MEMORY MODEL SENSITIVE ANALYSIS OF CONCURRENT DATA TYPES

Sebastian Burckhardt

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2007

Rajeev Alur, Milo Martin
Supervisors of Dissertation

Rajeev Alur
Graduate Group Chairperson

To my wife Andrea and
my children Sasha and Ellie

ABSTRACT

MEMORY MODEL SENSITIVE ANALYSIS OF CONCURRENT DATA TYPES

Sebastian Burckhardt

Rajeev Alur, Milo Martin

Concurrency libraries can facilitate the development of multi-threaded programs by providing concurrent implementations of familiar data types such as queues and sets. Many optimized algorithms that use lock-free synchronization have been proposed for multiprocessors. Unfortunately, such algorithms are notoriously difficult to design and verify and may contain subtle concurrency bugs. Moreover, it is often difficult to place memory ordering fences in the implementation code. Such fences are required for correct function on relaxed memory models, which are common on contemporary multiprocessors.

To address these difficulties, we describe an automated verification procedure based on bounded model checking that can exhaustively check all concurrent executions of a given test program on a relaxed memory model and can either (1) verify that the data type is sequentially consistent, or (2) construct a counterexample.

Given C implementation code, a bounded test program, and an axiomatic memory model specification, our *CheckFence* prototype verifies operation-level sequential consistency by encoding the existence of inconsistent executions as solutions to a propositional formula and using a standard SAT solver to decide satisfiability.

We applied *CheckFence* to five previously published algorithms and found several bugs (some not previously known). Furthermore, we determined where to place memory ordering fences in the implementations and verified their sufficiency on our test suite.

Contents

List of Figures

Chapter 1

Introduction

Shared-memory multiprocessor architectures are now ubiquitous. With today's advanced chip manufacturing technology, several processors can easily fit onto a single chip. In fact, because of the better performance/power ratio of multiprocessors, hardware architects now prefer to provide multiple cores rather than a highly optimized single core.

Unfortunately, even as multiprocessor hardware becomes cheap, writing software that can unleash its performance potential remains challenging [67]. We see this gap as an opportunity for researchers, as it is likely to lead to a high demand for strategies that can aid programmers to write simple, correct, efficient and portable multi-threaded programs.

In this introductory chapter, we first describe the specific problem this work addresses (Section 1.1). Next, we present our solution and report on our practical experiences applying it (Section 1.2). Then, we compare our work to prior results and tools in this general area (Section 1.3). Finally, we conclude this chapter with a description of the logical structure of this document and a brief summary of each of the remaining chapters (Section 1.5).

1.1 The Problem

The specific problem we are solving is the verification of concurrency libraries that use lock-free synchronization and target multiprocessor architectures with relaxed memory models.

Concurrency Libraries

Concurrency libraries can alleviate some of the problems associated with the development of concurrent software. For example, the java.util.concurrent package JSR-166 [46] or the Intel Threading Building Blocks [40] support the development of safe and efficient multi-threaded programs by providing *concurrent data types*, that is, concurrent implementations of familiar data abstractions such as queues, sets, or maps.

By raising the data abstraction level, concurrency libraries can free the programmer from some of the issues associated with safe sharing of data between threads. Moreover, libraries can also improve performance, as they may use algorithms that are optimized for concurrency.

Lock-free Synchronization

Many sophisticated algorithms for concurrent data types that use lock-free synchronization have been proposed [29, 30, 52, 54, 55, 66]. Such implementations are not race-free in the classic sense because they allow concurrent access to shared memory locations without using locks for mutual exclusion.

Algorithms with lock-free synchronization are notoriously hard to verify, as witnessed by many formal verification efforts [12, 25, 70, 74] and by bugs found in published algorithms [17, 53]. Many more interleavings need to be considered than for implementations that follow a strict locking discipline. Moreover, the deliberate use of races prohibits the use of classic race-avoiding design methodologies and race-detecting tools [1, 20, 31, 43, 57, 61, 63, 73].

Relaxed Memory Models

Most commonly used multiprocessor architectures use *relaxed memory ordering models* [2]. For example, a processor may reorder loads and stores by the same thread if they target different addresses, or it may buffer stores in a local queue. Whereas fully lock-based programs are insensitive to the memory model (because the lock and unlock operations are designed to guarantee the necessary memory ordering), implementations that use lock-free synchronization require explicit *memory ordering fences* to function correctly on relaxed memory models. Nevertheless, fence placements are rarely published along with the algorithm.

Memory ordering fences (or, as they are sometimes called, memory barriers) counteract the ordering relaxations by enforcing memory order between preceding and succeeding instructions. A lack of fences can lead to incorrect behavior, whereas an overzealous use of fences impacts performance.

1.2 Our Solution

To help designers and implementors develop correct and efficient programs for relaxed models, we present a verification method which is based on bounded model checking and can check the correctness of an implementation automatically, for a suite of bounded test programs written by the user. Furthermore, it can produce counterexample traces if the implementation is found to be incorrect.

We describe our solution in the following three sections. Section 1.2.1 describes our tool *CheckFence* from a “black-box” perspective. In Section 1.2.2, we describe in more detail how our tool checks the sequential consistency of an implementation.

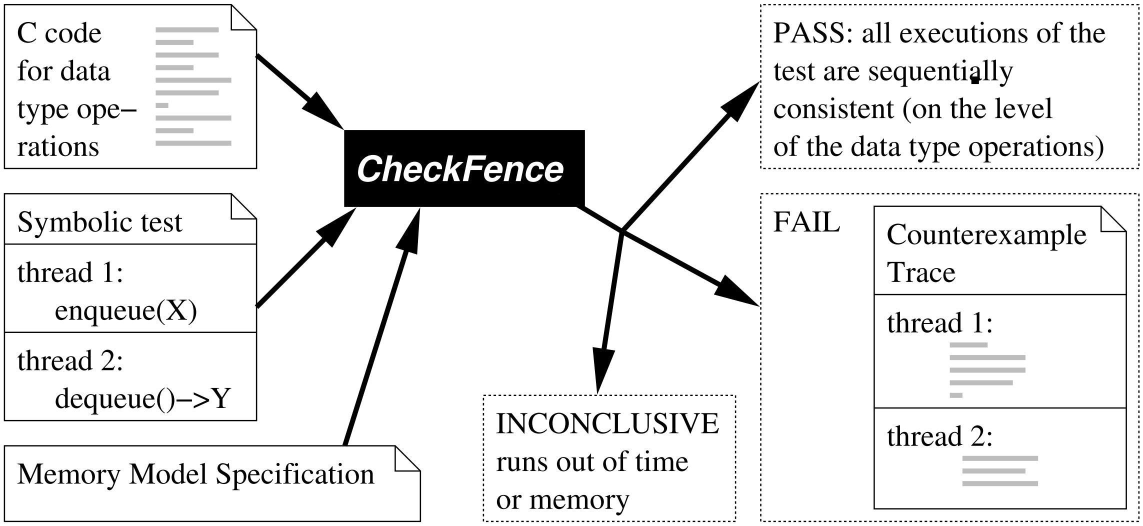

C code for data type ope– rations
Symbolic test
thread 1: enqueue(X)
thread 2: dequeue()–>Y
Memory Model Specification
CheckFence
PASS: all executions of the test are sequentially consistent (on the level of the data type operations)
FAIL
Counterexample Trace
thread 1:
thread 2:
INCONCLUSIVE runs out of time or memory

Figure 1.1: Black-box view of the *CheckFence* tool.

