Specifying and Enforcing Synchronization Disciplines in Multithreaded Programs

by
David Moon

Stephen N. Freund, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 23, 2016

Contents

List of Figures

Abstract

Multithreaded programs are notoriously prone to race conditions. Programmers try to avoid race conditions by adopting various synchronization disciplines, which are simple policies for ordering thread access to shared memory. As these disciplines are left implicit, most existing methods for dynamic race detection can only verify whether a set of memory accesses are ordered, not whether they are ordered as intended.

This thesis presents a new framework for specifying and enforcing synchronization disciplines in multithreaded programs. We introduce a simple model for synchronization disciplines called a *synchronization flow graph* (SFG for short) which captures the dynamic behavior of a discipline in terms of ordered sequences of access modes. Disciplines for individual program variables can be declared as SFGs using a compact text-based specification grammar. We also present a corresponding technique for dynamically enforcing conformance to specified SFGs. This technique automatically refines each specified SFG into a corresponding Petri net, which is then simulated at run time to enforce conformance. We prove that execution conformance to any specified SFG implies race freedom, and that our enforcement technique is sound with respect to conformance.

Acknowledgments

I would like to thank Stephen Freund for being an excellent advisor throughout the process of writing this thesis. I have learned a tremendous amount under his guidance. I would also like to thank Duane Bailey, my second reader, for providing helpful feedback and fresh insights. The computer science department as a whole has provided a fun and supportive environment. A special thanks goes to James Wilcox, class of 2013, for convincing me to take 136 four years ago, teaching me about field automorphisms in lab, taking me to my first PL conference, and countless other things. This thesis would not have been possible without his influence. Warm thanks are due as well to Derek Galvin for his friendship and support throughout this project. Last, but far from least, I would like to thank my parents Young Bae Moon and Nam Hee Yang, for their unending support and encouragement in all my endeavors.

Chapter 1

Introduction

The slowing improvement of raw processor speed has led to widespread adoption of multicore architectures and concurrent programming. Concurrent programs take advantage of multiple cores by executing multiple threads simultaneously. Because there are no inherent guarantees about the relative ordering operations by different threads, however, this performance comes at a cost: the programmer must manually synchronize accesses to shared resources, including memory. This is not an easy task in general, and synchronization mistakes are common. Without proper synchronization, two threads may attempt to update the same resource at the same time and cause unexpected behavior.

The most basic concurrency error is the *data race condition*. A race condition occurs when multiple threads access the same memory location, at least one of which is a write, and they are unordered by synchronization. As they may cause problems only on rare interleavings by the hardware scheduler, races can be extremely difficult to detect, reproduce, and eliminate with standard testing methods. Undetected races have had catastrophic consequences in the past, such as fatal overdoses by the Therac-25 radiation therapy machine [8] and the 2003 Northeast blackout [12].

Consequently, there is a need for automated detection of race conditions. Many race detection analyses have been developed over the past few decades, but no single method is perfect. The ideal analysis would be both *sound* and *complete*, meaning it detects all race conditions and reports no false positives, and it would run in a reasonable amount of time. Unfortunately, sound and complete race detection is undecidable in general, and so existing analyses make different trade-offs between accuracy, performance, and ease of use. For example, static analyses (which examine the source code) typically sacrifice completeness for the sake of performance. On the other hand, some dynamic analyses (which monitor the program as it executes) report no false alarms, but are only sound with respect to the observed trace and may not report any race conditions that would occur on different interleavings.

Most of these analyses work directly on unmodified source code and infer how shared variable accesses are synchronized. While such analyses are easiest to use, they must incur additional performance overhead in order to reconstruct this information. Thus, some past work [17, 19] takes a different approach and incorporates source-code specifications provided by the programmer. These

specifications declare the explicit *synchronization disciplines* by which programmers intend to order thread access to shared memory. This breaks the problem of race detection into two smaller ones: checking whether the specification precludes race conditions, and checking whether the program conforms to the specification.

The potential benefits of this approach are many. The most obvious benefit is that a checker for specification violations may be simpler and more performant than a general race detection analysis. This is already suggested by Eraser, an early dynamic race detector which checks that some mutex lock is held on every access to a shared variable, and can be viewed as verifying a pre-specified locking discipline on every variable. In exchange for sacrificing completeness, it remains one of the simplest and fastest dynamic race detectors so far devised. Another potential benefit is that, because the programmer provided the specification, errors may be reported in closer relation to the programmer's intent and therefore more easily understandable. In the case of dynamic analyses in particular, this may have the additional benefit of catching lurking race conditions that did not occur in the observed trace. Indeed, this is a unique advantage of Eraser in its aim to check for a locking discipline rather than general, and sometimes incidental, race freedom.

1.1 Goals

The goal of this thesis is to present a new framework for specifying and enforcing synchronization disciplines in multithreaded programs. In particular, we present a source-level annotation language for declaring disciplines and a corresponding technique for dynamically enforcing conformance to the annotations. Each annotation in our language decorates a shared variable and explicitly declares the discipline by which accesses to the variable should be ordered. A key contribution of this work is modeling each discipline as a *synchronization flow graph* (SFG for short), an automaton-like structure that abstracts away details of individual program operations and encodes the high-level "control flow" of the discipline. Using a few simple operators on SFGs, disciplines in our language are expressed recursively as compositions of sub-disciplines; these operators support compact expression of a richer variety of synchronization disciplines than in previous work, while guaranteeing that every specified discipline is race-free by construction.

Our enforcement technique refines each SFG into a more detailed structure called a *Petri net*. Petri nets are a mathematical model of computation that is particularly well-suited for modeling concurrent and distributed systems. The Petri net derived from an SFG explicitly encodes the individual synchronization operations implicit within the SFG abstraction, and is simulated at run time to ensure accesses are correctly synchronized. Unlike past work, this analysis is entirely automatic and requires no additional input from the programmer beyond the initial SFG specification.

We demonstrate that a variety of commonly used disciplines can be expressed using our language. We prove that execution conformance to any specified SFG implies race freedom for the annotated variable, and that our enforcement technique is sound with respect to conformance. Finally, we discuss the current limitations of our work—in particular, with respect to locking disciplines as well as performance—and present directions for building upon our framework.

1.2 Thesis Overview

This thesis is presented as follows.

Chapter 2 presents an overview of related work on race condition detection and synchronization discipline specification design.

Chapter 3 presents a high-level overview of our specification language and enforcement technique.

Chapter 4 formalizes our specification language and enforcement technique, proves that execution conformance to any SFG specified in our language implies race freedom, and proves that our enforcement technique is sound with respect to conformance.

Chapter 5 discusses the current limitations of our approach and presents directions for future work.

Chapter 6 reviews our accomplishments.

Chapter 2

Background

The widespread adoption of multicore processors has driven the development of multithreaded software to achieve high performance. Unfortunately, writing correct multithreaded programs is particularly difficult. There remains an unmet need for verification infrastructure that is both precise enough to detect all relevant concurrency errors and efficient enough to integrate into the programmer's workflow.

2.1 Race Conditions

Multithreaded programming is difficult, in part, due to the need to control interference between threads. Consider the simple bank account program in Figure 2.1. This program adds and removes money from a shared global `balance`. However, it does not require the threads to execute in a particular order, so the thread scheduler may interleave their steps in different ways. The desired behavior is for the program to execute the lines in the order 4, 5, 8, 9 or the order 8, 9, 4, 5—that is, for each thread procedure to execute atomically. Instead, the two threads may interleave their steps so that they execute in the order 4, 8, 5, 9, as in the execution trace in Figure 2.2, such that the final assertion in line 17 fails.

At the heart of this and many other concurrency errors is the *race condition*. Roughly speaking, a race condition occurs when two or more threads access the same memory region simultaneously and at least one of the accesses is a write. (See Section 2.2 for a precise definition.) In such occurrences, the programmer has no guarantee regarding the ordering of memory accesses by different threads, due to either arbitrary interleaving by the underlying thread scheduler or to the relaxed memory consistency rules present in many existing hardware systems. To avoid race conditions, the programmer must use *synchronization mechanisms* to coordinate accesses to shared data between multiple threads. Common mechanisms include:

Mutual exclusion locks Locks can be used to ensure a shared resource is accessed by a single thread at a time. They support two operations: `acq` (acquire) and `rel` (release). If one thread has

```
 1  int balance;
 2
 3  void deposit:
 4    int temp1 = balance;
 5    balance = temp1 + 100;
 6
 7  void withdraw:
 8    int temp2 = balance;
 9    balance = temp2 - 100;
10
11  void main:
12    balance = 500;
13    fork(deposit);
14    fork(withdraw);
15    join(deposit);
16    join(withdraw);
17    assert balance == 500;
```

Figure 2.1: A bank account program with a race condition on `balance`.

Step	main	deposit	withdraw	balance
1	`balance = 500`			500
2	`fork(deposit)`			500
3	`fork(withdraw)`			500
4		`temp1 = balance`		500
5			`temp2 = balance`	500
6		`balance = temp1 + 100`		600
7			`balance = temp2 - 100`	400
8	`join(deposit)`			400
9	`join(withdraw)`			400
10	`assert balance == 500` `// fails`			400

Figure 2.2: A sample run of the program in Figure 2.1, demonstrating a race condition.

acquired a lock, and a second thread tries to acquire the same lock, the second thread blocks until the first thread releases the lock.

Fork and join The `fork` operation creates a new thread or a set of threads. The `join` operation causes the invoking thread to block until the previously `fork`ed threads finish executing. Program steps by the `fork`-ing thread before the `fork` operation must execute before any steps by a `fork`-ed thread, which in turn must execute before any steps by `join`-ing thread after the `join` operation.

Barriers A barrier is shared by a fixed set of threads. When a thread enters a barrier, it blocks until every other thread has also entered the barrier, at which point all threads continue execution.

Volatile variables Volatile variables are a lower-level mechanism than those mentioned above, and do not cause any threads to block or spawn. Writes to a volatile variable go directly to main memory and are immediately visible by all threads. This imposes a total ordering on all accesses of the volatile variable, which is not the case for ordinary data variables. Section 2.2 makes this notion more precise.

Higher-order mechanisms Other abstractions built on top of the above may act as higher-order synchronization mechanisms. For example, some multithreaded programs consist of a producer thread and a consumer thread, where the producer pushes data packets into a queue and the consumer removes them. In this case, the actions by the producer before pushing a data packet must occur before the actions by the consumer after it removes that packet.

We may fix our bank account in Figure 2.1 by adding a lock `m` as in Figure 2.3; new added lines are 2, 5, 8, 11, and 14. Because of the locking, the `withdraw` thread must wait for the `deposit` thread to finish before it can begin execution, as shown in Figure 2.4.

Unfortunately, programmers frequently make mistakes in their use of synchronization mechanisms. Undetected races in mission critical applications have had catastrophic consequences in the past. For example, the 2003 Northeast blackout was the result of a subtle race condition in power transmission control systems that allowed local failures to cascade and become widespread [12]. Prior to that, between 1985 and 1987, a race condition in the Therac-25 radiation therapy machine caused massive overdosing of several patients, leading to deaths and serious injuries [8].

The issue is futher compounded by the nondeterminism of thread scheduling. While necessary for efficiency purposes, this means that some races may only manifest on rare interleavings, which greatly diminishes the effectiveness of traditional debugging techniques. Hence, there is a need for tools that can automatically and comprehensively detect race conditions.

2.2 Happens-Before Relation

In the previous section, we stated informally that a race condition occurs when two threads access the same memory region "simultaneously", i.e., the accesses are unordered in some sense. To define

```
1  int balance;
2  lock m;
3
4  void deposit:
5    acq(m);
6    int temp1 = balance;
7    balance = temp1 + 100;
8    rel(m);
9
10 void withdraw:
11   acq(m);
12   int temp2 = balance;
13   balance = temp2 - 100;
14   rel(m);
15
16 void main:
17   balance = 500;
18   fork(deposit);
19   fork(withdraw);
20   join(deposit);
21   join(withdraw);
22   assert balance == 500;
```

Figure 2.3: A race-free bank account program.

Step	main	deposit	withdraw	balance
1	balance = 500			500
2	fork(deposit)			500
3	fork(withdraw)			500
4		acq(m)		500
5		temp1 = balance		500
6		balance = temp1 + 100		600
7		rel(m)		600
8			acq(m)	600
9			temp2 = balance	600
10			balance = temp2 - 100	500
11			rel(m)	500
12	join(deposit)			500
13	join(withdraw)			500
14	assert balance == 500 // succeeds			500

Figure 2.4: A sample run of the race-free program in Figure 2.3.

race conditions precisely, we must formalize the ordering of events in a concurrent system. The total ordering imposed by physical time is too restrictive: two parallel threads accessing disjoint regions of memory and invoking no synchronization exhibit the same behavior no matter how their steps are interleaved. For a notion of order to be useful with respect to program behavior, it should be possible for an event a_1 to causally affect an event a_2 if a_1 comes before a_2 under this order.

Lamport's happens-before relation [7] precisely captures this idea. The execution trace of a multithreaded program is represented by a sequence ω of program operations, each performed by a thread. The operations represent execution steps that are externally visible to all threads; these may be memory reads, memory writes, or synchronization operations. The *happens-before relation* $\prec_\omega$ on ω is a strict partial order on the operations defined as follows:

- (program order) If o precedes o' in ω, and both are performed by the same thread, then $o \prec_\omega o'$.
- (synchronization order) If o precedes o' in ω, and any one of the following statements holds, then $o \prec_\omega o'$:
  - o releases a lock `m` and o' acquires `m`
  - o writes to a volatile variable `v` and o' reads from `v` [1]
  - o enters a barrier object `b` and o' exits `b`
  - o forks a thread t and o' is an operation performed by t
  - o is an operation performed by thread t and o' joins t
- (transitivity) If $a_1 \prec_\omega a_2$ and $a_2 \prec_\omega a_3$, then $a_1 \prec_\omega a_3$.

Events a and a' are *concurrent* if $a \not\prec_\omega a'$ and $a' \not\prec_\omega a$, i.e., they are unordered by happens-before. They are *conflicting* if they access the same shared memory and at least one of the events is a write. Finally, they form a *race condition* if they are both concurrent and conflicting.

For example, consider the directed graph in Figure 2.5, which visually represents the happens-before relation on the trace in Figure 2.2. There is no path between operations 6: `balance = temp1 + 100` and 5: `temp2 = balance`; hence, these operations form a race condition. On other hand, examining the happens-before relation on the trace in Figure 2.4 as depicted in Figure 2.6, we see that there is a path between every pair of accesses on `balance`.

We emphasize that a race condition is defined with respect to an observed trace and its happens-before ordering. It is possible that one trace contains a race condition while another trace of the same program, but with a different happens-before ordering, does not. For example, consider the execution trace in Figure 2.7. As there is a total happens-before ordering on the program steps, this particular trace is race-free. However, a different interleaving that switches the order of the lock acquires would reveal a race condition on `y`.

[1] Technically, any conflicting pair of accesses to a volatile variable are ordered by happens-before. However, only the ordering between write-read pairs can affect program execution.

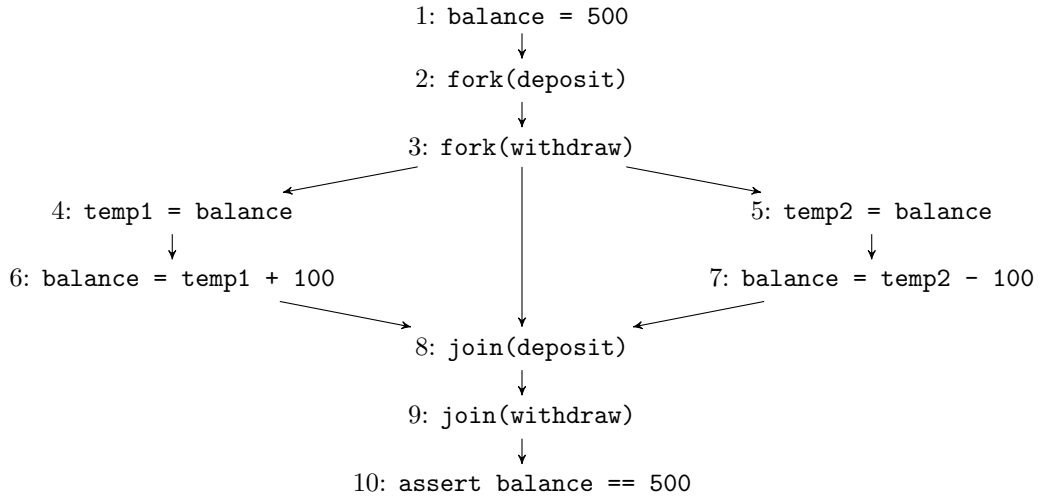


Figure 2.5: The happens-before graph for the racy bank account trace in Figure 2.2.

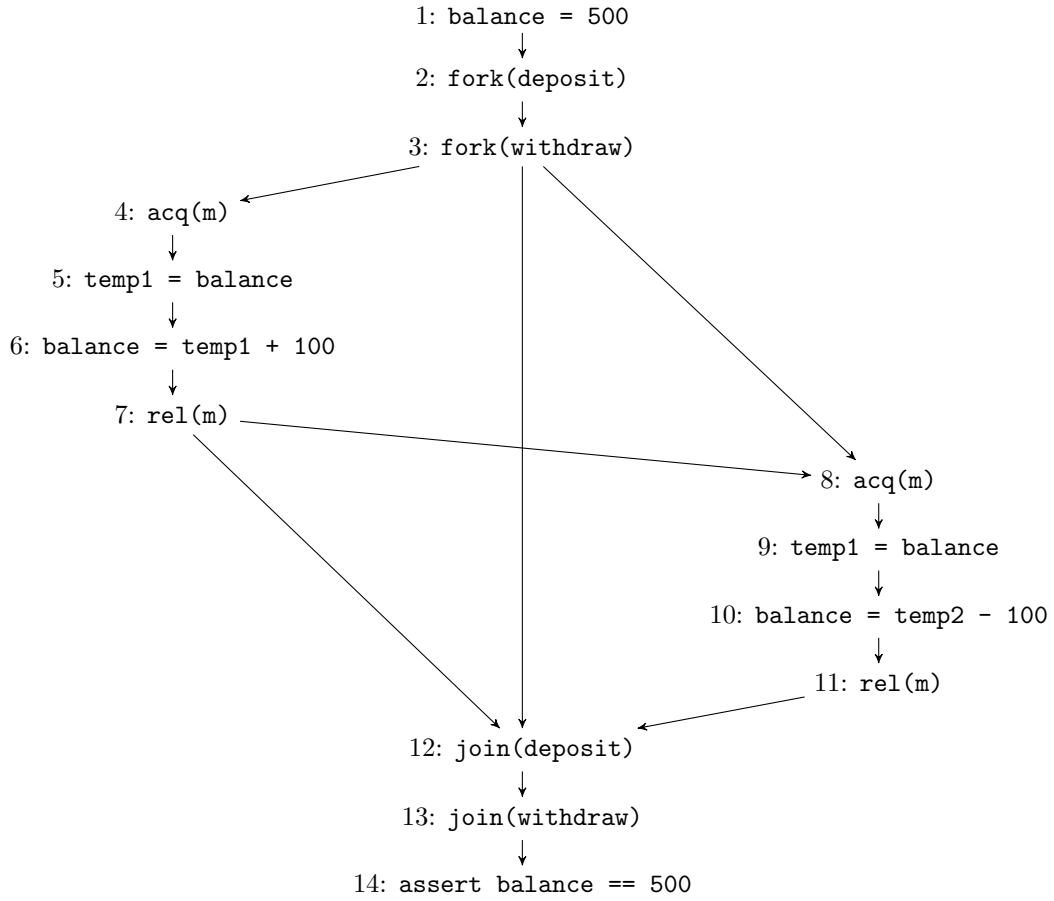


Figure 2.6: The happens-before graph for the race-free bank account trace in Figure 2.4.

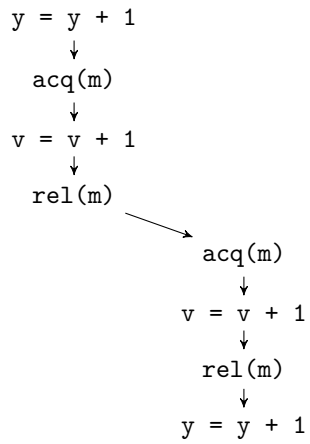


Figure 2.7: A program trace with a hidden race condition.

2.3 Race Detection

A wide variety of tools to detect race conditions automatically has been developed. These may use *static analyses*, which examine the source code of the target program, or *dynamic analyses*, which monitor the program as it executes, or both. We review some of these tools below in light of the following dimensions of the design space:

Soundness A dynamic analysis is *sound* if it reports all race conditions in the happens-before ordering of the observed trace; a static analysis is *sound* if it reports all race conditions in all possible execution traces.

Completeness A race detection analysis is *complete* if it does not report any false positives, i.e., all reported races are true race conditions in some execution trace.

Coverage While static analyses reason about all possible execution traces, dynamic analyses only observe a single trace at a time. Some dynamic analyses, however, generalize beyond the observed trace and detect race conditions in traces with different happens-before orderings. We say that such analyses exhibit greater *coverage* than the standard happens-before analysis.

Performance Dynamic analyses introduce time and space overhead into execution of the target program. Static analyses do not affect the run-time performance, but face fundamental computability limits or may require expensive analyses that preclude scalability; thus, they must carefully balance trade-offs in the dimensions above in order to run in a reasonable amount of time.

2.3.1 Dynamic analyses

The advantage of dynamic analyses is their increased precision combined with low computational overhead. This has allowed for effective use of dynamic race detectors in industry settings, such

as Google's ThreadSanitizer [16] and Helgrind in the Valgrind debugging tool [18]. Neither tool is both sound and complete, however. While a variety of work has lowered the overhead of sound and complete dynamic analyses to relatively modest levels, it currently remains too high to incorporate such tools into the developer's workflow.

A dynamic race detector instruments the target program and tracks access information by maintaining additional *shadow state* for each shared variable during execution. We describe three different shadow state paradigms below.

Vector clocks Vector clocks are a mechanism for precisely recording the happens-before relation on a multithreaded trace. In a vector clock analysis, each thread has a logical clock that increases with each action performed by the thread. For its shadow state, the analysis attaches to each thread and shared variable a *vector clock*, a vector of clock values, in which each index i of the vector clock corresponds to a thread t_i. The update rules for these vector clocks maintain the following invariants:

- The vector clock vc_x for variable x stores at index i the clock value of the last access of x by thread t_i.
- The vector clock vc_i for thread t_i stores at index i the current clock value of t_i; at every other index j, it stores the clock value of the last action by thread t_j of which t_i is "aware".

A thread becomes "aware" of another thread's clock value via synchronization: each synchronization object (e.g. a lock or a barrier) acts a message passer between threads by allowing threads to read and write clock values to the object upon each invocation of it. Under these invariants, the trace contains a race condition if and only if, when a thread t_i accesses a variable x, $vc_i[j] < vc_x[j]$ for some j—that is, another thread t_j has previously accessed variable x, but without communicating this access via synchronization to thread t_i.

Archetypical examples of the vector clock analysis are DJIT [6] and its optimized cousin DJIT$^+$ [13]. The advantage of these analyses is that they are sound and complete. The disadvantage is that the standard vector clock algorithm is computationally expensive: in a program with n threads, each vector clock operation (which occurs on every memory access) takes $O(n)$ time, which can incur total slowdowns of a few orders of magnitude.

The FASTTRACK algorithm [4] improves upon the standard approach by leveraging the insight that most accesses are totally ordered in race-free sections of the trace. This allows FASTTRACK to compact almost every vector clock to an *epoch* consisting of the last accessing thread and clock value, without sacrificing soundness or completeness. Epoch operations take constant time, and so FASTTRACK is significantly faster than the standard algorithm. Since its inception in 2009, FASTTRACK remains state-of-the-art in sound and complete dynamic race detection.

Eraser: locksets In 1997, Savage et al. introduced a fast but incomplete dynamic race detector called Eraser [15]. Existing dynamic analyses at the time used the standard vector clock algorithm to compute the happens-before relation and so incurred prohibitive overhead. Eraser, in contrast, avoids computing the happens-before relation altogether. Instead it enforces the following synchronization

discipline: in a given trace, every shared variable has an associated lock that is acquired before accessing the variable. Conformance to this discipline imposes a happens-before ordering on all accesses to each shared variable, and thus ensures race freedom.

The shadow state in Eraser takes the form of a set of locks, or a *lockset*, attached to each shared variable. Eraser maintains the lockset throughout program execution such that it contains all locks held on every access to the variable so far. An error is reported whenever a variable's lockset becomes empty, i.e., there is no common lock that protects every observed access.

Eraser excels in performance. Locksets are typically small, which reduces the average cost of lockset operations, and few distinct locksets are observed in any program execution, which allows for compact representation and caching optimizations. Compared to the standard happens-before analysis based on vector clocks or epochs, which can only report races conditions in the observed trace, Eraser has an additional advantage of greater coverage. Consider again the program trace in Figure 2.7, which contains a hidden race on `y` that is only revealed on a different interleaving by the scheduler. Because Eraser does not constrain itself to the happens-before ordering of the trace, it is able to catch this race.

On the other hand, though sound, Eraser is incomplete in general due to its restriction to a specific synchronization discipline. For example, Eraser would issue a false alarm on a variable whose accesses are not protected by locks but totally ordered by volatile accesses. Such false alarms can incur high development costs.

Goldilocks: Extended Locksets The vector clock algorithm is complete but slow, while the lockset algorithm is incomplete but fast. Introduced in 2007, the dynamic race detector Goldilocks [1] aims to get the best of both worlds by using a lockset-like structure to encode the full happens-before relation. "Goldilocks locksets" are extended to include volatiles and threads. For each shared variable x, its lockset $GLS(x)$ maintains the following invariants:

- Lock $l \in GLS(x) \Leftrightarrow$ the last access to x was protected by l.
- Volatile $v \in GLS(x) \Leftrightarrow v$ has been written to since the last access to x.
- Thread $t \in GLS(x) \Leftrightarrow$ the last access to x happens-before any subsequent access to x by t.

