t2$ = Va2. Then {t1, t2'} $\stackrel{f}{=>}$ {t1, t2} where t2' = t2 with Va2 being removed.

Proof: Consider a failure behavior B of the LHS. The conflict sequence of semaphore a in B must contain Va1 in some subset, say, ... Si={Va1}... . We could augment B by inserting Va2 in the first subset of Va in the conflict sequence and obtain a

failure behavior B' of the RHS. It is straightforward to verify that the augmentation preserves conditions (i) to (iv) of a behavior. QED.

6.5 Partial Order Checking

We could check for the existence of failure behavior in T by explicitly generating and checking all the behaviors of T. The process can terminate as soon as a synchronization failure is detected. In the worst case, the absence of synchronization failures would require complete checking of all behaviors. Even in the model checking, the computation can be minimized by configuring the behaviors of T in the form of a behavior tree: a branch in the tree represents a choice in some conflict sequence and a path in the tree represents a behavior. Avoidance of duplicate checking can be ensured by walking through each segment of the tree once regardless of the number of behaviors which include that segment.

6.5.1 Local Behavior Tree

Given a trace set T=<I, {t1, t2, ...}>, we have defined the local trace set Ta = <Ia, {ta1, ta2, ...}> in Definition 5.12. Obviously if we treat Ta as a trace set in the normal sense, we can derive behaviors from it by using the behavior extension algorithm. From Theorem 5.1 and Theorem 5.2, we know that a behavior is characterized by a set of canonical conflict sequences.

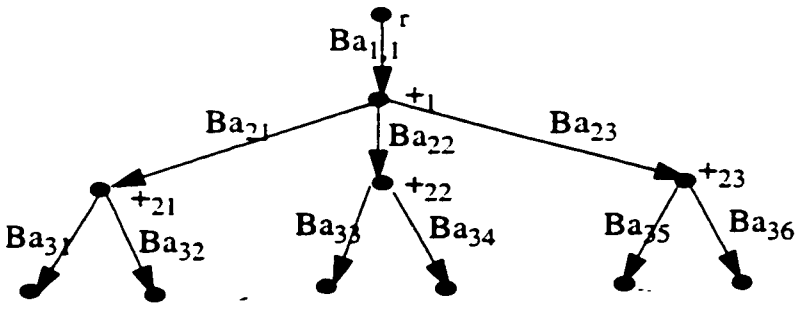
Theorem 6.14 Two conflict sequences of Ta must share a maximum common prefix that is either empty or ends with some {Va} or Pa.

Proof: i^{th} conflict sequence is identified by **Si1**;**Pi1**;**Si2**;**Pi2**; ... . Take any two distinct con

flict sequences, say the i^{th} and the j^{th}. Then it follows there exists a minimal index, say k, such that **Sik≠Sjk** or **Pik≠Pjk**. Then trivially the maximal common prefix shared by both sequences will be **Si1 = Sj1**; **Pi1 = Pj1**; ... ; **Si(k-1) = Sj(k-1)** or **Pi(k-1) = Pj(k-1)**. When k=1, the maximum common prefix is empty. QED.

From the above theorem, we could generate all the behaviors of Ta and organize them in the form of a tree; the first segment of behavior from the root is given by the maximal prefix shared by all behaviors of T. Notice that this could be an empty sequence. We use '$+_v$' to denote a choice in the extension between a Va and a Pa and '$+_p$' between two or more Pa's. Suppose Ba_{ij} is a distinct sub-sequence in level i of the tree. The following diagram depicts a local behavior tree:

Example 6.8 A local behavior tree.



In the above example, each node is drawn as a dark dot with r as the root, and $+_1$ or $+_{ij}$ representing a choice node (+v or +p). Each *branch* (arrow between two nodes) is distinctly labeled: Ba_1 is common to all behaviors and $+_1$ = +v or +p leads to Ba_{21}, Ba_{22} or Ba_{23}. In the former case (a *V choice* node), each Ba_{2j} starts with a distinct {Va}. This is caused by the choice between some Va and Pa. In the latter case (a *P choice* node) , each

Ba_{2j} starts with a distinct Pa. This is caused by the choice between two or more Pa's from different threads. The behavior tree could evolve until eventually each path ends with a successful or failure behavior. Notice that the root is either a V choice node in which case Ba_1 becomes empty, or a non-choice node in which case Ba_1 is non-empty.

We use BTa to denote the local behavior tree for Ta. Given a BTa, a *path* in the tree is the sequence of branches traversed from the root to any non-root node. For convenience, we also label each node in the following way:

The root r is said at level 0 and is labeled as <0>.

If the root r is not a choice node, then the choice node below r is labeled as <1>.

Any other node is labeled as <x.y> where x and y are positive numbers, x representing the level of the node from root r and y the number in that level.

Example 6.9 Consider the local trace set in Figure 41 and Figure 42. The local behavior B1={V1};P2:{V2};P3:{V3};P4:{V4} of Ta in Figure 41 is represented by the left-most path of the tree. The local behavior B2 = {V1,V4};P3 of Ta in Figure 42 is represented by the right-most path of the tree. The root in Figure 42 is a V-choice: the behavior must choose to have either {V1} or {V1,V4}. In other words, the canonical conflict sequence starts with either {V1} or {V1, V4}. The label of a node uniquely captures a path in a local behavior tree and we also use a label to refer to a corresponding path or local behavior. So B1 is also referred as local behavior <2.1> and B2 as <2.5>.

Definition 6.3 A path in a behavior tree is *maximal* if it ends at a leaf, otherwise it is *partial*.

A path in a behavior tree is *complete* if it contains every event of the corre-

sponding trace set. A path in a behavior tree is a *failure* if it is maximal but not complete.

In the example of Figure 42, three leaf nodes are marked with 'F' representing failure behaviors.

A local behavior tree is a more concise representation than a set of conflict sequences or partial orders; common prefixes of local behaviors appear only once in a tree. We will use the terms path, canonical conflict sequence, and a local behavior, interchangeably as they represent equivalent entities.

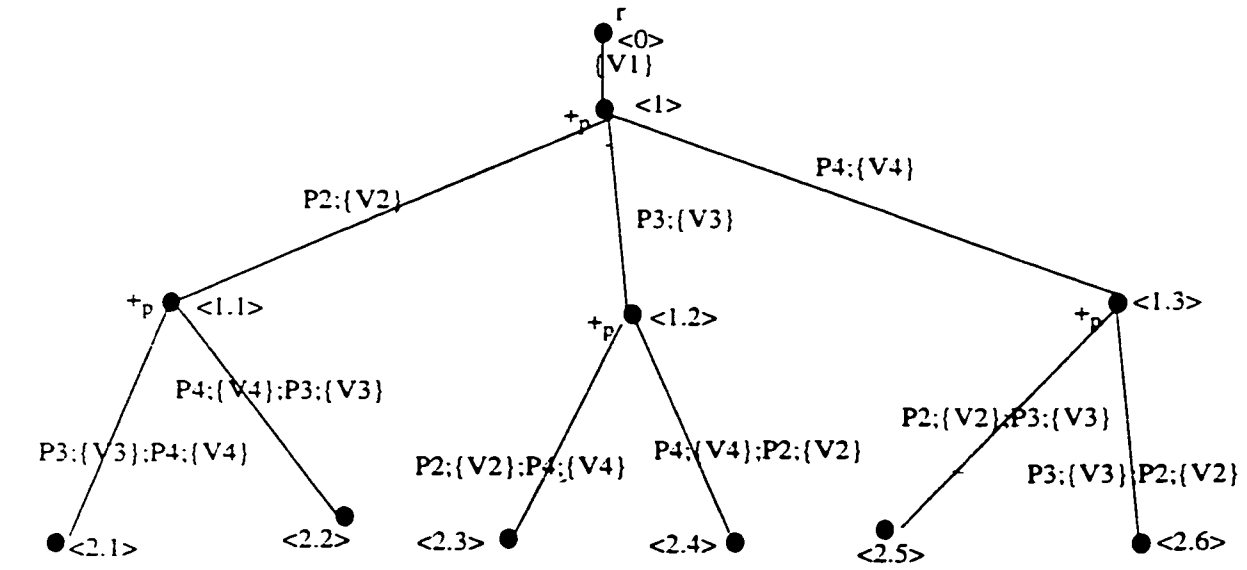


Figure 41 An example to illustrate local behavior tree

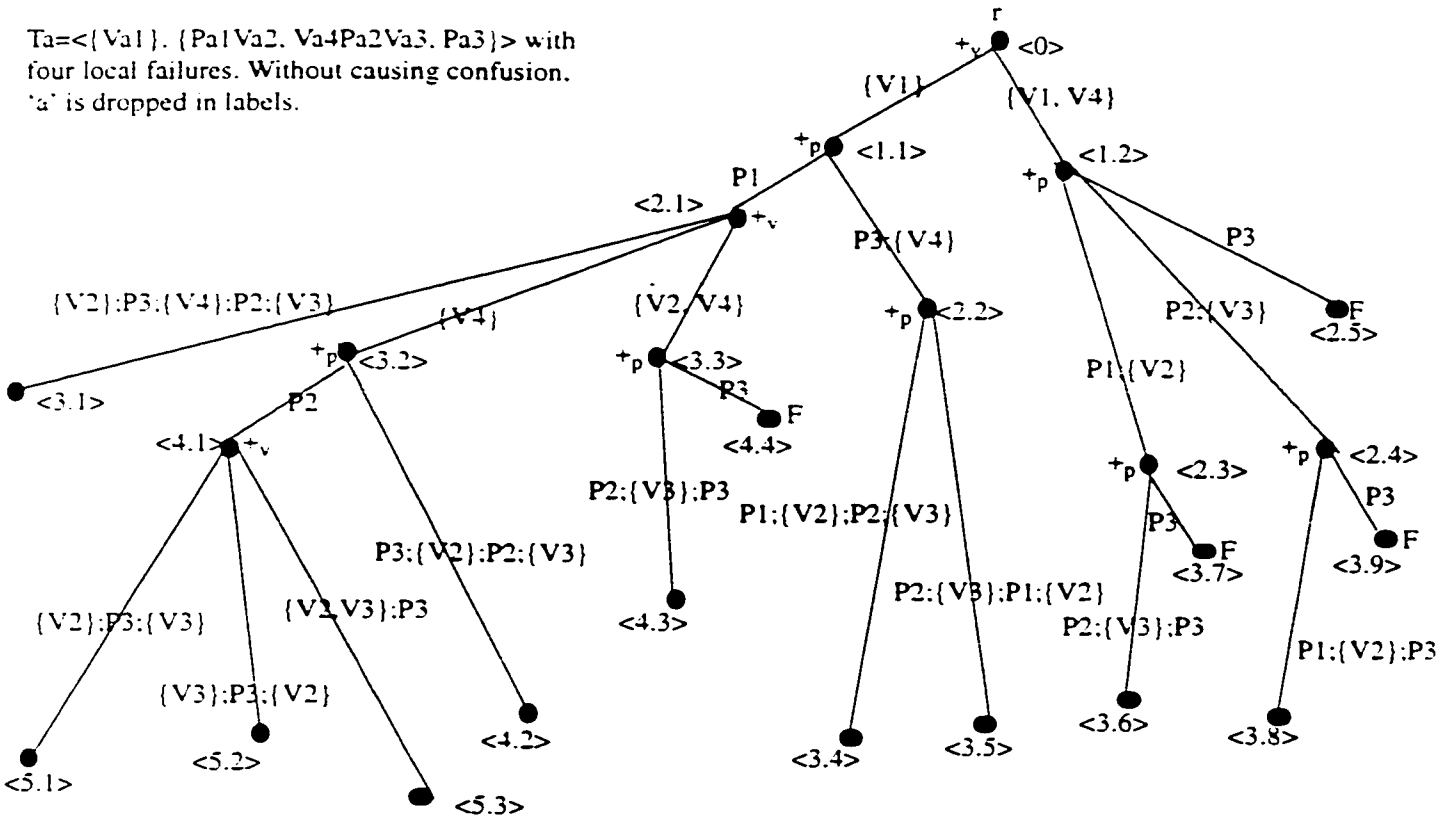


Figure 42 Another example to illustrate local behavior tree

The global behavior tree BT of a trace set T can be derived from the local behavior trees of T. As an example, suppose some T has two local behavior trees BTa and BTc only. The global behavior tree BT may look like the one in Figure 43 next. Every global behavior is represented by two paths in two different local behavior trees: g1 by a1 and c1, g2 by a2 and c1, and so on. Same as in BTa or BTc, common prefixes of global behaviors appear only once in BT.

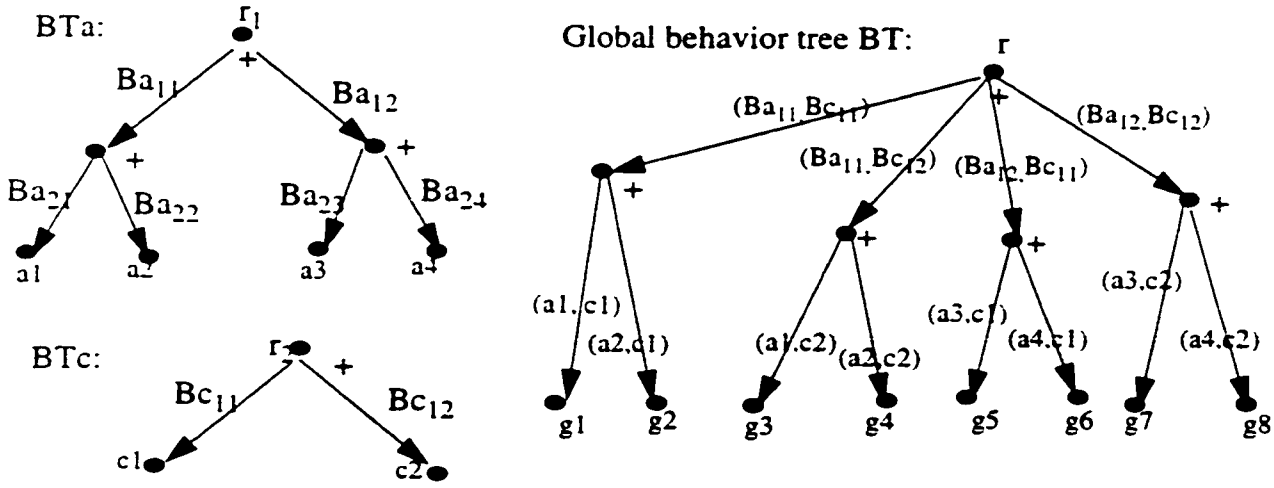


Figure 43 A global behavior tree

Even though we may not have to generate the whole BTa in analysis, we present an algorithm to construct BTa from Ta. We introduce some notations first. Given a subsequence of the local thread trace tai, vai is the maximum prefix of tai formed by consecutive V events. (va1, va2, ...) can be viewed as some trace set (of all V events). We call vai the *maximum V prefix* of tai and vai is empty if ^{0}tai=P.

Input: Ta=<Ia, {ta1, ta2, ...}>, a local trace set.

Output: The local behavior tree of Ta, BTa.

Procedure LTG: local_tree_generation(Ta);

push ({}, {ta1, ta2, ...}) onto the stack and add the root node;
while the stack is non-empty do
    pop (Av, Sf) from the stack where Av is a set of V and Sf = (sa1, sa2, ...) is a suffix of Ta;
    V-phase:
    if Av is empty then
        Sv:={W | W is a distinct prefix of (vt1, vt2, ...)} where vti is the maximum V prefix of sai;
        if |Sv| > 1, then add a V choice node;
        for each member C of Sv do
            if Sf' (obtained from Sf by deleting events that appear in C from Sf) is non-empty then push (C, Sf') onto stack;
    P-phase:
    if Av is non-empty then
        Sp := {^{0}sti | ^{0}sti=P for i = 1, 2, ...};
        if |Sp| > 1, then add a P-choice node;
        for each member C of Sp do
            if Sf' (obtained by deleting events that appear in C from Sf) is non-empty then push ({},Sf') onto stack;
end of while ... do.