Finally, Section 1.2.3 summarizes our experiences with applying the tool to some implementations from the literature.

1.2.1 The *CheckFence* Tool

We implemented our verification method in our *CheckFence* tool (Fig. 1.1). Given (1) the implementation code, (2) a bounded test, and (3) an axiomatic memory model specification, *CheckFence* checks all executions and returns one of the following answers:

- PASS: all executions of the test are correct.
- FAIL: an incorrect execution was found, and is presented to the user in the form of a counterexample trace.
- INCONCLUSIVE: the tool exhausted its time or space resources.

We use it in the following manner. First, we write a few small and simple bounded tests by hand that call the code being verified. For those, our method can decide quickly whether they are correct, or give us a counterexample if they are not. Then we can manually increase the size of the tests gradually (fixing any bugs we find along the way), until we reach the scalability limits. All counterexamples are sound, and no time is thus wasted on false warnings.

Although we can not completely verify an implementation in this manner (because our test suite may be incomplete and not cover all possible cases), our experience suggests that most bugs can be found on simple, small tests, especially the bugs that are caused by relaxations in the memory model.

1.2.2 Checking Operation-Level Sequential Consistency

Our correctness condition is *sequential consistency* [44] on the operation level, which requires that for all client programs and all executions, the observed outcome is consistent with some atomic interleaving of the operations by each thread. Note that the consistency of the data type operations is independent of the consistency model used by the underlying hardware: a concurrent data type may be sequentially consistent on the level of the data type operations even if the underlying multiprocessor's memory model (which is specified as one of the inputs to the tool) is not sequentially consistent.

Bounded Tests

Each bounded test specifies a finite sequence of operation invocations for each thread and may use nondeterministic input arguments. Because the total number of instructions is limited for each test, the number of executions is finite. Unlike deterministic tests, however, a bounded test has an exponential number of executions, because it considers all possible inputs, all instruction interleavings and reorderings, and nondeterministic dynamic memory allocation. It is thus easy to cover many executions with just a few tests.

Note that without bounding the executions in some way, the verification problem is generally undecidable because (1) C programs that access unbounded amounts of memory are Turing-complete, and (2) sequential consistency is undecidable even for implementations with finite memory [3].

Specification Mining

The definition of sequential consistency assumes that we have a formal specification of the serial semantics of the data type. Because we prefer not to require the user to write a formal specification, we use an automatic *specification mining* algorithm that can extract a specification directly from an implementation. The idea is that we assume that the serial executions of the implementation are correct and can thus serve as the serial specification of the data type.

Note that we can verify the serial executions separately, using more conventional techniques: for serial executions, only one thread executes at a time, and the threads are interleaved along operation boundaries only. They are thus much easier to verify. Also, we have the option of writing a separate reference implementation (which need not be concurrent and is thus much simpler) and extract the specification from the latter.

Encoding

At the heart of our method is an encoding that represents the executions of the test program (on a relaxed memory model) as solutions to a propositional formula. More specifically, we construct a formula that has a satisfying valuation if and only if there exists an incorrect execution. Then, we use a standard SAT solver to decide if such a solution exists; if it does, we can use the assignment provided by the SAT solver to construct the counterexample trace.

Handling C Code

The C language does not have a language-level memory model, and does not explicitly specify what load and store instructions a program may produce. To accurately model synchronization instructions and the effects of the memory model, we require a lower-level representation that makes the loads and stores explicit.

To this end, we define an intermediate language called LSL (load-store language) that has (a) a small syntax, (b) a well-defined semantics even for weak memory models, and (c) support for spin loops, atomic blocks, and assertions.

Our front end accepts a reasonable subset of C and performs an automatic translation into LSL.

Memory Model Specifications

Unfortunately, researchers, engineers and programmers do not always agree on how to interpret the memory model specifications found in the architecture manuals. Because precise specifications are an absolute prerequisite for formal verification, we establish a solid formal foundation for multiprocessor executions and develop a specification format for axiomatic memory models.

Moreover, we capture the most common relaxations (store buffers, out-of-order execution) by defining a memory model *Relaxed*. This model abstracts unneeded details and provides a common conservative approximation of the Sun SPARC v9 TSO/PSO/RMO [72], Alpha [13], and IBM zArchitecture [41]. We chose not to include a third relaxation (non-global store orders) in *Relaxed*, which is thus not quite as weak as IBM PowerPC [24], IA-32 [38], or IA-64 [39]. It remains on our list of future work to explore the impact of non-global store orders on the correctness of implementations.

We give a brief introduction to memory models and present examples of these relaxations in Section 3.1.

1.2.3 Experiments

We applied *CheckFence* to five previously published algorithms, writing C code that closely follows the published pseudocode, and creating test programs. We were thus able to

1. Reproduce two known bugs in a concurrent deque algorithm known as “snark” [14, 17, 45].

2. Uncover a not-previously-known bug in a lazy list-based set implementation. The published pseudocode [30] fails to initialize a field, which was missed by a prior formal verification using the PVS interactive proof system [12].

3. Show numerous failures on architectures with relaxed memory models (the original algorithms assume sequentially consistent architectures), and fix them by inserting appropriate memory ordering fences.

Our experiments confirm that the *CheckFence* method is very efficient at verifying or falsifying small testcases. As small testcases appear to be sufficient to find memory-model-related bugs (all of the testcases that we required to expose missing fences contained fewer than 200 memory accesses and took less than 10 minutes to verify), *CheckFence* achieves its main goal.

Furthermore, *CheckFence* proved to be useful to find algorithmic bugs that are not related to the memory model. In general, however, we found that the tool does not scale for testcases that require more than a few hundred memory accesses.

1.3 Related Work

Of course, many aspects of our method were inspired by previous work and build upon the results of others. The following list should be representative for the various research areas that we found relevant for our work.

Verification of Concurrent Data Types

Most prior work on verification of concurrent data types is based on interactive proof construction and assumes a sequentially consistent memory model [12, 25, 62, 70]. These methods are far from automated, but require a laborious proof construction. Shape analysis can provide a somewhat higher degree of automation[74], but still requires sophisticated user guidance. To our knowledge, analogous proof strategies for relaxed memory models have not been investigated.

Runtime verification is highly automatic and scalable (yet less complete) and has been applied successfully in this domain [18, 68]. These methods require the user to specify the commit points for each operation manually.

Relaxed Memory Models

Reasoning about execution traces by means of ordering relations and axioms was pioneered by Collier [11]. Using similar techniques, Gharacharloo [26] describes a number of practically relevant models and how they are connected to hardware implementations. A similar perspective is assumed by the tutorial [2], which is an excellent introduction to the subject. Contemporary architecture manuals [13, 24, 37, 38, 39, 41, 72] use an axiomatic style for memory model specifications, although they are often not quite formal, nor quite complete. The Java Memory Model [50] also uses an axiomatic style.

Model Checking Code on Relaxed Models

Most previous work on model checking executions on relaxed memory models has focused on relatively small and hand-translated code snippets (such as spinlocks or litmus tests). It can be divided into two categories: explicit-state model checking combined with operational memory models [16, 60], and constraint solving combined with axiomatic memory models [8, 27, 78].