Under these invariants, a race on x is discovered upon any access to x by t when $t \notin GLS(x)$.

Goldilocks is significantly faster than the standard vector clock algorithm, but still slower than Eraser and FastTrack. (See Table 1 in [4] for a detailed comparison.) It is worth noting, however, that the locksets in Goldilocks capture more than just the ordering of accesses: the synchronization primitives in the lockset reveal *how* the accesses may have been ordered, unlike the synchronization-mechanism-independent clock values in vector clocks. Such richer information is necessary in the context of enforcing specific synchronization disciplines.

2.3.2 Static analyses

Static analyses examine the source code of the target program to detect race conditions. The advantages of a static over dynamic analysis are: (1) it reports errors without needing to run the

program, allowing for earlier detection in the development process, and (2) it reasons about all program paths and therefore can identify all potential race conditions, not merely those observed as in the case of dynamic analyses. These advantages, however, are tempered by a significant disadvantage: sound and complete static race detection is undecidable.

One approach to static race detection is the use of *type systems*, where a type-safe program is guaranteed to be race-free. Type systems are typically efficient enough to be run often, and enforce properties that allow for modular checking of different parts of the program. On the other hand, such modular properties tend to be relatively simple and only apply to a small subset of correct programs. For example, RaceFreeJava [3] is an extended type system for Java in which programmers may declare data to be protected by specific locks. Like Eraser, though sound, RaceFreeJava is incomplete due to restricted reasoning about only lock-based synchronization.

Whole-program analyses often require less specification from the programmer and synthesize more precise control flow information than what can be expressed in most type systems. However, they tend to be slower and are not modular, which poses an issue for checking open programs such as libraries. Two examples are RacerX and Chord. Chord [9] is an analysis that computes all pairs of memory accesses in the source program that may form a race condition. It does this by taking an initial over-approximation of syntactic memory accesses that may affect the same memory region and then iteratively refining this set over a number of analyses, each of which filters out pairs that satisfy some sufficient condition for race freedom. For example, the final *UnlockedPairs* analysis checks each pair of accesses to determine if they are protected under a common lock. Chord is neither sound nor complete, but has been used effectively to detect new race conditions in real-world code. On the other hand, Chord does not scale to large programs. RacerX [2] constructs a control-flow graph of the program and statically simulates a lockset-based analysis over all paths in the program. With various crude approximations—for example, pointer variables are lumped together by type—RacerX is neither sound nor complete. Instead, it takes a developer-centric approach and focuses on post-processing the reported errors according to approximations of how likely they are false positives and how difficult they may be to inspect. With its aggressive focus on scalability and usability, RacerX has been used to detect a number of previously unknown race conditions in Linux and FreeBSD, both multi-million-line programs.

Lastly, static race detection may also be carried out via *model checking*. Model checking involves the systematic exploration of different thread schedules. As the number of different schedules grows explosively with the size of the program, model checkers must resort to search heuristics in order to maximize the number of errors found in reasonable time. One such heuristic for concurrent programs is *iterative context-bounding*, which prioritizes schedules with fewer preemptions by the scheduler (such as in the expiration of a time slice). Iterative context-bounding has been implemented in the model checker Chess to uncover bugs in real-world programs. Unlike other model checkers, Chess is stateless and checks each schedule with Goldilocks. Hence, it is sound and complete with respect to each checked schedule. However, checking any reasonably comprehensive set of schedules requires a significantly longer time compared to any of the static methods above.

2.3.3 Hybrid analyses

Many hybrid analyses, which combine different methods above, have also been developed. Within dynamic checking, a number of race detectors [10, 13] combine lockset and vector clock analyses; Goldilocks [1] emerged partially in response to this class of analyses. Such race detectors run a vector clock analysis to re-examine the potential races reported by a lockset analysis to determine whether they are actual races.

Other hybrid approaches mix static and dynamic checking. In these analyses, an initial static pass optimizes a subsequent dynamic pass by removing redundant checks. For example, a simple thread-local analysis could filter out a set of thread-local memory accesses from being checked dynamically. More sophisticated examples include RedCard [5] and, though it was originally designed as a standalone static analysis, Chord [9]. RedCard reduces the number of run-time checks by a dynamic race detector by close to 40% with no loss in precision.

2.4 Contract Systems for Synchronization Disciplines

Design by contract or *contract programming* is an approach to designing software that emphasizes the specification, meeting, and guaranteeing of conditions between software components. Programmers typically write code with implicit pre-conditions (e.g., the passed argument is a non-null pointer to an array of integers), post-conditions (e.g., the array is sorted in ascending order), and invariants (e.g., the pointer only points to memory within array bounds) in mind. Design by contract advocates explicit and early specification of such conditions in the development process.

Some programming languages integrate such specification into the language and provide some form of dynamic or static checking that the specified conditions are met. Examples of dynamic checking include Java assertions [11] and Racket contracts [14], while any static type system may be viewed as specifying and enforcing simple contracts. Well-designed contract systems tend to help both the programmer and the checker, the former by clarifying design during development, and the latter by providing direct access to the programmer's intent and thereby reducing what it must infer.

In the case of concurrent software, programmers usually follow *synchronization disciplines* that totally order accesses on each shared variable. A simple example is the discipline specified and enforced by the type system RaceFreeJava: every shared variable has at least one lock associated with it that protects all accesses of the variable. Outside of type systems, however, synchronization disciplines are usually not specified explicitly or precisely. Hence, other race detection tools devote significant computation toward inferring these disciplines. For example, Eraser dynamically enforces the same locking discipline as RaceFreeJava, but it must infer which locks are associated with each shared variable.

Eraser may do this inference quickly compared to other dynamic race detectors, but it also restricts itself to a specific, simple class of disciplines. Inferring arbitrary disciplines is much more expensive, as indicated in the thesis work of Benjamin Wood (Williams '08) [19]. Wood's thesis work comprises two parts. The first is a simple synchronization discipline specification language called Grits. A Grits specification for a shared variable is a *compound discipline* made up of a sequence of

simple disciplines; importantly, this expresses how methods of synchronization on a variable change over the course of program execution. For example, if the lock protecting a variables changes from `m` to `n` midway through execution, its compound discipline would be `guarded-by m; guarded-by n`.

The second part is a dynamic analysis tool, called Hominy, that infers a Grits specification for every shared variable. The key step in the analysis can be viewed as an extension of the Goldilocks dynamic race detection algorithm. Recall that, as opposed to clock-based methods, the modified locksets of Goldilocks begin to capture how, i.e., via which synchronization mechanisms, accesses are ordered. Hominy further refines the locksets of Goldilocks along this trajectory, to the point that it can determine exactly which synchronization primitives were necessary to order the current and last accesses and thereby generate an accurate Grits specification. While conceptually novel, Hominy incurs prohibitive slowdown factors of a few orders of magnitude and does not map specifications written over dynamic memory locations back to source-level specifications.

Another former Williams student, Antal Spector-Zabusky '12, developed a different synchronization specification language [17]. Like Grits, Spector-Zabusky's language attaches a synchronization discipline to each shared variable. However, it takes a more refined perspective on the effect of synchronization operations. Spector-Zabusky's language explicitly incorporates the notion of *ownership mode* of a variable. The possible modes are: exclusive (i.e., available only to a single thread); read-shared among a set of threads; and unavailable to (or unowned by) any thread. They reflect the three race-free modes for accessing shared data without any synchronization. A synchronization discipline specification in Spector-Zabusky's language is essentially a deterministic finite automaton in which synchronization operations form transitions from one ownership mode to another. For example, the automaton in Figure 2.8 captures the discipline that a variable is protected by a lock `m`.

To verify that synchronization disciplines are obeyed, Spector-Zabusky's language further incorporates line-end assertions that specify how the annotated line may change the ownership mode of a shared variable. As an example, consider the small program in Figure 2.9. It is written in PolyCoSM, a small imperative language implemented by Spector-Zabusky that integrates his specification language. Each step of the program, including the read of `total` within line 14, is annotated with an assertion that begins with `&`. The assertion in line 9 indicates `total` should begin in ownership mode `unavailable` and end in mode `local`; the assertion on the return of `total` in line 10 indicates

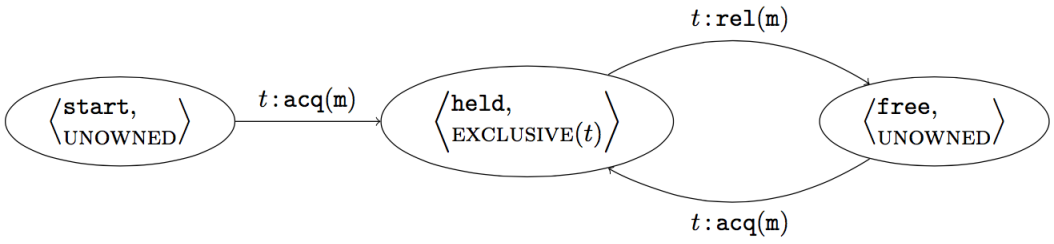


Figure 2.8: An automaton representing a simple locking discipline.

```
1  Globals: total : (start : unavailable)
2                   (acq(m) (held : local(current))
3                    rel(m) (free : unavailable))*
4  Locks:   m
5
6  Main:
7    mfork 5, Aux
8    mjoin Aux
9    acq m & total from unavailable to local
10   total & total in local
11
12 Aux:
13   acq m & total from unavailable to local
14   total = (total & total in local) + 1 & total in local
15   rel m & total from local to unavailable
```

Figure 2.9: A PolyCoSM program whose `Main` thread forks off `5` other `Aux` threads, each of which adds `1` to the global `total` value at the top. The annotation attached to `total` translates to the automaton in Figure 2.8.

`total` begins in mode `local` and ends in mode `local`. These assertions act as run-time checks that ensure obedience to the specified discipline.

Spector-Zabusky's specification language provides a clean representation of synchronization disciplines, and he demonstrates how to verify them in a restricted language. However, extending his system to a full language with references and aliasing poses a significant challenge. Furthermore, even in the restricted setting, using his specification language is difficult due to its high annotation cost. This would only increase when writing programs with more than one shared variable. It is often the case that many shared variables fall under the protection of a common synchronization object. In such cases, any operations called on the common synchronization object would require multiple distinct annotations to handle each variable's transitions.

Chapter 3

Overview

The main contributions of this thesis are: (1) an expressive, compact annotation language that may be used to specify synchronization disciplines on shared variables in multithreaded programs; and (2) a corresponding technique for dynamically enforcing conformance to a specified synchronization discipline. In this chapter, we provide an overview of the concept of synchronization disciplines, our motivation for specifying and enforcing such disciplines, and our contributions toward those ends. The chapter is presented as follows:

Section 3.1 introduces the concept of a synchronization discipline, presents examples of commonly used disciplines, and provides the motivation for specifying and enforcing such disciplines over traditional race detection.

Section 3.2 presents the design of our specification language for synchronization disciplines. In particular, we introduce a new abstraction called a *synchronization flow graph* for modeling synchronization disciplines.

Section 3.3 presents a corresponding technique for dynamically enforcing conformance to a specified discipline.

3.1 Synchronization Disciplines

When implementing multithreaded programs, programmers avoid race conditions by devising and adhering to various *synchronization disciplines*. In this section, we introduce the concept of a synchronization discipline, give several examples of commonly used disciplines, and discuss the benefits of specifying and enforcing such disciplines over traditional dynamic race detection.

3.1.1 What is a synchronization discipline?

A synchronization discipline is a policy governing accesses to a given shared variable that ensures race freedom. A simple and ubiquitous example is a locking discipline under which a specific lock is

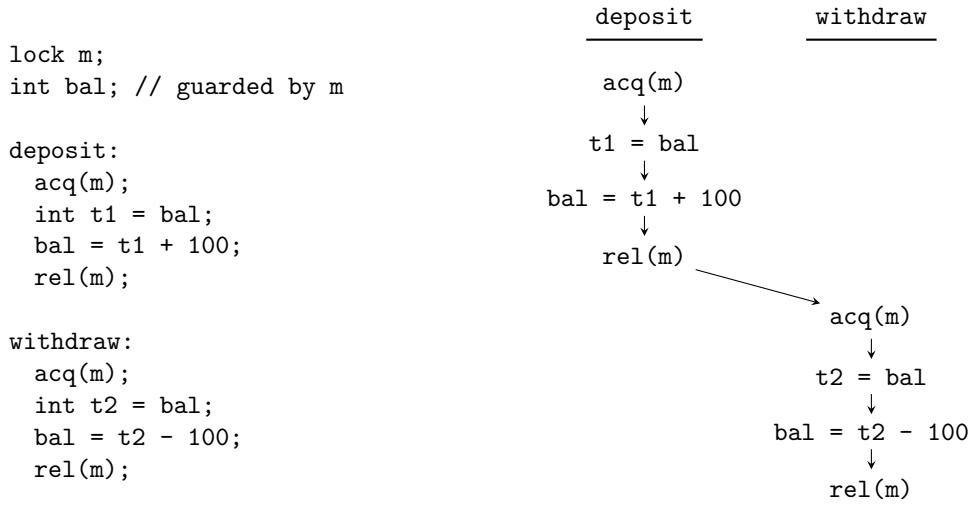


Figure 3.1: A bank account program using a locking discipline on `ball`.

used to "guard" the variable, which we used to fix the racy bank account program in the previous chapter; the fixed program is re-displayed in Figure 3.1. Under this discipline, each thread accesses the variable `bal` only when it holds the lock `m`. This ensures that only a single thread accesses `bal` at a time, thereby preventing any concurrent accesses.

In general, a program variable has no race conditions if and only if every access satisfies one of two properties: (1) the access is ordered by happens-before with every other access, or (2) the access is a read and concurrent only with other reads. A synchronization discipline on the variable guarantees this by assigning an implicit *access mode* at each point in execution. These modes may be either: (1) an exclusive access mode by a single thread, or (2) a read-shared access mode by a set of threads. The discipline further specifies synchronization mechanisms for transitioning from one access mode to the next.

We list some examples of commonly used disciplines:

Exclusive access The simplest discipline consists of a single exclusive access mode. For example, in a Java program, an object may be allocated in shared heap memory but with the intention of being accessed exclusively by the allocating thread.

Thread local Under this discipline, the variable is intended to be accessed exclusively by a thread, but the specific thread may not known and may change on different executions. For example, it may be known that each job in a job queue is processed by some worker thread, but not which thread in particular.

Initialize then read Some variables are only read after initialization. For example, in Figure 3.2, `Thread0` and `Thread1` perform different computations depending on the initialization of `x`. A `fork` operation orders the exclusive access by `main` with the subsequent read-shared access by `Thread0`

```
int x; // init then read

Thread0:
  y = f(x);

Thread1:
  z = g(x);

main:
  x = ...;
  fork(Thread0,Thread1);
```

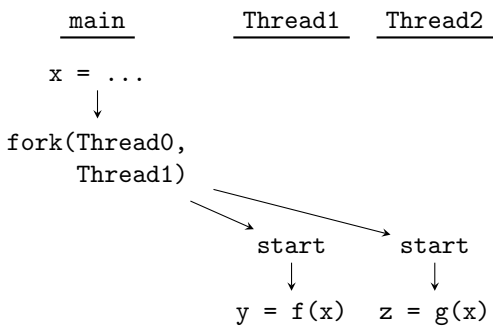


Figure 3.2: A program in which x is only read after initialization.

```
volatile v;
int x;

Thread0:
  loop {
    while (v != 0); // spin
    process(x);
    x = ...;
    v = 1;
  }

Thread1:
  loop {
    while (v != 1); // spin
    process(x);
    x = ...;
    v = 0;
  }
```

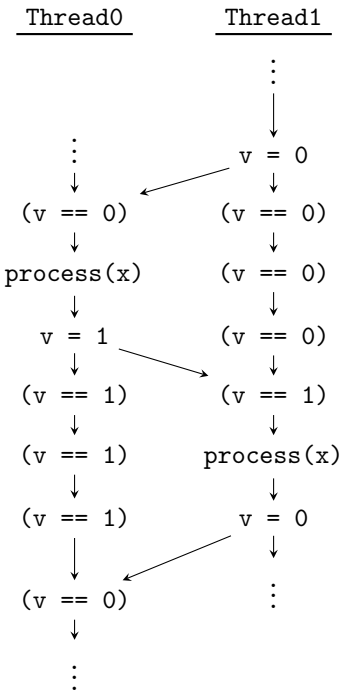


Figure 3.3: Two threads synchronized on a volatile flag.

and `Thread1`.

Guarded by lock Under the locking discipline on `bal` in Figure 3.1, the lock `m` be used to transition between different exclusive access modes on `bal`.

Volatile flag Threads may synchronize their behavior by setting and reading a shared volatile variable. For example, in Figure 3.3, each thread spins until the other thread flips the volatile flag `v`. The synchronization on `v` ensures ordering between the exclusive access modes on `x`.

Barrier synchronization Consider the particle simulation program in Figure 3.4, which employs a typical barrier-based discipline. Each of the particles `p0`, `p1`, `p2` is simulated by a correspondingly numbered thread. The simulation proceeds as a two-phase loop: first each thread reads the state of all particles and computes a new state for its own particle; then each thread updates the state of its particle. Accesses in one phase are ordered with those of another by synchronization on the barrier object `b`, thereby preventing any updates on a particle while it is being read. Under this discipline, each particle variable uses `b` to transition back and forth between read-shared access and exclusive access by its simulating thread.

Lock `m` then lock `n` The lock used to guard a shared variable may change over the course of execution. The lock change occurs in the form of a handshake as displayed in Figure 3.5: `Thread0` acquires the new lock `n` while still holding the old lock `m`.

3.1.2 Motivation

Programmers universally rely on synchronization disciplines to help manage the complexity of writing correct multithreaded programs. Put another way, an observed race can be ascribed to the failure to conform some intended discipline. This raises the question of whether one may specify and enforce disciplines directly.

The potential benefits of such an approach have already been partially demonstrated by the dynamic race detector Eraser [15]. Eraser's lockset algorithm enforces a specific synchronization discipline: within each execution, every shared variable must have at least one associated lock that is held on every access to the variable. In restricting itself to this fixed discipline, Eraser sacrifices completeness and cannot be used effectively on programs that rely on synchronization mechanisms other than locks. On programs that do ostensibly adhere to this discipline, however, Eraser holds two significant advantages over other dynamic race detectors.

First, it avoids the expensive vector clock computations typically used in precise dynamic race detection. Eraser remains today one of the simplest and fastest dynamic race detectors so far devised.

Second, by enforcing a stronger correctness property than race freedom, Eraser is capable of detecting certain *potential* race conditions, even if they have not explicitly manifested in the observed trace. For example, consider the two interleavings in Figure 3.6. The left interleaving is technically race-free. However, the programmer did not re-acquire the lock `m` before the last operation on `x` as intended. If `Thread1` had managed to acquire `m` before `Thread0`, then we would have observed the

```
particle p0,p1,p2;
barrier b;

Thread0:
  loop {
    doBarrier(b);
    particle q0
     = f(p0,p1,p2);
    doBarrier(b);
    p0 = q0;
  }

Thread1:
  loop {
    doBarrier(b);
    particle q1
     = f(p0,p1,p2);
    doBarrier(b);
    p1 = q1;
  }

Thread2:
  loop {
    doBarrier(b);
    particle q2
     = f(p0,p1,p2);
    doBarrier(b);
    p2 = q2;
  }
```

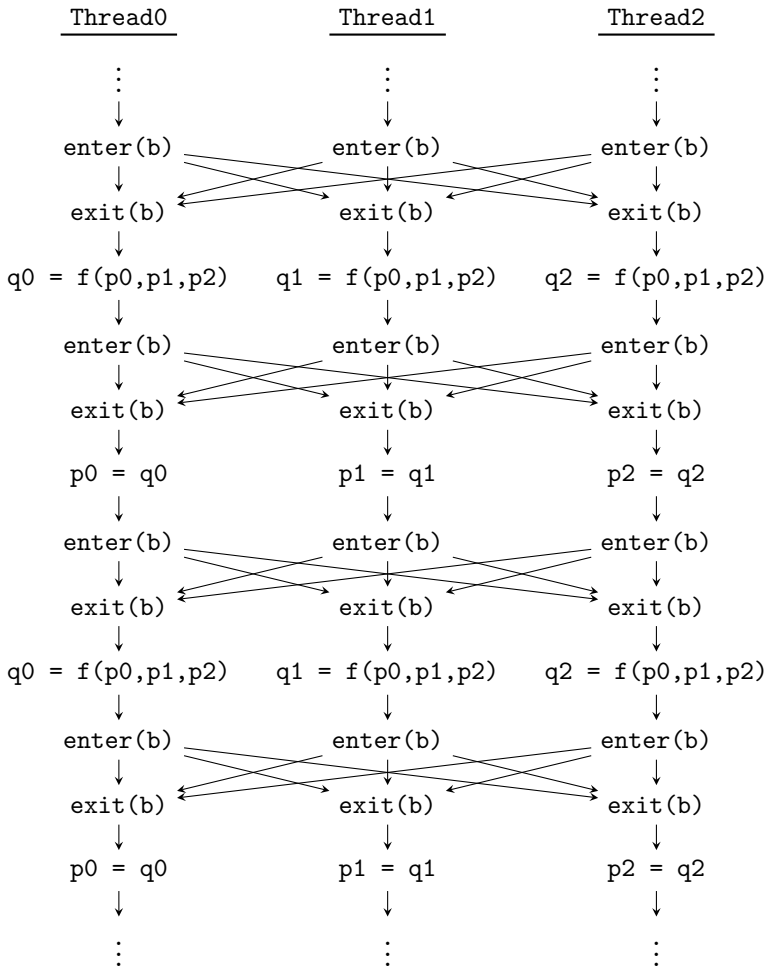


Figure 3.4: A particle simulator program.

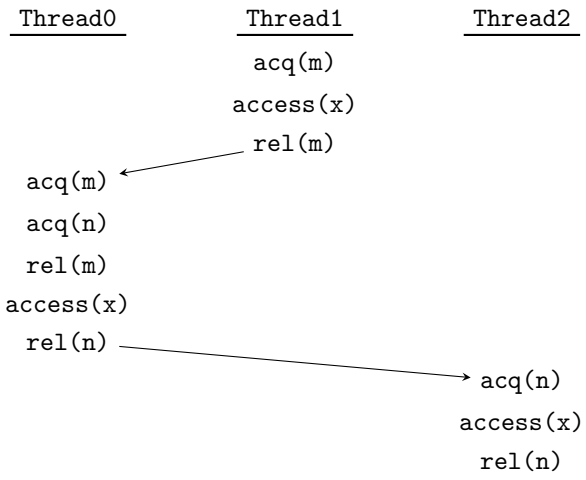


Figure 3.5: A lock handshake changing from `m` to `n`.

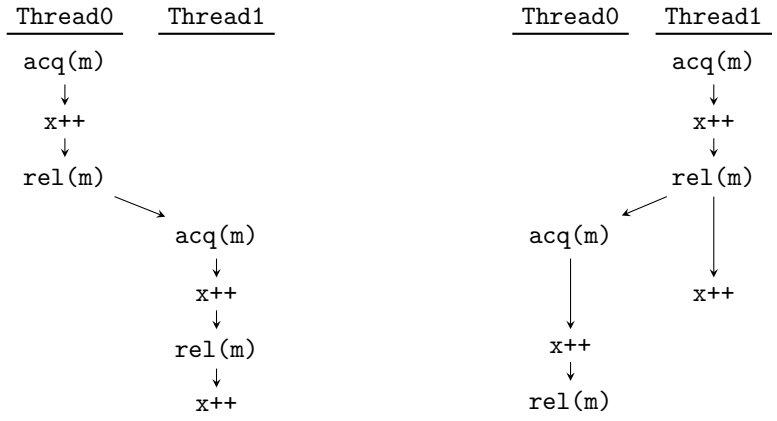


Figure 3.6: Two interleavings. The left interleaving is race-free, the right has a race condition.

right interleaving, in which there is a clear race condition. Whereas a precise dynamic race detector would only catch the symptomatic race condition upon its explicit appearance, Eraser would detect the source violation of the intended locking discipline on either interleaving. The greater detectability of a discipline violation translates to a greater confidence in the correctness of the analyzed code when no errors are reported—provided, of course, that Eraser's locking discipline was actually the one intended.

By devising a method of specifying and enforcing general synchronization disciplines, our ultimate goal is to extend the benefits of Eraser to race detection for all multithreaded programs. We surmise that synchronization disciplines are an instance where explicit specification helps more than it hurts. Given the error-proneness of multithreaded programming, such disciplines tend already to be at the front of the programmer's mind. At the same time, the difficulty of thinking concurrently sometimes means that the programmer's intent is itself incorrect; having a structured specification language reduces those sorts of errors.

Furthermore, specifying disciplines opens up possibilities of *modular* race detection. Often, it is a small subset of shared variables in a program that are untrusted and at risk of race conditions. In order to reduce the overhead of its analysis, a traditional dynamic race detector can be modified so that, rather than instrumenting every memory access, it instruments only accesses to those critical variables. However, it cannot avoid instrumenting every synchronization operation, since it does not have prior knowledge of which synchronization mechanisms are relevant. Although synchronization accounts for only about 3% of all program operations on average [4], the overhead on these operations can be quite significant. For example, FastTrack employs full vector clock computations on every synchronization operation; if the program employs many threads, this can incur overhead that is one or two orders of magnitude greater than that incurred by FastTrack's more typical epoch computations.