Here is some explanation. At initialization, it is assumed that every member of Sv contains I. The deterministic extension continues until either Sv or Sp contains multiple choices, in which case a choice node is added and the corresponding suffix of each choice, if non-empty, is pushed onto stack for further extension. If a corresponding suffix of some choice (either P or V choice) becomes empty, then it means a complete behavior has been

generated. After a V-phase, an non-empty Av will be pushed onto stack together with its non-empty suffix to indicate that V is available in the (partial) behavior. After a P-phase, an empty Av will be pushed onto stack together with its non-empty suffix to indicate that V is unavailable in the (partial) behavior and the next phase must be a V-phase. The algorithm terminates when the stack is empty, i.e., all choices are considered. The correctness of the algorithm is straightforward and its proof is omitted.

Look at the local trace and the local behavior tree in Figure 42 again. Starting from the root, the first Sv would be {{V1},{V1, V4}}. So the root is a V choice node with two branches that are labeled with {V1} and {V1, V4}. The suffix of Ta after the removal of {V1} is Sf1={P1V2, V4P2V3, P3}. The suffix of Ta after the removal of {V1, V4} is Sf2={P1V2, P2V3, P3}. Both suffixes are pushed onto stack to be further extended. For the branch labeled with {V1}, the P events that can be extended is a set {P1, P3}. For the branch labeled with {V1, V4}, the P events that can be extended is a set {P1, P2, P3}.

6.5.2 A Canonical Representation of Partial Order Behaviors

Given a trace set T, every (partial order) behavior B of T can be represented as a set of local behaviors, one per local behavior tree. In other words, B can be represented using a set of paths, one per local behavior tree. There is one to one mapping between local behaviors and paths. We say a path is (in)feasible if its corresponding local behavior is (in)feasible (section 6.2).

Let $\mathbf{B_T}$ ($\mathbf{B_{Ta}}$) denote the set of all maximal behaviors of T (Ta) or maximal paths in BT (BTa), and **Y** be the full cross product $\mathbf{B_{Ta}} \times \mathbf{B_{Tb}} \times \mathbf{B_{Tc}} \ldots = \{\langle ya, yb, \ldots\rangle \mid ya \in \mathbf{B_{Ta}}, yb \in \mathbf{B_{Tb}}, \ldots\}$ for semaphore a,b, ... of T. For every member of $\mathbf{B_T}$, there is a corresponding member of **Y** representing it. The reverse is not necessarily true. In Figure 43, $\mathbf{B_{Ta}}$={a1, a2, a3, a4}, $\mathbf{B_{Tc}}$={c1,c2} and **Y**={<a1, c1>, <a2, c1>, <a3,c1>, <a4,c1>, <a1, c2>, <a2, c2>, <a3,c2>, <a4,c2>}.

Consider the trace set and local behavior trees in Figure 44. $\mathbf{B_{Ta}}$ = {a1, a2, a3, a4}, $\mathbf{B_{Tb}}$={b1, b2}, $\mathbf{B_{Tc}}$={c1} and **Y** = {<a1, b1, c1>, <a1, b2, c1>, <a1, b1, c1>, <a2, b2, c1>, <a3, b1, c1>, <a3, b2, c1>, <a4, b1, c1>, <a4, b2, c1>}. H1$\in \mathbf{B_T}$ and is represented by <a3, b1, c1>$\in$ **Y**. It can be verified that local behavior (path) b1 is feasible. However, the

local behavior (path) b2 is infeasible: Vb1 cannot happen before Pb as it depends on Vc that comes after Pb. Notice that a3 is feasible and a failure.

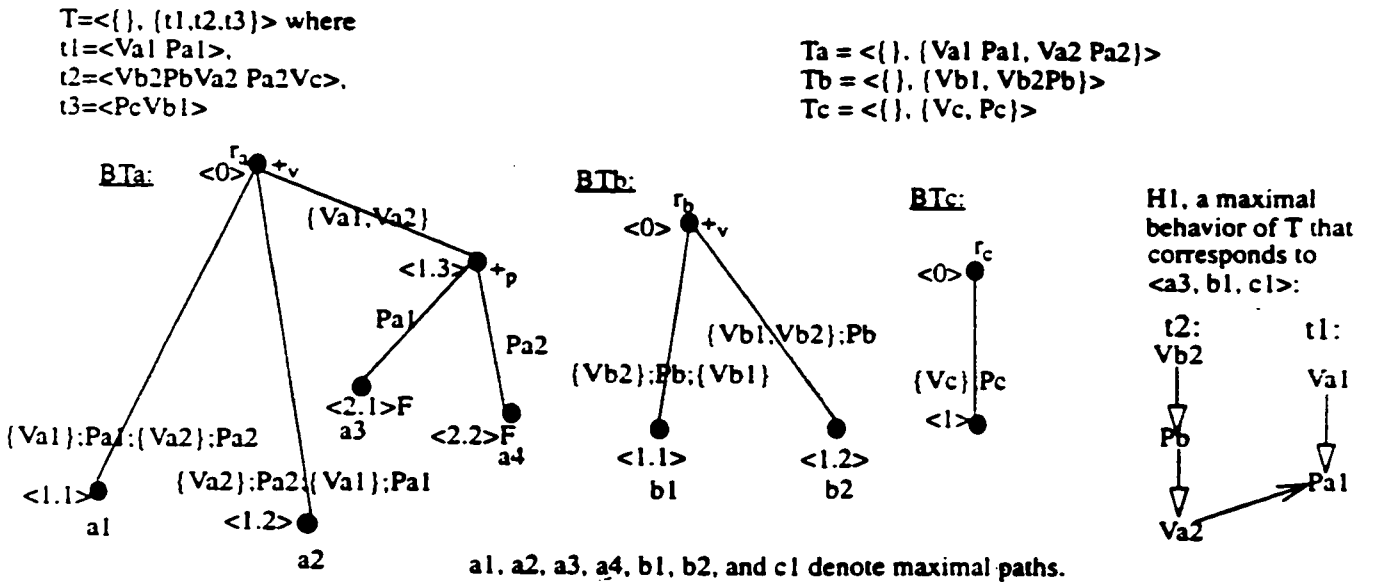


Figure 44 An example to illustrate feasible and infeasible local behaviors

Y captures every possible combination of local behaviors, and hence global behaviors of T. Therefore, we can check behaviors of T for failure by checking members of Y. We can revise the BBE (section 5.2.3) to check whether <ya, yb, ...> ∈ Y corresponds to a failure behavior of T.

The BBE can select any V in the beginning of a trace suffix or any Pa event in the beginning of a trace suffix if the input behavior B has available Va. Given <ya, yb, ...>, each local behavior ya, yb, ... specifies some program order between synchronization events. This restricts what can be selected for further extension. One can only select those that satisfy the program order in the suffixes of local behaviors. We present the revised behavior extension algorithm next.

6.5.3 Using Full Cross Product to Detect Synchronization Failure

Given a partial behavior B of T, the associated suffix, and some <ya, yb, ...> from Y, the new algorithm, called the *Guided* Behavior Extension (GBE), either extends B with one event from the suffix or terminates and concludes that B cannot be further extended with respect to <ya, yb, ...>. The GBE is similar to the BBE except that each extension step maintains consistency with the local behaviors. Unlike the BBE, when the GBE concludes that B cannot be further extended with respect to <ya, yb, ...>, it could mean either B is a failure behavior of T or <ya, yb, ...> is infeasible (that is, at least one of ya, yb, ..., is infeasible) and should be discarded. As long as there exists one member of Y that leads the GBE to terminate with a failure behavior then T can fail, otherwise, T cannot fail.

The GBE terminates when every trace suffix starts with a P for which there is no available V in B. Such a B is a failure since it cannot be extended further. It also terminates in two other cases. Case (i), some suffixes start with some V but the V cannot be selected for extension because of the restrictions on program orders that <ya, yb, ...> enforces. A local behavior contains some program orders that may not be consistent with some other local behaviors. Case (ii), every trace suffix starts with P, the compatible V for one or more such P's is available in B, and these P cannot be selected for extension because of the restrictions on program orders that <ya, yb, ...> enforces. B is not necessarily a failure in either case since it could been extended if another tuple <ya', yb', ...> of Y is used instead of <ya, yb, ...>. Therefore, we just discard the tuple <ya,yb,...>.

Input: A partial order behavior B=<Ev, O>, the associated suffix {st1, st2, ...} and a set of paths y=<ya, yb, ...>∈ Y. Symbols Pb^0 and $\{Vb\}^0$ are defined in BBE of section 5.2.3.

Output: B', a behavior extended from B by zero or more event from the suffix of B.

Procedure GBE: guided_behavior_extension (B, {st1, st2, ...}, <ya, yb, ..., >)

B' := <Ev', O'> := B;

if possible choose a ^{0}sti on some semaphore b such that *^{0}sti is in yb, every event that is ordered before ^{0}sti according to yb is already in B*, and ^{0}sti=Pb ⇒ Vb is available in the current B';

case (1) ^{0}sti = Vb and Pb^0 exists:

O' := O∪{(pti^0, ^{0}sti), (Pb^0,^{0}sti)}; Ev' := Ev∪{^{0}sti};

case (2) ^{0}sti=Vb and no Pb^0 exists (there may exist $\{Vb\}^0$):

if no $\{Vb\}^0$ exists then Ev' := Ev∪{^{0}sti};

else O' := O∪{(pti^0, ^{0}sti)}; Ev' := Ev∪{^{0}sti};

case (3) ^{0}sti=Pb:

O' := O∪{(pti^0, ^{0}sti)}∪{(Vb, ^{0}sti) | Vb∈ $\{Vb\}^0$};

Ev' := Ev∪{^{0}sti};

if B' is s complete behavior then return B' and indicate its completeness;

else return "Continue with B' ";

else *if the every ^{0}sti=P and there is no available V that is compatible with P in B* then return " B is a failure behavior ";

else return " *Discard <ya, yb, ...>* ".

Like the original BBE, it can be easily checked against the conditions (i) to (iv) of Definition 5.2 that when the algorithm returns B', B' is a a behavior of T. The correctness proof is omitted.

As an example, consider the trace set in Figure 44 again. We illustrate how B1 is generated step by step from the input <a3, b1, c1>. Suppose the algorithm starts with an empty behavior. The suffixes are Va1Pa1, Vb2PbVa2Pa2, PcVb1 respectively. Here is how the GBE generates B1 and terminates:

(1) chooses Va1 and the suffixes become {Pa1, Vb2PbVa2Pa2Vc, PcVb},

(ii) chooses Vb2 and the suffixes become {Pa1, PbVa2Pa2Vc, PcVb1}. Notice that it cannot choose Pa1 (as the BBE would do) since that will violate a3 which requires Va2 to be executed before Pa1,

(iii) chooses Pb and the suffixes become {Pa1, Va2Pa2Vc, PcVb1}. So we get Vb2 $\xrightarrow{p.l}$ Pb of B1,

(iv) chooses Va2 and the suffixes are {Pa1, Pa2Vc, PcVb1},

(v) chooses Pa1 and the suffixes become {Pa2Vc, PcVb1}, and we get the behavior B1 Va1 $\xrightarrow{p.l}$ Pa and Va2 $\xrightarrow{p.l}$ Pa. Notice that the GBE cannot choose Pa2 since that will violate a3 which requires that both Va1 and Va2 to enable Pa1. No more extension is possible, so the algorithm concludes that B1 is a failure and terminates.

Notice that what the GBE returns may not contain every event in the input tuple <ya, yb, ...>. For example, given the input of an empty behavior with <a3, b1, c1> in Figure 44, the algorithm returns the behavior B1 of T as shown in the right side which only contains five events while a3, b1 and c1 contain all events of T.

As another example, if we apply the GBE with input <a3, b2, c1> starting with a behavior B containing Va1 and Vb2 only, then it will return "*Discard <a3, b2, c1>*". This is because even though every trace suffix starts with P (Pa1, Pb, Pc) and there are compatible V (Va1, Vb2) available in B, the algorithm cannot select Pa1 or Pb for extension. According to a3, Pa1 need to be enabled by both Va1 and Va2 and according to b2, Pb need to be enabled by both Vb1 and Vb2.

The input to the GBE need not be a tuple of all maximal paths for the algorithm to work. The GBE works fine for an input <ya,yb, ...> where some of ya, yb, ... , are partial. In such a case, the GBE may still terminate with a failure of T or conclude that the input should be discarded, but it will not return a complete behavior.

6.6 Local Failure Validation

6.6.1 Use of Feasible Local Failure to Detect Global Failure

The projection of a behavior B of T onto a semaphore *a* is a local behavior Ba. If Ba is a local failure, then it is feasible because of such B and obviously such a B is either a global failure or prefix of a global failure of T. Therefore, knowing whether a partial behavior of T contains a local failure can help the GBE to detect a global failure more efficiently. The GBE can stop and say T can fail as soon as it has reached a partial behavior that contains a local failure. We show next how to improve upon the GBE using the above observation.

During the behavior extension by the GBE, some tuple <ya,yb, ...> may be discarded as no further extension is possible. However, this does not mean that B could not be a prefix of some failure behavior that is obtainable from another tuple. For example, consider the trace set of two semaphores in Figure 45. Each local behavior tree has two maximal paths and a1 is a local failure. **Y** = {<a1,b1>, <a1,b2>, <a2,b1>, <a2,b2>}. If we apply the GBE to the tuple <a1, b1>, we will get the partial behavior B1 that contains a1 (and hence a1 is feasible) as shown on the right in the same figure. If we continue to apply the GBE, the algorithm will then return *"Discard the tuple <a1,b1>"* because it cannot extend B1

according to a1 and b1. It can be verified, however, that B1 is a prefix of a global failure behavior that is obtainable by the same algorithm with another tuple <a1, b2> as input.

T = <{}, {t1,t2}> = <{}, {Va1Pa1Vb1,Vb2Pa2Va2Pb}>
Ta = <{}, {Va1Pa1, Pa2Va2}>
Tb = <{},{Vb1, Vb2Pb}>

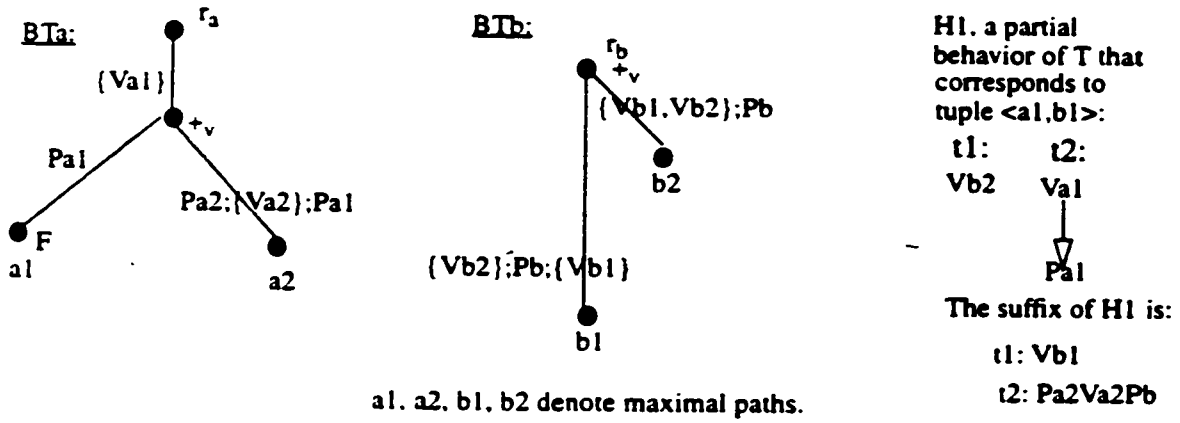


Figure 45 Making use of feasible local failure in the GBE

We modify the GBE to include an additional checking to see whether B contains a local failure behavior and if it does, then the algorithm immediately returns that T can fail. We show the modified GBE in Figure 46.

Input: A partial order B=<Ev, O>, the associated suffix {st1, st2, ...} and a set of paths y=<ya, yb, ...>, a member of Y. Symbols Pb^0 and {Vb} are defined in BBE of section 5.2.3.

Output: B', a behavior extended from B by one more event from the suffix of B.