We prefer the latter approach for two reasons: (1) axiomatic models can more easily capture official specifications because they use an axiomatic style, and (2) constraint-based encodings can leverage the advances in SAT solving technology.

Operational Memory Models

The first connection between memory model specifications and verification tools was made using explicit-state model checking [16, 60]. For this approach to work, the axiomatic specification must first be manually translated into an operational model, which can be challenging. This same method was applied again more recently to the C# memory model [36].

We are not aware of any work that demonstrates an application of operational memory models to the verification of programs that contain more than a few lines of code.

Axiomatic Memory Model Specifications

The use of constraint solvers rather than explicit-state model checkers was first applied to the validation of hardware execution traces [27], which inspired our encoding of executions.

Our axiomatic specification format is very similar also to the one used by the NEMOS project [75, 78] (Non-operational yet executable memory ordering specifications), which also proposes the use of SAT solvers to validate execution traces. The main differences are that (1) our approach uses a custom specification format rather than a subset of PROLOG, and (2) we connect the memory model specification with actual C programs and concurrent data type implementations rather than hardware execution traces. More recently, the NEMOS ideas were extended to apply to software verification as well [76, 77]. However, there is no mention of applying these techniques to concurrent data types. Also, they consider a custom, abstract programming language only, and do not demonstrate that their prototype can find issues in real implementations.

Encoding Program Executions

The idea of encoding program executions as formulae is of course not new, in particular for sequential programs. Jackson and Vaziri [42] demonstrate that constraint solving is useful for finding bugs in small, sequential code, using the Alloy constraint solver. The CBMC tool by Clarke, Kroening and Lerda [10] tackles a problem that is closely related to ours, as it translates sequential ANSI C code into SAT constraints. The brief paper on CBMC [10] gave us some important clues on how to encode programs; however, it does leave many aspects undiscussed and does not present an encoding algorithm.

For concurrent programs, a SAT encoding based on bounding the context switches was proposed by Rabinovitz and Grumberg [62]. This encoding is quite different from ours (and handles sequentially consistent executions only): it models the interleavings by associating with each instruction the number of context switches prior to its execution, rather than encoding the memory ordering as a binary relation.

Parts of the design of our intermediate language LSL were inspired by prior work in research areas not related to memory models or concurrent data types. For instance, we liked the elegant use of exceptions as the fundamental control structure in the intermediate language for ESC Java [47]. Also, the work on building verifiable compilers [6, 48] gave helpful clues on the design of a front end, because it describes in careful detail how to transform C into machine language through a series of steps that are then formally verified by the Coq proof assistant.

Fence Insertion

Specialized algorithms to insert memory fences automatically during compilation have been proposed early on [19, 64]. However, these methods are based on a conservative program analysis, which makes them less attractive for highly optimized implementations: inserting fences indiscriminately has a considerable performance impact [69, 71] and should be used sparingly.

Existing Verification Tools

Today's state-of-the-art software verification tools offer only indirect help for implementors of concurrent data types that use lock-free synchronization. Race-detectors are useless, as the implementations contain intentional data races. Many model checkers do not support concurrency at all (BLAST, SLAM, CBMC). The ones that do (Zing, F-Soft, SPIN) assume a sequentially consistent memory model and offer no support for checking high-level correctness conditions such as sequential consistency, which cannot be captured using monitors or assert statements.

Comparison to Our Earlier Publications

We first introduced our verification method in the CAV paper [8]. This paper is written as a case study that describes our encoding of executions of bounded test programs, and reports on our experiences applying it to a small example. In our PLDI paper [9], we add the front end that translates C code, study more examples, and describe the specification mining (replacing the less automatic and slower commit point method we used for the case study). Compared to those papers, this dissertation contains new material in the following areas:

1. We develop a generic specification format for axiomatic memory models. To demonstrate its use, we use it to formalize the Sparc RMO model (Chapter 3).

Furthermore, we give a quantitative comparison of the various memory model encodings in Chapter 8.

2. We describe our intermediate language LSL in detail, and present a formal syntax and semantics (Chapter 4).

3. We present a pseudocode algorithm that shows how we encode programs as formulae, and prove its correctness (Chapter 5).

1.4 Contributions

Verifying concurrent data type implementations that make deliberate use of data races and memory ordering fences is challenging because of the many interleavings and counterintuitive instruction reorderings that need to be considered. Conventional verification tools for multithreaded programs are not sufficient because they make assumptions on the programming style (race-free programs) or the memory model (sequential consistency).

Our verification method as embodied in the *CheckFence* implementation provides a valuable aid to algorithm designers and implementors because it (1) accepts implementations written as C code, (2) supports relaxed memory models and memory ordering fences, and (3) can verify that the implementation behaves correctly for a given bounded test and produce a counterexample trace if it does not.

Our experiments confirm that the *CheckFence* method is very efficient at finding memory-model related bugs. All of the testcases we required to expose missing fences contained fewer than 200 memory accesses and took less than 10 minutes to verify. Furthermore, *CheckFence* proved to be useful to find algorithmic bugs that are not related to the memory model.

1.5 Overview

To facilitate navigation, we provide a brief summary of each of the remaining chapters.

- Chapter 2 lays the technical foundations of our formalization. We establish what we mean by an execution of a program on a multiprocessor, including executions on relaxed memory models.

  Furthermore, we introduce the logic framework (formulae, vocabularies, valuations, etc.) needed for describing memory model specifications (Section 3.3.1) and our encoding of executions (Section 2.2).

- Chapter 3 introduces relaxed memory models and describes our specification format for axiomatic memory models. We conclude this chapter with a listing of the axiomatic specifications for sequential consistency, *Relaxed*, and the SPARC RMO model.

- Chapter 4 formalizes our notion of programs by introducing an intermediate language called LSL (load-store language) with a formal syntax and semantics. Furthermore, it describes how to unroll programs to obtain finite, bounded versions that can be encoded into formulae.

- Chapter 5 describes how to construct a propositional formula whose solutions correspond to the concurrent execution of an unrolled program on the specified memory model. This encoding lays the technical foundation for our verification method. It also contains a correctness proof (which is partially broken out into Appendix A). The construction of this proof heavily influenced our formalization of concurrent executions and memory traces.

- Chapter 6 introduces concurrent data types and shows how we define correctness and verify it by bounded model checking. This chapter shows how to assemble the techniques introduced in the earlier chapters, and explains our verification method.

- Chapter 7 describes the *CheckFence* tool, which implements our verification method. We discuss some of the implementation challenges, such as the translation from C to LSL, and how to encode formulae in CNF form suitable for SAT solving.

- Chapter 8 describes the results of our experiments. It lists both qualitative results (bugs we found in the studied implementations), and quantitative results (performance measurements for the *CheckFence* tool). For completeness, we list the C source code of the studied implementations (along with the fences we inserted) in Appendix B.

- Chapter 9 summarizes our contributions and discusses future work.