On the other hand, having discipline specifications on the variables allows for an enforcement analysis to ignore any operations on irrelevant synchronization mechanisms. For example, consider the bank account program in Figure 3.7, which extends our original bank account in Figure 3.1. In

```
lock m;
int bal; // guarded by m
int amt; // init then read

deposit:
  acq(m);
  int t1 = bal;
  int t2 = amt;
  bal = t1 + t2;
  rel(m);

withdraw:
  acq(m);
  int t3 = bal;
  int t4 = amt;
  bal = t3 - t4;
  rel(m);

main:
  amt = ...; // init
  fork(deposit,withdraw);
  join(deposit,withdraw);
```

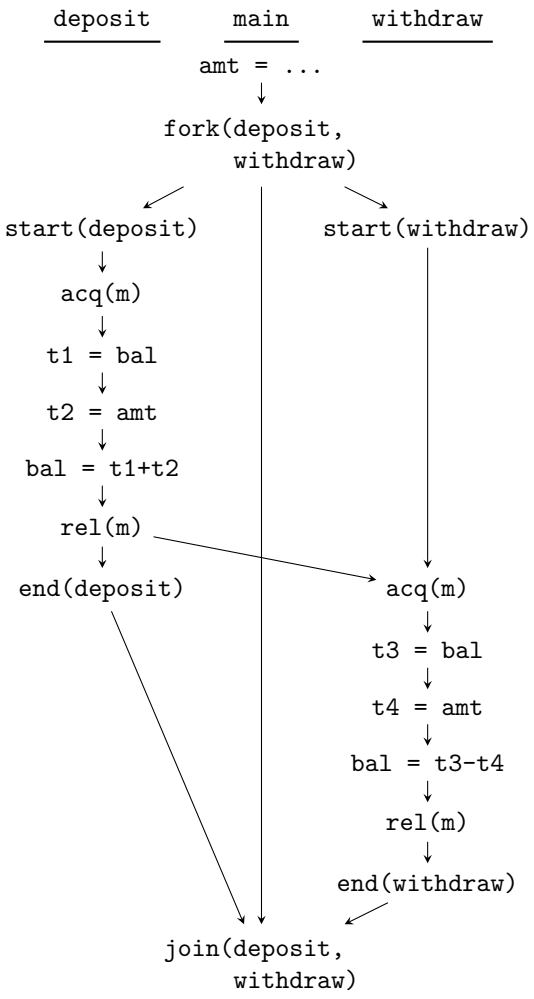


Figure 3.7: An extended bank account program and a sample execution.

the extended version, the threads `deposit` and `withdraw` now modify `bal` by a varying amount, depending on the value with which the `main` thread initializes the new variable `amt`. The variable `bal` is still guarded by lock `m`; the new variable `amt` is protected by the initialize-then-read discipline we described earlier in this section. Figure 3.8 highlights the minimal subsets of operations relevant to each discipline. Having access to each specified discipline means that an enforcement analysis need only instrument these minimal subsets.

3.2 Specifying Disciplines

In this section, we motivate and present the design of a new specification language for synchronization disciplines. We model disciplines as *synchronization flow graphs* (SFGs), which capture a high-level control flow between access modes for the underlying variable; each term in our specification

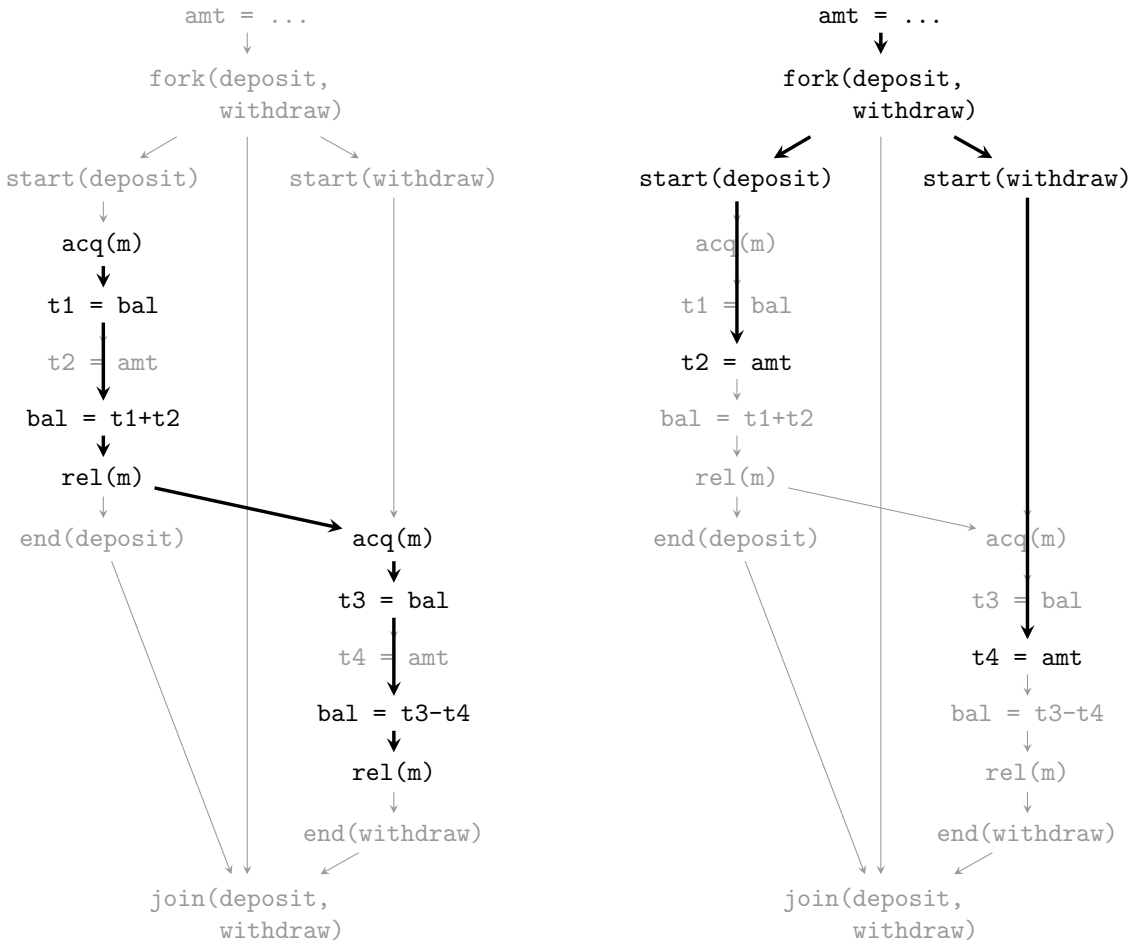


Figure 3.8: Left: the subtrace relevant to the locking discipline on `bal`. Right: the subtrace relevant to the initialize-then-read discipline on `amt`.

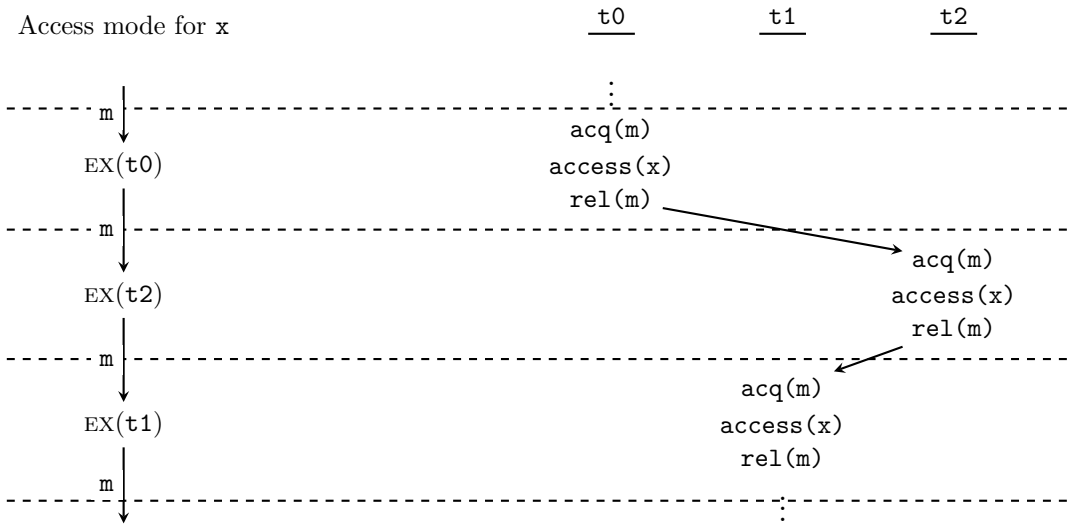


Figure 3.9: Phases in an execution conforming to a locking discipline for the variable x.

language represents such a synchronization flow graph. We begin by motivating this choice of model for disciplines and describing how a program execution *conforms* to a discipline with respect to this model. We then present our specification language and show how an SFG may be compactly expressed using our language.

3.2.1 Synchronization flow graphs

In the previous section, we presented several examples of commonly used synchronization disciplines. We described each discipline as consisting of two main components: access modes and synchronization mechanisms. The access modes, each of which may be exclusive access by a single thread or read-shared access by a set of threads, constitute all the ways a shared variable may be accessed without synchronization in a race-free manner. The synchronization mechanisms (if any) in a discipline are used to order accesses performed in different access modes allowed by the discipline.

Notably, synchronization disciplines have a *sequential* run-time structure. While the individual operations relevant to a discipline may be performed concurrently, we can group them into a higher-order sequence of execution phases that correspond to distinct access modes—that is, the accesses within each execution phase are those allowed by the current access mode. Operations on the relevant synchronization mechanisms are performed at the boundaries of these phases to ensure safe transition from one access mode to another. We highlight this structure for a few disciplines in Figures 3.9, 3.10, and 3.11. These observed sequences of access modes may, of course, vary across different executions but typically follow simple patterns. For example, under the locking discipline in Figure 3.9, the particular order of threads with exclusive access may vary, but we should expect to observe only exclusive access modes and transitions between modes via the lock m.

Our specification language directly encodes this high-level sequential structure by modeling disciplines as *synchronization flow graphs*. An SFG is simply a directed graph in which vertices are

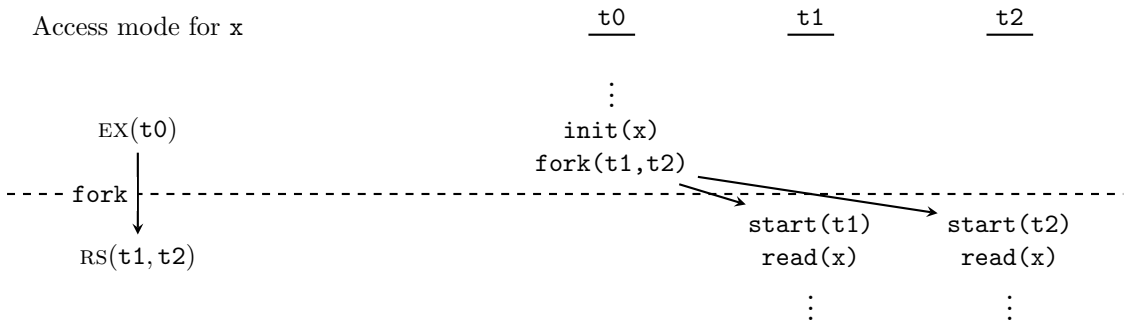


Figure 3.10: Phases in an execution conforming to the initialize-then-read discipline for the variable x.

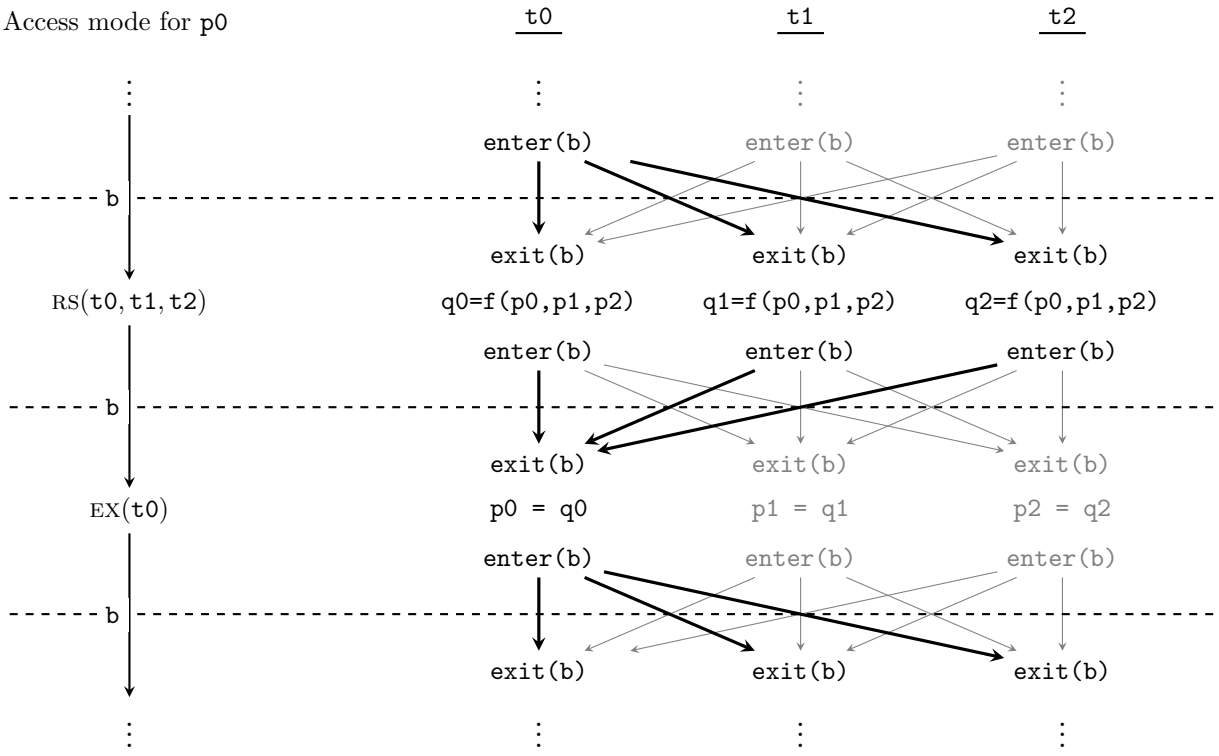
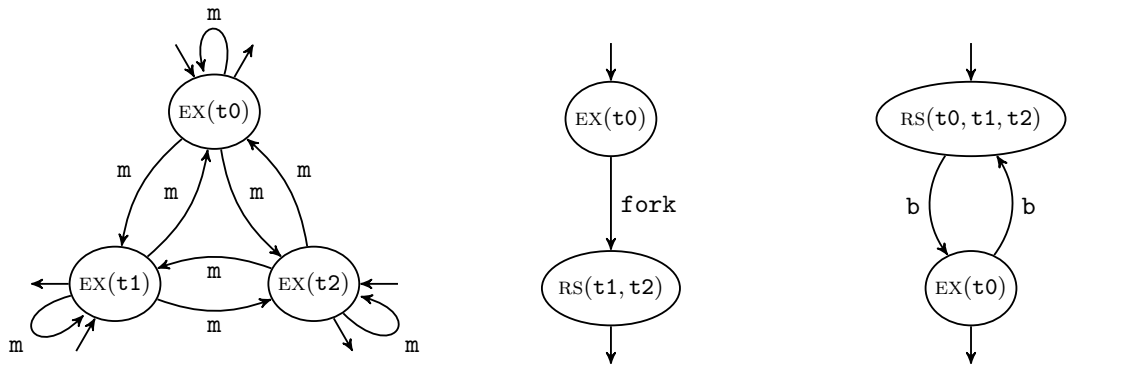


Figure 3.11: Phases in an execution conforming to a barrier discipline for the variable p0.



(a) Locking discipline in Figure 3.9

(b) Initialize-then-read discipline in Figure 3.10

(c) Barrier discipline in Figure 3.11

Figure 3.12: Synchronization flow graphs

labeled with access modes and edges are labeled with synchronization mechanisms. A subset of the vertices are designated as *in-modes* and *out-modes* and serve as entry and exit points when composing disciplines, which we discuss in the second half of this section; the in-modes also designate the possible starting access modes. An SFG directly encodes the high-level structure of the run-time behavior stipulated by a synchronization discipline. In particular, the sequences of access modes described above are encoded as paths on SFGs that begin with an in-mode. See Figure 3.12 for the SFGs that model the disciplines in Figures 3.9, 3.10, and 3.11; in-modes and out-modes are designated by the dangling edges leading in and out, respectively, of access modes.

Although SFGs abstract away details of individual program operations, it is easy to derive these details from their structure. An SFG may be viewed as a sort of nondeterministic finite automaton for the underlying variable. The current access mode dictates which and how threads may access the variable. Specifically, a variable at access mode $\textsc{ex}(t)$ for some thread t should only be accessed by t; a variable at access mode $\textsc{rs}(T)$ for some thread set T should only be read by threads in T. Let $Tid(a)$ denote the associated thread set of a; for any access modes a, a' and synchronization mechanism z, a transition $a \xrightarrow{z} a'$ is made when every thread $t \in Tid(a)$ performs a "send" operation on z, then every thread $t' \in Tid(a')$ performs a "receive" operation on z. For example, these are the synchronization operations performed at the boundaries of each execution phase in Figures 3.9, 3.10, and 3.11. Importantly, the "send" operations on z must be performed by t after every access by t in mode a, and the "receive" operation must be performed by t' before every access by t' in mode a'. This ensures that accesses performed in different access modes are ordered by happens-before.

SFGs are nondeterministic in that there may be multiple candidate access modes at any point in program execution. A simple example is captured by the locking discipline in Figure 3.12a. In this discipline, every access mode is an in-mode and therefore a candidate access mode before the first access to the underlying variable. A more subtle example is captured in the barrier discipline in Figure 3.12c. Suppose this discipline is protecting a variable `x`. The barrier object `b` may be used to protect a different variable `y` as well, so some operations on `b` might not be intended to change the access mode for `x`. There may occur a scenario in which `x` is in mode $\textsc{rs}(\texttt{t0}, \texttt{t1}, \texttt{t2})$, operations

(a) $\mathcal{EX}(t)$ (b) $\mathcal{RS}(T)$

Figure 3.13: Base graphs.

on `b` are performed, then `x` is read by thread `t0`. In this case, it is not clear the program should transition to access mode EX(`t0`) or ignore the operations on `b` and remain in mode RS(`t0`, `t1`, `t2`).

Finally, although we have described SFGs as nondeterministic finite automata for exposition purposes, we emphasize that they are never directly executed to enforce a discipline. They serve only to specify at a high level the run-time behaviors permitted by a discipline. Later, we describe how SFGs may be automatically refined into a more detailed computation model called a Petri net that is utilized for discipline enforcement.

3.2.2 Specification language

SFGs provide a simple model for synchronization disciplines that closely aligns with our high-level intuition. However, the obvious approach to specifying an SFG, which would be to list each individual access mode and transition, is tedious for even simple disciplines. This is undesirable in a specification language. Fortunately, the space of all possible SFGs is overly general for the purpose of specifying common or useful disciplines. By restricting the expressivity of our specification language to a space of useful disciplines, we can specify disciplines more naturally and compactly as compositions of smaller disciplines.

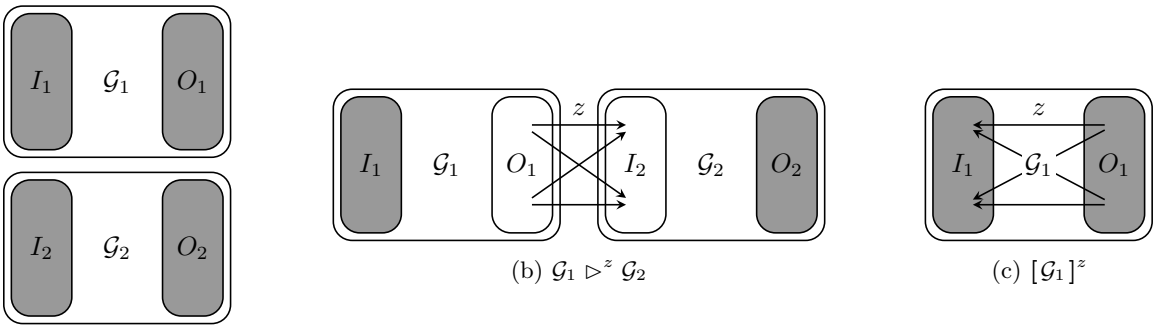
Every SFG expressed by our specification language can be constructed from a set of base graphs and three simple graph operators. The base graphs are:

- $\mathcal{EX}(t)$ for all threads t, each of which consists of a single access mode EX(t); and
- $\mathcal{RS}(T)$ for all thread sets T, each of which consists of a single access mode RS(T).

Essentially, these are the access modes themselves. These graphs are depicted in Figure 3.13. Our three graph operators, displayed schematically in Figure 3.14, are the following:

- the binary *disjunction* operator $\cdot \sqcup \cdot$ in Figure 3.14a, which simply takes the disjoint union of its graph operands and presents a choice between the two operands;
- the binary sequence operator $\cdot \rhd^z \cdot$ in Figure 3.14b, which adds edges labeled by z from the out-modes of the left operand to the in-modes of the right operand; and
- the unary loop operator $[\cdot]^z$ in Figure 3.14c, which adds edges labeled by z from the out-modes to the in-modes of its single operand.

Altogether, this gives us a simple, text-based specification language for synchronization disciplines with the following grammar:



(a) $\mathcal{G}_1 \sqcup \mathcal{G}_2$

(b) $\mathcal{G}_1 \rhd^z \mathcal{G}_2$

(c) $[\mathcal{G}_1]^z$

Figure 3.14: Graph operators. Each pair of boxes labeled I_i and O_i represent the sets of in-modes and out-modes, respectively, of the SFG $\mathcal{G}_i$ represented by the larger enclosing box. For each operator, I_i and O_i are shaded gray if they are part of the sets of in-modes and out-modes, respectively, of the total SFG.

$$
\begin{array}{lcll}
d & \in & \mathit{Discipline} & \rightarrow \quad \mathit{Discipline} \sqcup \mathit{Discipline} \\
 & & & \mid \quad \mathit{Discipline} \rhd^z \mathit{Discipline} \\
 & & & \mid \quad [\mathit{Discipline}]^z \\
 & & & \mid \quad \mathit{AccessMode} \\
a & \in & \mathit{AccessMode} & \rightarrow \quad \texttt{EX}(t) \mid \texttt{RS}(T)
\end{array}
$$

$$z \in \mathit{SyncMech} \qquad t \in \mathit{Thread} \qquad T \subseteq \mathit{Thread}$$

Here, the text specifications $\texttt{EX}(t)$ and $\texttt{RS}(T)$ represent the base graphs $\mathcal{EX}(t)$ and $\mathcal{RS}(T)$, and the text operators $\cdot \sqcup \cdot$, $\cdot \rhd^z \cdot$, and $[\cdot]^z$ represent the matching operators on graphs. While simple, our specification language is capable of expressing a wide variety of commonly used disciplines. In the table in Figure 3.15, we display sample specifications and corresponding SFGs for all the common disciplines described in the previous section.

3.3 Enforcing Disciplines

SFGs capture the high-level structure of a synchronization discipline in terms of access modes and transitions via synchronization mechanisms. They do not explicitly encode sufficient detail with respect to actual program operations to be directly useful when enforcing discipline conformance. However, they may be refined into such a suitable structure in a general, mechanical fashion. In particular, we derive from each SFG a more detailed computation model called a *Petri net*. This Petri net is then simulated at run time in a dynamic program analysis to enforce conformance to a specified discipline. In this section, we introduce the Petri net model and our Petri net constructions from SFGs.

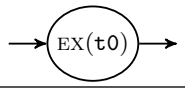
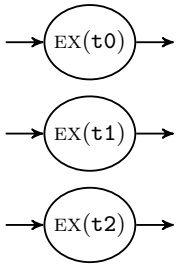
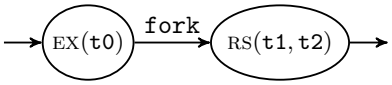
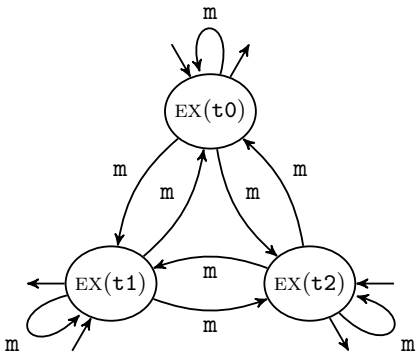
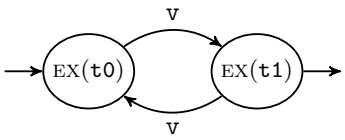
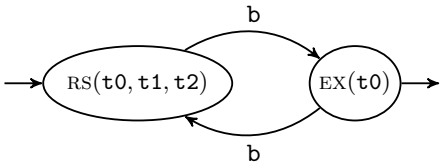
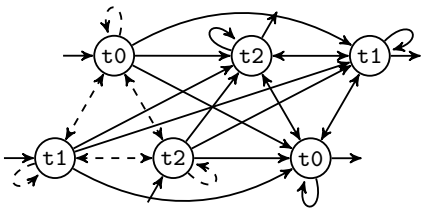
Discipline	Specification	SFG
`exclusive`	$\texttt{EX(t0)}$	
`thread-local`	$\texttt{EX(t0)} \sqcup \texttt{EX(t1)} \sqcup \texttt{EX(t2)}$	
`init-then-read`	$\texttt{EX(t0)} \rhd^{\texttt{fork}} \texttt{RS(t1, t2)}$	
`lock(m)`	$[\texttt{thread-local}]^{\texttt{m}}$	
`volatile(v)`	$[\texttt{EX(t0)} \sqcup \texttt{EX(t1)}]^{\texttt{v}}$	
`barrier(b)`	$\texttt{RS(t0, t1, t2)} \rhd^{\texttt{b}} \texttt{EX(t0)}$	
`lock(m)` then `lock(n)`	$\texttt{lock(m)} \rhd^{\texttt{n}} \texttt{lock(n)}$	

Figure 3.15: A table of the synchronization disciplines listed in Section 3.1.1, their specifications, and their SFGs.

3.3.1 Introducing Petri nets

A *Petri net* is a mathematical model of computation that is particularly well-suited for modeling concurrent and distributed systems. Petri nets may be understood as a generalization of finite state automata, which we take a moment to review in order to highlight this connection. A finite state automaton has the static structure of a directed graph, where each vertex represents a state of computation, and each labeled edge or transition represents a computation event that updates the state from one to another. We can view the computation itself as playing a certain token game on top of this structure. Between computation steps, a token occupies some state and enables every transition that leads out of that state. At each step, upon observing the corresponding event, an enabled transition fires and sends the token from its input state to its output state.