Procedure Modified GBE: modified_guided_behavior_extension (B, {st1, st2, ...}, <ya, yb, ...>)

B' := <Ev', O'> := B;
if possible, choose a ^{0}sti on some semaphore b such that ^{0}sti is in yb, every event that is ordered before ^{0}sti according to yb is already in B, and ^{0}sti=Pb ⇒ Vb is available in the current B';
    case (1) ^{0}sti := Vb and Pb^0 exists:
        O' := O∪{(pti^0, ^{0}sti), (Pb^0,^{0}sti)}; Ev' := Ev∪{^{0}sti};
    case (2) ^{0}sti=Vb and no Pb^0 exists (there may exist {Vb}^{0}):
        if no {Vb}^{0} exists then Ev' := Ev∪{^{0}sti};
        else O' := O∪{(pti^0, ^{0}sti)}; Ev' := Ev∪{^{0}sti};
    case (3) ^{0}sti=Pb:
        O' := O∪{(pti^0, ^{0}sti)}∪{(Vb, ^{0}sti)|Vb∈{Vb}^{0}};
        Ev' := Ev∪{^{0}sti};
    if B' is s complete behavior then return B' and indicate its completeness;
    else return "Continue with B' ";
else if the every ^{0}sti=P and there is no available V that is compatible with P in B
    then return "B is a failure behavior";
else if B contains some local failure, then return "B is a (prefix of some) failure behavior";
else return "Discard <ya,yb, ...>".

Figure 46 The modified GBE algorithm

The last '*else if*' clause checks whether there is some local failure that B contains and if so, the algorithm reports that the trace set T can fail. Therefore, in selecting tuples from Y, we should first select those tuples that contain at least one failure to validate whether such local failure is feasible. What remains to be shown is how to find some local failures efficiently.

6.6.2 Locate Simple Local Failures

The first step to find some local failures of a local trace set is to see whether there is any. This can be done efficiently by using Theorem 6.5. If the answer is yes, we then start to find some local failures. To find every local failure in a single Ta takes time exponential to the size of Ta in the worst case. This is true because a local behavior tree could have an exponential number of leaves and most of the leaves are failures. We only locate a subset of local failures that are "closest" to the root in BTa. Those failures can be generated fast. Next we use an example to illustrate these failures.

Suppose Ta = <{Va1}, {Pa1 Va2 Pa2 Va3, Pa7 Va6 Pa8 Va7 Pa3 Va4 Pa4, Pa9 Va8 Pa10 Va9 Pa5 Pa6 Va5}. Since there is no Va after Pa4, it is easy to see that BTa contains local failures whose last branch is labeled with Pa4. It is also easy to see that BTa contains another local failure whose last branch is labeled with Pa5. We call such Pa4 or Pa5 *failing* P of Ta.

Definition 6.4 A *failing* P of Ta is the *first* such Pa in a local thread trace tai of Ta that either (i) tai contains consecutive PaPa', or (ii) after this Pa there is no Va but there is another local thread trace that also contains some Pa'. A local thread trace has at most one failing P.

In BTa, more than one failure may end with a failing P.

Example 6.10 Consider the local behavior tree in Figure 42, there are four failures that end with a branch labeled as Pa3: node <4.4>, node <3.9>, node <3.7>, and node <2.5>. The last one is shortest or closest to the root.

We give a simple procedure to obtain a special local failure br that ends with a failing P for a local trace tai:

Input: a failing P of tai.

Output: a local failure ends with P.

Procedure GLF: generate_local_failure(P, tai).

(i) br := empty;
(ii) v_prefix := the maximal prefix of the current Ta that contains V only; extend br by adding v_prefix to it; remove v_prefix from the current Ta;
(iii) extend br by adding ^{0}tai, a P event; remove the P from tai.
(iv) repeat (ii) to (iii) until the failing P is in br.

Here is some explanation. We assume that the initial Va in Ia is also included in the first v_prefix generated. All V in v_prefix are immediately used to enable ^{0}tai only. Therefore, the failing P in tai will be enabled at the earliest possible time. We call such a local failure the *shortest* local failure of tai that ends with the failing P.

6.7 Local Behavior Tree Pruning

In searching for failure behavior, one can enumerate every combination in **Y** using the Modified GBE. This brute force approach checks for every combination of all local behaviors until a failure is found. Due to synchronization races, the number of combination could be exponential and their enumeration could be very time-consuming. In this section, we discuss how some paths in a local behavior tree may be pruned because it is infeasible or it is *dominated* by another. We define domination next.

6.7.1 Behavior Domination

Definition 6.5 Let br1 and br2 be two distinct paths of some local behavior tree BTa. We say that br2 is *dominated* by br1 iff br1 is infeasible $\Rightarrow$ br2 is infeasible.

Example 6.11 T=<{Va1, Vb1}, {Pc1 Pe1 Va2 Vb2 Pc2 Pe2 Va3, Pa1 Vc1 Vd Pa2 Pf1 Vc2 Vd Pa3 Pf2, Pb1 Pd1 Ve1 Vf1 Pb2 Pd2 Ve2 Vf2 }>, and Ta = <{Va1}, {Va2 Va3, Pa1 Pa2 Pa3}>. Let br1, br2, and br3 are three different paths in BTa:

br1 = {Va1}; Pa;{Va2}, br2={Va2}, and br3={Va1,Va2}.

br2 is infeasible because Va2 cannot be executed according to the trace set T. Because of that, br3 must be infeasible too because {Va1, Va2} is a *superset* of {Va2}. Therefore, br3 is dominated by br2. The reasoning is simple: since the rest of conditions are the same, if a set of events cannot be executed , then any set containing the set (a superset) cannot be executed either.

Theorem 6.15 Let br=br_c;{V} be an infeasible path and br'= br_c;{V}' be another path of

some BTa. If {V}' is a superset of {V}, then br' is dominated by br.

Proof: Assume br is infeasible. So there is no global behavior that contains every event of br. Suppose br' is feasible, then there is a global behavior B that contains every event of br'. Since br' and br have a common prefix and br ends with a branch containing less or equal amount of V events than that of br', B also contains every event of br, which implies that br is feasible, a contradiction. Therefore, br' must be infeasible too. QED.

The checking for dominance is straight-forward. The not so straight-forward task is to find some infeasible path first to be used. There is no efficient way to solve this problem for general local trace sets. However, we will show a special class of trace sets, whose local behavior trees can be checked efficiently.

6.7.2 Infeasible Paths

Let L={T} be a set of the trace sets such that *every* semaphore a of T satisfies the following two conditions:

Condition P: All Pa appear in one thread trace (say t1) only;

Condition V: All Va appear in another thread trace (say t2) only.

It is possible to verify whether some paths of local behavior trees of such T are infeasible in time proportional to the size of T. The intuition here is:

(i) program order constraints, that is, for some Va events to be executed , every P that is in the same trace and occurs before any of such Va events must be executed already,

(ii) Semaphore semantics constraints, that is, for a certain number of Pa to be executed, at least the same amount of Va's must be executed,

(iii) All V in a collision set {V} need to be collided (as if all were executed at the same time), and

(iv) Given a trace set suffix, if all Pa are in t1 only and all Va are in t2 only, then in the best case (TCF case), the i^{th} Pa, will be enabled by the i^{th} Va if there is no initial Va. If there is an initial Va, then the i^{th} Pa will be enabled by the i^{th} - 1 Va. When there is at least one token collision, the i^{th} Pa will be enabled by the k^{th} Va ($k > i$ if no initial Va, $k = i$ otherwise).

Therefore, before extending a path with a branch labeled with a collision set {Va}, we could check whether every Pb that is from the same trace set and is before any V in {Va} can be executed first. If any of such Pb's could not be executed due to the lack of Vb, then we do not extend the branch further.

Example 6.12 T=<{Va1, Vb1}, {Pc1 Pe1 Va2 Vb2 Pc2 Pe2 Va3, Pa1 Vc1 Vd Pa2 Pf1 Vc2 Vd Pa3 Pf2, Pb1 Pd1 Ve1 Vf1 Pb2 Pd2 Ve2 Vf2 }>, and Ta = <{Va1}, {Va2 Va3, Pa1 Pa2 Pa3}>. It is easy to verify that T∈ **L**. Let a path br1 = {Va1}; Pa1. We now try to extend br1 with branch {Va2,Va3}.

The following is the reasoning to conclude the extended br1 will be infeasible. Suppose it is feasible, then according to program order, for Va2 and Va3 to collide, Pc1 and Pc2 before Va3 must be executed already. Since all Vc are in one thread, the *best* case is that

Vc1 enabled Pc1 and Vc2 enabled Pc2, which implies that, Vc1 and Vc2 must be executed before the collision. According to program order, for both Vc1 and Vc2 to be executed, Pa1 and Pa2 must be executed already, which is impossible since there is no available Va for Pa2 (Va1 is already used by Pa1). Therefore, two V events in {Va2, Va3} can never collide. The extended br is infeasible and we should not extend br1 to include {Va2,Va3}. Using a similar reasoning, we can show that the path br2 = {Va1,Va2} is infeasible.

Let s1 and s2 are two distinct semaphores and br= br_p;{V_{s1}} be a path of B_{s1} where br_p represents the path from the root to the V-choice node inclusive. We formalize the above discussion next.

Theorem 6.16 The path br = br_p;{V_{s1}} in B_{s1} of T∈ **L** is infeasible if in the *suffix* of br, there is another semaphore s2 in T and $N_{s1} > 0$ where (i) N_{s2} = the total number of distinct Ps2 that is before any V in {V_{s1}}, and (ii) N_{s1}=the total number of distinct Ps1 that is before the k^{th} (k=N_{s2} -1 if there is initial Vs2, k = N_{s2} otherwise) Vs2.

Proof: Since T is in **L**, for any semaphore s, Vs is only in one thread trace and Ps is only in another one. There are altogether N_{s2} Ps2's before any Vs1 in {V_{s1}}. According to program order, before all Vs1 in {V_{s1}} are executed and collided, each of these Ps2 must be executed already. Even if s2 is TCF, N_{s2} Vs2's are required to enable these Ps2's. According to program order, before these Vs2's are executed, every Ps1 that is before the k^{th} Vs2 must be executed already. But

$Ns1 > 0$, which implies that there is at least one such Ps1 that has not been executed (not in br_p). Therefore, one need not consider the $\{V_{s1}\}$ in extending path br_p. In other words, the branch $\{V_{s1}\}$ can be pruned. QED.

Re-visit br1 = {Va1}:Pa1:{Va2,Va3} of Example 6.12 above. Let s1 = a and s2 = c. We have $N_{s2} = 2$ and $N_{s1} = 1 > 0$. According to the above theorem, br1 is infeasible and can be pruned. As for br2 = {Va1,Va2}, Let s1 = a and s2 = c. We have $N_{s2} = 1$ and $N_{s1} = 1 > 0$. According to the above theorem, br2 is infeasible and can be pruned too.

6.8 Algorithm to Detect Global Failures

We have presented various techniques to shorten the time to detect synchronization failures. In this section, we combine these techniques in an algorithm, named Failure Detection Algorithm (GFD).

6.8.1 The Major Steps of GFD

A natural approach to combine those techniques is to apply them in the following order to a given trace set T.

(i) Application of Short-cuts

The results in section 6.3 are used to see T can or cannot fail. If it is inconclusive, then we proceed to (ii).

(ii) Application of Reduction rules and Reduction heuristics

The trace reduction rules are used followed by the reduction heuristics as outlined in section 6.4.

(iii) Local Failure Validation

After the trace set is reduced, we then determine whether there exists any local failure. If there is none, then this phase is skipped. It there is some, we find the shortest ones and for each shortest local failure, apply the Modified GBE (section 6.6) to see whether it is feasible. If it is feasible, then T contains synchronization failure and we are done. Otherwise continue checking the next shortest local failure. If none of the shortest local failures is feasible, then we move on to phase (iv).

(iv) Partial Order Checking

Here we apply the Modified GBE for every member of **Y** (section 6.5) until a synchronization failure is found. The following improvements are made to improve the time efficiency of the algorithm:

(1) *on -demand* generation/unfolding of local behavior tree.

The algorithm GBE in section 6.5 could be used to generate every *maximal* path of every local behavior tree of T and hence the complete **Y**. We could then check every combination in **Y** to see whether it corresponds to a global failure. However, this requires every local behavior tree to be fully generated, which may be wasteful if one could detect synchronization failures with non-maximal inputs or discard some non-maximal inputs.

With that in mind, we start the Modified GBE with a combination of *shorter* paths: paths that are closer to the root. If such a combination corresponds to a global failure then we are done. If it is infeasible, we throw it away. In both case, we save the time to generate the rest of those paths. However, if the tuple is neither infeasible nor corresponding to some

global failure, then we continue with the extension of one of the paths. This technique of on-demand generation of a local behavior tree enables us to generate a local behavior tree to as far as necessary.

Example 6.13 Figure 47 depicts how a local behavior tree is generated on-demand by expanding one choice node at a time. We first generate the local behavior tree up to the first choice node, say a1. Suppose the Modified GBE cannot decide, we generate every branch of a1 up to the next set of choice nodes or leaves, say (a2, a3, a6). Suppose a2 can be pruned and a3 and a6 are choice nodes. The process continues with the branches of a3 and so on, using depth-first generation of a tree.

With this on-demand generation of local behavior trees, the Modified GBE algorithm still applies except for the interpretation of the return value "Discard ...". Given an input of a tuple of all *maximal* paths, the Modified GBE returns "Discard" to indicate that some path of the tuple is infeasible and the tuple can be ignored. When only *partial* paths are given as input, "Discard" only means that the algorithm cannot extend the partial behavior with the partial paths as the input. However, we may still extend the partial behavior after the partial paths in the input are extended.

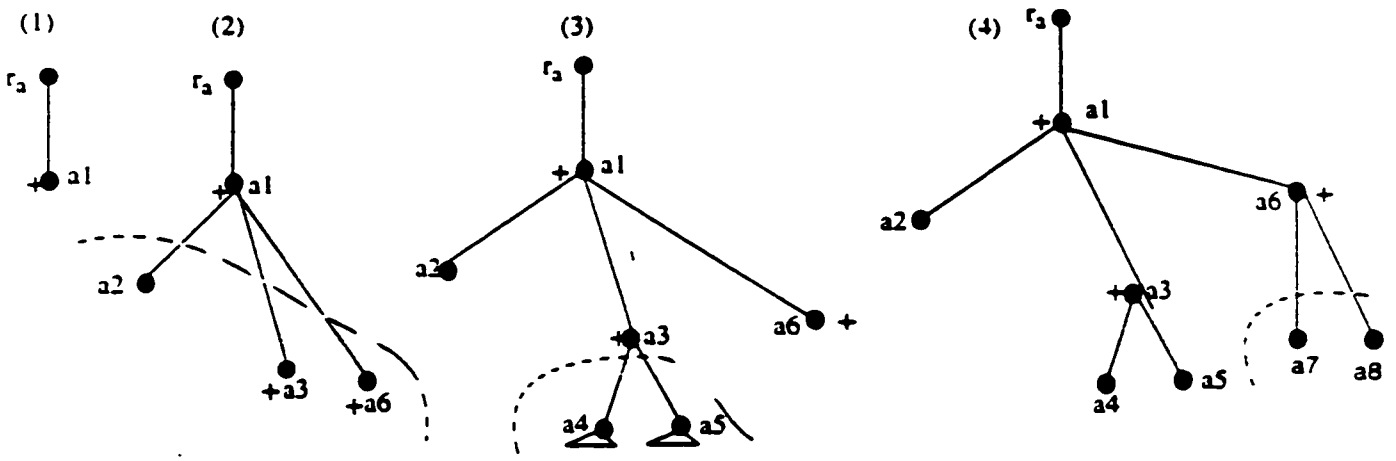


Figure 47 On demand generation of a local behavior tree

(2) pruning of local behavior tree (section 6.7).

During the on-demand generation of a local behavior tree, we will not extend a path if it can be pruned (infeasible or dominated). For example, if path a2 in Figure 47 is infeasible, then we will not extend it.

Example 6.14 Figure 48 illustrates how the synchronization failure detection algorithm works in this phase. A complete description of the algorithm is given in the section following that.