Chapter 2

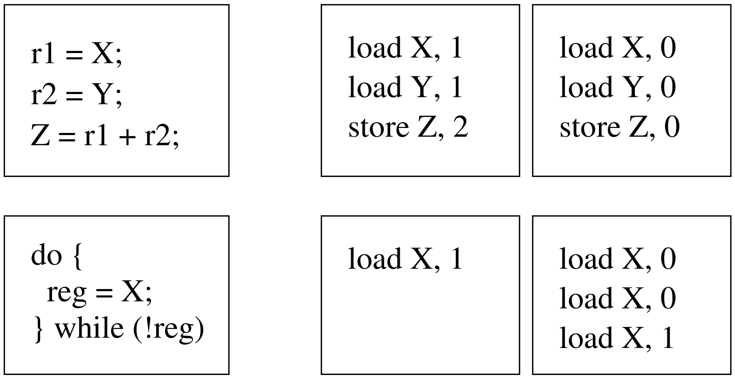
Foundations

In this chapter, we lay the technical foundations of our formalization. We address two topics:

- In Section 2.1, we establish what we mean by an execution of a program on a multiprocessor. While the execution semantics of uniprocessor machines is generally well understood by programmers and well specified by architecture manuals, the situation is less clear for multiprocessors. As we wish to use precise reasoning, we first establish a solid formal foundation for multiprocessor executions.
- In Section 2.2, we present the logic framework needed for describing our encoding. Specifically, we define a syntax and semantics for formulae (featuring types, quantifiers, equality, and limited support for second-order variables). Most of the material in that section is fairly standard and can be skimmed by most readers.

2.1 Multiprocessor Executions

To achieve a clean separation between the local program semantics (local program execution for each thread) and the memory model (interaction between different



```
r1 = X;
r2 = Y;
Z = r1 + r2;
```

```
load X, 1
load Y, 1
store Z, 2
```

```
load X, 0
load Y, 0
store Z, 0
```

```
do {
  reg = X;
} while (!reg)
```

```
load X, 1
```

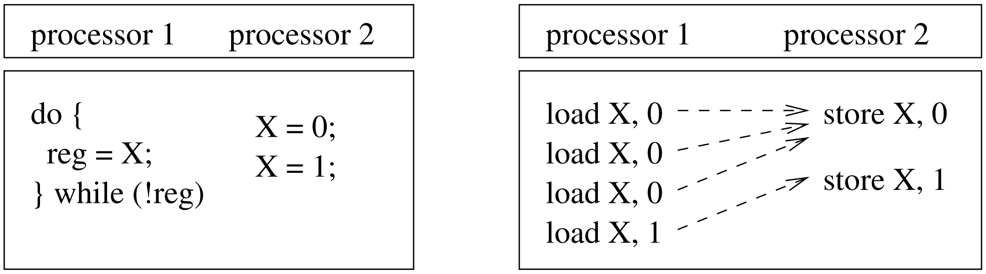
```
load X, 0
load X, 0
load X, 1
```

Figure 2.1: Informal Example: we show two programs on the left, and for each program two possible local memory traces it may produce on the right.

threads via shared memory), we abstractly represent the communication between the program and the memory system using *memory traces.* More specifically, we adopt the following perspective:

- During an execution, each local program performs a logical sequence of load, store, and fence instructions, which can be described abstractly by a *local memory trace.* Fig. 2.1 shows informal examples of programs and some local memory traces they may produce (many others are possible). We define the set *Ltr* of all local traces in Section 2.1.2.

- The memory system combines the local traces into a *global memory trace.* The global trace specifies for each load where its value originated. The origin of the value can be either a store appearing in one of the local traces, or the initial value of the memory location. Fig. 2.2 shows an informal example of a concurrent program and a global trace that it could possibly exhibit (again, other executions are possible). We define the set *Gtr* of all global traces in Section 2.1.4.



```
processor 1        processor 2

do {                X = 0;
  reg = X;          X = 1;
} while (!reg)
```

```
processor 1          processor 2

load X, 0            store X, 0
load X, 0
load X, 0            store X, 1
load X, 1
```

Figure 2.2: Informal Example: the concurrent program on the left may produce the global memory trace on the right (among many other possibilities). The arrows indicate for each load where the value was sourced from.

Trace Semantics

For the purpose of this chapter, we assume an abstract view of programs and memory models. We simply foreshadow the definition of suitable sets *Prog*, *Axm* and the semantic functions E_L and E_G as follows:

- We assume a definition for the set *Prog* of local programs (defined in Chapter 4) and a function

  $$E_L : \mathit{Prog} \to \mathcal{P}(\mathit{Ltr} \times \mathit{Exs})$$

  where *Ltr* is the set of local traces (to be defined in Section 2.1.2 below) and *Exs* is a set of execution states (to be defined in Chapter 4). $E_L(p)$ captures the semantics of the program p in the following sense: $(\boldsymbol{t}, e) \in E_L(p)$ if and only if there is an execution of program p (in some arbitrary environment) that produces the local memory trace $\boldsymbol{t}$ and terminates with execution state e.

- We assume a definition for the set *Axm* of axiomatic memory model specifications (defined in Chapter 3) and a function

  $$E_G : \mathit{Axm} \to \mathcal{P}(\mathit{Gtr})$$

  where *Gtr* is the set of global traces (to be defined in Section 2.1.4 below). $E_G(Y)$ captures the semantics of the memory model specification Y in the following sense: $\boldsymbol{T} \in E_G(Y)$ if and only if the global trace $\boldsymbol{T}$ is allowed by the memory model Y.

As we formalize the semantics of the memory system and the individual threads separately, they both become highly nondeterministic (because they assume the most general behavior of the environment). For a specific concurrent program and memory model, the executions are much more constrained because the possible traces must simultaneously match the semantics of each individual thread and the memory model. When we encode concurrent executions later on (Chapter 5) we express these individual constraints as formulae that must be simultaneously satisfied.

In the remainder of this section, we formalize the ideas outlined above and conclude with a definition of the set of concurrent executions (Section 2.1.6).

2.1.1 Instructions and Values

To formalize memory traces, we need some preliminary definitions that provide the framework for representing the machine instructions, memory addresses and data values used by a program. We also need to consider the fact that programs may terminate abnormally.

1. Memory traces are made up of individual *instructions*. We let $\mathcal{I}$ be a pool of instruction identifiers. We will use these identifiers to distinguish individual instruction *instances*. Note that a single statement in a program may produce more than one instruction instance[1]:

    - Some statements produce several memory accesses. For example, if x, y are variables residing in memory, the assignment $x = y + 1$ first loads the location y, then adds 1, then stores the result to location x, thus producing both a load and a store instruction.

[1] Computer architects sometimes use a terminology that distinguishes between "static" and "dynamic" instructions to express this concept.

- Program statements may be executed more than once if they appear within a loop or a recursive procedure. Each time it gets executed, a statement produces fresh instruction instances.

We keep identifiers unique within an execution, and say ‘instruction’ short for ‘instruction identifier’ or ‘instruction instance’.

2. There are different types of instructions. For our purposes, we care about instructions that are relevant for the memory model only and define the following types: $\mathit{Cmd} = \{\mathit{load}, \mathit{store}, \mathit{fence}\}$. We let the type of each instruction be defined by the function $\mathit{cmd} : \mathcal{I} \rightarrow \mathit{Cmd}$.