A similar token game forms the intuitive basis for computation on Petri nets. A Petri net has the static structure of a *net* or directed bipartite graph. One partite class consists of *places*, which correspond to conditions; the other consists of *transitions*, which correspond to events. Each transition t has a set of input places from which edges lead into t and a set of output places into which edges lead from t; these correspond, respectively, to the pre-conditions and post-conditions of the underlying event.

Where a token game state for an automaton consists of a single token occupying a single state, a net-based token game state may consist of many tokens occupying many places. In this thesis, we restrict our attention to *elementary* Petri nets, which means that tokens are identical and each place may be occupied by at most one token at a time. A transition is *enabled* if there is a token occupying each of its input places and no tokens occupying its output places. An enabled transition may *fire* upon observing the corresponding event. The firing of a transition removes the tokens from its input places and adds a token to each of its output places.

Petri nets provide a natural way to express many concurrent systems. Consider, for example, the Petri net modeling the producer/consumer problem in Figure 3.16. Circles denote places; squares denote transitions; the dots occupying the top two places denote tokens in their initial positions for this system. The Petri net may be viewed as consisting of three components: the producer, the consumer, and the buffer by which they communicate. The 'produce' and 'consume' transitions may fire concurrently; however, the producer must wait until the buffer is empty before it can 'push', while the consumer must wait until the buffer is full before it can 'pop'. In general, Petri nets easily support the expression of many different forms of behavior; see Figure 3.17.

3.3.2 Deriving Petri nets from synchronization flow graphs

Consider again the execution of the particle simulation program in Figure 3.11 and the barrier discipline in Figure 3.12c to which the execution conforms. The SFG structure reflects the high-level sequence of access modes, but does not encode the many possible interleavings of operations by different threads within each phase.

Petri nets are well-suited to capture such concurrency and can be automatically constructed from SFGs. Figure 3.18 shows a schematic for the derivation procedure on an arbitrary edge $a \xrightarrow{z} b$ of an SFG. Each access mode is expanded into a set of `access` places, one for every thread of the

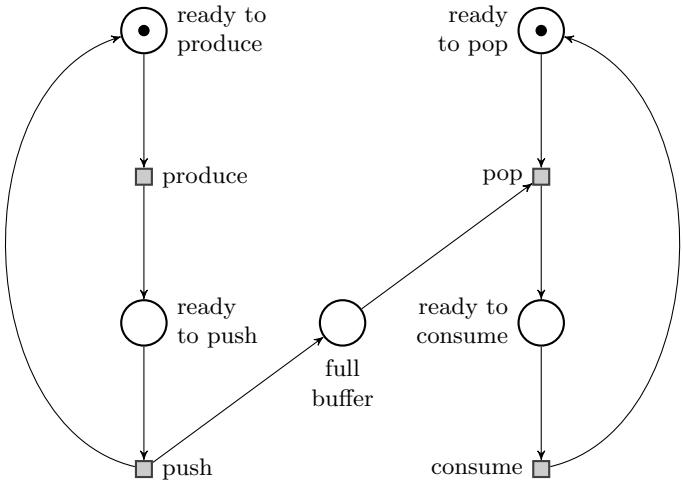


Figure 3.16: A Petri net modeling the producer/consumer problem with a one-element buffer.

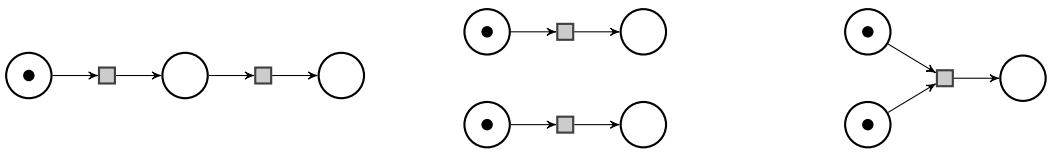


Figure 3.17: Petri net gadgets capturing (a) sequential, (b) concurrent, and (c) synchronized behavior.

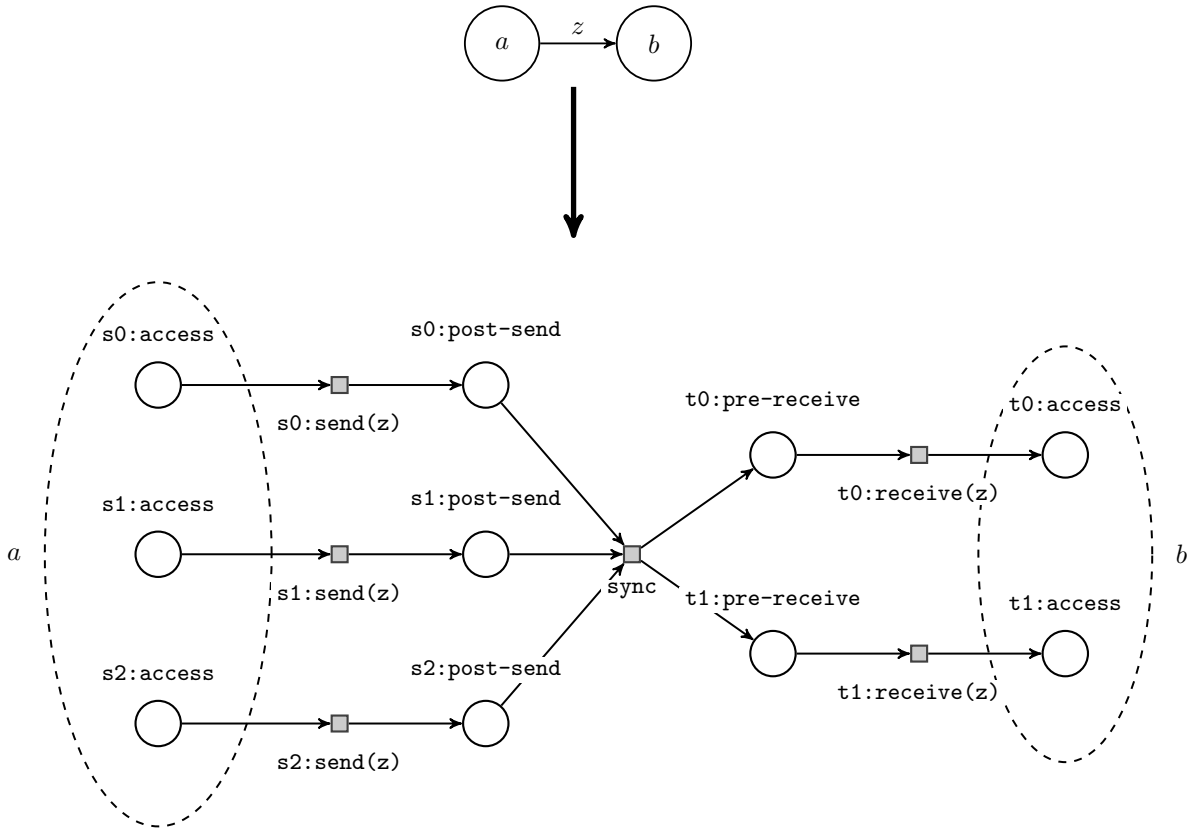


Figure 3.18: Derivation schematic for an arbitrary edge $a \xrightarrow{z} b$ in an SFG.

mode. Tokens in `access` places indicate that the underlying variable may be accessed. Each thread in access mode a has its own sequential track in the first half of the derived edge. This track consists of a single transition corresponding to a "send" operation on z, which transfers a token from an `access` place to a `post-send` place. When a token occupies a `post-send` places, this indicates that the corresponding thread has reached the boundary of the current access phase, and must wait until every `post-send` place is occupied by a token. When this occurs, this means that every thread has performed a "send" operation on z, and therefore the threads in b may begin performing "receive" operations. Accordingly, the `sync` transition fires and places tokens at the `pre-rcv` places. Tokens in these places indicate that execution has entered the next access phase, but cannot begin accessing the variable until they have performed the required "receive" operations. Upon doing so, the tokens occupy the `access` places of b, and the corresponding threads are now permitted to perform accesses.

For a concrete example, see the derived net of our barrier discipline in Figure 3.19. The circular tokens form the unique initial marking of this discipline. In general, however, the initial marking may cover the set of places derived from any in-mode of the SFG.

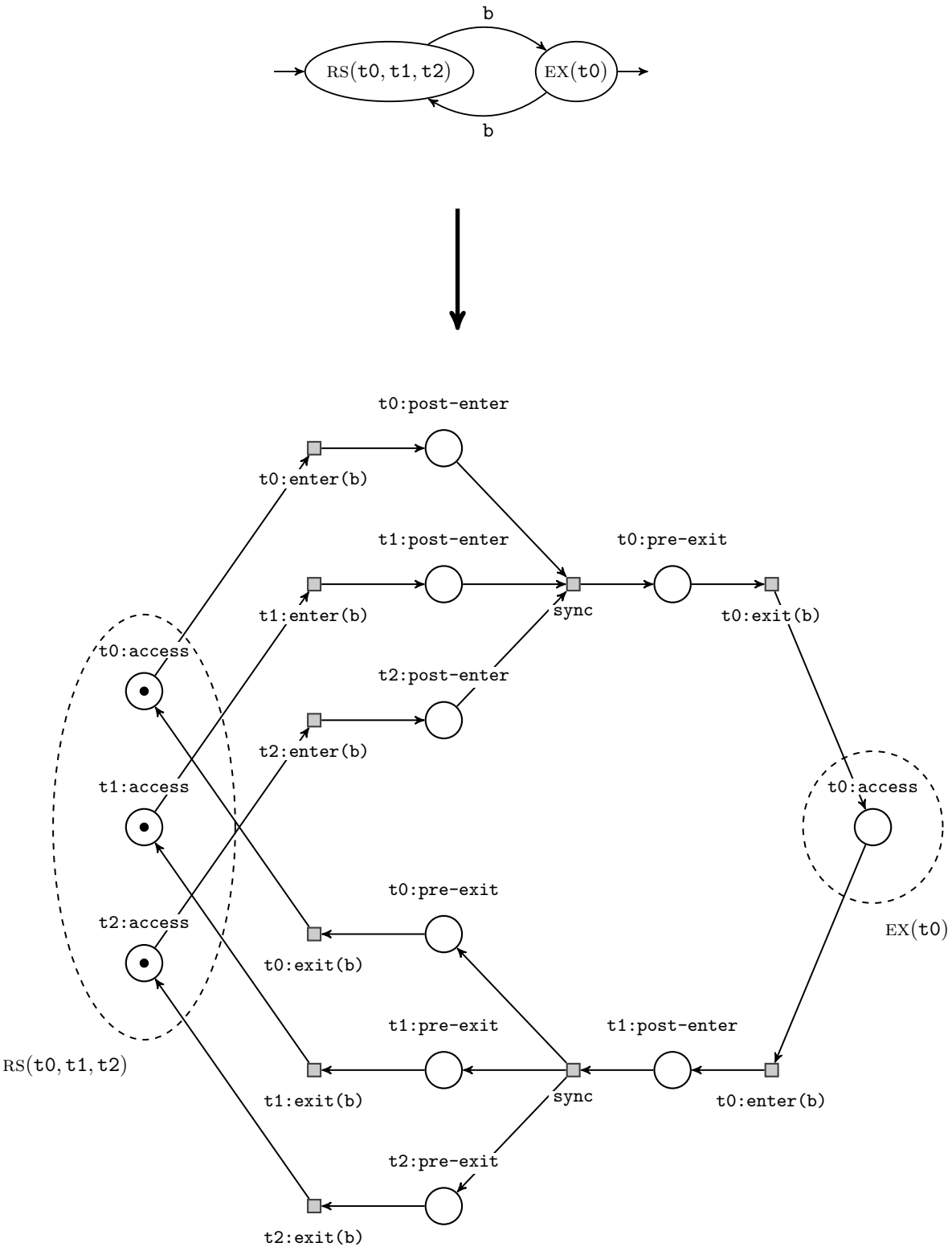


Figure 3.19: The derivation of a Petri net from an SFG.

3.4 Chapter Summary

Programmers avoid race conditions by devising and adhering to various synchronization disciplines. Specifying and enforcing these disciplines has many potential benefits, including detecting race conditions that would manifest in different interleavings as well as efficient, modular race detection.

We have introduced a new abstraction for modeling synchronization disciplines called a synchronization flow graph. SFGs directly encode the high-level execution structure of a discipline in terms of access modes and transitions via synchronization mechanisms. We have shown how SFGs may be expressed compactly using a small set of base graphs and graph operators, and have presented modeled several commonly used disciplines as SFGs.

Finally, we have introduced *Petri nets*, a computation model that is well-suited for reflecting concurrency. We have described how an SFG may be automatically refined into a more detailed Petri net structure, which is to be used to enforce disciplines in a dynamic program analysis presented in the next chapter.

Chapter 4

Analysis

In this chapter, we formalize and verify the technique presented in Chapter 3. A majority of the chapter is devoted to precisely defining new concepts. On the specification side, these include SFGs, the specification grammar for SFGs, and how a program execution conforms to an SFG. On the enforcement side, these include the construction of a corresponding Petri net from a specified SFG, and a dynamic program analysis that leverages the constructed Petri net to enforce conformance to the specified SFG. Given these definitions, we prove that conformance to an SFG implies race freedom, and that our dynamic analysis is sound with respect to conformance, i.e., it catches all non-conforming traces.

We develop these ideas with respect to a restricted program model with only a single memory location. This restriction is for the purpose of presentation clarity, as the details of handling multiple memory locations are simple and orthogonal to the key ideas of our formal analysis. Later, in Chapter 5, we explicitly cover those details.

This chapter is presented as follows:

Section 4.1 presents the formal program model upon which we base our analysis.

Section 4.2 develops our discipline specification language and the underlying SFG constructions that model the disciplines, defines discipline conformance, and verifies the properties above of conforming traces.

Section 4.3 shows how an SFG may be automatically refined to a Petri net that is used in our dynamic analysis and shows that our analysis correctly enforces discipline conformance.

4.1 Program Model

A program consists of a number of concurrently executing threads $s, t \in Thread$ that manipulate locks $m \in Lock$, volatile variables $v \in Volatile$, barriers $b \in Barrier$, and a single memory location. A *trace* $\omega \in Trace$ captures an execution of a multithreaded program as a sequence of operations

$$
\begin{array}{rcl}
\omega \in \mathit{Trace} & \to & \mathit{Operation}^* \\
o \in \mathit{Operation} & \to & \mathit{MemOp} \mid \mathit{SyncOp} \\
\\
\mathit{MemOp} & \to & \mathtt{rd}(t) \mid \mathtt{wr}(t) \\
\\
\mathit{SyncOp} & \to & \mathit{SyncSnd} \mid \mathit{SyncRcv} \\
\mathit{SyncSnd} & \to & \mathtt{rel}(t,m) \mid \mathtt{vwr}(t,v) \mid \mathtt{barEnt}(t,b) \mid \mathtt{fork}(t,S) \mid \mathtt{end}(s) \\
\mathit{SyncRcv} & \to & \mathtt{acq}(t,m) \mid \mathtt{vrd}(t,v) \mid \mathtt{barEx}(t,b) \mid \mathtt{start}(s) \mid \mathtt{join}(t,S)
\end{array}
$$

$s,t \in \mathit{Thread} \qquad S,T \subseteq \mathit{Thread} \qquad m \in \mathit{Lock} \qquad v \in \mathit{Volatile} \qquad b \in \mathit{Barrier}$

Figure 4.1: Program model.

performed by the various threads as defined in Figure 4.1. The set of operations that a thread t can perform are:

- $\mathtt{rd}(t)$ and $\mathtt{wr}(t)$, which read and write a value from the single memory location;
- $\mathtt{acq}(t,m)$ and $\mathtt{rel}(t,m)$, which acquire and release a lock $m \in \mathit{Lock}$;
- $\mathtt{vwr}(t,v)$ and $\mathtt{vrd}(t,v)$, which write and read a volatile variable $v \in \mathit{Volatile}$;
- $\mathtt{barEnt}(t,b)$ and $\mathtt{barEx}(t,b)$, which enter and exit a barrier $b \in \mathit{Barrier}$;
- $\mathtt{fork}(t,S)$, which creates a new set S of threads s, each of begins execution by performing $\mathtt{start}(s)$; and
- $\mathtt{join}(t,S)$, which stops the execution of t until every thread $s \in S$ concludes its execution by performing $\mathtt{end}(s)$.

For each operation $o \in \mathit{Operation}$, we write $\mathit{Tid}(o)$ to denote the thread that performs it. We assume each operation in a trace ω has an implicit positional index that distinguishes it from other occurrences of the same operation within ω; when needed, we capture these positional indices externally by writing $\omega = o_1 \ldots o_n$.

4.1.1 Synchronization mechanisms

The set

$$
\begin{array}{rcl}
\mathit{SyncMech} & := & \mathit{Lock} \cup \mathit{Volatile} \cup \mathit{Barrier} \\
& & \cup\ \{\mathtt{fork}(t,T) \mid t \in \mathit{Thread},\ T \subseteq \mathit{Thread}\} \\
& & \cup\ \{\mathtt{join}(t,T) \mid t \in \mathit{Thread},\ T \subseteq \mathit{Thread}\}
\end{array}
$$

denotes the set of all *synchronization mechanisms* available in our program model. As the core of our analysis is concerned with the correct usage of synchronization mechanisms, it is convenient to be able to reason about all synchronization operations in a general way. To this end, given threads

$\texttt{snd}(t, m)$	$\equiv \texttt{rel}(t, m)$	$\texttt{rcv}(t, m)$	$\equiv \texttt{acq}(t, m)$
$\texttt{snd}(t, v)$	$\equiv \texttt{vwr}(t, v)$	$\texttt{rcv}(t, v)$	$\equiv \texttt{vrd}(t, v)$
$\texttt{snd}(t, b)$	$\equiv \texttt{barEnt}(t, b)$	$\texttt{rcv}(t, b)$	$\equiv \texttt{barEx}(t, b)$
$\texttt{snd}(t, \texttt{fork}(t, T))$	$\equiv \texttt{fork}(t, T)$	$\texttt{rcv}(t, \texttt{fork}(t, T))$	$\equiv \texttt{start}(t)\ (t \in T)$
$\texttt{snd}(t, \texttt{join}(t, T))$	$\equiv \texttt{end}(t)\ (t \in T)$	$\texttt{rcv}(t, \texttt{join}(t, T))$	$\equiv \texttt{join}(t, T)$

Figure 4.2: Operations denoted by $\texttt{snd}(t, z)$ and $\texttt{rcv}(t, z)$ for all $t \in Thread$ and $z \in SyncMech$.

$t \in Thread$ and mechanism $z \in SyncMech$, we use $\texttt{snd}(t, z)$ and $\texttt{rcv}(t, z)$ to denote the operations invoking z in $SyncSnd$ and $SyncRcv$, respectively. Specifically, we have the identities listed in Figure 4.2.

Each type of synchronization mechanism has ordering semantics that impose a well-formedness criterion on observable traces. A trace is considered to be *well-formed* if it satisfies the following criteria:

- No thread acquires any lock previously acquired but not released by a thread. No thread releases a lock it did not previously acquire.
- Each $b \in Barrier$ is implicitly indexed by some thread set $T \subseteq Thread$ with which b was initialized. If $t \notin T$, then t never performs an operation on b. Thread t may exit b only if every thread in T has entered b, and t has not performed any operation since last entering b.
- The first and last operations performed by any thread t are $\texttt{start}(t)$ and $\texttt{end}(t)$, respectively. Thread t never performs $\texttt{start}(t)$ or $\texttt{end}(t)$ more than once each.
- If a thread s performs $\texttt{fork}(s, T)$, then every operation by a thread $t \in T$ occurs after it.
- If a thread t performs $\texttt{join}(t, S)$, then every operation by thread $s \in S$ occurs before it.

4.1.2 Happens-before relation

The *happens-before* relation relates pairs of operations in an execution trace that could not have occured at the same time. Precisely, the happens-before relation for a trace $\omega = o_1 \dots o_n$ is the smallest binary relation $\prec_\omega$ on operations in ω such that:

- (program order) $o_i \prec_\omega o_j$ if $i < j$ and $Tid(o_i) = Tid(o_j)$;
- (synchronization order) $o_i \prec_\omega o_j$ if $i < j$, $o_i = \texttt{snd}(s, z)$, and $o_j = \texttt{rcv}(t, z)$; [1]
- (transitivity) $o_i \prec_\omega o_j$ if there is an operation o_k such that $o_i \prec_\omega o_k$ and $o_k \prec_\omega o_j$.

Two operations o, o' in ω are *concurrent* if $o \not\prec_\omega o'$ and $o' \not\prec_\omega o$. If $o, o' \in MemOp$, they are called *conflicting* if at least one of them is a write operation. They form a *race condition* if they are both concurrent and conflicting. A trace is called *race-free* if it has no operations that form a race condition.

[1] As mentioned in Section 2.2, any conflicting pair of accesses to a volatile variable is technically ordered by happens-before. However, only the ordering between `vwr-vrd` pairs can affect program execution.

4.2 Specifying Disciplines

In the previous chapter, we showed that a program that correctly follows a synchronization discipline on a given variable can partition its execution into a sequence of phases. In each phase, memory operations follow some safe access pattern, while appropriate synchronization operations are invoked to transition safely between phases. We captured the set of all such run-time sequences for a given discipline as paths within a synchronization flow graph. We then discussed how the graphs corresponding to commonly used disciplines could be constructed and expressed in a precise and compact manner.

In this section, we formally develop these ideas with respect to our program model. We begin by formalizing synchronization flow graphs and the operators that give us a compact specification grammar. We then define how a trace in our program model conforms to any such specified discipline, and verify that conformance is indeed a property worth enforcing by showing it implies race freedom. Finally, we revisit our graph operators and refine their definitions in order to prevent constructing any SFGs that use synchronization mechanisms in a non-idiomatic manner, or are not relevant to well-formed traces.

4.2.1 Synchronization flow graphs

A *synchronization flow graph*, or SFG for short, is a four-tuple $\mathcal{G} = (A, S, I, O)$, where

- A is a set of *access modes*, where each access mode is $\textsc{ex}(t)$ for some $t \in Thread$ or $\textsc{rs}(T)$ for some $T \subseteq Thread$;
- $S \subseteq A \times SyncMech \times A$ is a set of transitions $a \xrightarrow{z} a'$ between access modes $a, a' \in A$ and labeled by synchronization mechanisms $z \in SyncMech$;
- $I \subseteq A$ is a set of *in-modes*; and
- $O \subseteq A$ is a set of *out-modes*.

Like trace operations, we assume that every access mode has an implicit index that distinguishes it from other access modes of the same type. The in-modes designate the possible starting points in $\mathcal{G}$; both the in-modes and out-modes of $\mathcal{G}$ form the interface points at which $\mathcal{G}$ may be combined with other SFGs. Given an access mode a of $\mathcal{G}$, we write $Tid(a)$ to denote the underlying thread set. A path through $\mathcal{G}$ is written in the form $a_0 \xrightarrow{z_1} a_1 \xrightarrow{z_2} \cdots \xrightarrow{z_{n-1}} a_{n-1} \xrightarrow{z_n} a_n$. If a_0 is an in-mode, then the path is called *complete*.

Although SFGs provide a useful model for synchronization disciplines, expressing them in the tuple form above can be tedious. This is undesirable in a specification language. Fortunately, the space of all possible SFGs is overly general for the purpose of specifying common or useful disciplines. By restricting the expressivity of our specification language to a space of useful disciplines, we can specify disciplines more naturally and compactly using a small set of base graphs and graph operators.

The set of base graphs consists of individual access modes with no additional edges; these are the graphs $\mathcal{EX}(t)$ and $\mathcal{RS}(T)$ for all $t \in Thread$, $T \subseteq Thread$ defined in Figure 4.3. All other SFGs are

constructed inductively from the base graphs and three graph operators. These are the *disjunction* operator $\cdot \sqcup \cdot$, *sequence* operator $\cdot \rhd^z \cdot$, and *loop* operator $[\cdot]^z$ for all $z \in SyncMech$, as defined in Figure 4.4. The disjunction operator takes the disjoint union of its graph operands and presents a choice between the two operands. The sequence operator $\cdot \rhd^z \cdot$ connects each out-mode of the left graph to each in-mode of the right graph with an edge labeled by z. The loop operator $[\cdot]^z$ is similar but acts upon a single operand, connecting each out-mode to each in-mode in the operand with an edge labeled by z. We defer to Section 4.2.3 discussion of the predicate $Valid(o \xrightarrow{z} i)$ in the definitions of $\cdot \rhd^z \cdot$ and $[\cdot]^z$, as it is orthogonal to the rest of our development.

Altogether, this allows us to specify synchronization disciplines in a text-based annotation language with the following compact grammar:

$$\begin{array}{lcll} d & \in & Discipline & \rightarrow & Discipline \sqcup Discipline \\ & & & \mid & Discipline \rhd^z Discipline \\ & & & \mid & [Discipline]^z \\ & & & \mid & AccessMode \\ a & \in & AccessMode & \rightarrow & \texttt{EX}(t) \mid \texttt{RS}(T) \end{array}$$

$$z \in SyncMech \qquad t \in Thread \qquad T \subseteq Thread$$

An annotation $d \in Discipline$ has an underlying SFG $[\![d]\!]$ defined simply as follows:

$$\begin{aligned} [\![d_1 \sqcup d_2]\!] &:= [\![d_1]\!] \sqcup [\![d_2]\!] \\ [\![d_1 \rhd^z d_2]\!] &:= [\![d_1]\!] \rhd^z [\![d_2]\!] \\ [\![[d_1]^z]\!] &:= [[\![d_1]\!]]^z \\ [\![\texttt{EX}(t)]\!] &:= \mathcal{EX}(t) \\ [\![\texttt{RS}(T)]\!] &:= \mathcal{RS}(T) \end{aligned}$$

Note that annotations of this form hide the details of constructing and manipulating the tuple components in the fully general definition of an SFG.

4.2.2 Discipline conformance

In this section, we precisely define how a program execution trace conforms to a synchronization discipline encoded as an SFG $\mathcal{G}$. Any trace conforming to $\mathcal{G}$ should "walk" along some path in $\mathcal{G}$. In so doing, it should perform only those memory operations permitted by the current access mode at each point, and perform the correct synchronization operations to transition from one access mode to another.