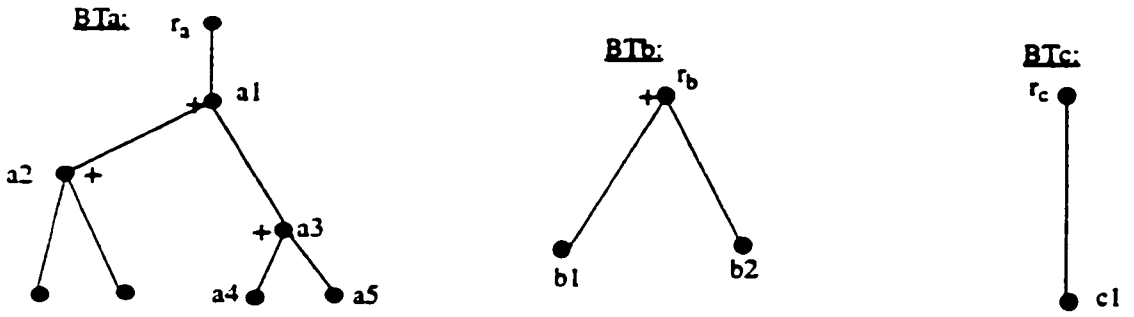


Figure 48 Illustration of synchronization failure detection algorithm

Suppose that the tuple <a5, b2, c1> is the only synchronization failure. Our detection algorithm starts with <a1, Rb, c1> and finds it does not correspond to a synchronization failure but a partial behavior including a1. It then extends to a2 and a3. Suppose that a2 is pruned. It continues with tuple <a3, Rb, c1>, which does not correspond to a synchronization failure either and does not contain a3. So the only choice left is to extend to b1 and b2 and continue with <a3, b1, c1>. It finds out that the extension containing a3 is not a synchronization failure. So it continues with a4 and a5 using tuple <a4, b1, c1> followed by <a5, b1, c1>. No synchronization failure is found so it checks <a1, b2, c1>, followed by <a3, b2, c1> and <a4, b2, c1>. Eventually the algorithm will find a synchronization failure associated with the tuple <a5, b2, c1>.

6.8.2 Synchronization Failure Detection Algorithm

The following is the algorithm to detect global failures in a trace set T. It combines the techniques we have discussed above together.

form and then repeat steps (iii) to (v) of the above algorithm. This is because *Reduction Heuristics* work one-way only: 'the reduced T fails' ⇒ 'T fail', not vice versa.

Input: A trace set T.

Output: FAILURE if T contains failure behavior, NO_FAILURE otherwise.

Procedure GFD: global_failure_detection(T).

(i) apply short-cuts to T. if it is determined that T can fail then return FAILURE, else if it is determined that T cannot fail, then return NO_FAILURE;
(ii) apply trace set reductions and reduction heuristics T; repeat (i) once;
(iii) if T has local failure then for every shortest local failure do
validate whether it is feasible using the Modified GBE algorithm;
if it is feasible then return FAILURE.
(iv) push (B_0, <Ca,Cb, ...>) into stack where Ca (Cb, ...) is the path of BTa (BTb, ...) of T that terminates at a choice node or a leaf node and B_0 is the empty behavior;
(v) while stack is non empty do
pop (B, tp1) from stack;
apply the Modified GBE to see what it returns:
case "Failure": return FAILURE;
case "Discard":
if B *contains* at least one path of tp1 which ends with a choice node,
then generate all branches of any of such paths;
for every such branch br that cannot be pruned do
push (B, tp2) where tp2 is similar to tp1 except that pa is extended with br;
else discard tp1;
case " Continue with B' ": push(B', tp1);
end of while ... do;
(vi) return NO_FAILURE. Here the stack is empty, which implies that every tuple of Y has been checked and none of them is a synchronization failure.

Figure 49 The algorithm to detect global failures.

The algorithm terminates since the size of the trace set T is finite. As we point out in section 5.3.2, the worst case time complexity of the algorithm is exponential unless P=NP. Notice that in case that the algorithm returns NO_FAILURE for a T that is reduced by applying *Reduction Heuristics* (section 6.4.2), T needs to be rolled back to its original

Chapter 7 Some Experimental Results

7.1 Introduction

To evaluate the effectiveness of the algorithm GFD, we use four applications: two with producer/consumer type of synchronization, and two with critical sections. Trace sets of successful executions are derived manually from the algorithms. For each application we give (i) a description of the problem, (ii) the correct algorithm that solves the problem, (iii) the modified algorithms that contain errors (but still have a successful execution), and (iv) the effectiveness of the GFD.

To evaluate the effectiveness of the GFD, we take the problem size as a parameter. For example, the problem size for the L-U decomposition problem is the dimension of the matrix under decomposition. For a given algorithm, the problem size depends on the size of the trace set: the number of threads (Nt), the number of semaphores (Ns), and the number of events (Nn).

In particular, we compare the GFD against the GNR. The latter is same as the GFD except that it does not use reductions. The speedup of the GFD over the GNR is the time it takes for the latter to detect synchronization failures (t_n) divided by the time the former takes (t_f).

For each application, we introduce two errors that may cause synchronization failure and we construct three trace sets with different problem sizes. For each such trace set, the

execution times for the GFD and the GNR to detect synchronization failure are measured. The time is measured in milliseconds (ms).

7.2 Experiment Results From Application A, Zero Search

7.2.1 Description of Problem

Given a continuous function f having opposite signs at the end points of length L, locate a zero of f within a unit interval. That is, given a function f(x) and an interval [I1, I2] (I2 - I1 + 1 = L), f(I1) <0 and f(I2) >0, find [K,K+1] so that f(K) < 0 and f(K+1) > 0 where I1≤K ≤ I2 - 1.

7.2.2 Correct Algorithm

Let Nt be the number of threads working together on this problem. Initially, the length L is divided into Nm intervals. Both ends of an interval is checked for signs by one thread. After this first iteration, at least one thread will find opposite signs on the two ends of an interval. Let all threads work on this interval by dividing it into Nt (smaller) intervals and having each thread work on one interval. Repeat k= $\lceil \log_{Nt} L \rceil$ times. Opposite signs will be found for a unit interval.

A detailed algorithm for the case Nt=4 is presented next. Let l_i ($1 \le i \le k$) be the interval of the i^{th} iteration. Variable a and b are used to record the two ends of an interval with opposite signs found after each iteration. s1 to s8 are binary semaphores with initial value 1.

The initial length of intervals is $\lfloor L/N_t \rfloor$ (if $\lfloor L/N_t \rfloor \neq L/N_t$. The length for the last thread is $\lfloor L/N_t \rfloor$+reminder of L/N_t). Thread T5 is the coordinator for other four threads. It assigns new intervals to them.

```
T1                  T2                  T3                  T4                  T5

i1=0;               i2=0;               i3=0;               i4=0;               j=0;
while(i1<k)         while(i2<k)         while(i3<k)         while(i4<k)         while(j<k)
{                   {                   {                   {                   {
P(s1);              P(s2);              P(s3);              P(s4);              P(s5);
V(s5);              V(s6);              V(s7);              V(s8);              P(s6);
if two ends         if two ends         if two ends         if two ends         P(s7);
have opposite       have opposite       have opposite       have opposite       P(s8);
sign, record        sign, record        sign, record        sign, record        read [a,b] and
the ends in [a,b];  the ends in [a,b];  the ends in [a,b];  the ends in [a,b];  check if it is from
i1:= i1+1;          i2:= i2+1;          i3:= i3+1;          i4:= i4+1;          a unit interval
}                   }                   }                   }                   if yes, stop,
                                                                                otherwise assign
                                                                                new intervals for
                                                                                T1 to T4;
                                                                                V(s1);
                                                                                V(s2);
                                                                                V(s3);
                                                                                V(s4);
                                                                                j:=j+1;
                                                                                }
```

7.2.3 Algorithms with Errors

1. Error A_E1

An error is injected into the algorithm by inserting an additional V(s1) before V(s5) in thread T1. It is easy to see that the algorithm allows a successful execution but also contains synchronization failure.

2. Error A_E2

A different error is injected into the correct algorithm by replacing P(s5) with V(s5) in thread T5.

7.2.4 Speedup of GFD over GNR

TABLE 1. Speedup for Application A

	Nt	Ns	Nn	t_n (ms)	t_f(ms)	Speedup
A_E1	5	8	136	1480	10	148
A_E1	9	16	264	2970	10	297
A_E1	17	32	520	7770	10	777
A_E2	5	8	128	1100	10	110
A_E2	9	16	256	5350	10	530
A_E2	17	32	512	33600	10	3360

7.3 Experiment Results From Application B, L-U Decomposition

7.3.1 Description of Problem

Given A which is an N * N matrix, compute matrices L and U such that A=L*U, where L is a N by N matrix with all zero in the upper triangle and U is a N by N matrix with all zero in the lower triangle. The following is an example for N=3.

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 1 & 2 & 1 \\ 1 & 0 & -1 \end{bmatrix} = L * U = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 3 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 3 & 2 \\ 0 & -1 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

7.3.2 Correct Algorithm

We give an algorithm that uses four threads (Nt=4) to solve the L-U decomposition. The columns are statically allocated to the threads in an interleaved fashion: thread T1 works on column 1, 5, 9, ...; thread T2 on column 2, 6, 10, ...; thread T3 on column 3, 7, 11, ...; and thread T4 on column 4, 8, 12, ... Each thread processes its columns from left to right. A column is used to modify all columns to its right. Binary semaphores s1, s2, ... are used to coordinate the accesses to the matrix elements by different threads.

```
T1                  T2                  T3                  T4

i1=1;               i2=1;               i3=1;               i4=1;
V(s1);              P(s1);              P(s2);              P(s3);
V(s2);              V(s4);              P(s5);              P(s6);
V(s3);              V(s5);              V(s7);              P(s9);
                    V(s6);              V(s8);              V(s10);
                                        V(s9);              V(s11);
                                                            V(s12);
while (i1≤N)        while (i2≤N)        while (i3≤N)        while (i4≤N)
{                   {                   {                   {
i1=i1+4;            i2=i2+4;            i3=i3+4;            i4=i4+4;
P(s4);              P(s1);              P(s2);              P(s3);
P(s7);              P(s8);              P(s5);              P(s6);
P(s10);             P(s11);             P(s12);             P(s9);
V(s1);              V(s4);              V(s7);              V(s10);
V(s2);              V(s5);              V(s8);              V(s11);
V(s3);              V(s6);              V(s9);              V(s12);
}                   }                   }                   }
```

7.3.3 Algorithms with Errors

1. Error B_E1

An error is injected into thread T3 (the second thread to the last) by interchanging each subsequent V operation with the preceding P operation.

2. Error B_E2

An error is injected into thread T2 (the second thread) by interchanging each subsequent V operation witn the proceeding P operation.

7.3.4 Speedup of GFD over GNR

TABLE 2. Speedup for Application B

	Nt	Ns	Nn	t_n (ms)	t_f(ms)	Speedup
B_E1	3	6	251	1250	10	125
B_E1	4	12	372	20810	10	2081
B_E1	8	56	861	76400	10	7640

TABLE 2. Speedup for Application B

	Nt	Ns	Nn	t_n (ms)	t_1(ms)	Speedup
B_E2	3	6	122	1250	10	125
B_E2	4	12	372	28110	10	2811
B_E2	8	56	861	122530	10	12253

7.4 Experiment Results From Application C, Dining Philosophers

7.4.1 Description of Problem

The well-known Dining Philosophers (Tanenbaum 1992, chapter 2, section 2.3.1, pp.56-59) is used here.

7.4.2 Correct Algorithm

The algorithm described in (Tanenbaum 1992, chapter 2, section 2.3.1, pp.56-59) is adopted but modified so that it terminates by limiting each philosopher to repeat the 'think-eat' cycles a fixed number of times.

7.4.3 Algorithms with Errors

Let Ti, i =0, 1, ..., N-1, to represent the algorithm for philosopher i. The forks are numbered as 0 to N-1. Philosopher i has fork i as his left fork and fork (i+1) % N, where % is the modulo operation, as its right fork. We use N binary semaphores whose initial values are 1. If the operations of take_fork and put_fork are already guaranteed to be atomic, then there is no need to use the binary semaphores at all.

1. Error C_E1

A classical error is for each philosopher to take the left fork first and then the right one.

```
while (True)
{       think();
        P(si);
        take_fork(i);              ( take the left fork )
        V(si);
        P((s(i+1) % N));           ( take the right fork )
        take_fork((i+1) % N);
        V(s((i+1) % N));
        eat();
        P(si);
        put_fork(i);               ( put down the left fork )
        V(si);
        P(s((i+1) % N));
        put_fork((i+1) % N);       ( put down the right fork )
        V(s((i+1) % N));
}
```

2. Error C_E2

The above algorithm can be modified so that T4 (the 5^{th} philosopher) requests right forks first:

```
while (True)
{       think();
        P(s0);
        take_fork(0);       ( take the right fork )
        V(s0);
        P(s4);              ( take the left fork )
        take_fork(4);
        V(s4);
        eat();
        P(s4);
        put_fork(4);        ( put down the left fork )
        V(s4);
        P(s0);
        put_fork(0);        ( put down the right fork )
        V(s0);
}
```

This modified algorithm has no synchronization failure. But we introduce some error into this algorithm: V(s2) at the end of T1 (the 2^{nd} philosopher) and V(s3) at the end of T2 (the 3^{rd} philosopher) are interchanged.

7.4.4 Speedup of GFD over GNR

TABLE 3. Speedup for Application C

	Nt	Ns	Nn	t_n (ms)	t_f(ms)	Speedup
C_E1	8	8	520	220	30	7
C_E1	16	16	1040	740	100	7
C_E1	32	32	2080	2800	400	7
C_E2	5	5	45	500	10	50
C_E2	5	5	85	24600	10	2460
C_E2	8	8	72	115390	10	11539

Notice that trace sets of larger sizes such as (5, 5, 165) or (8, 8,136) for C_E2, The GNR takes more than an hour CPU time to run but the GFD only needs 10 ms.

7.5 Experiment Results From Application D, Shared Resources

7.5.1 Description of Problem

A collection of resources are shared mutually exclusively by a collection of threads. There is no other synchronization requirement among the threads, that is, a thread does not have to wait until another thread has done something. A thread can always go ahead and request for some resources. Each thread simply repeats the following: request for resources, make use of resources, and release resources.

7.5.2 Correct Algorithms

Consider the following algorithms for 8 threads that share 8 different numbers of resources of different types, protected by 8 binary semaphores, s1, ... , s8, initially all containing value 1:

```
T1: while(True) {P(s1) V(s1) P(s2) V(s2);}
T2: while(True) {P(s1) V(s1) P(s3) V(s3);}
T3: while(True) {P(s4) V(s4);}
T4: while(True) {P(s4) P(s3) V(s4) V(s3);}
T5: while(True) {P(s4) P(s5) V(s4) V(s5);}
T6: while(True) {P(s6) P(s7) V(s6) V(s7);}
T7: while(True) {P(s8) V(s8) P(s1) V(s1);}
T8: while(True) {P(s2) V(s2) P(s8) V(s8);}
```

As before, to ensure termination, we limit each thread to execute its while loop a finite number of times.

7.5.3 Algorithm with Errors

1. Error D_E1

A possible error may be introduced by interchanging P(s1) of T7 with P(s8) of T8 as shown next:

```
T7: while(True) {P(s8) V(s8) P(s8) V(s1);}
T8: while(True) {P(s2) V(s2) P(s1) V(s8);}
```

It is easy to see that V(s8) in T8 and the initially available V(s8) can collide and thus may cause a synchronization failure.

2. Error D_E2

A different error is obtained by interchanging V(s2) of T1 and P(s3) of T2 as shown next:

```
T1: while(True) {P(s1) V(s1) P(s2) P(s3);}
T2: while(True) {P(s1) V(s1) V(s2) V(s3);}
```

Similarly, V(s2) and V(s3) in T2 can collide and may cause synchronization failures.

7.5.4 Speedup of GFD over GNR

TABLE 4. Speedup for Application D

	Nt	Ns	Nn	t_n (ms)	t_l(ms)	Speedup
D_E1	8	8	148	110	10	11
D_E1	8	8	192	260	10	26
D_E1	8	8	580	1830	10	183
D_E2	8	8	128	140	10	14
D_E2	8	8	248	350	10	35
D_E2	8	8	488	1540	10	154

Notice that for this application, Local Failure Validation is a very effective technique. If running the GFD without Local Failure Validation, then t_n for every row will be more than 1,000 seconds.