   We assume that $\mathit{cmd}^{-1}(c)$ is infinite for all all $c \in \mathit{Cmd}$, so there is an infinite supply of instruction identifiers for each type. For notational convenience, we define the following functions on sets of instructions:

$$\begin{array}{rcl rcl}
\mathit{loads} & : & \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I}) & \mathit{loads}(I) & = & \{i \in I \mid \mathit{cmd}(i) = \mathit{load}\} \\
\mathit{stores} & : & \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I}) & \mathit{stores}(I) & = & \{i \in I \mid \mathit{cmd}(i) = \mathit{store}\} \\
\mathit{fences} & : & \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I}) & \mathit{fences}(I) & = & \{i \in I \mid \mathit{cmd}(i) = \mathit{fence}\} \\
\mathit{accesses} & : & \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I}) & \mathit{accesses}(I) & = & \mathit{loads}(I) \cup \mathit{stores}(I)
\end{array}$$

3. We let *Val* denote the set of values that appear in the traces. We give a full definition of *Val* in Section 4.2. For now, we only state its properties as far as they are relevant for understanding this section:

   - *Val* can represent address and data values used by programs
   - *Val* contains a special value $\perp$ to represent an undefined value

2.1.2 Local Memory Traces

A local memory trace describes the logical sequence of memory instructions issued by the local program, and it includes the specific values that are loaded and stored, and the addresses of all accessed memory locations. It is defined as follows:

Definition 1 *A* local memory trace *is a tuple* $\boldsymbol{t} = (I, \prec, adr, val)$ *such that*

- $I \subset \mathcal{I}$ *is a set of instructions*
- $\prec$ *is a total order over* I *(called the* program order*)*
- adr *is a function of type* $accesses(I) \rightarrow Val$
- val *is a function of type* $accesses(I) \rightarrow Val$

Let Ltr *be the set of all local traces. Let* $\boldsymbol{t}^0$ *be the local trace for which* $I = \emptyset$*.*

For an instruction i, $adr(i)$ is the address of the accessed memory location, and $val(i)$ is the value loaded or stored.

2.1.3 Local Trace Semantics

What memory traces are possible depends on the local program. The same local program may exhibit many different memory traces, because the details of its execution (say, the path taken) usually depends on the behavior of other threads. We account for this nondeterministic behavior by defining the *trace semantics* of a program to be the set of all memory traces that the program may exhibit *in an arbitrary environment.* This set must cover all local program executions, for arbitrary values returned by each load: when running in an arbitrary environment, other threads may store any value to any location at any time, so a load may return any value.

In Chapter 4, we will formally define the set $Prog$ of local programs, the set Exs of execution states and the semantic function $E_L : Prog \rightarrow \mathcal{P}(Ltr \times Exs)$ such that $(\boldsymbol{t}, e) \in E_L(p)$ if and only if there is an execution of program p (in some arbitrary environment) that produces the local memory trace $\boldsymbol{t}$ and terminates with execution state e.

2.1.4 Global Memory Traces

A global memory trace is somewhat similar to a local trace, but with a few important differences. First of all, the trace contains instructions from different processors. As a result, the program order is no longer a total order, but a partial order. Moreover, a global trace specifies for each load where the loaded value originated. A loaded value may originate from some prior store in the trace (if that store targeted the same address), or it may be the initial value of the loaded memory location (which we represent using the special value $\bot$). Note that a global trace does *not* specify any temporal relationships between the instructions, only the logical relationship between loads and the stores they get their value from.

Definition 2 *A* global trace *is a tuple* $\boldsymbol{T} = (I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat})$ *such that*

- $I \subset \mathcal{I}$ *is a set of instructions*
- $\prec$ *is a* partial *order over* I
- proc *is a function of type* $I \to \mathbb{N}$ *(the processor issuing the instruction)*
- $\prec$ *is a total order on* $\mathit{proc}^{-1}(k)$ *for each* $k \in \mathbb{N}$
- adr *is a function of type* $\mathit{accesses}(I) \to \mathit{Val}$
- val *is a function of type* $\mathit{accesses}(I) \to \mathit{Val}$
- seed *is a partial function of type* $\mathit{loads}(I) \rightharpoonup \mathit{stores}(I)$
- *if* $\mathit{seed}(l) = s$*, then* $\mathit{adr}(l) = \mathit{adr}(s)$ *and* $\mathit{val}(l) = \mathit{val}(s)$
- *if* $l \in (\mathit{loads}(I) \setminus \operatorname{dom} \mathit{seed})$*, then* $\mathit{val}(l) = \bot$
- stat *is a partial function of type* $\mathbb{N} \to \mathit{Exs}$ *such that* $(\operatorname{rg} \mathit{proc} \subset \operatorname{dom} \mathit{stat})$

Let Gtr *be the set of all global traces. Let* $\boldsymbol{T}^0$ *be the global trace for which* $I = \emptyset$*.*

We can project a global trace onto its local components as follows:

Definition 3 *For a global trace* $\boldsymbol{T} \in \mathit{Gtr}$ *and an integer* $k \in \mathbb{N}$ *define the projection* $\pi_k(\boldsymbol{T})$ *of* $\boldsymbol{T}$ *onto processor* k *to be the tuple* $(I', \prec', \mathit{adr}', \mathit{val}')$ *where*

- $I' = \{i \in I \mid proc(i) = k\}$
- $\prec' = \prec|_{(I' \times I')}$
- $adr' = adr|_{I'}$
- $val' = val|_{I'}$

It follows directly from the definition that for any global trace $\boldsymbol{T}$, the projection $\pi_k(\boldsymbol{T})$ is a local trace.

2.1.5 Memory Model Semantics

The memory model defines restrictions on what global traces are possible. For example, the simplest memory model (sequential consistency) requires that the global trace is consistent with some *interleaving* of the local traces. However, not all memory models are that strict. We will discuss this topic in more detail in Chapter 3 where we define the set Axm of axiomatic memory model specifications and the semantic function $E_G : Axm \to \mathcal{P}(Gtr)$ such that $\boldsymbol{T} \in E_G(Y)$ if and only if the global trace $\boldsymbol{T}$ is allowed by the memory model Y.

2.1.6 Concurrent Executions

We can now bring together the notions of programs, memory models, and executions. First we define concurrent programs.

Definition 4 *A* concurrent program *is a tuple $P = (p_1, \ldots, p_n)$ such that $n \geq 0$ and for each i, $p_i \in Prog$ is a local program.*

We now conclude this section with the definition of concurrent executions.

Definition 5 *For a concurrent program $P = (p_1, \ldots, p_n)$ and a memory model Y, define the set $E(P, Y)$ of* concurrent executions of P on Y *to consist of all global traces $\boldsymbol{T} = (I, \prec, proc, adr, val, seed, stat)$ that satisfy the following conditions:*

- $\boldsymbol{T} \in E_G(Y)$
- *for all* $k \in \{1, \ldots, n\}$*, we have* $(\pi_k(\boldsymbol{T}), \mathit{stat}(k)) \in E_L(p_k)$
- *for all other* k*,* $\pi_k(\boldsymbol{T}) = \boldsymbol{t}^0$

Because memory models are nondeterministic, there are typically many executions for a given program P. Reasonable memory models will likely satisfy additional properties (for example, all programs P should have at least one execution).

2.2 Formulae

In this section, we establish the logical framework for formulae. Most of this material is fairly standard; we start with a quick informal overview which is likely to be sufficient for most readers.