To develop a more detailed understanding, consider again the particle simulator program in Figure 3.4, the execution of which is re-displayed in Figure 4.6. This execution conforms to $\mathcal{G} = [\![[\texttt{RS(t0,t1,t2)} \rhd^{\texttt{b}} \texttt{EX(t0)}]^{\texttt{b}}]\!]$, displayed in Figure 4.5, for the variable `p0`. Any path through $\mathcal{G}$ alternates between RS(`t0`, `t1`, `t2`) and EX(`t0`), and every transition between access modes is on the barrier object `b`. Accordingly, as depicted in Figure 3.4, the execution can be partitioned into

$$t \in Thread \qquad T \subseteq Thread$$

$$\mathcal{EX}(t) \quad := \quad \begin{pmatrix} A & := & \{\textsc{ex}(t)\} \\ S & := & \varnothing \\ I & := & \{\textsc{ex}(t)\} \\ O & := & \{\textsc{ex}(t)\} \end{pmatrix}$$

$$\mathcal{RS}(T) \quad := \quad \begin{pmatrix} A & := & \{\textsc{rs}(T)\} \\ S & := & \varnothing \\ I & := & \{\textsc{rs}(T)\} \\ O & := & \{\textsc{rs}(T)\} \end{pmatrix}$$

Figure 4.3: Base graphs.

$$\mathcal{G}_1 = (A_1, S_1, I_1, O_1) \qquad \mathcal{G}_2 = (A_2, S_2, I_2, O_2) \qquad z \in SyncMech$$

$$\mathcal{G}_1 \sqcup \mathcal{G}_2 \quad := \quad \begin{pmatrix} A & := & A_1 \cup A_2 \\ S & := & S_1 \cup S_2 \\ I & := & I_1 \cup I_2 \\ O & := & O_1 \cup O_2 \end{pmatrix}$$

$$\mathcal{G}_1 \rhd^z \mathcal{G}_2 \quad := \quad \begin{pmatrix} A & := & A_1 \cup A_2 \\ S & := & S_1 \cup \left\{ o \xrightarrow{z} i \;\middle|\; \begin{matrix} o \in O_1,\ i \in I_2 \\ Valid(o \xrightarrow{z} i) \end{matrix} \right\} \cup S_2 \\ I & := & I_1 \\ O & := & O_2 \end{pmatrix}$$

$$[\mathcal{G}_1]^z \quad := \quad \begin{pmatrix} A & := & A_1 \\ S & := & S_1 \cup \left\{ o \xrightarrow{z} i \;\middle|\; \begin{matrix} o \in O_1,\ i \in I_1 \\ Valid(o \xrightarrow{z} i) \end{matrix} \right\} \\ I & := & I_1 \\ O & := & O_1 \end{pmatrix}$$

Figure 4.4: Graph operators.

alternating phases of read-shared access and exclusive access on `p0`. In each phase, every thread behaves in the same structured manner:

> If permitted to access `p0` by the current access mode,
>
> 1. exit the barrier,
> 2. perform permitted memory operations on `p0`, and
> 3. re-enter the barrier.
>
> Otherwise, do not access `p0`.

This is easily generalized. If an arbitrary discipline is modeled by $\mathcal{G}$, and an execution reaches an access mode a along some path $\cdots \xrightarrow{z} a \xrightarrow{z'} \cdots$ in $\mathcal{G}$, then every thread t behaves as follows:

> If $t \in Tid(a)$,
>
> 1. perform $\texttt{rcv}(t, z)$,
> 2. perform permitted memory operations, and
> 3. perform $\texttt{snd}(t, z')$.
>
> Otherwise, do not perform any memory operations.

With this intuition in mind, we now present the formal definition of conformance for our program model. Given a trace ω, a subtrace ω' of ω is called a *memory projection* if ω' includes every memory operation of ω. For any $t \in Thread$, the subtrace of ω containing only the operations performed by t is called the *t-projection* of ω, which we denote by $Proj_t(\omega)$. Note that memory projections are non-unique and t-projections are unique.

Definition 4.2.1. Let $p = a_0 \xrightarrow{z_1} a_1 \xrightarrow{z_2} \cdots \xrightarrow{z_n} a_n$ be a path in an SFG $\mathcal{G}$. A trace ω *conforms* to p if there is a memory projection $\omega_0\omega_1 \ldots \omega_n$ of ω such that, for all $t \in Thread$,

- $Proj_t(\omega_0) \in \begin{cases} (\ \texttt{wr}(t) \mid \texttt{rd}(t)\ )^* \cdot \texttt{snd}(t, z_1) & \text{if } t \in Tid(a_0),\ a_0 = \text{EX}(t) \\ (\ \texttt{rd}(t)\ )^* \cdot \texttt{snd}(t, z_1) & \text{if } t \in Tid(a_0),\ a_0 = \text{RS}(T) \\ \epsilon & \text{if } t \notin Tid(a_0) \end{cases}$
- for all $0 < i < n$,
  $Proj_t(\omega_i) \in \begin{cases} \texttt{rcv}(t, z_i) \cdot (\ \texttt{wr}(t) \mid \texttt{rd}(t)\ )^* \cdot \texttt{snd}(t, z_{i+1}) & \text{if } t \in Tid(a_i),\ a_i = \text{EX}(t) \\ \texttt{rcv}(t, z_i) \cdot (\ \texttt{rd}(t)\ )^* \cdot \texttt{snd}(t, z_{i+1}) & \text{if } t \in Tid(a_i),\ a_i = \text{RS}(T) \\ \epsilon & \text{if } t \notin Tid(a_i) \end{cases}$
- $Proj_t(\omega_n) \in \begin{cases} \texttt{rcv}(t, z_n) \cdot (\ \texttt{wr}(t) \mid \texttt{rd}(t)\ )^* & \text{if } t \in Tid(a_n),\ a_n = \text{EX}(t) \\ \texttt{rcv}(t, z_n) \cdot (\ \texttt{rd}(t)\ )^* & \text{if } t \in Tid(a_n),\ a_n = \text{RS}(T) \\ \epsilon & \text{if } t \notin Tid(a_n) \end{cases}$

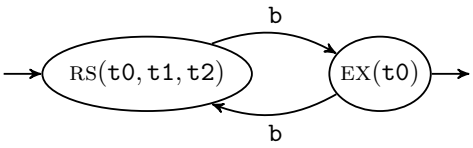


Figure 4.5: $[\![[\texttt{RS}(\texttt{t0},\texttt{t1},\texttt{t2}) \rhd^{\texttt{b}} \texttt{EX}(\texttt{t0})]^{\texttt{b}}]\!]$

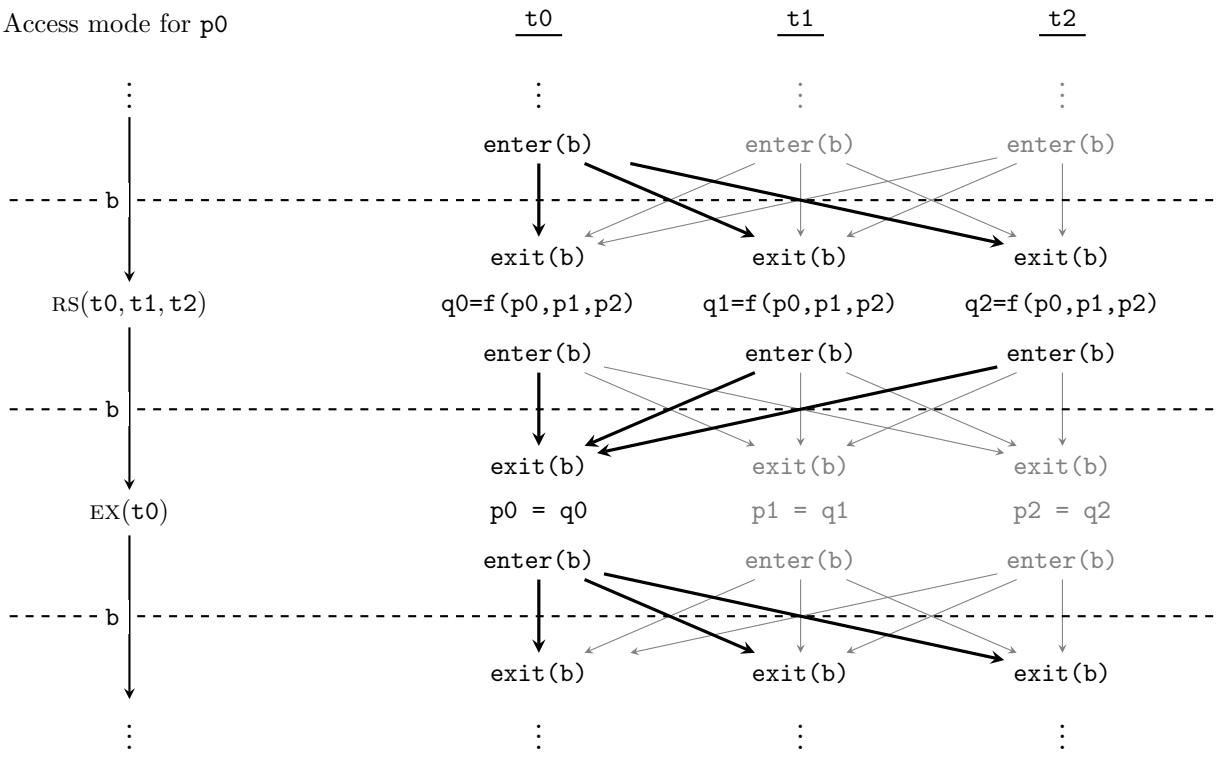


Figure 4.6: An execution trace of our particle simulator program in Figure 3.4. The trace conforms to $[\![[\texttt{RS}(\texttt{t0},\texttt{t1},\texttt{t2}) \rhd^{\texttt{b}} \texttt{EX}(\texttt{t0})]^{\texttt{b}}]\!]$ for the the variable p0.

The trace ω *conforms* to $\mathcal{G}$ if ω conforms to a complete path in $\mathcal{G}$.

Theorem 4.2.2 (Conformance implies race freedom). *If a trace ω conforms to an SFG $\mathcal{G}$, then ω is race-free.*

Proof. Let $p = a_0 \xrightarrow{z_1} a_1 \xrightarrow{z_2} \cdots \xrightarrow{z_n} a_n$ be the path in $\mathcal{G}$ to which ω conforms, and let $\omega' = \omega_0\omega_1 \ldots \omega_n$ be the corresponding memory projection of ω. Since ω' contains all the memory operations of ω, it suffices to show that ω' is race-free. In turn, it suffices to show that: (1) each ω_i is race-free, and (2) every memory operation in ω_i happens-before every memory operation in ω_{i+1}.

(1) If $a_i = \textsc{ex}(s)$, then $Proj_t(\omega_i) = \epsilon$ for all $t \neq s$. Thus, all memory operations in ω_i are performed by s and totally ordered by program order. If $a_i = \textsc{rs}(S)$, then $Proj_t(\omega_i) = \epsilon$ for all $t \notin S$, and every memory operation in $Proj_t(\omega_i)$ is a read for all $t \in S$. Thus, all memory operations in ω_i are reads and therefore non-conflicting.

(2) Let o be a memory operation performed by some thread s in ω_i, and o' a memory operation performed by some thread t in ω_{i+1}. It is clear from the definition of conformance that the operations $o, \mathsf{snd}[s : z_{i+1}], \mathsf{rcv}[t : z_{i+1}], o'$ occur in that order in ω'. Thus,

$$o \prec_{\omega'} \mathtt{snd}(s, z_{i+1}) \prec_{\omega'} \mathtt{rcv}(t, z_{i+1}) \prec_{\omega'} o',$$

the first and last pairs by program order, and the middle pair by synchronization.

□

4.2.3 Valid transitions

In this section, we motivate and define the $Valid$ predicate used in the definition of our graph operators in Figure 4.4. Note that it is orthogonal to the definition of conformance and the proof of Theorem 4.2.2: conformance to an SFG $\mathcal{G}$ is defined with respect to individual paths in $\mathcal{G}$, while application of the $Valid$ predicate in the sequence and loop operators simply filters the collection of all paths in $\mathcal{G}$.

The $Valid$ predicate takes into account the semantics of each synchronization mechanism; without it, some specified disciplines would be irrelevant to well-formed program traces, or not utilize synchronization mechanisms in a useful or idiomatic fashion. Before getting into the details of the $Valid$ predicate , we consider a couple problematic disciplines that would be specifiable in its absence.

First consider the discipline modeled by the SFG $[\![\mathtt{RS}(T) \rhd^b \mathtt{EX}(t)]\!]$. If we were to omit the predicate $Valid(o \xrightarrow{z} i)$ from the definition of $\mathcal{G}_1 \rhd^z \mathcal{G}_2$ in Figure 4.4, then every out-mode $o \in O_1$ of $\mathcal{G}_1$ would be indiscriminately connected to every in-mode $i \in I_2$ of $\mathcal{G}_2$. Then $[\![\mathtt{RS}(T) \rhd^b \mathtt{EX}(t)]\!]$ would be as displayed in Figure 4.7a. If a trace conforms to the path $\textsc{rs}(T) \xrightarrow{b} \textsc{ex}(t)$, then we should observe $\mathtt{barEnt}(t', b)$ performed by every thread $t' \in T$ and $\mathtt{barEx}(t, b)$ performed by thread $t \in T$. However, it may be that the barrier object b was initialized with a thread set S that is entirely disjoint from T and $\{t\}$, in which case a well-formed trace would contain no operations on b performed by $t' \in T$ or t. That is, conforming to $[\![\mathtt{RS}(T) \rhd^b \mathtt{EX}(t)]\!]$ would be at odds with being well-formed.

(a) $[\![\texttt{RS}(T) \rhd^b \texttt{EX}(t)]\!]$ (b) $[\![\texttt{RS}(T) \rhd^m \texttt{EX}(t)]\!]$.

Figure 4.7: SFGs constructible without the $Valid$ predicate.

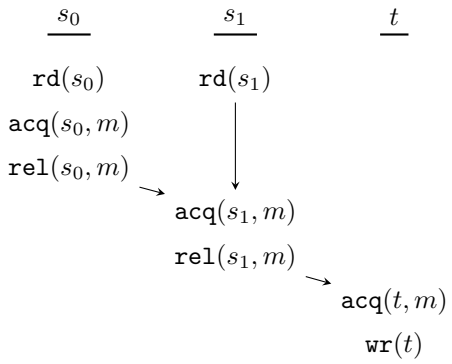


Figure 4.8: A trace conforming to $[\![\texttt{RS}(T) \rhd^m \texttt{EX}(t)]\!]$ in Figure 4.7b, where $T = \{s_0, s_1\}$.

A more subtle example is presented in Figure 4.7b, which shows the SFG $[\![\texttt{RS}(T) \rhd^m \texttt{EX}(t)]\!]$. If a trace conforms to the path $\text{RS}(T) \xrightarrow{m} \text{EX}(t)$, then we should observe a read-shared access phase and an exclusive access phase separated by operations $\texttt{rel}(t', m)$ for all $t' \in T$ and $\texttt{acq}(t, m)$. This is technically possible in a well-formed trace, but it is inefficient if T consists of many threads. The semantics of locks require that every thread in T first acquire the lock m before releasing, meaning the $\texttt{rel}(t', m)$ operations cannot happen concurrently; see an example of such a trace in Figure 4.8. Consequently, programmers typically do not utilize locks in such a manner.

To prevent the expression of such problematic disciplines, we first define the sets $Msg(z) \subseteq \mathcal{P}(Thread) \times \mathcal{P}(Thread)$ for all $z \in SyncMech$ as follows:

$$
\begin{aligned}
Msg(\quad m \quad) &:= \{\ (\{s\}, \{t\}) \mid s \in Thread,\ t \in Thread\ \} \\
Msg(\quad v \quad) &:= \{\ (\{s\}, \{t\}) \mid s \in Thread,\ t \in Thread\ \} \\
Msg(\quad b \quad) &:= \{\ (T, T)\ \} \text{ (where } T \text{ is the thread set with which } b \text{ was initialized)} \\
Msg(\ \texttt{fork}(t, T)\ ) &:= \{\ (\{t\}, T)\ \} \\
Msg(\ \texttt{join}(t, T)\ ) &:= \{\ (T, \{t\})\ \}
\end{aligned}
$$

We discuss the meaning of the sets $Msg(z)$ in more depth further below. Given $Msg(z)$, the predicate $Valid(a \xrightarrow{z} a')$ on a transition $a \xrightarrow{z} a'$ is then defined as follows:

$$Valid(a \xrightarrow{z} a') \quad \equiv \quad Tid(a) \subseteq S \text{ and } Tid(a') \subseteq T \text{ for some } (S, T) \in Msg(z)$$

(a) $[\![\texttt{RS}(T) \rhd^b \texttt{EX}(t)]\!]$ (b) $[\![\texttt{RS}(T) \rhd^m \texttt{EX}(t)]\!]$.

Figure 4.9: SFGs constructed with the $Valid$ predicate.

If we now apply this predicate in the definition of our graph operators, the problematic transitions in the example disciplines described above are filtered out. The resulting SFGs are displayed in Figure 4.9.

As we have defined them above, we can think of the sets $Msg(z)$ as conservative approximations of the semantics of z. Each element of $Msg(z)$ is a pair (S, T) of thread sets such that, at run time, a set of $\texttt{snd}(s, z)$ operations for all $s \in S$ followed by a set of $\texttt{rcv}(t, z)$ operations for all $t \in T$ is deemed an acceptable "unit of synchronization" or "message" via z. In particular, it should be possible for the $\texttt{snd}(s, z)$ operations to be pairwise concurrent, and for the $\texttt{rcv}(t, z)$ operations to be pairwise concurrent; for $z = m \in Lock$, this property is broken by the pair $(S, T) = (\{s_0, s_1\}, \{t\})$ in Figure 4.8, indicating that an ordering via m between threads in S and threads in T requires multiple "messages" of m.

This description of $Msg(z)$ is imprecise; however, As we have already seen, how exactly we choose to define the sets $Msg(z)$ is orthogonal to the rest of our work. Changing $Msg(z)$ may modify the specific SFGs constructible by our language, but our definition of conformance, our proof that conformance implies race freedom, and the soundness of our enforcement technique with respect to conformance (demonstrated in the next section) are unaffected. The use of the sets $Msg(z)$ lies in the simple, uniform format by which they specify the semantics for all mechanisms z.

Consequently, our specification language is highly *extensible*: to specify disciplines incorporating a synchronization mechanism z outside of those in our program model, a programmer need only specify the new mechanism's `snd` operation, its `rcv` operation, and the set $Msg(z) \subseteq \mathcal{P}(Thread) \times \mathcal{P}(Thread)$. We discuss some of the ramifications of this fact in the next chapter.

4.3 Enforcing Disciplines

Synchronization flow graphs directly encode the high-level structure of a synchronization discipline in terms of access modes and transitions via synchronization mechanisms. This abstraction is useful for specifying disciplines, but less so for enforcing them. For example, consider again the execution of our particle simulator program in Figure 4.6. While an SFG captures the high-level sequence of execution phases, it does not provide sufficiently detailed state to reflect the many possible interleavings of operations by different threads within each phase.

Petri nets, on the other hand, are well-suited to reflect such concurrency. In this section, we present a Petri-net-based dynamic analysis that enforces conformance to a specified SFG $\mathcal{G}$. Our analysis pre-processes $\mathcal{G}$ and constructs a corresponding Petri net $\mathscr{N}(\mathcal{G})$; at run time, it monitors the execution of the target program and simulates $\mathscr{N}(\mathcal{G})$ to enforce conformance to $\mathcal{G}$.

We begin by establishing in Section 4.3.1 the basic groundwork for constructing and reasoning about Petri nets. We present the construction of $\mathscr{N}(\mathcal{G})$ from $\mathcal{G}$ in Section 4.3.2 and our dynamic analysis in Section 4.3.3. Finally, in Section 4.3.4, we prove that our analysis is sound with respect to conformance.

4.3.1 Petri net basics

A *Petri net* is a four-tuple $\mathscr{N} = (P, R, F, \mathcal{M}_0)$, where

- P is a finite set of *places*;
- R is a finite set of *transitions*;[2]
- $F \subseteq (P \times R) \cup (R \times P)$ is the *flow relation*; and
- $\mathcal{M}_0 \subseteq \mathcal{P}(P)$ is a set of *initial markings*.

More generally, places and transitions are called *components* of $\mathscr{N}$. For each component $x \in P \cup R$, its *pre-set* ${}^\bullet x$ and *post-set* $x^\bullet$ are defined by

$$\begin{aligned} {}^\bullet x &:= \{y \in P \cup R \mid y \to x \in F\}, \\ x^\bullet &:= \{y \in P \cup R \mid x \to y \in F\}. \end{aligned}$$

We lift this notation to sets $X \subseteq P \cup R$ of components such that

$$\begin{aligned} {}^\bullet X &:= \bigcup_{x \in X} {}^\bullet x, \\ X^\bullet &:= \bigcup_{x \in X} x^\bullet. \end{aligned}$$

Given a marking $M \subseteq P$, a transition r is *enabled* by M if ${}^\bullet r \subseteq M$ and $r^\bullet \cap M = \varnothing$. An enabled transition may *fire*, in which case the tokens in its pre-set are removed and replaced with tokens in its post-set; we refer to this as the *firing rule*. We write $M \overset{r}{\Rightarrow} M'$ if t is enabled by M and M' is the result of the firing of r, i.e., $M' = M \setminus {}^\bullet r \cup r^\bullet$. For any finite sequence $\tau = r_1 \dots r_n$ of transitions, we write $M \overset{\tau}{\Rightarrow} M'$ if there exist markings $M_0, \dots, M_n \subseteq P$ such that $M_0 = M$, $M_n = M'$, and $M_0 \overset{r_1}{\Rightarrow} \cdots \overset{r_n}{\Rightarrow} M_n$. We write $M \Rightarrow M'$ if there exists a finite (possibly empty) sequence τ of transitions such that $M \overset{\tau}{\Rightarrow} M'$. A marking M' is called *reachable* if $M \Rightarrow M'$ for some $M \in \mathcal{M}_0$. See Figure 4.10 for examples of some of these definitions.

Finally, we conclude this section with the following two lemmas, both of which follow immediately from the firing rule. Both state that, if a place is not shared by two markings connected by a firing sequence, then a neighboring transition must have fired in that sequence.

Lemma 4.3.1. *Suppose $M \overset{\tau}{\Rightarrow} M'$ and $p \in M' \setminus M$. Then there exists a transition r in τ such that $r \in {}^\bullet p$.*

[2] To denote transitions, we use 'r' and 'R' instead of the expected 't' and 'T' in order to avoid notation conflict with threads.

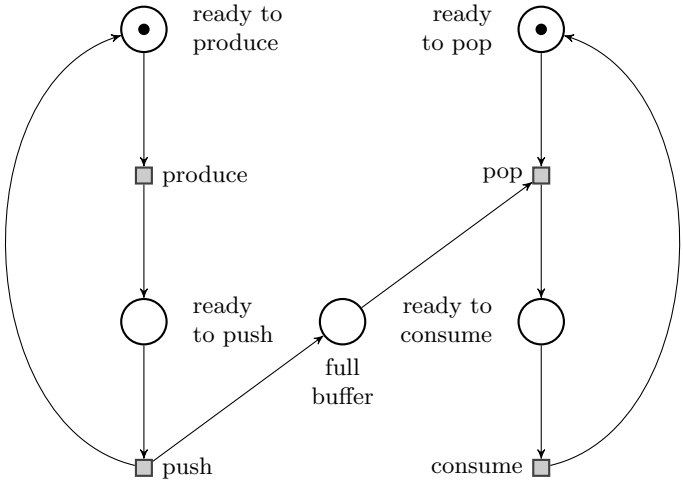


Figure 4.10: The current marking is $M = \{\text{'ready to produce'}, \text{'ready to pop'}\}$. The transition 'produce' is enabled. If 'produce' fires, then $M \xRightarrow{\text{'produce'}} \{\text{'ready to push'}, \text{'ready to pop'}\}$. The transition 'pop' is not enabled by M, since ${}^\bullet\text{'pop'} = \{\text{'full buffer'}, \text{'ready to pop'}\} \not\subseteq M$.

Lemma 4.3.2. *Suppose $M \stackrel{\tau}{\Rightarrow} M'$ and $p \in M \setminus M'$. Then there exists a transition r in τ such that $r \in p^\bullet$.*

4.3.2 Petri net construction

In this section, we present the construction of the Petri net $\mathcal{N}(\mathcal{G})$, the simulation of which enforces conformance to $\mathcal{G}$. The net $\mathcal{N}(\mathcal{G})$ is essentially a refined version of $\mathcal{G}$; whereas $\mathcal{G}$ abstracts away the details of individual program operations and their interleavings, $\mathcal{N}(\mathcal{G})$ brings them into explicit focus.

Let $\mathcal{G} = (A, S, I, O)$ be the specified SFG. The full definition for the Petri net $\mathcal{N}(\mathcal{G})$ is presented in Figure 4.11. As shown in the top-level definition of $\mathcal{N}(\mathcal{G})$ in this figure, the total construction consists of localized constructions $P(a)$, $P(a \xrightarrow{z} b)$, $R(a \xrightarrow{z} b)$, and $F(a \xrightarrow{z} b)$ for every access mode $a \in A$ and edge $a \xrightarrow{z} b \in S$. Understanding these localized constructions is sufficient for an understanding of the total net $\mathcal{N}(\mathcal{G})$. Below, we step through each of the constructions in turn. In the bulleted items, we highlight the different types of components in $\mathcal{N}(\mathcal{G})$ and describe their semantics with respect to our program model. For a visual aid, see Figure 4.12, which depicts a schematic of the local constructions for a pair of neighboring access modes a, b and an edge $a \xrightarrow{z} b$ connecting them.