7.6 Some Analysis of Experiment Results

In our experiment, the GFD detects the existence of failure behaviors efficiently: in all cases but one, it takes less than 10 ms (the recorded time is at multiples of 10 ms). For the error C_E1 of Application C it takes 30 ms for the problem size of (N_t=8, N_s=8, N_n=520), 100ms for problem size of (N_t=16, N_s=16, N_n=1040) and 400ms for problem size of (N_t=32, N_s=32, N_n=2080). In this case, when N_s and N_t each doubles, the detection time increases approximately by four times. We believe that the upper bound of the detection time for Error C_E1 of Application C is in the order of $O(N_t*N_s)$. Notice that even for the 10 ms cases, it is still necessary to scan through each individual trace set during the reduction phase, an activity that takes a time in the order of $O(N_n)$. Therefore, for all applica-

tions and errors we have experimented, the detection time is in the order of $O(N_n)$ or $O(N_s*N_t)$. The following are some more specific observations:

(1) Reductions and Reduction Heuristics are in general effective in reducing the behavior tree.

(2) Local Failure Validation is a very effective technique to reduce the behavior space to be searched, such as those in Application D, although it does not help in applications such as Application C which does not have local failure.

(3) 'Short-cuts' created by special Failure Rules are effective, especially when they are used together with Reductions and Reduction Heuristics.

(4) Partial Order Checking is an effective technique for behavior checking, when compared with the interleaving based approach, especially when the derived behaviors contains a lot of concurrencies.

Chapter 8 Future Work and Conclusion

8.1 On Memory Consistency Models

8.1.1 Future Works

(1) Link Consistency for parallel programs can also be extended to parallel programs that use Lock and Unlock. One can treat Unlock as V and Lock as P. More over, Link Consistency can be extended to allow parallel programs with both binary and counting semaphores.

We also need to study how to relax sequential program order between such an operation and another operation for parallel programs that include counting semaphores. The validity of our relaxations in Figure 24 has to be checked and there can be new relaxations that work only for counting semaphores. For example, since the notion of token collision is gone for counting semaphores, we conjecture that the relaxation V(a) $\xrightarrow{p}$ Q(b), where Q is either a P or a V operation, is always valid if semaphore a is a counting semaphore.

(2) We have assumed the absence of branching operations in program threads in the above discussion. With the presence of branching operations, our machine model has to be extended to include branching operation too. Further work is needed to adopt our machine model to include branching operations. The simplest way is to just apply our result to each contiguous piece of straight-line code segment.

(3) It is interesting to investigate the absence of symmetry between a V operation and a P operation in rule R3 and R4. Little is known on how to relax P $\xrightarrow{p}$ V' without other

explicit knowledge of a program.

In the long term, the following issues should be addressed:

(4) Debugging with Link Consistency. Similar to RMrc, RMls assumes that a parallel program is data race free on BM. In addition RMls also assumes that a parallel program does not contain synchronization failure on BM. In real life, a program may violate these assumptions. Such violations have to be taken care of by a RM (Adve and Hill 1991) before the execution results can be treated as valid.

(5) We have compared, via some examples, Link Consistency (with or without scope) with existing consistency models such as Release Consistency. It would be instructive to actually run these examples on a real machine, such as the work done for Release Consistency or Processor Consistency (Gharachorloo, Gupta, and Hennessy 1991, Yuanyuan Zhou et al. 1997, Ranganathan, Pai and Adve 1997), and to analyze the performance gains of Link Consistency. Our experiences on working on memory consistency suggests that the problem to relax sequential program orders between synchronization operations to tolerate long memory latency still calls for more investigation, especially on the real functionality of synchronization operations in a parallel program.

8.1.2 Conclusions

We go beyond *Release Consistency* in exploiting knowledge of actual precedence relationship among synchronization operations and others. This is perhaps as far as one should go in relaxing access order in a sequential thread. If we wish to accomplish more

overlap and thus suffer less from long memory latency, it seems the only avenue left is to relax *Sequential Consistency*. Unfortunately, such existing candidates, such as *Processor Consistency* or *Slow Memory* (Hutto and Ahamad 1990) in , is difficult to reason with in programming. The true challenge is to find a program model that is general yet simple enough for a sequential consistency based programmer to express what the programmer requires and to analyze the program easily. Until then, relaxation of operation ordering will be limited by the abstract program semantics involving *Sequential Consistency*.

8.2 On Synchronization Failure Detection

8.2.1 Some Conjecture on Binary Trace Sets

We conjecture that the following general form of failure rule is valid:

> A trace set T contains synchronization failure if it contains a thread trace with two PaAPa where A is in Δ^* and $a \notin \text{sem}(\Delta^*)$.

For example, if a trace set T contains PaPbPa or PaPbVbPa as one of its thread trace, then T contains synchronization failure. We have proved the case when A is empty.

8.2.2 Synchronization Failure Detection for Trace Set with Control Variables

8.2.2.1 Definition of Trace Set with Control Variable Accesses (CVA)

A trace set with CVA is a set of local traces, each being a sequence of synchronization events and read or write events to control variables. Suppose T is such a trace set. We use Ts to represent the trace set obtained from T by removing all CVA, and Tx to represent the

trace set obtained from T by removing all operations that do not operate on the control variable X.

For example, consider the following program with two threads and one control variable:

```
Initially control variable X=0.
BINARY SEMAPHORE a=0, b=1.

T1                                    T2

(1) Va1;                              (6) Pb2;
(2) Pa1;                              (7) R(X);
(3) Pb1;                              (8) Vb2;
(4) W(X,1);                           (9) if X=1
(5) Vb1;                                  then
(12)HALT                              (10)  Va2;
                                      (11)  Pa2;
                                      (13)HALT
```

This program has more than one execution. One such execution will have R(X, 0), that is R(X) returns 0, in T2. Because of that, Va2 and Pa2 will not be executed. One trace set of such execution is the following:

T=<{Vb}, {Va1 Pa1 Pb1 W(X, 1) Vb1, Pb2 R(X, 0) Vb2}>

The projection of T on to synchronization events is Ts=<{Vb}, {Va1 Pa1 Pb1 Vb1, Pb2 Vb2}>, and the projection of T on to the control variable X Tx is Tx=<{ X=0 }, {W(X, 1), R(X, 0)}>.

Another execution will have R(X, 1), that is R(X) returns 1, in T2, and because of that,Va2 and Pa2 will be executed. One trace set of such execution is the following:

T'=<{Vb}, {Va1 Pa1 Pb1 W(X,1) Vb1, Pb2 R(X,1) Vb2 Va2 Pa2}>

The projection of T on to synchronization events is Ts'=<{Vb}, {Va1 Pa1 Pb1 Vb1, Pb2 Vb2 Va2 Pa2}>, and the projection of T on to the control variable X Tx isTx'=<{X=0}, {W(X,1), R(X,1)}>.

8.2.2.2 An Example of Invalid Failure Behavior

We can apply the GFD to detect the synchronization failure behavior of Ts. Not every behavior of Ts is valid with respect to T, however. For example, the CVA in T may prevent some V from enabling some P even though it is possible with Ts.

Therefore, it may not be accurate if we only apply the current GFD to Ts and reports whatever the GFD returns as an indication of the existence or absence of synchronization failures in T.

Consider the example trace set T' in the above section. Ts' has a failure behavior B1 if the initial Vb enables Pb2 and then both Va1 and Va2 enable Pa1 as follows.

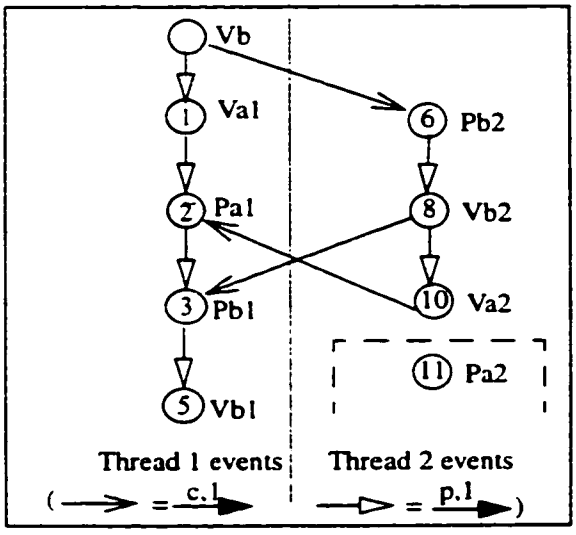


Figure 50 A failure behavior of Ts'.

Now if we add the omitted CVA to the above failure behavior we will get:

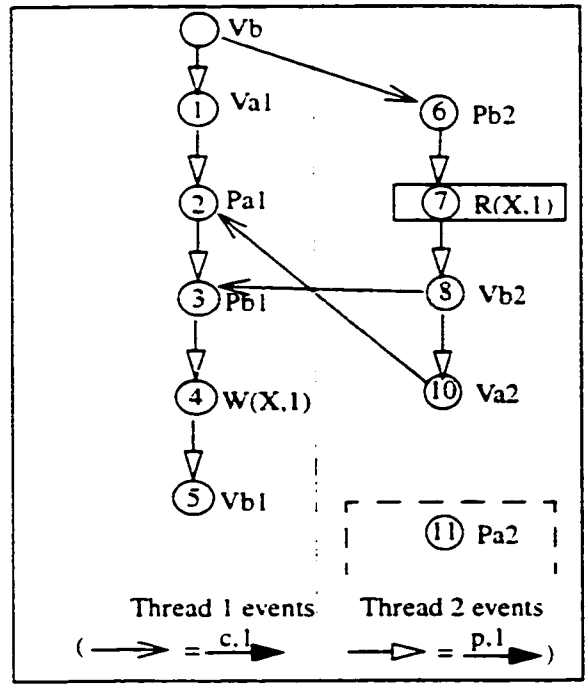


Figure 51 A failure behavior of Ts' that is invalid with respect to T' due to CVA.

It is easy to see that in the above figure R(X,1) is impossible since the initial value of X is 0 and the W(X,1) is ordered after R(X). In such a case, we say the failure behavior B1 of Ts is *invalid* with respect to T. In general one may or may not be able to augment a behavior of such Ts with the missing CVA and make it a 'valid' derived behavior of T. Further work has to be done to check which rules or algorithms developed in this thesis are valid.

8.2.2.3 Complexity of Synchronization Failure Detection

Theorem 6.5 shows that if a trace set contains only one binary semaphore, then presence of synchronization failure can be decided in polynomial time. When such a simple trace

set also contains CVA, the synchronization failure detection problem becomes complicated. The synchronization failure detection problem for a trace set that contains CVA and only one binary semaphore is NP complete as is proved in Appendix D.

To verify whether a behavior of Ts corresponds to some valid behavior of T, there are two things to check: whether the semaphore semantics is satisfied and whether the access semantics is satisfied. Further investigation is needed.

8.2.3 Synchronization Failure Detection for Trace Set with Counting Semaphores

The study of synchronization failure detection so far has been restricted to binary trace sets. A trace set may also contain events on counting semaphores. We call such a trace set *General* trace sets. Some results that are invalid for binary trace sets may become valid for general trace sets; while the opposite is true for other results.

For example, Theorem 6.1 is not valid for general semaphores. However, our preliminary work shows that the following rules, invalid for binary semaphores, are valid for counting semaphores:

(1) V Permutation Rule: $VaVb \stackrel{f}{=} VbVa$ if semaphore a and b are counting semaphores, and

(2) Reduction Rule: $A_1A_2A_3 \stackrel{f}{=} A_1$ where A_1 and A_3 are all-P sequences, $A_1=A_3$, A_2 is a matching V sequence of A_1, and $sem(A_1)$ are all counting semaphores.

According to the above reduction rule, $PaVaPa \stackrel{f}{=} Pa$ if a is a counting semaphore. This

result is more general than that of Theorem 6.7.

8.2.4 Conclusions

Our experiment result shows that for a binary trace set that contains synchronization failure, the algorithm GFD can efficiently determine the existence of such failure in time approximately proportional to the size of a trace set, measured by N_n or N_t*N_s. Our algorithm tries to unveil the synchronization failure as early as possible if there is any. It can be a valuable aid for parallel program development. For a trace set that contains no synchronization failure, our algorithm still works, but may take longer time to terminate. Techniques such as reductions, on-demand unfolding of local behavior trees, pruning, and partial order checking are still useful for synchronization failure free trace sets.

References

Adve, S.V. and M.D. Hill. Weak Ordering - A New Definition. In *17th International Symposium on Computer Architecture*, pp. 2-14, (May 1990).

_____. A Unified Formalization of Four Shared-Memory Models. In *IEEE Transactions on Parallel and Distributed Systems* **4**(6), pp. 613-624, (June 1993).

Adve, S.V., M.D. Hill, B.P. Miller and R.H.B. Netzer. Detecting Data Races on Weak Memory Systems. In *18th International Symposium on Computer Architecture*, pp. 234-243, (May 1991).

Adve, S.V. and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. In *Computer*, **29**(12), pp. 66-76, (December 1996).

Bershad, B.N., M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *IEEE COMPCON '93 Conference*, pp. 528-537 (February 1993).

Blume, W., R. Eigenmann, K. Faigin, et al. Polaris: Improving the Effectiveness of Parallelizing Compilers. In *7th International Workshop in Languages and Compilers for Parallel Computing (LCPC'94)*, pp. 141-154, (August 1994).

Blume, W., R. Eigenmann, K. Faigin, et al. Restructuring Programs for High-Speed Computers with Polaris. In *ICPP workshop on Challenges for Parallel Processing*, pp. 149-161, (August 1996).

Blume, W., R. Doallo, R. Eigenmann, et al. Parallel programming with Polaris. In *COMPUTER* **29**(12), pp. 78-82, (December 1996).

Chandy, K. M., J. Misra and L. M. Haas. Distributed Deadlock Detection. In *ACM Transactions on Computer Systems* **1**(2), pp. 144-156, (May 1983).

Corbett, James C. Evaluating Deadlock Detection Methods for Concurrent Software. In *IEEE Transactions on Software Engineering* **22**(3), pp. 161-180, (March 1996)

Dekker, T. Two Process Mutual Exclusion Algorithm. See Dijsttra, E.W. Co-operating sequential processes, in *Programming Languages*, edited by F. Genuys, Academic Press, New York, pp. 43-112, (1965).

Diniz, Pedro and Martin Rinard. Synchronization Transformations for Parallel Computing. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'97), pp. 187-200, (January 1997).

Dubois, M., C. Scheuich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *13rd International Symposium on Computer Architecture*, pp. 434-442, (June1986).

Duri, S., U. Buy, R. Davarapalli, and S.M. Shatz. Using State Space Reduction Methods for Deadlock Analysis. *1993 International Symposium on SoftwareTesting and Analysis*, as *ACM SIGSOFT Software Engineering Notes* **18**(3), pp. 51-60, (July 1993).

Feitelson, D.G. Deadlock detection without wait-for graphs. In *Parallel Computing* **17**(12), pp. 1377-1383, (December 1991).

Fu, T. A General Consistency Model for Parallel Programming andTolerating Memory Latency of Scalable Multiprocessor. *Doctoral Thesis Proposal*, Department of Computer Science, Concordia University, Montreal, Canada (February 1992).

Garey, M.R. and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, (1979).

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *17th International Symposium on Computer Architecture*, pp. 15-26, (May 1990).

Gharachorloo, K., A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.245-257, (1991).

Gharachorloo, K., S. V. Adve, A. Gupta, J. Hennessy and M. D. Hill. Programming for Different Memory Consistency Models. In *Journal of Parallel and Distributed Computing* **15**(4), pp. 399-407, (August 1992).

Gibbons, P. B. and E. Korach. The Complexity of Sequential Consistency. In *4th IEEE Symposium on Parallel and Distributed Processing*, pp. 317-325, (December 1992).

Gibbons, P. B. and E. Korach. On Testing Cache-Coherent Shared Memories. In *6th Annual Symposium on Parallel Algorithms and Architectures*, pp. 177-188, (June 1994).

Gibbons, P.B., M.Merritt, and K.Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *1991 ACM Symposium on Parallel Architectures and Algorithms*, pp. 292-303, (July 1991).

Godefroid, P. and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Saftey Properties. In *3th International Workshop on Computer Aided Verification (CAV'91)*, pp. 332-342, (July 991).

Gonzalez, D. M. and J.R. Garitagoitia. A model for deadlock detection based on automata and languages theory. *Computers and Mathematics with Applications* **25**(6) pp. 47-55, (March 1993).