We construct formulae and terms over a vocabulary of set symbols $\mathcal{S}$, typed function symbols $\mathcal{F}$ and typed variable symbols $\mathcal{V}$, using the syntax shown in Fig. 2.3. For some interpretation $[\![.]\!]$ of $\mathcal{S} \cup \mathcal{F}$ and a valuation ν of the variables in $\mathcal{V}$, we then define the evaluation function $[\![\phi]\!]^\nu$ on a formula or term ϕ in the usual way.

The following points of interest are marked by an asterisk (*) in Fig. 2.3:

- We provide an (if-then-else) construct $\phi \; ? \; \phi_1 : \phi_2$ which evaluates to either ϕ_1 or ϕ_2, depending on whether ϕ evaluates to *true* or *false*, respectively.

- We provide a limited quantifier syntax $(\forall \; p \; X : \phi)$ and $(\exists \; p \; X : \phi)$, where p is a predicate that restricts the range over which the variable X is being quantified. The reason is that we use these quantifiers when expressing memory model axioms (Section 3.3.1). Our encoding does not support arbitrary quantification, but requires that the quantifiers range over subsets of instructions in the trace (which are finite). Using this syntax, we can control the range of quantifiers by limiting the predicates that are available in the vocabulary.

- We provide second-order relation variables.

For completeness, we provide a more formal exposition of these concepts in the remainder of this section. We encourage the reader to skim the details.

2.2.1 Vocabularies

To construct formulae, we need some elementary symbols to represent sets, functions, relations, and variables. Such symbols are provided by the vocabulary, defined as follows:

Definition 6 *An* uninterpreted vocabulary *is a tuple $\sigma = (\mathcal{S}, \mathcal{F}, \mathcal{V})$ where*

- *$\mathcal{S}$ is a set of set symbols.*
- *We define types as follows: a type T is of the form*

  $$T ::= \ \ \mathit{bool} \ \ | \ \ S \ \ | \ \ S_1 \times \cdots \times S_k \rightarrow \mathit{bool} \ \ | \ \ S_1 \times \cdots \times S_k \rightarrow S$$

  where $k \geq 1$ and $S, S_1, \ldots, S_k \in \mathcal{S}$.
- *$\mathcal{F}$ is a set of typed function symbols. We let $\mathcal{F}^T$ be the functions of type T.*
- *$\mathcal{V}$ is a set of typed variable symbols. We let $\mathcal{V}^T$ be the variables of type T.*
- *the sets $\mathcal{S}, \mathcal{F}, \mathcal{V}$ are pairwise disjoint*

An uninterpreted vocabulary does not assume any particular meaning of the symbols. If we wish to establish such a meaning, we use an interpretation, defined as follows:

Definition 7 *Given an uninterpreted vocabulary σ, define an* interpretation *to be a function $[\![.]\!]$ on $(\mathcal{S} \cup \mathcal{F})$ that "interprets" the symbols as follows:*

- *each set symbol $S \in \mathcal{S}$ is mapped to a set $[\![S]\!]$*
- *each constant $c \in \mathcal{F}^S$ is mapped to an element $[\![c]\!] \in [\![S]\!]$*
- *each constant $c \in \mathcal{F}^{bool}$ is mapped to a truth value $[\![c]\!] \in \{\mathit{false}, \mathit{true}\}$*

(set)	S	$\in$	$\mathcal{V}$
(function of type T)	f^T	$\in$	$\mathcal{F}^T$
(variable of type T)	X^T	$\in$	$\mathcal{V}^T$
(term of type S)	ϕ^S	$\in$	$\mathcal{T}^S$
(term constant)		$::=$	f^S
(term variable)			$\mid X^S$
(function)			$\mid f^{S_1\times\cdots\times S_k\rightarrow S}(\phi_1^{S_1},\ldots,\phi_k^{S_k})$
*(if-then-else)			$\mid \phi \; ? \; \phi_1^S : \phi_2^S$
(boolean formula)	ϕ	$\in$	$\mathcal{T}^{bool}$
(false)		$::=$	*false*
(true)			$\mid$ *true*
(negation)			$\mid \neg\phi$
(conjunction)			$\mid \phi_1 \wedge \phi_2$
(disjunction)			$\mid \phi_1 \vee \phi_2$
(implication)			$\mid \phi_1 \Rightarrow \phi_2$
(equivalence)			$\mid \phi_1 \Leftrightarrow \phi_2$
*(forall)			$\mid \forall\, f^{S\rightarrow bool}\, X^S : \phi$
*(exists)			$\mid \exists\, f^{S\rightarrow bool}\, X^S : \phi$
(equality)			$\mid \phi_1^S = \phi_2^S$
(boolean constant)			$\mid f^{bool}$
(boolean variable)			$\mid X^{bool}$
(relation)			$\mid f^{S_1\times\cdots\times S_k\rightarrow bool}(\phi_1^{S_1},\ldots,\phi_k^{S_k})$
*(relation variable)			$\mid X^{S_1\times\cdots\times S_k\rightarrow bool}(\phi_1^{S_1},\ldots,\phi_k^{S_k})$

Figure 2.3: The syntax for formulae over a vocabulary $(\mathcal{S},\mathcal{F},\mathcal{V})$.

- *each function* $f \in \mathcal{F}^{S_1 \times \cdots \times S_k \to S}$ *is mapped to a function* $[\![f]\!] : [\![S_1]\!] \times \cdots \times [\![S_n]\!] \to [\![S]\!]$
- *each predicate* $p \in \mathcal{F}^{S_1 \times \cdots \times S_k \to bool}$ *is mapped to a function* $[\![p]\!] : [\![S_1]\!] \times \cdots \times [\![S_n]\!] \to \{\mathit{false}, \mathit{true}\}$

The combination of a vocabulary and an interpretation is called an interpreted vocabulary.

Definition 8 *An* interpreted vocabulary *is a tuple* $\sigma = (\mathcal{S}, \mathcal{F}, \mathcal{V}, [\![.]\!])$ *such that* $(\mathcal{S}, \mathcal{F}, \mathcal{V})$ *is an uninterpreted vocabulary and* $[\![.]\!]$ *is an interpretation for the latter.*

2.2.2 Formula Syntax

We now introduce the formula syntax.

Definition 9 *Given a vocabulary* $(\mathcal{S}, \mathcal{F}, \mathcal{V})$*, define*

- *the set of boolean formulae* $\mathcal{T}^{bool}$
- *for each set symbol* S*, the set* $\mathcal{T}^{S}$ *of terms of type* S

recursively as specified in Fig. 2.3.

We define free variables as usual.