First, we remark on the notation. In the constructions $P(a)$, $P(a \xrightarrow{z} b)$, and $R(a \xrightarrow{z} b)$, every component is denoted by a type and an index in square brackets; for example, the place $\mathsf{Acc}[t : a]$ has type Acc and index $[t : a]$. Further note that, other than transitions of type sync, the index of each component consists of a thread on the left side of the colon and an access mode $a \in A$ or edge $a \xrightarrow{z} b \in S$ in $\mathcal{G}$ on the right side; the index of a sync-transition consists only of an edge $a \xrightarrow{z} b \in S$

We now describe the meaning of each component in turn. The constructions $P(a)$ and $P(a \xrightarrow{z} b)$ produce places of three different types. For every access mode $a \in A$, $P(a)$ produces the following

$\mathcal{G} = (A, S, I, O)$

$$\mathscr{N}(\mathcal{G}) := \left(\begin{array}{lcc} P & := & \bigcup_{a \in A} P(a) \ \cup \ \bigcup_{a \xrightarrow{z} b \in S} P(a \xrightarrow{z} b) \\ R & := & \bigcup_{a \xrightarrow{z} b \in S} R(a \xrightarrow{z} b) \\ F & := & \bigcup_{a \xrightarrow{z} b \in S} F(a \xrightarrow{z} b) \\ \mathcal{M}_0 & := & \{P(i) \mid i \in I\} \end{array} \right)$$

$$P(a) := \{ \ \mathsf{Acc}[t : a] \mid t \in Tid(a) \ \}$$

$$P(a \xrightarrow{z} b) := \{ \ \mathsf{PostSnd}[s : a \xrightarrow{z} b] \ , \ \mathsf{PreRcv}[t : a \xrightarrow{z} b] \mid s \in Tid(a), t \in Tid(b) \ \}$$

$$R(a \xrightarrow{z} b) := \{ \ \mathsf{snd}[s : a \xrightarrow{z} b] \ , \ \mathsf{sync}[a \xrightarrow{z} b] \ , \ \mathsf{rcv}[t : a \xrightarrow{z} b] \mid s \in Tid(a), t \in Tid(b) \ \}$$

$$F(a \xrightarrow{z} b) := \left\{ \begin{array}{c} \mathsf{Acc}[s : a] \to \mathsf{snd}[s : a \xrightarrow{z} b] \to \mathsf{PostSnd}[s : a \xrightarrow{z} b] \\ \mathsf{PostSnd}[s : a \xrightarrow{z} b] \to \mathsf{sync}[a \xrightarrow{z} b] \to \mathsf{PreRcv}[t : a \xrightarrow{z} b] \\ \mathsf{PreRcv}[t : a \xrightarrow{z} b] \to \mathsf{rcv}[t : a \xrightarrow{z} b] \to \mathsf{Acc}[t : b] \end{array} \middle| \begin{array}{l} s \in Tid(a) \\ t \in Tid(b) \end{array} \right\}$$

Figure 4.11: The refinement procedure that takes an SFG $\mathcal{G}$ and produces a Petri net $\mathscr{N}(\mathcal{G})$. The refinement consists of local constructions $P(a)$, $P(a \xrightarrow{z} b)$, $R(a \xrightarrow{z} b)$, and $F(a \xrightarrow{z} b)$ for each access mode a and edge $a \xrightarrow{z} b$ in $\mathcal{G}$. In the definition of $F(a \xrightarrow{z} b)$, we write $p_1 \to r \to p_2$ to denote two distinct flow edges $p_1 \to r$ and $r \to p_2$.

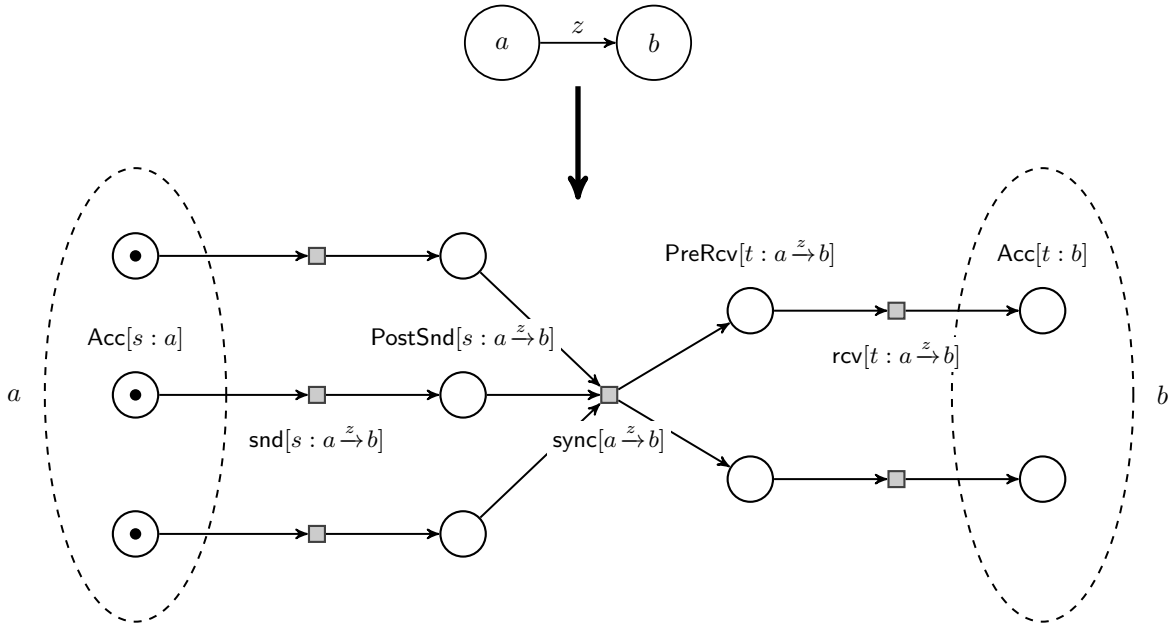


Figure 4.12: A schematic of the refinement procedure for a pair of neighboring access modes a, b and an edge $a \xrightarrow{z} b$ connecting them.

places:

- $\mathsf{Acc}[t : a]$ for all $t \in Tid(a)$. A token at $\mathsf{Acc}[t : a]$ indicates that thread t has entered access mode a and may perform the memory operations permitted by a.

For every edge $a \xrightarrow{z} b \in S$, $P(a \xrightarrow{z} b)$ produces the following places:

- $\mathsf{PostSnd}[s : a \xrightarrow{z} b]$ for all $s \in Tid(a)$. A token at $\mathsf{PostSnd}[s : a \xrightarrow{z} b]$ indicates that thread s has exited access mode a by performing $\mathtt{snd}(s, z)$ and currently may not perform any memory operations. This corresponds to thread s reaching the very end of an execution phase such as those depicted in Figure 4.6.

- $\mathsf{PreRcv}[t : a \xrightarrow{z} b]$ for all $t \in Tid(b)$. A token at $\mathsf{PreRcv}[t : a \xrightarrow{z} b]$ indicates that thread t is about to enter access mode b by performing $\mathtt{rcv}(t, z)$ and currently may not perform any memory operations. This corresponds to thread t reaching the very beginning of an execution phase like those in Figure 4.6.

For each edge $a \xrightarrow{z} b \in S$, the construction $R(a \xrightarrow{z} b)$ produces transitions of three different types:

- $\mathsf{snd}[s : a \xrightarrow{z} b]$ for all $s \in Tid(a)$. $\mathsf{snd}[s : a \xrightarrow{z} b]$ is enabled when thread s is in access mode a and fires when s performs $\mathtt{snd}(s, z)$. The firing of $\mathsf{snd}[s : a \xrightarrow{z} b]$ sends a token from place $\mathsf{Acc}[s : a]$ to place $\mathsf{PostSnd}[s : a \xrightarrow{z} b]$, thereby signaling the exit from a by s.

- $\mathsf{sync}[a \xrightarrow{z} b]$. This single transition is enabled when every thread $s \in Tid(a)$ has exited a by performing $\mathtt{snd}(s, z)$, thereby triggering $\mathsf{snd}[s : a \xrightarrow{z} b]$ and placing a token at $\mathsf{PostSnd}[s : a \xrightarrow{z} b]$ for all threads $s \in Tid(a)$. It fires immediately upon becoming enabled, removing the tokens from the PostSnd-places on its left and replacing them with tokens in the PreRcv[:-]places on its right.

- $\mathsf{rcv}[t : a \xrightarrow{z} b]$ for all $t \in Tid(b)$. $\mathsf{rcv}[t : a \xrightarrow{z} b]$ is enabled by the presence of a token at place $\mathsf{PreRcv}[t : a \xrightarrow{z} b]$ and fires when thread t performs $\mathtt{rcv}(t, z)$. The firing of $\mathsf{rcv}[t : a \xrightarrow{z} b]$ sends a token from $\mathsf{PreRcv}[t : a \xrightarrow{z} b]$ to $\mathsf{Acc}[t : b]$, thereby signaling the entry into access mode b by thread t.

The flow construction $F(a \xrightarrow{z} b)$ is already implicit in our component descriptions above and visually captured in Figure 4.12. However, it is worth highlighting a few structural patterns. The net constructed from an edge $a \xrightarrow{z} b$ consists of two dual halves, one capturing the exit from a and the other capturing the entry into b. The flow relation is defined such that, in the first half, every thread $s \in Tid(a)$ has its own sequential track $\mathsf{Acc}[s : a] \to \mathsf{snd}[s : a \xrightarrow{z} b] \to \mathsf{PostSnd}[s : a \xrightarrow{z} b]$; similarly, in the second half, every thread $t \in Tid(b)$ has its own sequential track $\mathsf{PreRcv}[t : a \xrightarrow{z} b] \to \mathsf{rcv}[t : a \xrightarrow{z} b] \to \mathsf{Acc}[t : b]$. That these tracks in each half are pairwise disjoint reflects the concurrency with which the threads may perform their respective synchronization operations.

On the other hand, the flow relation joins the ends of the tracks in each half as the pre-set and post-set of the single transition $\mathsf{sync}[a \xrightarrow{z} b]$. By the firing rule and our semantics above, this forces every thread $s \in Tid(a)$ to exit a by performing $\mathtt{snd}(s, z)$ before any thread t enters b by performing $\mathtt{rcv}(t, z)$. This orders all memory operations performed in a with all those performed in b.

There remains one final detail in the construction of $\mathcal{N}(\mathcal{G})$ that our discussion above has not covered: the set $\mathcal{M}_0$ of initial markings. This set is defined simply as the collection of constructions $P(i)$ for each in-mode $i \in I$. For example, if a is an in-mode in Figure 4.12, then the depicted marking would be an element of $\mathcal{M}_0$. Though straightforward, the definition of $\mathcal{M}_0$ is crucial to the sensibility of the semantics described above. In particular, as we demonstrate in Section 4.3.4, it guarantees that there is at most one token associated to each thread in every reachable marking of $\mathcal{N}(\mathcal{G})$. If this were not the case, we might encounter a marking including both $\mathsf{Acc}[s : a]$ and $\mathsf{PostSnd}[s : a \xrightarrow{z} b]$; by our semantics above, the former would mean s may perform memory operations while the latter would mean s may not.

We conclude this section with an example of a complete net construction. Figure 4.13 depicts the total net $\mathcal{N}(\mathcal{G})$ for the SFG $\mathcal{G} = [\![[\texttt{RS}(\texttt{t0}, \texttt{t1}, \texttt{t2}) \rhd^{\texttt{b}} \texttt{EX}(\texttt{t0})]^{\texttt{b}}]\!]$ from Figure 4.5. As we can see, $\mathcal{N}(\mathcal{G})$ shares the same high-level structure as $\mathcal{G}$, but provides a greater amount of local detail with respect to program operations and their interleavings.

4.3.3 Program analysis

In the previous section, we informally discussed the execution semantics of the constructed net $\mathcal{N}(\mathcal{G})$ with respect to our program model. We now formalize these semantics in a dynamic program analysis that simulates $\mathcal{N}(\mathcal{G})$ to enforce conformance to $\mathcal{G}$. Prior to run time, the analysis preprocesses $\mathcal{G}$ and constructs the net $\mathcal{N}(\mathcal{G}) = (P, R, F, \mathcal{M}_0)$ as defined in Figure 4.11. At run time, it monitors the execution of the target program and maintains an analysis state consisting of a marking M of $\mathcal{N}(\mathcal{G})$. The initial analysis state may be any marking in $\mathcal{M}_0$; we address this nondeterminism further below.

When the target program performs an operation o, the analysis may update its state via the relation $M \overset{o}{\Rrightarrow} M'$. The rules for the single-step relation are presented in Figure 4.14; we describe each rule in the bulleted items below. For a trace $\omega = o_1 \dots o_n$, we write $M \overset{\omega}{\Rrightarrow} M'$ to denote the existence of markings $M_0, M_1, \dots, M_{n-1}, M_n$ such that $M = M_0 \overset{o_1}{\Rrightarrow} M_1 \overset{o_2}{\Rrightarrow} \cdots \overset{o_{n-1}}{\Rrightarrow} M_{n-1} \overset{o_n}{\Rrightarrow} M_n = M'$.

Memory operations If there is a token at $\mathsf{Acc}[t : a]$ in the current marking, then thread t may perform a memory operation permitted by access mode a. The rules [WRITE EXCLUSIVE], [READ EXCLUSIVE], and [READ SHARED] cover the different cases:

- [WRITE EXCLUSIVE] permits the operation $\texttt{wr}(t)$ if $a = \textsc{ex}(t)$.
- [READ EXCLUSIVE] permits the operation $\texttt{rd}(t)$ if $a = \textsc{ex}(t)$.
- [READ SHARED] permits the operation $\texttt{rd}(t)$ if $a = \textsc{rs}(T)$ (which implies $t \in T$ since there are no places $\mathsf{Acc}[t : \textsc{rs}(T)]$ in $\mathcal{N}(\mathcal{G})$ such that $t \notin T$).

On all permitted memory operations, the analysis state M remains constant.

Synchronization The rules [SEND], [RECEIVE], and [SYNC] advance the analysis if there is some corresponding transition in $\mathcal{N}(\mathcal{G})$ enabled by the current marking M. Specifically:

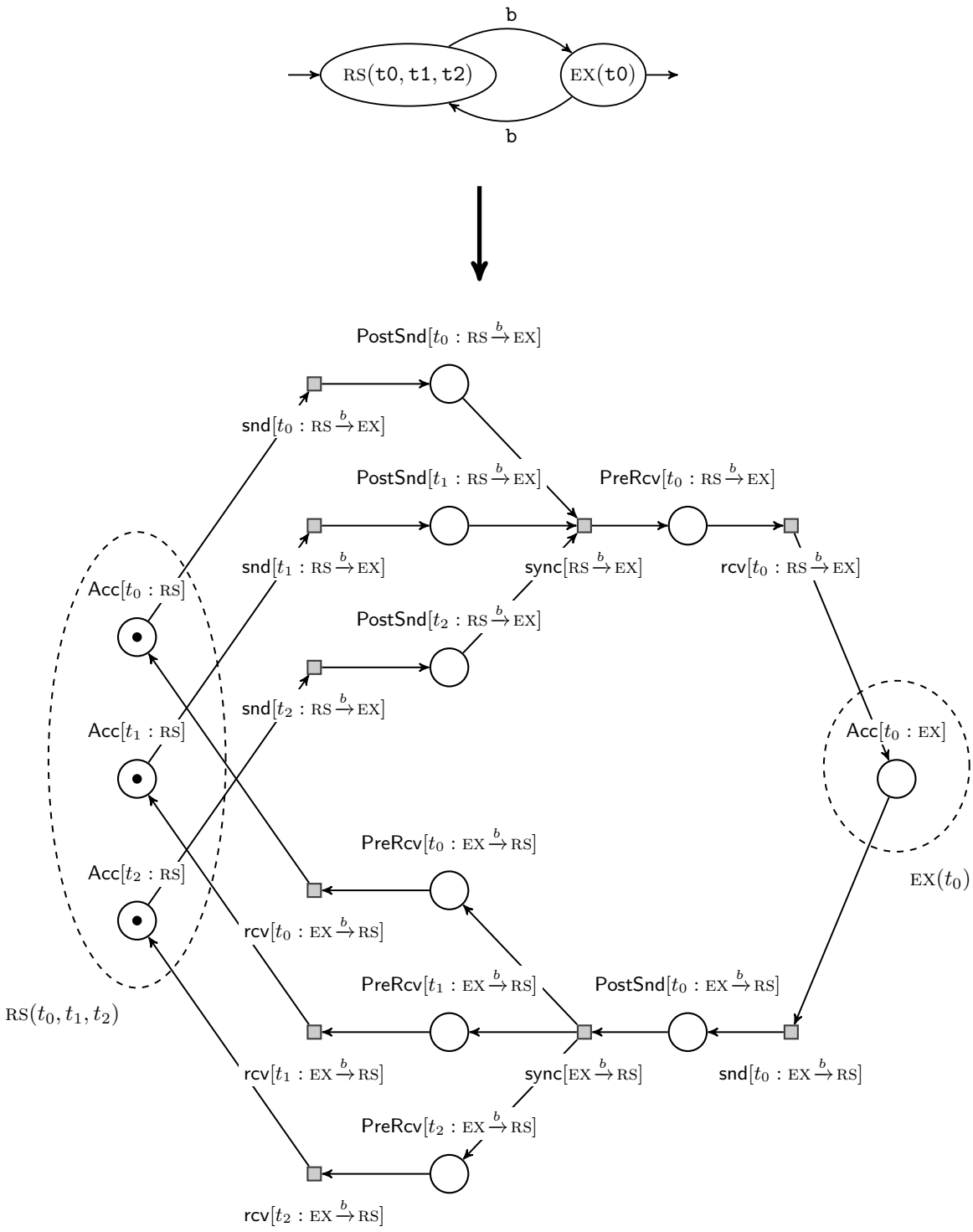


Figure 4.13: The net $\mathscr{N}(\mathcal{G})$ for the SFG $\mathcal{G} = [\![[\texttt{RS(t0,t1,t2)} \rhd^{\texttt{b}} \texttt{EX(t0)}]^{\texttt{b}}]\!]$ from Figure 4.5. In the index of each component, we abbreviate $\text{RS}(t_0, t_1, t_2)$ and $\text{EX}(t_0)$ to RS and EX, respectively, to reduce notational clutter. The places marked by tokens form the unique initial marking.

$$\frac{\mathsf{Acc}[t : \textsc{ex}(t)] \in M}{M \overset{\mathtt{wr}(t)}{\Rrightarrow} M} \text{ [WRITE EXCLUSIVE]} \qquad \frac{\mathsf{Acc}[t : \textsc{ex}(t)] \in M}{M \overset{\mathtt{rd}(t)}{\Rrightarrow} M} \text{ [READ EXCLUSIVE]} \qquad \frac{\mathsf{Acc}[t : \textsc{rs}(T)] \in M}{M \overset{\mathtt{rd}(t)}{\Rrightarrow} M} \text{ [READ SHARED]}$$

$$\frac{M \xrightarrow{\mathsf{snd}[t : a \xrightarrow{z} b]} M'}{M \overset{\mathtt{snd}(t,z)}{\Rrightarrow} M'} \text{ [SEND]} \qquad \frac{M \xrightarrow{\mathsf{rcv}[t : a \xrightarrow{z} b]} M'}{M \overset{\mathtt{rcv}(t,z)}{\Rrightarrow} M'} \text{ [RECEIVE]} \qquad \frac{M \xrightarrow{\mathsf{sync}[a \xrightarrow{z} b]} M'}{M \overset{\epsilon}{\Rrightarrow} M'} \text{ [SYNC]}$$

$$\frac{o \in \mathit{SyncOp}}{M \overset{o}{\Rrightarrow} M} \text{ [SKIP]}$$

Figure 4.14: Update rules for a dynamic program analysis that nondeterministically enforces conformance to a specified discipline.

- [SEND] permits the operation $\mathtt{snd}(t, z)$ if there is some transition $\mathsf{snd}[t : a \xrightarrow{z} b]$ enabled by M. Applying this rule triggers the firing of $\mathsf{snd}[t : a \xrightarrow{z} b]$ and updates the analysis state to the resulting marking M'.
- [RECEIVE] permits the operation $\mathtt{rcv}(t, z)$ if there is some transition $\mathsf{rcv}[t : a \xrightarrow{z} b]$ enabled by M. Applying this rule triggers the firing of $\mathsf{rcv}[t : a \xrightarrow{z} b]$ and updates the analysis state to the resulting marking M'.
- [SYNC] permits no operations, but updates the current marking as soon as it enables some transition $\mathsf{sync}[a \xrightarrow{z} b]$. Applying this rule triggers the firing of $\mathsf{sync}[a \xrightarrow{z} b]$ and updates the analysis state to the resulting marking M'.

The analysis may also proceed by skipping a synchronization operation:

- [SKIP] permits any operation $o \in \mathit{SyncOp}$. It does not modify the analysis state.

The [SKIP] rule is motivated by the fact that the trace format required by conformance need only be satisfied by some memory projection of the observed trace. That is, we may ignore any synchronization that is not relevant for conformance.

Note that this analysis is nondeterministic. As mentioned at the top of the section, the initial analysis state is chosen from any of the markings in $\mathcal{M}_0$. The analysis also updates its state nondeterministically on some synchronization operations, since any operation $o \in \mathit{SyncOp}$ permitted by either of the rules [SEND] or [RECEIVE] is also permitted by the rule [SKIP]. (Though it is not immediately apparent that the analysis is deterministic on memory operations, reads in particular, this follows by our proofs in the next section.)

We may run the analysis deterministically by using the rule in Figure 4.15, where $\mathcal{L} \overset{o}{\Mapsto} \mathcal{L}'$ in the consequent denotes the new deterministic step relation. The analysis state expands from a single

$$\frac{\begin{array}{rcl} o & \in & \mathit{Operation} \\ \mathcal{L} & \neq & \varnothing \\ \mathcal{L}' & = & \bigcup_{M \in \mathcal{L}} \left\{ M' \;\middle|\; M \overset{o}{\Rrightarrow} M' \right\} \end{array}}{\mathcal{L} \overset{o}{\Longrightarrow} \mathcal{L}'}$$

Figure 4.15: The update rule for a deterministic version of the analysis in Figure 4.14.

marking M to a collection $\mathcal{L}$ of candidate markings. The initial analysis state is simply the set $\mathcal{M}_0$ of initial markings. When the target program performs an operation o, the deterministic analysis steps through each candidate marking $M \in \mathcal{L}$ of the current state and replaces it with every possible marking M' such that $M \overset{o}{\Rrightarrow} M'$ by the original nondeterministic analysis. It is evident that $\mathcal{L} \overset{\omega}{\Longrightarrow} \mathcal{L}'$ if and only if $M \overset{\omega}{\Rrightarrow} M'$ for some $M \in \mathcal{L}$, $M' \in \mathcal{L}'$; in other words, the deterministic analysis admits ω if and only if there is some choice of initial marking and rules in Figure 4.14 by which the nondeterministic analysis admits ω.

4.3.4 Proof of soundness

In this section, we prove that the dynamic analysis defined in the previous section is sound with respect to conformance, i.e., an admitted trace conforms to the specified graph $\mathcal{G}$. Because of the equivalence between the nondeterministic and deterministic forms, we restrict our attention to the original nondeterministic form presented in Figure 4.14.

Our proof can be broken up into three main parts.

A. First, we establish some basic structural properties of the constructed net $\mathcal{N}(\mathcal{G})$. These are covered by Lemmas 4.3.3 through 4.3.5.

B. Second, we reason about the behavior of reachable markings on top of the net structure. In particular, we show that every reachable marking M of $\mathcal{N}(\mathcal{G})$ satisfies a property called *b-feasibility* for some access mode b. This property tells us that: (1) M is in the "local neighborhood" of b, e.g., within or near the dashed ellipse for $b = \textsc{rs}(t_0, t_1, t_2)$ in Figure 4.13; and (2) M has exactly one token associated with each thread $t \in \mathit{Tid}(b)$. This portion is covered by Definitions/Lemmas 4.3.6 through 4.3.11.

C. Third, we use the results shown above to prove soundness in Theorem 4.3.12.

Let $\mathcal{N}(\mathcal{G}) = (P, R, F, \mathcal{M}_0)$ be the Petri net constructed by our analysis from the specified SFG $\mathcal{G} = (A, S, I, O)$. For any component $x \in P \cup R$, let $\mathit{Tid}(x)$ denote the thread in its square-bracketed index. For any set $X \subseteq P \cup R$ of components and thread t, let

$$[X]_t := \{x \in X \mid \mathit{Tid}(x) = t\}.$$

Given a sequence τ of transitions and thread t, let $\mathit{Proj}_t(\tau)$ denote the subsequence of τ containing only the transitions r for which $\mathit{Tid}(r) = t$.

We begin Part A. Recall the schematic in Figure 4.12. Our first lemma, Lemma 4.3.3, simply formalizes this structure by listing the pre-set and post-set of every component in $\mathcal{N}(\mathcal{G})$. We apply Lemma 4.3.3 implicitly throughout the proof whenever evaluating pre-sets or post-sets.