Goodman, J.R. Cache Consistency and Sequential Consistency. *Technical Report 1006*, Dept. of Computer Science, University of Wisconsin, Madison, Wisconsin, USA, (February 1989).

Hall, M. W., B. R. Murphy, S. P. Amarasinghe, S. Liao, M. S. Lam. Interprocedural Analysis for Parallelization". In *8th International Workshop on Languages and Compilers for Parallel Computing (LCPC'95)*, pp. 61-80, (August 1995).

Hall, M. W., J. M. Anderson, S. P. Amarasinghe, et al. Maximizing Multiprocessor Performance with the SUIF Ccompiler. In *COMPUTER* **29**(12), pp. 84-89, (December 1996).

Helmbold, D. P., C. E. McDowell, and J. Z. Wang. Determining Possible Event Orders by Analyzing Sequential Traces. In *IEEE Transactions on Parallel and*

Distributed Systems, **4**(7), pp. 827-839, (July 1993).

Hoare, C.A.R. An axiomatic basis for computer programming. In *Communications of ACM*, **12**(10), pp. 576-583 (October 1969).

_____. Parallel programming: an axiomatic approach. In *Computer Languages*, **1**(2), pp. 151-60 (1975).

Hutto, P.W. and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *10th International Conference on Distributed Computing Systems*, pp. 302-311, (1990).

Iftode, L., J. P. Singh and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 277-287, (June 1996).

Krishnamurthy, Arvind and Katherine Yelick. Optimizing Parallel SPMD Programs. In *7th International Workshop on Languages and Compilers for Parallel Computing*,
pp. 331-345, (August 1994).

Krishnamurthy, Arvind and Katherine Yelick. Optimizing Parallel Programs with Explicit Synchronization. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pp. 196-204, (June 1995).

Kuck, D. J., R. H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *8th ACM Symposium on Principles of Programming Languages*, pp. 207-218, (January 1981).

Lamport, L. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. In *IEEE Transactions on Computers* **C-28**(9), pp. 690-691, (September 1979).

Li, H. F. and T. Fu. Implementing Sequential Consistency More Efficiently. *Technical Report*, Concordia University, Montreal, Quebec, Canada, (July 1992)

Lim, A. W. and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transformation. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pp. 201-214, (January 1997).

Lipton, R. The Reachability Problem Requires Exponential Space. *Research Report 62*, Department of Computer Science, Yale University, New Haven, Connecticut, USA, (January 1976).

Masticola, S. P. and B. G. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *ACM SIGPLAN Notices* **26**(12), pp. 97-107, (December 1991).

Mayr, E. An Algorithm for the General Petri Net Reachability Problem. In *SIAM Journal of Computing*, **13**, pp. 441-460, (1984).

McMillan, K. Using Unfolding to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In *4th International Conference on Computer Aided Verification (CAV'92)*, pp. 164-173, (June 29-July 1, 1992).

Navarro, A.G., Y. Paek, E.L. Zapata, D. Padua. Compiler Techniques for Effective Communication on Distributed-memory multiprocessors. In *the 1997*

International Conference on Parallel Processing (ICPP '97), pp. 74-77, (August 1997).

Netzer, R. H. B. and Barton P. Miller. Detecting Data Races in Parallel Program Executions. In *Languages and Compilers for Parallel Computing*, edited by D. Gelernter, T. G ross, A. Nicolau, and D. Padau, The MIT Press, pp. 109-129 (1990).

Netzer, R. H. B. and Barton P. Miller. Improving the Accuracy of Data Race Detection. In *3rd ACM Symposium on Principles and Practice of Parallel Programming*, pp. 133-144, (April 1991).

Perkovic, Dejian and Peter J. Keleher. Online Data-Race Detection via Coherency Guarantees. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pp. 47-57, (October 1996).

Peterson, G. L. Myths about the mutual exclusion problem. In *Information Processing Letters*, **12**, pp. 115-116, (June 1981).

Peterson, J. L. *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, (1981).

Pratt, V. R. Modelling concurrency with partial orders. *International Journal of Parallel Programming* **15**(1), pp. 710-720, (February 1986).

Probst, D.K. and H. F. Li. Using Partial-Order Semantics to Avoid the State Explosion Problem in Asynchronous Systems. In *2nd International Conference on Computer Aided Verification (CAV '90)*, pp. 146-155, (June 1990).

_____. Partial-Order Model Checking: A Guide for the Perplexed. In *3rd International Conference on Computer Aided Verification (CAV '91)*, pp. 322-331, (July 1991).

_____. Verifying Timed Behavior Automata with Input/Output Critical Races. In *5th International Conference on Computer Aided Verification (CAV '93)*, pp. 424-437, (June 28-July 1, 1993).

Ranganathan, P., V.S. Pai and S.V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow Performance Gap between Memory Consistency Models. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pp 199-210, (June 1997).

Rontogiannis, P., G. Pavlides, and A. Levy. Distributed algorithm for communication deadlock detection. In *Information and Software Technology*, **33**(7), pp. 483-488, (September 1991).

Shasha, D. and Marc Snir. Efficient and Correct Execution of Parallel Programs That Share Memory. In *ACM Transactions on Programming Languages and Systems*, **10**(2), pp. 282- 312, (April 1988).

Singla, Aman, Umakishore Ramachandran and Jessica Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pp 211-220, (June 1997).

Tanenbaum, Andrew S. *Modern Operating Systems*, 1st edition, Pretice Hall, Englewood Cliffs, NJ, USA, (1992).

Stocks, P., and D. Carrington. Test Template Framework: A specification-based testing

case study. In *1993 International Symposium on Software Testing and Analysis, ACM SIGSOFT Software Engineering Notes* **18**(3), pp. 19-27, (July 1993).

Tu, S., S. M. Shatz, and T. Murata. Applying Petri net reduction to support Ada-tasking deadlock detection Proceedings. In *10th International Conference on Distributed Computing Systems*, pp. 96-103, (1990).

Zhou, Yuanyuan, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 193-205, (June 1997).

Appendix A Proofs of Theorems For Link Consistency

A.1 Proof Strategy

Our approach to prove that a machine M1 is a sequentially consistent for programs in Pdsf is to show that for any maximal execution E1 on machine M1, we can *construct* another execution E2 on a machine M2, which is known to be sequentially consistent for programs from Pdsf, such that result(E1) = result(E2).

Definition of RMlc_0: RMlc_0 is the same as BM except that it uses relaxation R0 to relax sequential program orders among data operations; it does not relax sequential program orders between a data operation and a synchronization operation.

Definition of RMlc_1: RMlc_1 is the same as RMlc_0 except that it also uses relaxations R1 and R2 to relax sequential program orders among some data operations and synchronization operations.

Definition of RMlc_2: RMlc_2 is the same as RMlc_1 except that it also uses relaxation R4 to relax sequential program order among synchronization operations.

Knowing that BM is sequentially consistent for programs in Pdsf, we first use BM to prove that RMlc_0 is sequentially consistent. We then use RMlc_0 to prove that RMlc_1 is sequentially consistent and so on. Figure 52 shows the major steps of ours proofs.

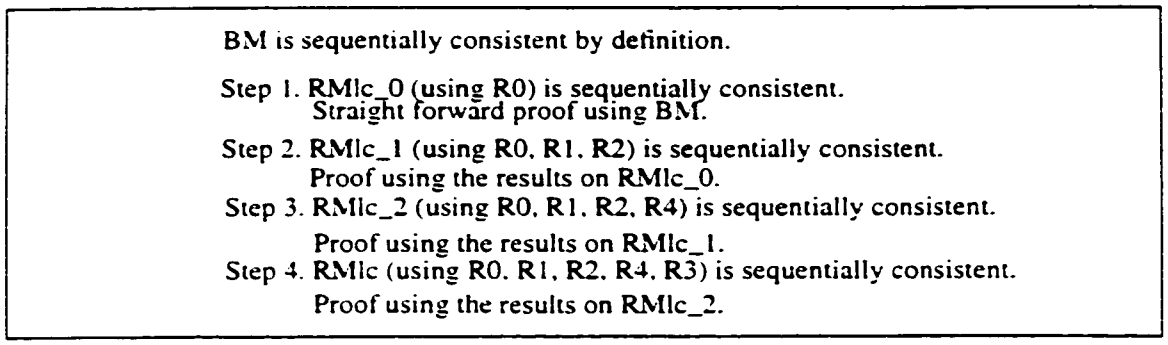


Figure 52 Major steps of proofs for RMlc.

A.2 A Lemma

We introduce a lemma that will be used in the later proofs.

Lemma A Let E1 be an execution of a program on some machine. If E1 contains a data race, race(ev1, ev2, E1), then there exists an execution E2 of that program on that machine such that ev1 occurs immediately before ev2 (or ev2 occurs immediately before ev1) in E2.

Proof: By construction. If there exist data event ev1 and ev2 from two different program threads and ev1 occurs immediately before ev2 in E1, then we are done.

Now suppose this is not true. Since race(ev1, ev2, E1) is true, there is no path_s(ev1, ev2) in PO(E1). Here are the steps to construct a PO(E2) for some execution E2 from PO(E1). 'before' refers to program order in a thread trace.

(i) Let po be a partial order, initially containing ev2 and all events before it, together with their program order.

(ii) For every P event that is newly added to po, add the V that enables it and all

events that are before the V, if they are not already in po.

(iii) Repeat (ii) and (iii) until every P in po is enabled. This is allowed since every P is enabled in PO(E1). Add $\xrightarrow{p.1}$ to po according to program order and $\xrightarrow{c.1}$ according to PO(E1).

The following is true for po: (1) For any data event ev3 that is in conflict with ev 2 and from a thread different from that of ev2, path_s(ev3, ev2) is true in po, (2) if path(ev, ev') in po, then there is path(ev, ev') in PO(E1), and (3) ev1, or any event that is after ev1, must not be in po. Otherwise path_s(ev1,ev2) is true in po and hence in PO(E1), which contradicts the existence of race(ev1, ev2, E1).

(iv) Similar to (i), (ii) and (iii), add ev1 and all events before it into po.

(v) At last, add ev1 $\xrightarrow{c.1}$ ev2 (or ev2 $\xrightarrow{c.1}$ ev1) into po.

From its construction, po satisfies the conditions of Definition 3.20. Let E2 be the projection of E1 onto events in po. Remove ev1 from E2 and insert it immediately before ev2 in E2. This is possible: there is no event after ev1 and the execution of ev1, therefore, ev1 can be postponed to the last. E2 contains data race too. QED.

Example A1. Consider the program in Figure 53 and a PO(E1) from some possible execution E1 in the top left of Figure 54. The intermediate partial order po after each step is also shown. The final po is given in the bottom right of the same figure. E2 can be derived easily from po such that PO(E2) = po.

```
T1                  T2                  T3
(1) V(a);           (6)P(b);            (10) P(c);
(2) P(a);           (7)X:= ...;(ev1)    (11) V(a);
(3) V(b);           (8)V(a);            (12) P(a);
(4) V(c);           (9)HALT             (13) ... := X;(ev2)
(5) HALT                                (14) HALT
```

Figure 53 A program with data race

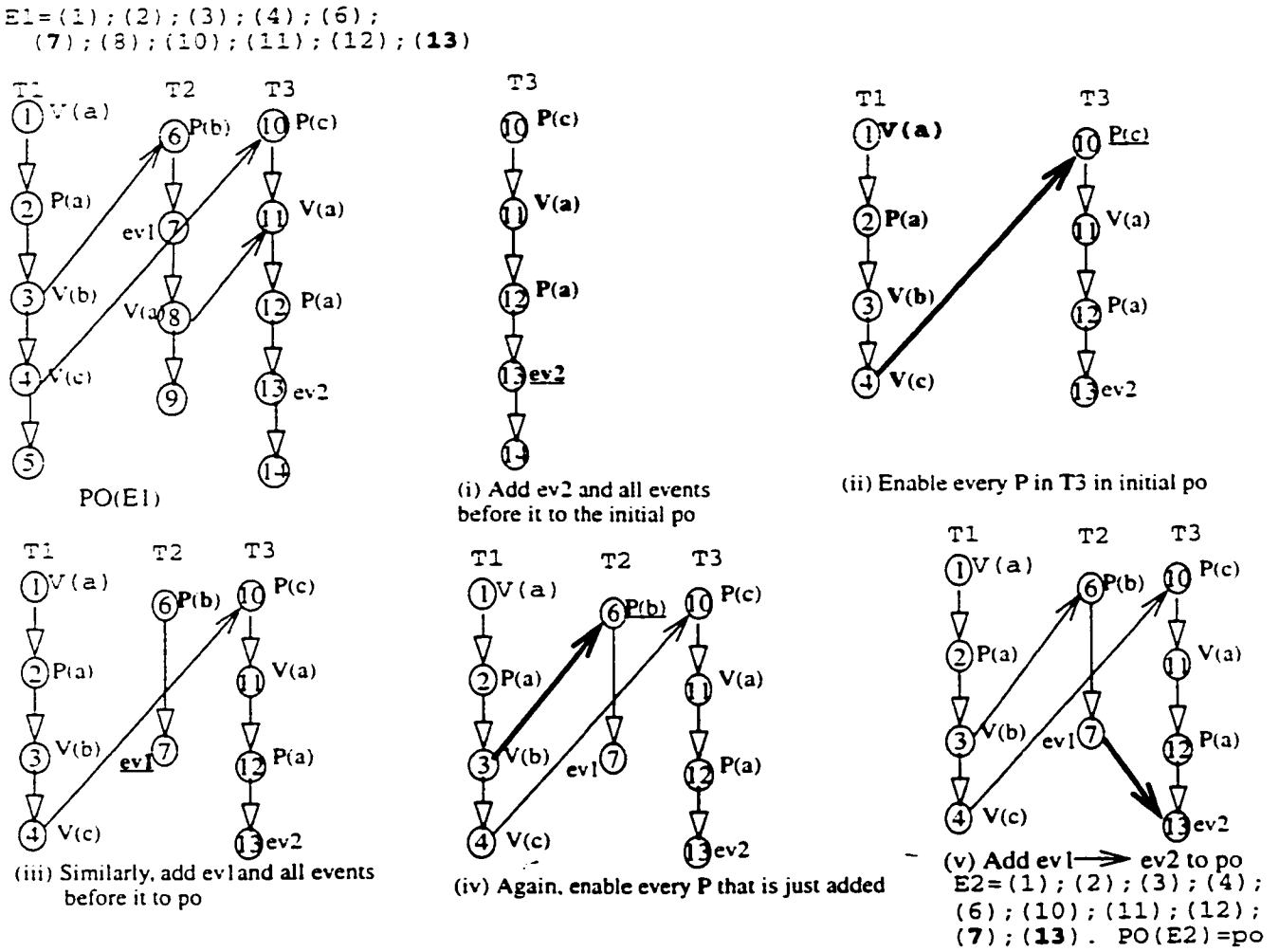


Figure 54 A PO(E1) with data race and a derived PO(E2)

A.3 Proof of Results on RMlc_0

Lemma A1 Any program from Pdsf is data race free on RMlc_0.

Proof: Suppose there is an execution E1 with data race on RMlc_0. According to Lemma A, E1 contains D1;D2 (D1 followed immediately by D2). RMlc_0 only relaxes program orders between two non-conflicting data operations, say it relaxes $D \xrightarrow{c.l} D'$. We could construct an execution E2 on BM from E1 as follows: (i) each time RMlc_0 executes D' before such D, BM executes D and then D', (ii) each time RMlc_0 executes such D at a later time, BM does nothing, and (iii) in any other case, both machines behave the same way. This is allowed since such D is non-blocking operation. Since E1 contains D1;D2, E2 will contain either D1;D2 or D1;D;D2 (D1 and D2 separated only by data events only). In either case, we see a data race in E2 on BM. A contradiction. QED.

Lemma A2 If a semaphore in a program from Pdsf is TCF on BM, then it is TCF on RMlc_0 too.

Proof: Suppose there is an execution E1 with token collision on RMlc_0. Hence PO(E1) contains $V1 \xrightarrow{c.l} P$ and $V2 \xrightarrow{c.l} P$. The construction of E2 in the preceding proof is an execution on BM that has token collision, which is a contradiction. Thus the claim. QED.

Lemma A3 Any program from Pdsf is synchronization failure free on RMlc_0.