Definition 10 *Given a vocabulary* σ*, define the set of free variables* $FV(\phi)$ *of a formula or term* ϕ *inductively as follows:*

- $FV(\mathrm{true}) = FV(\mathit{false}) = \emptyset$
- $FV(\neg\phi) = FV(\phi)$
- $FV(\phi_1 \wedge \phi_2) = FV(\phi_1) \cup FV(\phi_2)$
- $FV(\phi_1 \vee \phi_2) = FV(\phi_1) \cup FV(\phi_2)$
- $FV(\phi_1 \Rightarrow \phi_2) = FV(\phi_1) \cup FV(\phi_2)$

- $FV(\phi_1 \Leftrightarrow \phi_2) = FV(\phi_1) \cup FV(\phi_2)$
- $FV(\forall\, p\, X : \phi) = FV(\phi) \setminus \{X\}$
- $FV(\exists\, p\, X : \phi) = FV(\phi) \setminus \{X\}$
- $FV(\phi_1 = \phi_2) = FV(\phi_1) \cup FV(\phi_2)$
- $FV(\phi \mathbin{?} \phi_1 : \phi_2) = FV(\phi) \cup FV(\phi_1) \cup FV(\phi_2)$
- $FV(X) = \{X\}$
- $FV(f) = \emptyset$
- $FV(X(\phi_1, \ldots, \phi_k)) = \{X\} \cup FV(\phi_1) \cup \cdots \cup FV(\phi_k)$
- $FV(f(\phi_1, \ldots, \phi_k)) = FV(\phi_1) \cup \cdots \cup FV(\phi_k)$

A formula ϕ is called closed *if $FV(\phi) = \emptyset$.*

2.2.3 Evaluation of Formulae

We can now conclude this chapter with a definition of valuations, and what it means to evaluate a formula for a given valuation.

Definition 11 *Given an interpreted vocabulary σ, define a* valuation *ν to be a partial function on $\mathcal{V}$ which assigns values to variable symbols as follows:*

- *each term variable $X \in \mathcal{V}^S$ is mapped to an element $\nu(X) \in [\![S]\!]$*
- *each boolean variable $X \in \mathcal{V}^{bool}$ is mapped to truth value $\nu(X) \in \{\mathit{false}, \mathit{true}\}$*
- *each function variable $X \in \mathcal{V}^{S_1 \times \cdots \times S_k \to S}$ is mapped to a function $\nu(X) : [\![S_1]\!] \times \cdots \times [\![S_n]\!] \to [\![S]\!]$*
- *each predicate variable $X \in \mathcal{V}^{S_1 \times \cdots \times S_k \to bool}$ is mapped to a function $\nu(X) : [\![S_1]\!] \times \cdots \times [\![S_n]\!] \to \{\mathit{false}, \mathit{true}\}$*

Definition 12 *Given an interpreted vocabulary σ and a valuation ν, we define the* evaluation function *$[\![\phi]\!]^\nu$ of a formula or term ϕ that satisfies $FV(\phi) \subset \operatorname{dom} \nu$ inductively as follows:*

- $[\![\mathit{false}]\!]^{\nu} = \mathit{false}$
- $[\![\mathit{true}]\!]^{\nu} = \mathit{true}$
- $[\![\neg\phi]\!]^{\nu} = \neg[\![\phi]\!]^{\nu}$
- $[\![\phi_1 \wedge \phi_2]\!]^{\nu} = [\![\phi_1]\!]^{\nu} \wedge [\![\phi_2]\!]^{\nu}$
- $[\![\phi_1 \vee \phi_2]\!]^{\nu} = [\![\phi_1]\!]^{\nu} \vee [\![\phi_2]\!]^{\nu}$
- $[\![\phi_1 \Rightarrow \phi_2]\!]^{\nu} = \neg[\![\phi_1]\!]^{\nu} \vee [\![\phi_2]\!]^{\nu}$
- $[\![\phi_1 \Leftrightarrow \phi_2]\!]^{\nu} = \begin{cases} \mathit{true} & \mathit{if}\ [\![\phi_1]\!]^{\nu} = [\![\phi_2]\!]^{\nu} \\ \mathit{false} & \mathit{otherwise} \end{cases}$
- $[\![\forall\ p\ X^S : \phi]\!]^{\nu} = \begin{cases} \mathit{true} & \mathit{if}\ ([\![p(X^S) \Rightarrow \phi]\!]^{\nu[X^S \mapsto x]} = \mathit{true})\ \mathit{for\ all}\ x \in [\![S]\!] \\ \mathit{false} & \mathit{otherwise} \end{cases}$
- $[\![\exists\ p\ X^S : \phi]\!]^{\nu} = \begin{cases} \mathit{true} & \mathit{if}\ ([\![p(X^S) \wedge \phi]\!]^{\nu[X^S \mapsto x]} = \mathit{true})\ \mathit{for\ some}\ x \in [\![S]\!] \\ \mathit{false} & \mathit{otherwise} \end{cases}$
- $[\![\phi_1 = \phi_2]\!]^{\nu} = \begin{cases} \mathit{true} & \mathit{if}\ [\![\phi_1]\!]^{\nu} = [\![\phi_2]\!]^{\nu} \\ \mathit{false} & \mathit{otherwise} \end{cases}$
- $[\![\phi\ ?\ \phi_1 : \phi_2]\!]^{\nu} = \begin{cases} [\![\phi_1]\!]^{\nu} & \mathit{if}\ ([\![\phi]\!]^{\nu} = \mathit{true}) \\ [\![\phi_2]\!]^{\nu} & \mathit{otherwise} \end{cases}$
- $[\![X]\!]^{\nu} = \nu(X)$
- $[\![f]\!]^{\nu} = [\![f]\!]$
- $[\![X(\phi_1, \ldots, \phi_k)]\!]^{\nu} = \nu(X)([\![\phi_1]\!]^{\nu}, \ldots, [\![\phi_k]\!]^{\nu})$
- $[\![f(\phi_1, \ldots, \phi_k)]\!]^{\nu} = [\![f]\!]([\![\phi_1]\!]^{\nu}, \ldots, [\![\phi_k]\!]^{\nu})$

Chapter 3

Memory Models

In this chapter, we introduce the reader to memory models and describe our specification format for axiomatic memory models. It is structured as follows:

- Section 3.1 provides a brief introduction to memory models. We give some background information about the underlying motivations for relaxed memory models, and we provide an informal description of the most common relaxations.

- Section 3.2 describes the two memory models that were most important for our work: sequential consistency (which is the strictest model and does not relax any of the ordering guarantees), and *Relaxed* (which is a weak model that we use as a simple, conservative approximation for several relaxed models).

- Section 3.3 describes the basic format of axiomatic specifications that we use to define memory models. For illustration purposes, we show how to express sequential consistency in this format.

- Section 3.4 describes how we extend the basic format to include atomic blocks, fences, and control- or data-dependencies.

- Section 3.5 introduces the specification format used by our *CheckFence* tool, and gives the full axiomatic specifications for sequential consistency and *Relaxed*.

- Section 3.6 shows and discusses our formalization of the SPARC RMO model.

3.1 Introduction

Memory models describe how to generalize the semantics of memory accesses in a von Neumann architecture to a setting where several processors access a shared memory concurrently. Memory models specify the behavior of loads and stores to shared memory locations, and their interplay with special fence and synchronization operations.

The simplest memory model is *Sequential Consistency* [44], which specifies that a multiprocessor should execute like a single processor that interleaves the loads and stores of the different threads nondeterministically, yet preserves their relative order within each thread.

Although sequential consistency is simple to understand and popular with programmers, most commercially available multiprocessors do not support it. The reason is that it comes with a price: the memory system hardware is forced to maintain global ordering properties even though the vast majority of all memory accesses do not require such strict guarantees (usually, the ordering guarantees are important only with respect to accesses that serve a synchronization purpose).