Lemma 4.3.3 (Pre-sets and post-sets in $\mathcal{N}(\mathcal{G})$). *For any edge $a \xrightarrow{z} b \in S$ and threads $s \in Tid(a)$, $t \in Tid(b)$, the following statements hold:*

$$
\begin{aligned}
\mathsf{Acc}[s:a]^\bullet &= \{\ \mathsf{snd}[s:a \xrightarrow{z} b'] \mid a \xrightarrow{z} b' \in S\ \} \\
{}^\bullet\mathsf{snd}[s:a \xrightarrow{z} b] &= \{\ \mathsf{Acc}[s:a]\ \} \\
\mathsf{snd}[s:a \xrightarrow{z} b]^\bullet &= \{\ \mathsf{PostSnd}[s:a \xrightarrow{z} b]\ \} \\
{}^\bullet\mathsf{PostSnd}[s:a \xrightarrow{z} b] &= \{\ \mathsf{snd}[s:a \xrightarrow{z} b]\ \} \\
\mathsf{PostSnd}[s:a \xrightarrow{z} b]^\bullet &= \{\ \mathsf{sync}[a \xrightarrow{z} b]\ \} \\
{}^\bullet\mathsf{sync}[a \xrightarrow{z} b] &= \{\ \mathsf{PostSnd}[s':a \xrightarrow{z} b] \mid s' \in Tid(a)\ \} \\
\mathsf{sync}[a \xrightarrow{z} b]^\bullet &= \{\ \mathsf{PreRcv}[t':a \xrightarrow{z} b] \mid t' \in Tid(b)\ \} \\
{}^\bullet\mathsf{PreRcv}[t:a \xrightarrow{z} b] &= \{\ \mathsf{sync}[a \xrightarrow{z} b]\ \} \\
\mathsf{PreRcv}[t:a \xrightarrow{z} b]^\bullet &= \{\ \mathsf{rcv}[t:a \xrightarrow{z} b]\ \} \\
{}^\bullet\mathsf{rcv}[t:a \xrightarrow{z} b] &= \{\ \mathsf{PreRcv}[t:a \xrightarrow{z} b]\ \} \\
\mathsf{rcv}[t:a \xrightarrow{z} b]^\bullet &= \{\ \mathsf{Acc}[t:b]\ \} \\
{}^\bullet\mathsf{Acc}[t:b] &= \{\ \mathsf{rcv}[t:a' \xrightarrow{z} b] \mid a' \xrightarrow{z} b \in S\ \}
\end{aligned}
$$

Proof. This follows by a simple, mechanical inspection of the definition of $\mathcal{N}(\mathcal{G})$ in Figure 4.11. □

In the informal discussion that accompanied Figure 4.12, we noted that the net derived from any edge consists of two halves, each half containing disjoint sequential tracks for every thread. Such parallel patterns allow us to simplify our reasoning about the behavior of $\mathcal{N}(\mathcal{G})$ to a per-thread basis. The following two lemmas, Lemma 4.3.4 and 4.3.5, formalize these patterns.

Lemma 4.3.4. *Let $M \subseteq P$ be a marking and $t \in Thread$.*

(1) If M does not contain a PreRcv*-place, then ${}^\bullet[M]_t = [{}^\bullet M]_t$.*

(2) If M does not contain a PostSnd*-place, then $[M]_t^\bullet = [M^\bullet]_t$.*

Proof. Since the operations ${}^\bullet(\cdot)$, $(\cdot)^\bullet$, and $[\cdot]_t$ all distribute over taking unions, it suffices to show that the lemma holds when $M = \{p\}$ is a singleton. This follows by a straightforward inspection of the pre-sets and post-sets listed in Lemma 4.3.3. □

Lemma 4.3.5. *Suppose $M \overset{r}{\Rightarrow} M'$ for any markings M, M' and non-*sync*-transition r. If $t \neq Tid(r)$, then $[M]_t = [M']_t$.*

Proof. The firing rule states that $M' = M \setminus {}^\bullet r \cup r^\bullet$. Another inspection of Lemma 4.3.3 shows that, since r is not a sync-transition, ${}^\bullet r = \{p\}$ and $r^\bullet = \{q\}$ for some places $p, q \in P$ such that $Tid(p) = Tid(q) = Tid(r)$. Thus, if $t \neq Tid(r)$, then $[M']_t = [M]_t \setminus [{}^\bullet r]_t \cup [r^\bullet]_t = [M]_t$. □

Definition 4.3.6. Given an access mode $b \in A$, the *neighborhood* of b is the set of places

$$\begin{aligned} Q(b) \;:=\; & \{\mathsf{PreRcv}[t : a \xrightarrow{z} b] \mid a \xrightarrow{z} b \in S,\ t \in Tid(b)\} \\ & \cup \{\mathsf{Acc}[t : b] \mid t \in Tid(b)\} \\ & \cup \{\mathsf{PostSnd}[t : b \xrightarrow{z} c] \mid b \xrightarrow{z} c \in S,\ t \in Tid(b)\}. \end{aligned}$$

Definition 4.3.7. A marking M of $\mathscr{N}(\mathcal{G})$ is called *b-feasible* if $M \subseteq Q(b)$ and

$$|[M]_t| \;=\; \begin{cases} 1 & \text{if } t \in Tid(b) \\ 0 & \text{if } t \notin Tid(b) \end{cases}.$$

A marking is called *feasible* if it is b-feasible for some access mode b.

The neighborhood of b consists of those places that are at most a single transition away from the places produced by $P(b)$. Alternatively, one could view the neighborhoods as forming a partition of all the places in $\mathscr{N}(\mathcal{G})$, where the partition is formed by "cutting" along each sync-transition. For example, Figure 4.16 shows the subnet of the Petri net in Figure 4.13 that contains the neighborhood $Q(b)$ of $b = \textsc{rs}(t_0, t_1, t_2)$.

A b-feasible marking is contained within the neighborhood $Q(b)$, with an additional constraint that it has exactly one token associated with every thread $t \in Tid(b)$ and no others. For example, in Figure 4.16, the marking denoted by the circular tokens is an example of a b-feasible marking. On the other hand, the marking denoted by the triangular tokens is not b-feasible because $|[M]_{t_0}| = 2$ and $|[M]_{t_2}| = 0$. The bolded subnet highlights the sequential track along which each token in a b-feasible marking travels; in general, the Acc-places may have many incoming and outgoing flow arrows.

One of the nice features of feasible markings is that containment between feasible markings is equivalent to equality, as demonstrated by the following lemma. This is useful in conjunction with the firing rule, which says that if $M \overset{r}{\Rightarrow} M'$, then ${}^\bullet r \subseteq M$ and $r^\bullet \subseteq M'$.

Lemma 4.3.8 (Containment between feasible markings implies equality). *If M is b-feasible, M' is b'-feasible, and $M \subseteq M'$, then $b = b'$ and $M = M'$.*

Proof. Since the markings are feasible, we have $M \subseteq Q(b)$ and $M \subseteq M' \subseteq Q(b')$, i.e., $Q(b)$ and $Q(b')$ have nonempty intersection. It is straightforward to verify that the neighborhoods $Q(b)$ for all access modes b form a partition of the total set P of places, and therefore $b = b'$. Again by feasibility, we have

$$|[M]_t| = |[M']_t| = \begin{cases} 1 & \text{if } t \in Tid(b) \\ 0 & \text{if } t \notin Tid(b) \end{cases}$$

for all $t \in Thread$. Since $M \subseteq M'$, it must be that $[M]_t = [M']_t$ for all $t \in Tid(b)$, and therefore $M = M'$. □

In the next two lemmas, we show that every reachable marking of $\mathscr{N}(\mathcal{G})$ is feasible. The first lemma, Lemma 4.3.9, says that the firing of any non-sync-transition preserves feasibility. The second

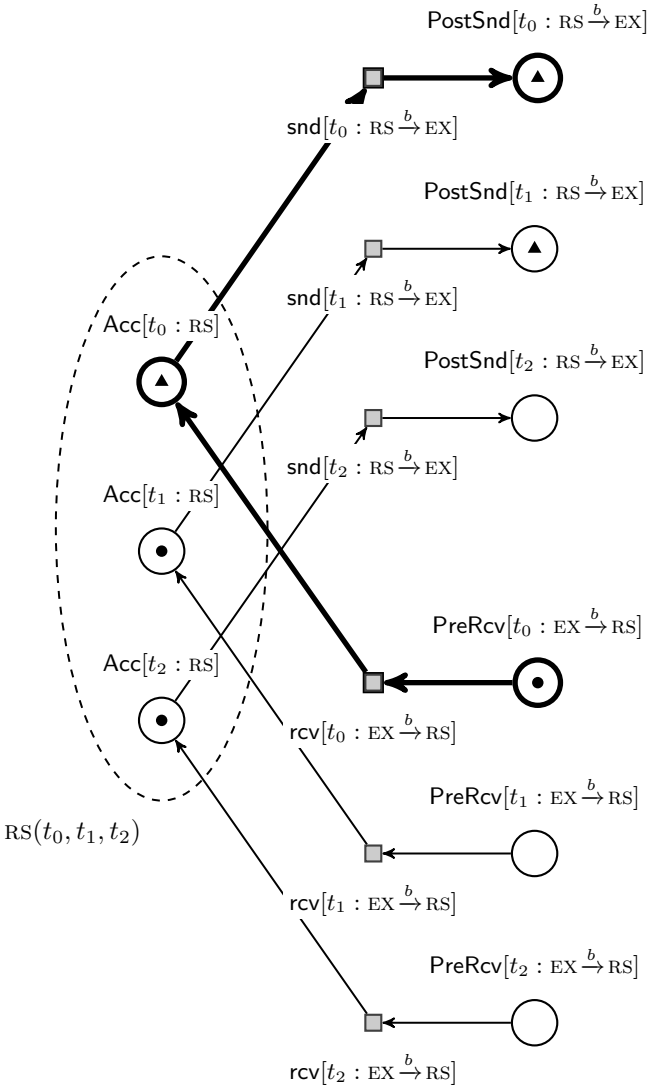


Figure 4.16: The subnet of the Petri net $\mathcal{N}(\mathcal{G})$ in Figure 4.13 that contains the neighborhood $Q(b)$ of access mode $b = \text{RS}(t_0, t_1, t_2)$.

lemma, Lemma 4.3.10, says that, if a transition $\mathsf{sync}[a \xrightarrow{z} b]$ fires from a reachable marking M and results in the marking M', then M and M' must coincide exactly with the pre-set ${}^\bullet\mathsf{sync}[a \xrightarrow{z} b]$ and post-set $\mathsf{sync}[a \xrightarrow{z} b]^\bullet$. As we can check that both ${}^\bullet\mathsf{sync}[a \xrightarrow{z} b]$ and $\mathsf{sync}[a \xrightarrow{z} b]^\bullet$ are feasible, and that every initial marking is feasible, it follows that every reachable marking is feasible.

Lemma 4.3.9 (Non-sync-transitions preserve b-feasibility). *Suppose $M \overset{\tau}{\Rightarrow} M'$ and τ is sync-free. For any access mode $b \in A$, M is b-feasible if and only if M' is b-feasible.*

Proof. Proceed by induction on the length of τ. If τ is empty, and therefore $M = M'$, then the theorem certainly holds.

Now let $\tau = r_1 \dots r_n$ for some $n > 0$, and suppose the theorem holds for any sequence of transitions of length $n-1$. There exist markings $M_0, \dots, M_n$ such that

$$M \;=\; M_0 \overset{r_1}{\Rightarrow} M_1 \overset{r_2}{\Rightarrow} \cdots \overset{r_{n-1}}{\Longrightarrow} M_{n-1} \overset{r_n}{\Rightarrow} M_n \;=\; M'.$$

First suppose M_0 is b-feasible. By our inductive hypothesis, M_{n-1} is b-feasible, i.e., $M_{n-1} \subseteq Q(b)$ and $|[M_{n-1}]_t| = 1$ for all $t \in Tid(b)$. The transition r_n must be an element of the set $M_{n-1}^\bullet$, which in turn is a subset of

$$\begin{aligned} Q(b)^\bullet &= \{\mathsf{PreRcv}[t : a \xrightarrow{z} b] \mid a \xrightarrow{z} b \in S,\ t \in Tid(b)\}^\bullet \\ &\quad \cup \{\mathsf{Acc}[t : b] \mid t \in Tid(b)\}^\bullet \\ &\quad \cup \{\mathsf{PostSnd}[t : b \xrightarrow{z} c] \mid b \xrightarrow{z} c \in S,\ t \in Tid(b)\}^\bullet \\ &= \{\mathsf{rcv}[t : a \xrightarrow{z} b] \mid a \xrightarrow{z} b \in S,\ t \in Tid(b)\} \\ &\quad \cup \{\mathsf{snd}[t : b \xrightarrow{z} c] \mid b \xrightarrow{z} c \in S,\ t \in Tid(b)\} \\ &\quad \cup \{\mathsf{sync}[b \xrightarrow{z} c] \mid b \xrightarrow{z} c \in S\}. \end{aligned}$$

Since r_n is not a sync-transition by assumption, there is some $s \in Tid(b)$ such that $r_n = \mathsf{rcv}[s : a \xrightarrow{z} b]$ for some edge $a \xrightarrow{z} b \in S$ or $r_n = \mathsf{snd}[s : b \xrightarrow{z} c]$ for some edge $b \xrightarrow{z} c \in S$. By the firing rule, we have ${}^\bullet r_n \subseteq M_{n-1}$ and $M_n \;=\; M_{n-1} \setminus {}^\bullet r_n \cup r_n^\bullet$, where

$${}^\bullet r_n \;=\; \begin{cases} \{\mathsf{PreRcv}[s : a \xrightarrow{z} b]\} & \text{if } r_n = \mathsf{rcv}[s : a \xrightarrow{z} b] \\ \{\mathsf{Acc}[s : b]\} & \text{if } r_n = \mathsf{snd}[s : b \xrightarrow{z} c] \end{cases}$$

and

$$r_n^\bullet \;=\; \begin{cases} \{\mathsf{Acc}[s : b]\} & \text{if } r_n = \mathsf{rcv}[s : a \xrightarrow{z} b] \\ \{\mathsf{PostSnd}[s : b \xrightarrow{z} c]\} & \text{if } r_n = \mathsf{snd}[s : b \xrightarrow{z} c] \end{cases}.$$

In either case, the b-feasibility of M_n follows from the b-feasibility of M_{n-1}.

Now suppose M_n is b-feasible. The proof follows similarly as above. We know that M_1 is b-feasible by our inductive hypothesis, which tells us that the first transition r_1 must be an element of ${}^\bullet Q(b)$. This sufficiently restricts the possible choices for r_1 for us to conclude that M_0 is also b-feasible. □

Lemma 4.3.10 (Reachable markings before and after sync-transitions are feasible). *If M, M' are reachable markings of $\mathscr{N}(\mathcal{G})$ such that $M \xRightarrow{\mathsf{sync}[a \xrightarrow{z} b]} M'$ for some edge $a \xrightarrow{z} b \in S$, then $M = {}^\bullet\mathsf{sync}[a \xrightarrow{z} b]$ and $M' = \mathsf{sync}[a \xrightarrow{z} b]^\bullet$.*

Proof. Since $M' = M \setminus {}^\bullet\mathsf{sync}[a \xrightarrow{z} b] \cup \mathsf{sync}[a \xrightarrow{z} b]^\bullet$ by the firing rule, it suffices to show $M = {}^\bullet\mathsf{sync}[a \xrightarrow{z} b]$. Since M and M' are reachable, there is some initial marking $L \in \mathcal{M}_0$ of $\mathscr{N}(\mathcal{G})$ and sequence τ of transitions such that

$$L \overset{\tau}{\Rightarrow} M \xRightarrow{\mathsf{sync}[a \xrightarrow{z} b]} M'.$$

Proceed by induction on the number n of sync-transitions in τ.

First suppose $n = 0$. By definition of $\mathcal{M}_0$, we have that $L = P(i) = \{\mathsf{Acc}[t : i] \mid t \in Tid(i)\}$ for some in-mode i. Inspection shows that L is i-feasible, and therefore M is i-feasible by Lemma 4.3.9. Since the preset ${}^\bullet\mathsf{sync}[a \xrightarrow{z} b] = \{\mathsf{PostSnd}[t : a \xrightarrow{z} b] \mid t \in Tid(a)\}$ is a-feasible by inspection and ${}^\bullet\mathsf{sync}[a \xrightarrow{z} b] \subseteq M$, we have by Lemma 4.3.8 that $a = i$ and $M = {}^\bullet\mathsf{sync}[a \xrightarrow{z} b]$.

Now suppose $n > 0$, and we have shown the statement holds for $n - 1$. We may write τ in the form

$$\tau_0\ \mathsf{sync}[a_1 \xrightarrow{z_1} b_1]\ \tau_1\ \mathsf{sync}[a_2 \xrightarrow{z_2} b_2]\ \cdots\ \mathsf{sync}[a_{n-1} \xrightarrow{z_{n-1}} b_{n-1}]\ \tau_{n-1}\ \mathsf{sync}[a_n \xrightarrow{z_n} b_n]\ \tau_n$$

such that each τ_i is sync-free. There exists a corresponding sequence $L_0, L_0', L_1, L_1', \ldots, L_n, L_n'$ of markings such that $L_0 = L$, $L_n' = M$, and

$$L_0 \underbrace{\overset{\tau_0}{\Rightarrow} L_0' \xRightarrow{\mathsf{sync}[a_1 \xrightarrow{z_1} b_1]} \cdots \xRightarrow{\mathsf{sync}[a_{n-1} \xrightarrow{z_{n-1}} b_{n-1}]} L_{n-1} \overset{\tau_{n-1}}{\Rightarrow} L_{n-1}' \xRightarrow{\mathsf{sync}[a_n \xrightarrow{z_n} b_n]} L_n \overset{\tau_n}{\Rightarrow}}_{\tau} L_n'.$$

By our inductive hypothesis, we have that $L_{n-1}' = {}^\bullet\mathsf{sync}[a_n \xrightarrow{z_n} b_n]$. By the firing rule, we have that

$$L_n = \mathsf{sync}[a_n \xrightarrow{z_n} b_n]^\bullet = \{\mathsf{PreRcv}[t : a_n \xrightarrow{z_n} b_n] \mid t \in Tid(b_n)\},$$

which is b_n-feasible by inspection. Then the same argument as in the base case shows that $L_n' = {}^\bullet\mathsf{sync}[a \xrightarrow{z} b]$. □

Now consider again the neighborhood $Q(b)$ for $b = \textsc{rs}(t_0, t_1, t_2)$ depicted in Figure 4.16. Since every reachable marking is feasible, we can think about firing sequences of markings in $Q(b)$ as concurrent, thread-local token-firings along each sequential track. In particular, for every thread, the corresponding token should first fire a rcv-transition, then a snd-transition. The following lemma verifies this behavior.

Lemma 4.3.11 (Thread-local behavior within a neighborhood). *Suppose M, M', N, N' are feasible markings and τ is a sync-free sequence of transitions such that*

$$M \xRightarrow{\mathsf{sync}[a \xrightarrow{z} b]} N \overset{\tau}{\Rightarrow} M' \xRightarrow{\mathsf{sync}[b \xrightarrow{z'} c]} N'.$$

Then

$$Proj_s(\tau) \;=\; \begin{cases} \mathsf{rcv}[s : a \xrightarrow{z} b]\ \mathsf{snd}[s : b \xrightarrow{z'} c] & \text{if } s \in Tid(b) \\ \epsilon & \text{if } s \notin Tid(b) \end{cases}$$

Proof. Write $\tau = r_1 \cdots r_n$ and let $L_0, L_1, \cdots, L_n$ be the corresponding sequence of markings such that

$$M \xrightarrow{\mathsf{sync}[a \xrightarrow{z} b]} N \;=\; L_0 \underbrace{\xRightarrow{r_1} L_1 \xRightarrow{r_2} \cdots \xRightarrow{r_n}}_{\tau} L_n \;=\; M' \xrightarrow{\mathsf{sync}[b \xrightarrow{z'} c]} N'.$$

First suppose $s \notin Tid(b)$. If $Proj_s(\tau) \neq \epsilon$, then there exists a transition r_i with $Tid(r_i) = s$. Inspection of Lemma 4.3.3 shows that, since r_i is a non-sync-transition, ${}^\bullet r_i = \{p\}$ for some place p such that $Tid(p) = s$. Furthermore, by the firing rule, we have $p \in L_{i-1}$ and therefore $|[L_{i-1}]_s| \geq 1$. But we also have by Lemma 4.3.9 that L_{i-1} is b-feasible and therefore $|[L_{i-1}]_s| = 0$, a contradiction. Thus, $Proj_s(\tau) = \epsilon$.

Now suppose $s \in Tid(b)$. Our goal is to show $Proj_s(\tau) = \mathsf{rcv}[s : a \xrightarrow{z} b]\ \mathsf{snd}[s : b \xrightarrow{z} c]$. It suffices to show the following statements:

(1) $Proj_s(\tau)$ is nonempty (and therefore there exist $1 \leq j \leq k \leq n$ such that r_j and r_k are, respectively, the first and last transitions r in τ such that $Tid(r) = s$).

(2) $r_j = \mathsf{rcv}[s : a \xrightarrow{z} b]$.

(3) $r_k = \mathsf{snd}[s : b \xrightarrow{z} c]$.

(4) $Tid(r_i) \neq s$ for all $j < i < k$.

(1) By Lemma 4.3.10, we have that

$$\begin{aligned} L_0 \;&=\; \mathsf{sync}[a \xrightarrow{z} b]^\bullet \;=\; \{\ \mathsf{PreRcv}[t : a \xrightarrow{z} b] \mid t \in Tid(b)\}, \\ L_n \;&=\; {}^\bullet\mathsf{sync}[b \xrightarrow{z'} c] \;=\; \{\mathsf{PostSnd}[t : b \xrightarrow{z'} c] \mid t \in Tid(b)\}, \end{aligned}$$

and therefore $\mathsf{PreRcv}[s : a \xrightarrow{z} b] \in L_n \setminus L_0$. Since $\mathsf{PreRcv}[s : a \xrightarrow{z} b]^\bullet = \{\mathsf{rcv}[s : a \xrightarrow{z} b]\}$, it follows by Lemma 4.3.2 that the transition $\mathsf{rcv}[s : a \xrightarrow{z} b]$ fires at some point in τ.

(2) By definition of r_j, $Tid(r_i) \neq s$ for all $1 \leq i \leq j-1$. It follows by Lemma 4.3.5 that

$$\{\mathsf{PreRcv}[s : a \xrightarrow{z} b]\} \;=\; [L_0]_s \;=\; [L_1]_s \;=\; \cdots \;=\; [L_{j-2}]_s \;=\; [L_{j-1}]_s.$$

Since $Tid(r_j) = s$, we have by Lemma 4.3.4(2) that

$$r_j \;\in\; [L_{j-1}^\bullet]_s \;=\; [L_{j-1}]_s^\bullet \;=\; \{\mathsf{PreRcv}[s : a \xrightarrow{z} b]\}^\bullet \;=\; \{\mathsf{rcv}[s : a \xrightarrow{z} b]\}.$$

That is, $r_j = \mathsf{rcv}[s : a \xrightarrow{z} b]$ as desired. Note for later use in the proof of Statement (4) that, by the firing rule,

$$\begin{aligned} L_j &= L_{j-1} \setminus {}^\bullet\mathsf{rcv}[s : a \xrightarrow{z} b] \cup \mathsf{rcv}[s : a \xrightarrow{z} b]^\bullet \\ &= L_{j-1} \setminus \{\mathsf{PreRcv}[s : a \xrightarrow{z} b]\} \cup \{\mathsf{Acc}[s : b]\} \end{aligned}$$

and therefore

$$[L_j]_s = \{\mathsf{Acc}[s : b]\}. \tag{4.1}$$

(3) The proof proceeds similarly as the proof of statement (2). By definition of r_k, $Tid(r_i) \neq s$ for all $k \leq i \leq n$. It follows by Lemma 4.3.5 that

$$[L_k]_s \;=\; [L_{k+1}]_s \;=\; \cdots \;=\; [L_{n-1}]_s \;=\; [L_n]_s \;=\; \{\mathsf{PostSnd}[s : b \xrightarrow{z'} c]\}.$$

Since $Tid(r_k) = s$, we have by Lemma 4.3.4(1) that

$$r_k \;\in\; [{}^\bullet L_k]_s \;=\; {}^\bullet[L_k]_s \;=\; {}^\bullet\{\mathsf{PostSnd}[s : b \xrightarrow{z'} c]\} \;=\; \{\mathsf{snd}[s : b \xrightarrow{z'} c]\}.$$

That is, $r_k = \mathsf{snd}[s : b \xrightarrow{z'} c]$ as desired. Note for later use in the proof of Statement (4) that, by the firing rule,

$$\{\mathsf{Acc}[s : b]\} \;=\; {}^\bullet\mathsf{snd}[s : b \xrightarrow{z'} c] \;\subseteq L_{k-1},$$

and therefore

$$[L_{k-1}]_s = \{\mathsf{Acc}[s : b]\} \tag{4.2}$$

since L_{k-1} is feasible.

(4) Suppose there exists some $j < l < k$ such that $Tid(r_l) = s$. Without loss of generality, we may suppose l is the smallest such index, i.e., $Tid(r_i) \neq s$ for all $j + 1 \leq i \leq l - 1$. By Lemma 4.3.5 and (4.1), we have that

$$\{\mathsf{Acc}[s : b]\} = [L_j]_s = [L_{j+1}]_s = \cdots = [L_{l-2}]_s = [L_{l-1}]_s.$$

Since $Tid(r_l) = s$, we have by Lemma 4.3.4(2) that

$$r_l \;\in\; [L^\bullet_{l-1}]_s \;=\; [L_{l-1}]^\bullet_s \;=\; \{\mathsf{Acc}[s : b]\}^\bullet \;=\; \{\mathsf{snd}[s : b \xrightarrow{y} d] \mid b \xrightarrow{y} d \in S\}.$$

For some $b \xrightarrow{y} d \in S$, we have by the firing rule that

$$\{\mathsf{PostSnd}[s : b \xrightarrow{y} d]\} \;=\; \mathsf{snd}[s : b \xrightarrow{y} d]^\bullet \;\subseteq\; L_l.$$

Since $[L_{k-1}]_s = \{\mathsf{Acc}[s : b]\}$ by (4.2), and therefore L_{k-1} does not contain the place $\mathsf{PostSnd}[s : b \xrightarrow{y} d]$, we have that that $l < k-1$ and $\mathsf{PostSnd}[s : b \xrightarrow{y} d] \in L_l \setminus L_{k-1}$. By Lemma 4.3.2, there exists some $l < m \leq k-1$ such that $r_m \in \mathsf{PostSnd}[s : b \xrightarrow{y} d]^\bullet = \{\mathsf{sync}[b \xrightarrow{y} d]\}$. But this contradicts the fact that τ is sync-free.

□

We are now ready to prove the main theorem.