Proof: Suppose E1 is an execution with synchronization failure on RMlc_0. This means that at least one P operation of the program is not in E1. RMlc_0 only relaxes the program order between two non-conflicting data operations. Given E1, we can construct, same as in Lemma A1, an execution E2 on BM which has the same

sequence of synchronization operations as E1. E2 is an execution on BM and does not contain those P either. Therefore E2 contains token collision on BM, a contradiction. QED.

Theorem A1 RMlc_0 is sequentially consistent for programs in Pdsf.

Proof: We show that for an execution E1 on RMlc_0, there is another execution E2 on BM such that result(E1) = result(E2). As we point out in the proof of Lemma A1, given E1, we can construct another execution E2 which is similar to E1 except that the program order of two non-conflicting data events are not relaxed. E2 is an execution on BM. Since both E1 and E2 are data race free, consider the projection of PO(E1) and PO(E2) on a same data variable. If D $\longrightarrow$ D' is in result(E1), then it must exist a path_s(D1,D2) in both E1 and E2 or they are program ordered. Therefore result(E2) = result(E1). QED.

A.4 Proof of Results on RMlc_1

Lemma A4 Any program from Pdsf is data race free on RMlc_1.

Proof: Suppose E is an execution on RMlc_1 which contains data race between D1 and D2. According to Lemma A, E1 contains D1;D2 (D1 followed immediately by D2). RMlc_1 only relaxes program orders between a data operation and a synchronization operation (on top of RMlc_0), say it relaxes D $\xrightarrow{c.l}$ P and V $\xrightarrow{c.l}$ D'. We could construct an execution E2 on RMlc_0 from E1 as follows: (i) each time RMlc_1 executes P (D') before such D (V), RMlc_0 executes D (V)

and then P (D'); (ii) each time RMlc_1 executes such D or V at a later time, RMlc_0 does nothing, and (iii) in any other case, both machines behave the same way. This is allowed since such D (V) is non-blocking operation. Since E1 contains D1;D2, E2 will contain one of (D1;D2), (D1;V;D2), (D1;P;V;D2) or (D2;P;D1). The last scenario occurs when the program contains D2 $\xrightarrow{p}$ P and RMlc_1 executes P before D2, followed by D1 and D2. In such a case, during the construction of E2, RMlc_0 executes D2 followed by P (using (i) above) and then D1. In either case, we see a data race in E2 on RMlc_0. A contradiction. QED.

Lemma A5 If a semaphore in a program from Pdsf is TCF on RMlc_0, then it is TCF on RMlc_1 too.

Proof: Suppose there is an execution E1 with token collision on RMlc_1. PO(E1) contains V1 $\xrightarrow{c.1}$ P and V2 $\xrightarrow{c.1}$ P. RMlc_1 only relaxes the program order between a data operation and a synchronization operation (on top of RMlc_0). Given E1, we can also construct an execution E2 on RMlc_0 which has the same sequence of synchronization operations as E1. E2 is an execution on RMlc_0 and PO(E2) contains V1 $\xrightarrow{c.1}$ P and V2 $\xrightarrow{c.1}$ P too. Therefore E2 contains token collision on RMlc_0, a contradiction. QED.

Lemma A6 Any program from Pdsf is synchronization failure free on RMlc_1.

Proof: RMlc_1 only relaxes the program order between a data operation and a synchronization operation (on top of RMlc_0) and progress is dependent on only synchronization operations. As in the proof of Lemma A5, given an execution E1

on RMlc_1, we could construct another execution E2 on RMlc_0. E2 has the same sequence of synchronization operations as E1. Hence synchronization failure freeness in E2 requires synchronization failure freeness in E1 (E1 and E2 differ only in the arrangement of data operations, where E2 preserves program orders between data and synchronization operations). QED.

Theorem A2 RMlc_1 is sequentially consistent for programs in Pdsf.

Proof: We show that for an execution E1 on RMlc_1, there is another execution E2 on RMlc_0 such that result(E1) = result(E2). This is proved by construction. As in the proof of Lemma A4, given E1, we can construct another execution E2 on RMlc_0. Notice that data evens will be constructed by (iii) only. Since both E1 and E2 are data race free, consider the projection of PO(E1) and PO(E2) on a same data variable. If D $\longrightarrow$ D' is in result(E1), then it must exist a path_s(D1,D2) in both E1 and E2 or they are program ordered. Therefore result(E2) = result(E1). QED.

A.5 Proof of Results on RMlc_2

Lemma A7 Any program from Pdsf is data race free on RMlc_2.

Proof: Suppose E is an execution on RMlc_2 which contains data race between D1 and D2. According to Lemma A, E1 contains D1;D2 (D1 followed immediately by D2). RMlc_2 only relaxes program orders between a Ve (exclusive producer) and a synchronization operation (on top of RMlc_1), say it relaxes Ve $\xrightarrow{c,l}$ Q. We

could construct an execution E2 on RMlc_1 from E1 as follows: (i) each time RMlc_2 executes Q before such Ve, RMlc_1 executes Ve and then Q; (ii) each time RMlc_2 executes such Ve at a later time, RMlc_1 does nothing; and (iii) in any other case, both machines behave the same way. This is allowed since such Ve is non-blocking operation. E1 contains D1:D2, according to (iii), E2 will contain D1:D2 too. Therefore, we see a data race in E2 on RMlc_1. A contradiction. QED.

Lemma A8 If a semaphore in a program from Pdsf is TCF on RMlc_1, then it is TCF on RMlc_2 too.

Proof: According to the definition of Ve, its semaphore, say a, is TCF. We prove by contradiction. Suppose there is an execution E1 with token collision on semaphore a on RMlc_2 and a is TCF on RMlc_1. As in the proof of Lemma A7, given E1, we could construct an execution E2 on RMlc_1. We show a has token collision in E2, which contradicts Lemma A5. Let E1' and E2' denote the executions which are partially constructed from E1 for RMlc_2 and RMlc_1 respectively. After an application of (i) for semaphore a, a token for semaphore a will be available in E2' but not in E1'; and subsequently, *only* after the application of (ii) for semaphore a, a token for semaphore a will also become available in E1' (it is impossible to apply (iii) to execute some V(a) since that will cause a token collision in E2', contradicting to our assumption). But from the assumption, at some application of (iii), there will be a token collision for

semaphore a in E1'. That implies that there is already a token available for semaphore a in E1' just before the application of (iii) and hence available in E2' too. Therefore there will be a token collision for semaphore a in E2', a contradiction. Notice that the token collision for semaphore a could not happen at some application of (i) or (ii): (i) is impossible since no V(a) is executed and (ii) is impossible since prior to that no token for semaphore a is available in E1'. QED.

Lemma A9 Any program from Pdsf is synchronization failure free on RMlc_2.

Proof: We prove by construction. Given an execution E1 with synchronization failure on RMlc_2, we could construct an execution E2 on RMlc_1 same as in the proof of Lemma A7. We show E2 contains a synchronization failure too, which contradicts Lemma A6. Let E1' and E2' denote the same as in the preceding proof. After an application of (i) for Ve, a token for the semaphore of the Ve will be available in E2' but not in E1'; and subsequently, *only* after the application of (ii) for the Ve, a token for its semaphore will also become available in E1'. Therefore, after each application of (i), (ii) or (iii), either E1' and E2' contain the same available tokens or E2' contains some extra tokens available for semaphores of Ve operations only. By assumption, E1' will become a synchronization failure, which means RMlc_2 has executed *every* possible V using (ii) or not, and contains no available token for the next P operation in the remaining E1. Since (ii) is not further applicable, E2' will contain no extra tokens available for

semaphore of those Ve. that is, both E1' and E2' will contain the same set of available tokens. Therefore E2' becomes a synchronization failure too, a contradiction. Thus. such E1 cannot exist and RMlc_2 cannot fail. QED.

Theorem A3 RMlc_2 is sequentially consistent for programs in Pdsf.

Proof: The proof is similar to that for Theorem A2. We show that for an execution E1 on RMlc_2, there is another execution E2 on RMlc_1 such that result(E1) = result(E2). This is proved by construction. As in the proof of Lemma A7, given E1. we can construct another execution E2 on RMlc_1. Notice that data evens will be constructed by (iii) only. Since both E1 and E2 are data race free, consider the projection of PO(E1) and PO(E2) on a same data variable. If $D \longrightarrow D'$ is in result(E1), then it must exist a path_s(D1,D2) in both E1 and E2 or they are program ordered. Therefore result(E2) = result(E1). QED.

A.6 Proof of Result on RMlc

Lemma A10 Any program from Pdsf is data race free on RMlc (Theorem 4.2).

Proof: Suppose E is an execution on RMlc which contains data race between D1 and D2. According to Lemma A, E1 contains D1;D2 (D1 followed immediately by D2). RMlc only relaxes program orders between a synchronization operation Q and a Pe (exclusive consumer), say it relaxes $Q \xrightarrow{c,l} Pe$. We could construct an execution E2 on RMlc_2 from E1 as follows: (i) each time RMlc executes Pe before such Q, RMlc_2 does nothing; (ii) each time RMlc executes such Q at a

later time, RMlc_2 executes Q and then Pe; and (iii) in any other case, both machines behave the same way. This is allowed since such Pe is an exclusive consumer (there is no P on the same semaphore in other threads): the token that is available for the semaphore of Pe at (ii) continues to be available until this Pe is executed. E1 contains D1:D2, according to (iii), E2 will contain D1:D2 too. Therefore, we see a data race in E2 on RMlc_2. A contradiction. QED.

Lemma A11 If a semaphore in a program from Pdsf is TCF on RMlc_2, then it is TCF on RMlc too (Theorem 4.3).

Proof: According to the definition of Pe, its semaphore, say a, is TCF. We prove by contradiction. Suppose there is an execution E1 with token collision on semaphore a on RMlc and a is TCF on RMlc_2. We could construct an execution E2 on RMlc_2 that is same as in the proof of Lemma A10. We show semaphore a has token collision in E2, which contradicts Lemma A8. Let E1' and E2' denote the executions which are partially constructed from E1 for RMlc and RMlc_2 respectively. After an application of (i) for semaphore a, a token for semaphore a will be available in E2' but not in E1'. Subsequently, *no* V(a) or some other P(a) will be executed by RMlc until after the corresponding Q and the Pe(a) are also executed by RMlc_2 according to (ii). This is because a token for semaphore a is still available in E2', and to execute another V(a) will cause a token collision in E2'. But from the assumption, at some application of (iii), there will be a token collision for semaphore a in E1'. That implies that there is already a token

available for semaphore a in E1' just before the application of (iii) and RMlc is to execute a V(a). This contradicts what we have pointed out above. Therefore E1' must be TCF. QED.

Lemma A12 Any program from Pdsf is synchronization failure free on RMlc (Theorem 4.4).

Proof: We prove by construction. Given an execution E1 with synchronization failure on RMlc, we could construct an execution E2 on RMlc_2 that is the same as in the proof of Lemma A10. We show E2 would contain a synchronization failure too, which contradicts Lemma A9. Let E1' and E2' denote the same as in the preceding proof. After an application of (i) for Pe, a token for Pe will be available in E2' but not in E1'. Subsequently, *only* after the application of (ii) for the corresponding Q, will the token for its semaphore become unavailable in E2'. Therefore, after each application of (i), (ii) or (iii), either E1' and E2' contain the same available tokens, or E2' contains some extra tokens available for Pe only. By the initial assumption, E1' will become a synchronization failure, which means RMlc does not contain available token for the next P operation in E1 - E1'. The next P is either a Pe or a non-exclusive consumer P.

Both E1' and E2' contain the same tokens available for non exclusive consumers. So if the next P is not an exclusive consumer and there is no token available in E1' for the P, then there will be no token available for the P in E2' either, which means E2' is a synchronization failure. But the next P cannot be a Pe: there can

be either (1) E1' and E2' does not contain a token available for the Pe just after (ii) is applied, or (2) E2' but not E1' contains a token available for the Pe after (i) but not before (ii) is applied. In the case (1) , E2' is a synchronization failure and we are done. In the case (2), we let RMlc_2 execute the Pe, which is already executed by RMlc. E2' will contain no token for the Pe either and becomes a synchronization failure as well. Therefore, such E1 cannot exist and RMlc cannot fail. QED.

Theorem A4 RMlc is sequentially consistent for programs in Pdsf (Theorem 4.1).

Proof: The proof is similar to that of Theorem A3. We show that for an execution E1 on RMlc, there is another execution E2 on RMlc_2 such that result(E1) = result(E2). This is proved by construction. As in the proof of Lemma A10, given E1, we can construct another execution E2 on RMlc_2. Notice that data evens will be constructed by (iii) only. Since both E1 and E2 are data race free, consider the projection of PO(E1) and PO(E2) on a same data variable. If D $\longrightarrow$ D' is in result(E1), then there must exist a path_s(D1,D2) in both E1 and E2 or they are program ordered. Therefore result(E2) = result(E1). QED.

Appendix B nsyncL in Release Consistency

The following is an example to show that Release Consistency (Gharachorloo et al. 1990) may not satisfy sequential consistency if it allows nsyncL in the program.

B.1 Introduction

(Gharachorloo et al. 1990) claims that given any properly labeled program, P_{PL}, *RC* is sufficient to guarantee that a multiprocessor (*RM*rc for short) only produces result which is also obtainable by running the P_{PL} on some M_{SC} machine:

> The relation *A=B* say that for a certain program, (consistency) models *A* and *B* cannot be distinguished based on the results of the program.
>
> Let us first restrict the programs to *PL* programs under sequential consistency (*SC*). Given such programs, we have proved the following equivalences: *SC=RC*.

B.2 A Counter-Example

Unfortunately, the above claim is not true. We give a counter-example in Figure 55. Consider the program in Figure 55 (a), and its execution result, particularly to our interest, those values returned by read operations to variables X and Y. For example, if we run the program on an M_{SC} machine, the result is either {<*P2, X=0, Y=0*>} or {<*P2, X=1, Y=1*>}, in which *P2* is Processor 2's identifier.

Next we run the program on a *RMrc* machine. But first we need to label its statements, specifically for our purpose, we want it to be *properly labeled*. A possible labeling is given

in Figure 55 (a). *Locks* are treated as *acquire* and *unlocks* as *release*. And deliberately, accesses to variable *X* are labeled as $ordinary_L$ while accesses to variable *Y* are labeled as $nsync_L$. Clearly this labeling makes the program *properly labeled* since under any legal interleaving, an ordinary access will not conflict with another access: ordinary accesses are perfectly protected by *acquires* and *releases*. But run it on an *RMrc* machine, we may get the result {<*P2, X=1, Y=0*>} from some execution as shown in Figure 55 (b). The execution satisfies *release consistency*: any ordinary access is performed after the *acquire* which precedes them has been performed (Condition 1), a *release* is performed only after all previous ordinary accesses have been performed (Condition 2), and the result concerning special accesses (statement 1, 3, 4, 5, 7, 8) is {<*P1, a=0*>, <*P2, a=0*>, <*P2, Y=0*>}, which is also obtainable from an execution of them (the corresponding program containing special accesses only is shown in Figure 55 (d)) on some M_{SC} machine as shown in Figure 55 (c) (Condition 3). But the result {<*P2, X=1, Y=0*>} is not possible if the program is run on any M_{SC} machine. This contradicts the above claim about *RC*'s sufficiency for P_{PL} programs.

Initially a=X=Y=0;

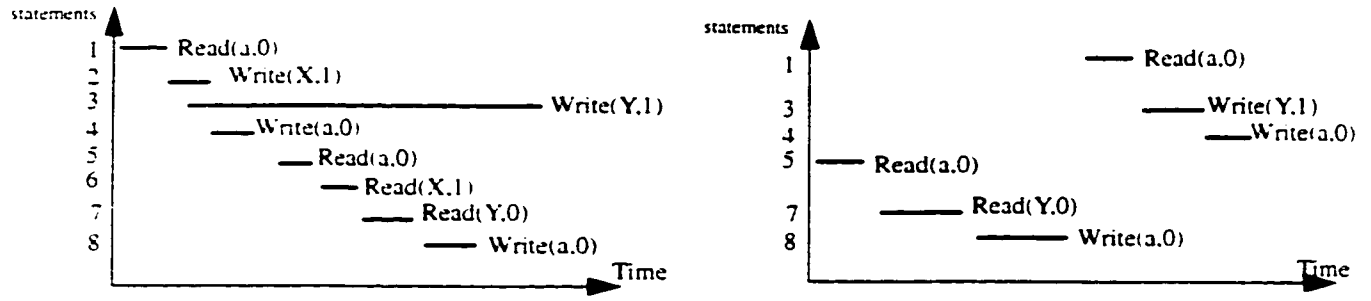
Processor 1		*Processor 2*	
(1) Lock(a);	$[acq_L]$	*(5) Lock(a);*	$[acq_L]$
(2) X:=1;	$[ordinary_L]$	*(6) Read(X);*	$[ordinary_L]$
(3) Y:=1;	$[nsync_L]$	*(7) Read(Y);*	$[nsync_L]$
(4) Unlock(a);	$[rel_L]$	*(8) Unlock(a);*	$[rel_L]$
HALT		*HALT*	

(a) A Sample Program

Initially a=X=Y=0;

Processor 1	*Processor 2*
(1) Lock(a);	*(5) Lock(a);*
(3) Y:=1	*(7) Read(Y);*
(4) Unlock(a);	*(8) Unlock(a);*
HALT	*HALT*

(d) Special accesses in the sample program



(b) An execution of the program on an *RMrc*

(c) An execution of program in (d) on an M_{SC}

Figure 55 A counter-example. A horizontal line to the right of a statement in (b) and (c) indicates the start point and completion point of the statement.

B.3 Concluding Remarks

Certainly, we have typed the statements in the above program in a very particular way. The necessity to do so may not be seen in most cases. We provide this counter-example only to show the imperfection of an assertion which can be remedied in one way or another. Also notice that the above counter-example is still valid if *Locks* are labeled both as *acquire* and *release* as suggested in (Gharachorloo et al. 1990). Finally it can be easily observed that given a parallel program, there always exists a way to label it using *acquire* and *release* so that the program running on a *RMrc* machine will produce a result which is also obtainable by running it on some M_{SC} machine. One of the simplest ways is to label every statement as *acquire* or as *release*. Our counter-example obviously has nothing to do with this observation which is somehow a trivial one, also true with *Weak Ordering* (Dubois et al. 1986) or DRF0 or DRF1 (Adve and Hill, 1990, 1993).

Appendix C Some Negative Results

Our research starts by many conjectures, some of which are provably correct, some are provably incorrect and some remain un-resolved. In this section, we report some other key conjectures are against our intuition. Knowing exactly why the intuition fails may help one to improve the understanding of some of difficulties of the problems we are tackling in this thesis.

C.1 Memory Consistency

We give two examples that explain the risk of further relaxing the conditions of Link Consistency. Such relaxation may lead to either data race or synchronization failure.

Example MC1.The program order between Va and Vb is important and should not be relaxed. Consider the following program:

```
T1          T2          T3

1) Va       3) Va       9) X:= ...
2) Vb       4) Pb       10) Pc
            5) Pa       11) Va
            6) Vc
            7) Pa
            8) X:= ...
```

On BM, the program contains no data race since operation 6 must be executed before operation 10 and operation 11 before 7. If we allow RMlc to further relax the program order between operation 1 and 2 then operation 2 can be executed before operation 1 and the following execution contains data race:

2;3;4;5;6;1;7;8;9; ... or (Vb;Va;Pb;Pa;Vc;Va;Pa;X:=; X:=; ...).

Example MC2.The program order between PaPb is important if Pb is not an exclusive consumer. Consider the following program:

```
T1          T2          T3

1) Vb       3)Pb        7) Pa
2) Vc       4) Pc       8) Pb
            5) Va
            6) Vb
```

It is easy to verify that the program contains no synchronization failure on BM. If we allow RMlc to relax the program order operations 7 and 8, then the same program contains synchronization failure in the following execution:

1: 8: 2 or (Vb:Pb:Vc). No more progress is possible.

C.2 Synchronization Failure Detection

Here we give several examples to show some erroneous intuitions in reasoning about a binary trace sets. These counter examples concern some incorrect intuitions about the state dominance, the token collision, the PV pruning, permutation or decomposition. Notice that every example next has a successful execution.

Example FD1. Incorrect State Dominance.

Let T=<I, {t1, t2, ...}> and T'=<I', {t1, t2, ...}>, I⊂ I'. A possible state dominance lemma says that T' $\stackrel{f}{=>}$ T. Unfortunately it is untrue in general.

Consider the following trace sets: T=<{Vb}, {PbVbVa, PaPb}> and T'=<{Va, Vb}, {PbVbVa, PaPb}>. It is not difficult to verify that T' contains synchronization failure while T does not.

Example FD2. Incorrect Token Collision, that is "A VaVa B $\stackrel{f}{\Leftarrow}$ A Va B" is false.

Consider the following trace sets:

T=<{Vb}, {VaVa, PcPbVbVa, PdPaPb, PaVcVd}> and

T'=<{Vb}, {Va, PcPbVbVa, PdPaPb, PaVcVd}>.

It is not difficult to verify that T contains synchronization failure but T' does not.

Example FD3. Incorrect Pruning, that is, "A PaVaPaVa B $\stackrel{f}{\Leftarrow}$ A PaVa B" is false.

Consider the following trace sets:

T=<{}, {Va1 Pa1 Vb1 Pc1 Vd1, Pd1 Vd6 Pd5 Vd2 Pb1 Va2,

Vd3 Vd4 Pd2 Vc1 Pd3 Vd5 Pa2Vb2Pd4 }>, and

T'=<{}, {Va1Pa1Vb1Pc1Vd1, ~~Pd1Vd6~~ Pd5Vd2Pb1Va2,

Vd3Vd4Pd2Vc1Pd3Vd5Pa2Vb2Pd4 }>.

T contains synchronization failures in the following scenario of execution order: Vd3, Pd1, Vd4, Pd2, Vd6, Vc1, Pd3, Vd5, Va1, Pa2, Vb2, Pd4. The failure suffix is <{Pa1Vb1Pc1Vd1, Pd1Vd6Pd5Vd2Pb1Va2}>. It can be shown that T' contains no synchronization failure.

Example FD4. False Permutation, that is, "A PaVaPbVb B $\stackrel{f}{\cong}$ A PbVbPaVa B" is false.

Consider the following trace sets:

T=<{Vb1}, {Pa1 Va1 Pb1 Vb2 Vc1 Pc1, Pb2 Vc2 Va2 Pc2 Vb3}>, and

T'=<{Vb1}, {Pb1 Vb2 Pa1 Va1 Vc1 Pc1, Pb2 Vc2 Va 2 Pc2 Vb3}> in which Pb1Vb2 is moved before Pa1Va1.

T' contains synchronization failures in the following scenario of execution order: Vb1, Pb1, Vb2, Pb2, Vc2, Va2, Pa1, Va1, Vc1, Pc1. The failure suffix is <{Pc2Vb3}>.

It is easy to show that T contains no synchronization failure.

Example FD5. Incorrect Decomposition: "A PaVaPbVb B $\stackrel{f}{=}$ (APaVaB or APbVbB)" is false.

This can be shown using the result of Example FD4. Suppose the Decomposition rule is true then we will have APaVaPbVb B $\stackrel{f}{=}$ (A PaVaB or APbVb B) $\stackrel{f}{=}$A PbVbPaVa B. which implies that the Permutation rule is true, contradicting what we have described in Example FD4 above.

Notice that in examples FD4 and FD5 we do not make assumptions on semaphores such as whether they are TCF, binary , or persistent (a term borrowed from Petri Net (Peterson, J. L. 1981), meaning that when there is a partial behavior with some Va available and a ready Pa in the behavior's suffix, no matter how the partial behavior is extended, there will *always* exist an available Va' (Va' = Va or not) for this P until it is enabled).

Appendix D Proof of a Theorem For Synchronization Failure Detection

Here we prove that the synchronization failure detection problem for a trace set that contains CVA and one binary semaphore only is NP complete. The proof is by constructing a reduction from the 3SAT problem (Garey and Johnson 1979).

Let C = {C1, C2, ... , Cn}, a set of clauses. Ci = {c1, c2, c3}, where c1 (c2, c3) is an instance of some variable or its negation, x or $\bar{x}$, x∈ U which is a set of variables, {u1, u2, ..., u_m}.

1. Reduction from 3SAT to the synchronization failure detection problem

We use the following simple set of clauses to illustrate how to construct a trace set from a given set of clauses. Consider C={C1, C2} where C1={a + b + $\bar{c}$}, C2={$\bar{a}$ + b + d}, and a, b, c and d are four different variables.

The following construction rules specify how to construct a trace set from a set of clauses given for a 3SAT problem:

(1) For each *variable a*, use three unique control variables: a, A and $\bar{A}$.

Similarly for variable b, c and d.

(2) For *each clause*, use a unique control variable. X1 for C1 and X2 for C2 in our case.

(3) For *each occurrence* in Ci use a unique control variables Ai, i=1 or 2 for our case.

(4) Construct the following four special threads involving four unique control variables S1,

S2, S3 and S4, plus one binary semaphore s:

```
Thread 0:   Initialization
      W(S1,0);
      W(S2,0);
      W(S3,0);
      W(S4,0);
      W(X1,0);
      W(X2,0);
      W(A,0);
      W(Ā,0);
      W(B,0);
      W(B̄,0);
      W(C,0);
      W(C̄,0);
      W(D,0);
      W(D̄,0);
      W(A1,0);
      W(A2,0);
      W(B1,0);
      W(B2,0);
      W(C1,0);
      W(D2,0);
      W(S1,1);
```

```
Thread 1: R(S1, 1);
          V(s);
          P(s);
          W(S4,1);
          W(S2,1);

Thread 2: R(S1, 1);
          R(S4,1);
          V(s);
          P(s);
          W(S2,1);

Thread 3: R(S1, 1);
          R(X1,1);
          R(X2,1);
          W(S4,1);
```

(5) For *each literal* in a clause construct a thread in the way specified next (simulation of clauses):

```
Thread 4: R(S3,1); R(a,1); W(X1,1); W(A1,1) for literal a in C1
Thread 5: R(S3,1); R(b,1); W(X1,1); W(B1,1) for literal b in C1
Thread 6: R(S3,1); R(c,0); W(X1,1); W(C1,2[4]) for literal c̄ in C1
Thread 7: R(S3,1); R(a,0); W(X2,1); W(A2,2) for literal ā in C2
Thread 8: R(S3,1); R(b,1); W(X2,1); W(B2,1) for literal b in C2
Thread 9: R(S3,1); R(d,1); W(X2,1); W(D2,1) for literal d in C2
```

(6) Construct two special threads as follows:

Thread 10: To allow Thread 4 through Thread 9 to proceed when there is no satisfiable assignment for the clause set C so the corresponding trace set will not fail.

```
R(S1,1);
R(S2,1);
```

4. Use value 1 if the literal is not in negative form. Use value 2 otherwise.

```
                R(S3,1);
                W(S4,1);
                W(a,1);
                W(b,1);
                W(d, 1);
                R(A1,1);
                R(B1,1);
                R(B2,1);
                R(D2,1);
                W(a, 0);
                W(c, 0);
                R(A2,2);
                R(C1,2);

Thread 11:  Wait until assignment is simulated.

                R(S1,1);
                R(A,1);
                R(Ā,1);
                R(B,1);
                R(B̄,1);
                R(C,1);
                R(C̄,1);
                R(D,1);
                R(D̄,1);
                W(S3, 1);
```

(7) For *each variable* construct two threads as follows (simulation of variable assignments):

```
Thread 12:R(S1, 1);
                W(a, 1);
                W(A, 1);

Thread 13:R(S1, 1);
                W(a, 0);
                W(Ā, 1);

Thread 14:R(S1, 1);
                W(b, 1);
                W(B, 1);

Thread 15:R(S1, 1);
                W(b, 0);
                W(B̄, 1);

Thread 16:R(S1, 1);
                W(c, 1);
                W(C, 1);

Thread 17:R(S1, 1);
                W(c, 0);
                W(C̄, 1);

Thread 18:R(S1, 1);
                W(d, 1);
                W(D, 1);

Thread 19:R(S1, 1);
                W(d, 0);
```

W($\bar{D}$, 1);

So the trace set T= <{}, {ti}> where i ranges from 0 to 29 inclusive in this case.

In general, there will be 3m (construction rule 1) + n (construction rule 2) + 3n(construction rule 3) + 4 (construction rule 4), or equivalently, (3m+4n+4) control variables used, and 6(construction rule 4 and 6) + 3n (construction rule 5) + m (construction rule 7), or equivalently, (3n+m+6) threads. Using the above trace set, we now prove the theorem.

2. The Proof

(i) The problem is in NP.

Since it takes only polynomial time to construct an interleaving of events in trace set T and to verify whether the interleaving corresponds a synchronization failure, the problem is in NP.

(ii) The construction takes polynomial time.

This is easy to see from construction rules (1) to (7). In fact the construction takes O(max(m,n)) time only. What remains to be shown is the following:

(iii) The trace set from above construction contains synchronization failure iff C is satisfiable.

'if' part: C is satisfiable with some assignment of true and false for every variable. That implies that at least one of three W(X1, 1) *and* at least one of three W(X2, 1) (see construction rule 5) must be able to be executed, which further implies that W(S4,1) in Thread 3 must be able to be executed before R(S4,1) in Thread 2 and V(s) in Thread 1, which implies that V(s) in Thread 1 and V(s) in Thread 2 can collide and the trace set T has a failure behavior.

'only if' part: C is not satisfiable with any assignment of true and false for every variable. That implies that there is at least one Ci whose components are all assigned a false value. That is, if not both, either none of three W(X1, 1) or none of three W(X2, 1) (see construction rule 5) can be executed before Thread 10 can be started. Therefore Thread 3 cannot be completed before R(S2,1) is complete, which implies its W(S4, 1) can only be executed after W(S2,1) of Thread 1 or Thread 2. Since the R(S4, 1) that Thread 2 contains must wait until after some W(S4, 1) is executed and there is only one other W(S4, 1) in Thread 1, W(S4, 1) in Thread 1 must be executed before R(S4, 1) of Thread 2. That implies then V(s) of Thread 1 must be consumed by P(s) of the same thread, and V(s) of Thread 2 must be consumed by the P(s) in Thread 2. No more token collision is possible. Also notice that when either W(S2, 1) (in Thread 1 or Thread 2) is executed, Thread 10 will be able to proceed. As one can see Thread 10 will enable Thread 4 through Thread 9 to complete if they are not completed yet. When Thread 4 through Thread 9 are all completed, Thread 3 will complete and every event of the trace set T will be executed eventually. Thus T cannot fail. QED.

Appendix E Experiment Environment

The experiments for running synchronization failure detection algorithms use the SPARC 2 workstation from Sun Microsystems. The software consists of the following five projects:

Project 1 includes the algorithm GFD and the related algorithms: the algorithm for on-demand unfolding of semaphore trees, the Modified GBE and GNR.

Project 2 includes the interleaving-based simulation algorithm IL.

Project 3 includes the algorithm FC, which uses full-cross product Y to detect global failures.

Project 4 estimates the number of leaves of a local behavior tree for Project 3 so that we can control the memory space and CPU time for Project 3 to run.

Project 5 is a user interface for the conversion of trace set from its external representation to its internal representation required by the above projects.

The software is written using OSF GNU C++ language and consists of about 2000 lines C++ code.

Glossary of Arrows

ev1 $\xrightarrow{p,l}$ ev2 (or ev1 $\longrightarrow$ ev2): Program order in an execution. ev1 and ev2 are from the same program thread and ev1 is ordered immediately before ev2 in an partial order execution or a partial order representation of a total order execution.

ev1 $\xrightarrow{c,l}$ ev2 (or ev1 $\Longrightarrow$ ev2): ev1 is the last event that is executed before ev2 and in conflict with ev2 in an execution.

$\xrightarrow{p}$: Transitive closure of $\xrightarrow{p,l}$.

$\xrightarrow{l}$: Either $\xrightarrow{p,l}$ or $\xrightarrow{c,l}$.

$\longrightarrow$: Transitive closure of $\xrightarrow{l}$.

ev1 $\xrightarrow{a,l}$ ev2: if ev1 $\xrightarrow{c,l}$ ev2 but not ev1 $\xrightarrow{p}$ ev2.

op1 $\longrightarrow$ op2: Represents inter-processor data dependency between data operation op1 and op2.