Theorem 4.3.12 (Analysis is sound with respect to conformance). *Suppose $M \overset{\omega}{\Rrightarrow} M'$ for some markings M, M' of $\mathscr{N}(\mathcal{G})$ with $M \in \mathcal{M}_0$. Then ω conforms to $\mathcal{G}$.*

Proof. We first establish some preliminary facts about the corresponding sequence $M \Rightarrow M'$ that must have fired in $\mathscr{N}(\mathcal{G})$ in order to have derived $M \overset{\omega}{\Rrightarrow} M'$. Since the [SKIP] rule does not modify the marking and may only be applied to synchronization operations, there exists a memory projection ω' of ω such that $M \overset{\omega'}{\Rrightarrow} M'$ by a [SKIP]-free derivation. There exist reachable markings $M_0, M_0', M_1, M_1', \ldots, M_n, M_n'$ such that

$$M \;=\; M_0 \overset{\omega_0}{\Rrightarrow} M_0' \overset{\epsilon}{\Rrightarrow} M_1 \overset{\omega_1}{\Rrightarrow} M_1' \overset{\epsilon}{\Rrightarrow} \cdots \overset{\epsilon}{\Rrightarrow} M_n \overset{\omega_n}{\Rrightarrow} M_n' \;=\; M',$$

where $\omega' = \omega_0\omega_1 \ldots \omega_n$, $M_{i-1}' \overset{\epsilon}{\Rrightarrow} M_i$ by the rule [SYNC] for all $1 \leq i \leq n$, and $M_i \overset{\omega_i}{\Rrightarrow} M_i'$ by a [SKIP]- and [SYNC]-free derivation for all $0 \leq i \leq n$. There exist corresponding transitions $\mathsf{sync}[a_1 \xrightarrow{z_1} b_1], \mathsf{sync}[a_2 \xrightarrow{z_2} b_2], \ldots, \mathsf{sync}[a_n \xrightarrow{z_n} b_n]$ and sync-free sequences $\tau_0, \tau_1, \ldots, \tau_n$ of transitions in $\mathscr{N}(\mathcal{G})$ such that

$$M_0 \overset{\tau_0}{\Longrightarrow} M_0' \xrightarrow{\mathsf{sync}[a_1 \xrightarrow{z_1} b_1]} M_1 \overset{\tau_1}{\Longrightarrow} M_1' \xrightarrow{\mathsf{sync}[a_2 \xrightarrow{z_2} b_2]} \cdots \xrightarrow{\mathsf{sync}[a_n \xrightarrow{z_n} b_n]} M_n \overset{\tau_n}{\Longrightarrow} M_n'.$$

Before showing conformance, we must define the path p in $\mathcal{G}$ to which ω conforms. Since the firing sequence $M_i \overset{\tau_i}{\Longrightarrow} M_i'$ is sync-free and M_i is b_i-feasible, it follows by Lemma 4.3.9 that M_i' is b_i-feasible. Since M_i' is also a_{i+1}-feasible, it follows by Lemma 4.3.8 that $b_i = a_{i+1}$. As we are given that $a_i \xrightarrow{z_i} b_i$ is an edge for all $1 \leq i \leq n$, we have that the sequence $p = a_1 \xrightarrow{z_1} b_1 \xrightarrow{z_2} b_2 \xrightarrow{z_3} \cdots \xrightarrow{z_{n-1}} b_{n-1} \xrightarrow{z_n} b_n$ forms a path in $\mathcal{G}$. For notational convenience, set $c_0 := a_1$ and $c_i := b_i$ for all $1 \leq i \leq n$ so that

$$p \;=\; c_0 \xrightarrow{z_1} c_1 \xrightarrow{z_2} \cdots \xrightarrow{z_{n-1}} c_{n-1} \xrightarrow{z_n} c_n.$$

We now show that ω conforms to p. In particular, recalling from above that $\omega' = \omega_0\omega_1 \ldots \omega_n$ is a memory projection of ω, we show that

$$Proj_s(\omega_i) \;\in\; \begin{cases} \mathtt{rcv}(s, z_i) \cdot (\; \mathtt{wr}(s) \;\mid\; \mathtt{rd}(s) \;)^* \cdot \mathtt{snd}(s, z_{i+1}) & \text{if } s \in Tid(c_i),\ c_i = \textsc{ex}(s) \\ \mathtt{rcv}(s, z_i) \cdot (\quad\; \mathtt{rd}(s) \quad\;)^* \cdot \mathtt{snd}(s, z_{i+1}) & \text{if } s \in Tid(c_i),\ c_i = \textsc{rs}(S) \\ \epsilon & \text{if } s \notin Tid(c_i) \end{cases}$$

for all $0 < i < n$ and $s \in \mathit{Thread}$; the cases $i = 0$ and $i = n$ follow by specialized but similar arguments.

We may write $\omega_i = \mu_0 o_1 \mu_1 o_2 \dots o_m \mu_m$ such that $\mu_j \in MemOp^*$ for all $0 \leq j \leq m$ and $o_j \in SyncOp$ for all $1 \leq j \leq m$. Since our analysis does not modify its state on memory operations, there exist markings $N_0, N_1, \dots, N_{m-1}, N_m$ such that

$$\cdots \overset{\epsilon}{\Rrightarrow} M_i \;=\; N_0 \underbrace{\overset{\mu_0}{\Rrightarrow} N_0 \overset{o_1}{\Rrightarrow} N_1 \overset{\mu_1}{\Rrightarrow} N_1 \overset{o_2}{\Rrightarrow} \cdots \overset{o_m}{\Rrightarrow} N_m \overset{\mu_m}{\Rrightarrow}}_{\omega_i} N_m \;=\; M_i' \overset{\epsilon}{\Rrightarrow} \cdots . \tag{4.3}$$

As this holds by a [SKIP]- and [SYNC]-free derivation, we have that each step $N_{j-1} \overset{o_j}{\Rrightarrow} N_j$ is derived from a corresponding non-sync transition $N_{j-1} \overset{r_j}{\Longrightarrow} N_j$ by one of the rules [SEND] or [RECEIVE]. Note in particular that, by inspection of [SEND] or [RECEIVE], $Tid(o_j) = Tid(r_j)$. Overall, we have the corresponding firing sequence

$$\cdots \xRightarrow{\mathsf{sync}[a_i \xrightarrow{z_i} b_i]} M_i \;=\; N_0 \underbrace{\overset{r_1}{\Longrightarrow} N_1 \overset{r_2}{\Longrightarrow} \cdots \overset{r_m}{\Longrightarrow}}_{\tau_i} N_m \;=\; M_i' \xRightarrow{\mathsf{sync}[a_{i+1} \xrightarrow{z_{i+1}} b_{i+1}]} \cdots$$

By Lemma 4.3.11, we have

$$Proj_s(\tau_i) \;=\; \begin{cases} \mathsf{rcv}[s : c_{i-1} \xrightarrow{z_i} c_i]\ \mathsf{snd}[s : c_i \xrightarrow{z_{i+1}} c_{i+1}] & \text{if } s \in Tid(c_i) \\ \epsilon & \text{if } s \notin Tid(c_i) \end{cases}.$$

First suppose $s \notin Tid(c_i)$. Our goal is to show $Proj_s(\omega_i) = \epsilon$. By our observation that $Tid(o_j) = Tid(r_j) \neq s$ for all $1 \leq j \leq m$, it follows that no synchronization operations in ω_i are performed by s. It remains to show that no memory operations in ω_i are performed by s. By Lemma 4.3.5, we have that $[N_j]_s = [N_0]_s$ for all $1 \leq j \leq m$. Since $s \notin Tid(c_i)$ and N_0 is c_i-local, it follows that $|[N_j]_s| = 0$ for all $0 \leq j \leq m$. Now, note by inspection of the rules [WRITE EXCLUSIVE], [READ EXCLUSIVE], and [READ SHARED] in Figure 4.14 that a memory operation performed by s on any marking N_j implies $\mathsf{Acc}[s : a] \in N_j$ for some access mode a. However, this would contradict the fact that $|[N_j]_s| = 0$. Thus, no memory operations are performed by s in ω_i either.

Now suppose $s \in Tid(c_i)$. There exist indices $1 \leq k \leq l \leq m$ such that

- $r_k = \mathsf{rcv}[s : c_{i-1} \xrightarrow{z_i} c_i]$;
- $r_l = \mathsf{snd}[s : c_i \xrightarrow{z_{i+1}} c_{i+1}]$; and
- $Tid(r_j) \neq s$ for all $j \neq l$ or $j \neq k$.

By the firing rule and the feasibility of the markings N_j for the first two bulleted items, and by Lemma 4.3.5 for the third bulleted item, it follows that

- $[N_{k-1}]_s \;=\; {}^\bullet\mathsf{rcv}[s : c_{i-1} \xrightarrow{z_i} c_i]$ and $[N_k]_s \;=\; \mathsf{rcv}[s : c_{i-1} \xrightarrow{z_i} c_i]^\bullet$;
- $[N_{l-1}]_s \;=\; {}^\bullet\mathsf{snd}[s : c_i \xrightarrow{z_{i+1}} c_{i+1}]$ and $[N_l]_s \;=\; \mathsf{snd}[s : c_i \xrightarrow{z_{i+1}} c_{i+1}]^\bullet$; and

- $[N_j]_s = \begin{cases} [N_{k-1}]_s & \text{if } 0 \leq j \leq k-1 \\ [N_k]_s & \text{if } k \leq j \leq l-1 \\ [N_l]_s & \text{if } l \leq j \leq m \end{cases}$

Computing the pre-sets and post-sets, we have that

$$[N_j]_s = \begin{cases} \{\mathsf{PreRcv}[s : c_{i-1} \xrightarrow{z_i} c_i]\} & \text{if } 0 \leq j \leq k-1 \\ \{\mathsf{Acc}[s : c_i]\} & \text{if } k \leq j \leq l-1 \\ \{\mathsf{PostSnd}[s : c_i \xrightarrow{z_{i+1}} c_{i+1}]\} & \text{if } l \leq j \leq m \end{cases}$$

By inspection of our update rules in Figure 4.14, it follows that

$$Proj_s(\mu_j) \in \begin{cases} \epsilon & \text{if } 0 \leq j \leq k-1 \\ (\mathtt{wr}(s) \mid \mathtt{rd}(s))^* & \text{if } k \leq j \leq l-1,\ c_i = \textsc{ex}(s) \\ (\quad \mathtt{rd}(s) \quad)^* & \text{if } k \leq j \leq l-1,\ c_i = \textsc{rs}(S) \\ \epsilon & \text{if } l \leq j \leq m \end{cases}$$

and

$$Proj_s(o_j) = \begin{cases} \mathsf{rcv}[s : z_i] & \text{if } j = k \\ \mathsf{snd}[s : z_i + 1] & \text{if } j = l \\ \epsilon & \text{otherwise} \end{cases}$$

This completes the proof. □

4.4 Chapter Summary

We have formally defined the SFG model for synchronization disciplines, a compact specification language in which terms represent SFG constructions, and how an execution trace conforms to a specified SFG $\mathcal{G}$. We have also presented an accompanying dynamic analysis that enforces conformance to $\mathcal{G}$. In particular, we have shown how $\mathcal{G}$ may be refined into a more detailed Petri net $\mathscr{N}(\mathcal{G})$ whose simulation at run time enforces conformance to $\mathcal{G}$. We have verified that conformance implies race freedom, and that our analysis is sound with respect to conformance.

Chapter 5

Extensions and Future Work

There are a number of interesting directions for future work. These go toward improving either the expressiveness of our specification language or the efficiency of our enforcement technique. We discuss the current limitations of our work with respect to these objectives along with specific ideas for addressing them.

5.1 Higher-Order Synchronization Mechanisms

In Section 4.2.3, we discussed how a new synchronization mechanism z may be added to our specification language simply by specifying its `snd` operation, its `rcv` operation, and the set $Msg(z) \subseteq \mathcal{P}(Thread) \times \mathcal{P}(Thread)$. Neither conformance nor the soundness of our enforcement technique depend on the choice of the set $SyncMech$ or $Msg(z)$, so our work would immediately extend to disciplines incorporating the new mechanism z. This presents interesting possibilities for specifying and enforcing higher-order synchronization disciplines.

For example, consider again the program in Figure 3.3, where threads `Thread0` and `Thread1` synchronize their accesses to `x` via the volatile variable `v`. We might specify this discipline as $[\texttt{EX}(\texttt{Thread0}) \sqcup \texttt{EX}(\texttt{Thread1})]^{\texttt{v}}$; conformance to this discipline guarantees that any two accesses to `x` by different threads is ordered by some volatile-write-read pair. However, this specification misses the important fact that `Thread0` should wait to access `x` until the condition `(v == 0)` is satisfied, and `Thread1` should wait to access `x` until the condition `(v == 1)` is satisfied.

To specify the stronger discipline, we can add two new synchronization mechanisms `vflag(0)`

and `vflag(1)` with the following specifications:

$$\texttt{vflag(0)} \;:\; \left\{ \begin{array}{rcl} \texttt{snd(vflag(0))} & \equiv & \texttt{v = 0} \\ \texttt{rcv(vflag(0))} & \equiv & \texttt{(v == 0)} \\ Msg(\texttt{vflag(0)}) & \equiv & \{\ (\{\texttt{Thread1}\}, \{\texttt{Thread0}\})\ \} \end{array} \right\}$$

$$\texttt{vflag(1)} \;:\; \left\{ \begin{array}{rcl} \texttt{snd(vflag(1))} & \equiv & \texttt{v = 1} \\ \texttt{rcv(vflag(1))} & \equiv & \texttt{(v == 1)} \\ Msg(\texttt{vflag(1)}) & \equiv & \{\ (\{\texttt{Thread0}\}, \{\texttt{Thread1}\})\ \} \end{array} \right\}$$

Then the desired discipline may be specified as $[\texttt{EX(Thread0)} \rhd^{\texttt{vflag(1)}} \texttt{EX(Thread1)}]^{\texttt{vflag(0)}}$. Notably, this discipline is the first we have specified so far that exhibits data-dependence.

For another example, consider a program that utilizes a thread-safe queue `q` such that objects `o` are not accessed while they are enqueued. The internal synchronization of `q` guarantees that the enqueueing of `o` happens-before the dequeuing of `o`. Thus, we may specify a new synchronization mechanism `q(o)` as follows:

$$\texttt{q(o)} \;:\; \left\{ \begin{array}{rcl} \texttt{snd(q(o))} & \equiv & \texttt{enqueue(q,o)} \\ \texttt{rcv(q(o))} & \equiv & \texttt{(dequeue(q) == o)} \\ Msg(\texttt{q(o)}) & \equiv & \{\ (\{s\}, \{t\}) \mid s \in Thread,\ t \in Thread\ \} \end{array} \right\}.$$

If the object `o` is repeatedly enqueued, dequeued, and processed by a set T of worker threads, then we might specify its synchronization discipline as $[\bigsqcup_{t \in T} \texttt{EX}(t)]^{\texttt{q(o)}}$. Note that enforcing this higher-order discipline rather than running a traditional dynamic race detector has the advantage of being able to avoid instrumentation on any of the internal synchronization in `q`, which may incur significant savings in overhead as discussed in Section 3.1.2.

These examples suggest that a more thorough investigation in this direction could yield good results for efficient, modularized race detection.

5.2 Locking Disciplines

The SFG model in its current form is limited with respect to locking disciplines, where a specified lock is held on every access to a variable. The closest approximation to this is something like the SFG $\mathcal{G} = [\![[\texttt{EX(t0)} \sqcup \texttt{EX(t1)} \sqcup \texttt{EX(t2)}]^{\texttt{m}}]\!]$ displayed in the fourth row of Table **??**. Conformance to $\mathcal{G}$ ensures that any pair of accesses of the protected variable performed by different threads are ordered by synchronization on `m`. This is not sufficient to ensure that `m` is actually *held* on every access.

For example, consider the trace ω displayed in Figure 5.1. The highlighted memory projection ω' of ω shows that ω conforms to $\mathcal{G}$ (specifically, by noting $\omega' = \omega_1\omega_2\omega_3$ with $\omega_1 = \texttt{wr(t0)}\ \texttt{rel(t0,m)}$, $\omega_2 = \texttt{acq(t1,m)}\ \texttt{rd(t1)}\ \texttt{rel(t1,m)}$, $\omega_3 = \texttt{acq(t2,m)}\ \texttt{wr(t2)}$, and therefore ω' conforms to the path $\texttt{EX(t0)} \xrightarrow{\texttt{m}} \texttt{EX(t1)} \xrightarrow{\texttt{m}} \texttt{EX(t2)}$ in $\mathcal{G}$). However, no access in ω is performed while the accessing thread holds `m`. The fundamental issue is that the `m`-labeled transitions in $\mathcal{G}$ describe how one should order pairs of accesses performed across different access modes, whereas the condition of holding `m` on

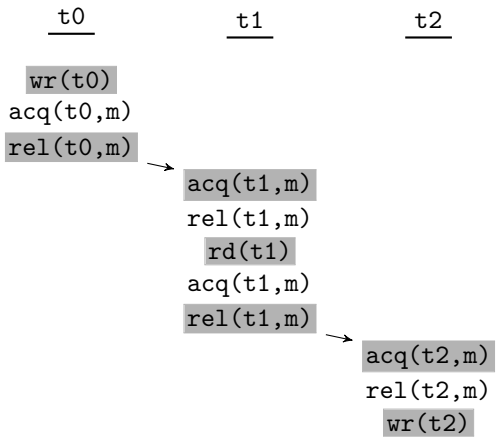


Figure 5.1: A trace ω that conforms to $\mathcal{G} = [\![[\texttt{EX(t0)} \sqcup \texttt{EX(t1)} \sqcup \texttt{EX(t2)}]^{\texttt{m}}]\!]$. Highlighted in gray is the memory projection ω' that satisfies the definition of conformance.

every access must be additionally enforced *within* each access mode.

A possible solution to this issue is to generalize the concept of an access mode. In addition to the synchronization-free access modes $\textsc{ex}(t)$ and $\textsc{rs}(T)$, SFGs could incorporate the access mode $\textsc{lock}_m(T)$ to represent mutually exclusive access by any of the threads in $T \subseteq \mathit{Thread}$ while the lock m is held. Whereas $\textsc{ex}(t)$ and $\textsc{rs}(T)$ simply allow unfettered access so long as the accesses are of the appropriate type, $\textsc{lock}_m(T)$ would additionally stipulate that the accessing thread has acquired but not released m. The challenge lies in identifying the exact details of how $\textsc{lock}_m(T)$ should compose with arbitrary SFGs and what Petri net structure should be refined from this mode such that the specification grammar remains flexible.

5.3 Scaling to Many Disciplines

Recall that our dynamic analysis in Figure 4.14 was defined with respect to a program model with a single memory location. Unfortunately, in its current form, the analysis does not scale well to many memory locations.

To see this, suppose we extend our program model so that it has a set Var of any number of memory locations, that some set $X \subseteq \mathit{Var}$ of variables has been annotated using our specification language, and that we would like to verify that the accesses to each variable $x \in X$ conform to its specified SFG $\mathcal{G}_x$. Figure 5.2 shows the obvious solution, which is simply to run copies of our original analysis for each variable $x \in X$ simultaneously. The total analysis state consists of a map $\mathbf{M}$ from variables $x \in X$ to markings $\mathbf{M}[x]$ of the respective Petri nets $\mathscr{N}(\mathcal{G}_x)$. Each marking $\mathbf{M}[x]$ may update its state by the rules enclosed in the box; these are identical to our original analysis in Figure 4.14, but with an additional subscript x on the relations to indicate that they are defined with respect to the Petri net $\mathscr{N}(\mathcal{G}_x)$.

How the total analysis handles operations is given by the rules below the box. Memory operations are simple because each is relevant only to the variable that it accesses; hence, as shown in the

$$\frac{\mathsf{Acc}[t : \textsc{ex}(t)] \in M}{M \overset{\mathtt{wr}(t,x)}{\Rrightarrow}_x M} \;[\textsc{write exclusive}]_x \qquad \frac{\mathsf{Acc}[t : \textsc{ex}(t)] \in M}{M \overset{\mathtt{rd}(t,x)}{\Rrightarrow}_x M} \;[\textsc{read exclusive}]_x \qquad \frac{\mathsf{Acc}[t : \textsc{rs}(T)] \in M}{M \overset{\mathtt{rd}(t,x)}{\Rrightarrow}_x M} \;[\textsc{read shared}]_x$$

$$\frac{M \xrightarrow{\mathsf{snd}[t : a \xrightarrow{z} b]}_x M'}{M \overset{\mathtt{snd}(t,z)}{\Rrightarrow}_x M'} \;[\textsc{send}]_x \qquad \frac{M \xrightarrow{\mathsf{rcv}[t : a \xrightarrow{z} b]}_x M'}{M \overset{\mathtt{rcv}(t,z)}{\Rrightarrow}_x M'} \;[\textsc{receive}]_x \qquad \frac{M \xrightarrow{\mathsf{sync}[a \xrightarrow{z} b]}_x M'}{M \overset{\epsilon}{\Rrightarrow}_x M'} \;[\textsc{sync}]_x$$

$$\frac{o \in \mathit{SyncOp}}{M \overset{o}{\Rrightarrow}_x M} \;[\textsc{skip}]_x$$

$$\frac{\mathbf{M}[x] \overset{\mathtt{wr}(t,x)}{\Rrightarrow}_x \mathbf{M}'[x]}{\mathbf{M} \overset{\mathtt{wr}(t,x)}{\Rrightarrow} \mathbf{M}'} \;[\textsc{write}] \qquad \frac{\mathbf{M}[x] \overset{\mathtt{rd}(t,x)}{\Rrightarrow}_x \mathbf{M}'[x]}{\mathbf{M} \overset{\mathtt{rd}(t,x)}{\Rrightarrow} \mathbf{M}'} \;[\textsc{read}] \qquad \frac{o \in \mathit{SyncOp} \qquad \forall x \,.\, \mathbf{M}[x] \overset{o}{\Rrightarrow}_x \mathbf{M}'[x]}{\mathbf{M} \overset{o}{\Rrightarrow} \mathbf{M}'} \;[\textsc{slow sync}]$$

Figure 5.2: An extension of our original analysis in Figure 4.14 so that it enforces specified disciplines for multiple variables. The antecedent highlighted in gray shows that the overhead on synchronization operations grows linearly with the number of memory locations for which a discipline is being enforced.

rules [WRITE] and [READ], only a single marking needs to be updated. On the other hand, a synchronization operation is potentially relevant to the analysis for every variable. Thus, the marking for every variable must be checked and possibly updated on synchronization operations, as highlighted in grey in the rule [SLOW SYNC]. In other words, the overhead on synchronization operations scales linearly with the number of checked variables.

Avoiding this kind of prohibitive overhead would require a technique that does not update the instrumentation state for any memory location on synchronization operations. One possible approach is the incorporation of a lazy update policy, such as that employed by the dynamic race detector Goldilocks [1]. Rather than globally updating its instrumentation store on synchronization operations, Goldilocks maintains a synchronization event queue that records each synchronization event. On a memory access to x, Goldilocks analyzes the portion of this queue up to the last access to x to check for races.

Such an approach may be particularly fruitful in the context of enforcing specified synchronization disciplines. Using a similar lazy policy, our analysis would not need to examine every synchronization operation since the last access, but could safely ignore those operations on mechanisms that are not relevant to the current state within the discipline. This suggests that the synchronization event queue could be partitioned across synchronization mechanisms for fast lookup, and even possibly compressed for each mechanism for fast analysis.

Chapter 6

Conclusions

We have presented a new framework for specifying and enforcing synchronization disciplines in multithreaded programs. In particular, we have presented a source-level annotation language for disciplines and a corresponding technique for dynamically enforcing conformance to specified disciplines. Our work introduces a new abstraction for modeling synchronization disciplines called synchronization flow graphs; these directly encode the high-level ordering patterns imposed upon program execution in terms of access modes and transitions via synchronization mechanisms. The SFG model allows for the expression of a wide variety of time-varying synchronization disciplines; at the same time, by using a small set of base graphs and graph operators, SFGs are expressed compactly in our specification language.

Our enforcement technique pre-processes and automatically refines a specified SFG into a more detailed Petri net, which is then simulated at run time to enforce conformance to the SFG. We have verified that this technique is sound with respect to conformance, and that a trace that conforms to any SFG is race-free.

Our framework is currently limited with respect to expressing proper locking disciplines as well as performance; however, we have discussed some interesting and promising directions for addressing these limitations. The general flexibility with which our framework incorporates synchronization mechanisms makes it highly extensible and thereby a potential platform for a more universal, higher-order discipline specification language.

Bibliography

[1] Elmas, T., Qadeer, S., and Tasiran, S. Goldilocks: A race and transaction-aware Java runtime. In *Conference on Programming Language Design and Implementation (PLDI)* (2007), pp. 245–255.

[2] Engler, D. R., and Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP)* (2003), pp. 237–252.

[3] Flanagan, C., and Freund, S. N. Type-based race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)* (2000), pp. 219–232.

[4] Flanagan, C., and Freund, S. N. FastTrack: Efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)* (2009), pp. 121–133.

[5] Flanagan, C., and Freund, S. N. RedCard: Redundant check elimination for dynamic race detectors. In *European Conference on Object-Oriented Programming* (2013), pp. 255–280.

[6] Itzkovitz, A., Schuster, A., and Zeev-Ben-Mordehai, O. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput. 59*, 2 (1999), 180–203.

[7] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[8] Leveson, N. G., and Turner, C. S. Investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (1993), 18–41.

[9] Naik, M., Aiken, A., and Whaley, J. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)* (2006), pp. 308–319.

[10] O'Callahan, R., and Choi, J. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2003), pp. 167–178.

[11] Oracle. Java documentation: Programming with assertions. `http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`.

[12] Poulsen, K. Tracking the blackout bug. `http://www.securityfocus.com/news/8412`.

[13] Pozniansky, E., and Schuster, A. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience 19*, 3 (2007), 327–340.

[14] Racket Developers. Racket documentation: Contracts. `http://docs.racket-lang.org/guide/contracts.html`.

[15] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. Eraser: A dynamic data race detector for multi-threaded programs. In *Symposium on Operating Systems Principles (SOSP)* (1997), pp. 27–37.

[16] Serebryany, K., and Iskhodzhanov, T. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (2009), WBIA '09, pp. 62–71.

[17] Spector-Zabusky, A. Dynamic race condition detection via synchronization specification automata. Bachelor's thesis, Williams College, 2012.

[18] Valgrind Developers. Helgrind: A thread error detector. `http://valgrind.org/docs/manual/hg-manual.html`.

[19] Wood, B. Hominy Grits: Specification and inference of synchronization disciplines for concurrent programs. Bachelor's thesis, Williams College, 2008.