For that reason, hardware architects tend to prefer *Relaxed Memory Models* [2] that provide more freedom for the hardware to optimize performance by reordering accesses opportunistically. Many different kinds of memory models have been formulated for various purposes [65]. We focus our attention on hardware-level memory models for multiprocessor architectures where the following three relaxations are common:

Initially: $A = Flag = 0$

thread 1	thread 2
store A, 1	load Flag, 1
store Flag, 1	load A, 0

Figure 3.1: An execution trace that is not sequentially consistent, but allowed on relaxed memory models that allow out-of-order execution (and may swap either the stores in thread 1 or the loads in thread 2).

Out-of-order execution. Modern processors may execute instructions out of order where it can speed up execution. On uniprocessors, these “tricks” are completely invisible to the programmer, but on multiprocessors, they become visible in some cases, because the order in which loads and stores commit to memory can be observed by a remote processor. Fig. 3.1 shows an example of a relaxed execution that could be caused either by out-of-order stores, or out-of-order loads.

Store Buffers. On most architectures, stores do not commit atomically; when executed, they update a local buffer first (a store queue or a memory order buffer) before they become visible to other processors. Subsequent loads by the same processor ‘see’ the store (an effect called *store-load forwarding*) even if it hasn’t committed globally yet.

No Global Store Order. Some architectures allow stores to different addresses to happen in an order that is not globally consistent. This means that different processors may observe such stores in a different order.

Of course, if a processor could reorder instructions at will, it would be impossible to write meaningful programs. The relaxations are therefore formulated with the following goals in mind:

1. Single processor semantics must be preserved; if each processor executes an independent program, the results are the same as if each program were executed separately on a uniprocessor.

2. If all accesses to shared memory locations are fully synchronized (i.e., guarded by critical sections or other suitable synchronization constructs), the memory model should be indistinguishable from sequential consistency.

3. Special memory ordering instructions (called memory fences, memory barriers, or memory synchronization instructions) are provided, which can be used to selectively avoid out-of-order execution, or to flush store buffers. Such instructions allow the implementation of mutual exclusion locks that have the desired property as stated above. To avoid confusion with cross-thread synchronization barriers or with synchronization operations such as test-and-set or compare-and-swap, we consistently use the term *memory ordering fences* or just *fences* to refer to such instructions.

Most architecture manuals provide some code examples to illustrate the use of fences, and include code for a simple spinlock that can be used to build critical sections. The precise name and semantics of the fences vary across architectures (for example, some of the fence instructions are called: MB, WMB on Alpha; sync, lwsync, eieio on PowerPC; load-load/load-store/store-load/store-store fences on SPARC; mf on Itanium, etc.)

3.2 Memory Model Examples

In this section, we present formal presentations of the two main memory models we used. The first one is the classic *sequential consistency* [44], which requires that the loads and stores issued by the individual threads are interleaved in some global, Initially: $X = Y = 0$

thread 1	thread 2
X = 1	r_1 = Y
Y = 1	r_2 = X

Eventually: $r_1 = 1$, $r_2 = 0$

Figure 3.2: An execution trace that is not sequentially consistent, but allowed by *Relaxed*, because the latter may swap the order of the stores in thread 1, or the order of the loads in thread 2.

total order. Sequential consistency is the easiest to understand; however, it is not guaranteed by most multiprocessors.

The second memory model we describe in this section is *Relaxed* [8]. The purpose of this model is to provide an abstract, conservative approximation of several commonly used memory models. Of the relaxations listed above, it models the first two (out-of-order execution and store buffers), but not the third (it does assume that stores are globally ordered). Specifically, the relaxations permit (1) reorderings of loads and stores to different addresses, (2) buffering of stores in local queues, (3) forwarding of buffered stores to local loads, (4) reordering of loads to the same address, and (5) reordering of control- or data-dependent instructions.

To illustrate some of the relaxations, we now discuss a few examples. Such examples are often called *litmus tests* and are included as part of the official memory model specification.

- The execution trace in Fig. 3.2 illustrates how out-of-order execution may become visible to the programmer. The values loaded by thread 2 are not consistent with any sequentially consistent execution, but may result from a reordering of the stores by thread 1 or a reordering of the loads by thread 2.

- The trace in Fig. 3.3 illustrates how fences can prevent out-of-order execution. It shows the same sequence of loads and stores as the previous example, but with two fences inserted: a store-store fence that enforces the order of preceding

Initially: $X = Y = 0$

thread 1	thread 2
X = 1	r_1 = Y
store-store fence	load-load fence
Y = 1	r_2 = X

Eventually: $r_1 = 1$, $r_2 = 0$

Figure 3.3: An execution trace that is not allowed by *Relaxed* because neither the stores in thread 1 nor the loads in thread 2 may be reordered due to the respective fences.

Initially: $X = Y = 0$

thread 1	thread 2
	Y = 2
X = 1	r_1 = Y
store-store fence	load-load fence
Y = 1	r_2 = X

Eventually: $r_1 = 2$, $r_2 = 0$, $Y = 2$

Figure 3.4: An execution trace that is not sequentially consistent, but allowed by *Relaxed*, because the latter allows store-load forwarding.

and succeeding stores, and a load-load fence that enforces the order of preceding and succeeding loads. These fences prevent this trace on *Relaxed*.

- The trace in Fig. 3.4 illustrates store-load forwarding. The value 2 stored by $(Y = 2)$ is loaded by $(r_1 = Y)$ while the store is still sitting in the store buffer. Thus, it is possible for the store $(Y = 2)$ to take global effect after the load $(r_1 = Y)$. Without store-load forwarding, this trace is impossible because

  - $(Y = 2)$ must happen before $(r_1 = Y)$.
  - $(r_1 = Y)$ must happen before $(r_2 = X)$ because of the load-load fence.
  - $(r_2 = X)$ must happen before $(X = 1)$ because it loads value 0, not 1.
  - $(X = 1)$ must happen before $(Y = 1)$ because of the store-store fence.

Initially: X = Y = 0

thread 1	thread 2	thread 3	thread 4
X = 1	Y = 1	r_1 = X	r_3 = Y
		load-load fence	load-load fence
		r_2 = Y	r_4 = X

Eventually: $r_1 = r_3 = 1$, $r_2 = r_4 = 0$

Figure 3.5: An execution trace that is theoretically possible on PPC, IA-32, and IA-64, but not allowed by *Relaxed* because the latter requires that all stores be globally ordered.

This implies that $(Y = 2)$ must happen before $(Y = 1)$ which contradicts the final value 2 of Y.

- The trace in Fig. 3.5 illustrates an execution without a globally consistent ordering of stores. The threads 3 and 4 observe a different ordering of the stores $(X = 1)$ and $(Y = 1)$. We do not include this relaxation in *Relaxed*, which is thus not quite as weak as IBM PowerPC [24], IA-32 [38], or IA-64 [39]. It remains on our list of future work to explore the impact of non-global store orders on the correctness of implementations.

3.2.1 Formal Specification

Clearly, memory models can not be completely and precisely specified by giving examples only. Because precise specifications are an absolute prerequisite for formal verification, we now proceed to a formal definition of sequential consistency and *Relaxed*. We use the formal framework for multiprocessor executions introduced in Sections 2.1.4 and 2.1.5. Briefly summarized, we define executions as tuples $(I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat})$ where I is the set of instructions in the global trace, $i \prec j$ iff i precedes j in program order, $\mathit{proc}(i)$ is the processor that issued instruction i, $\mathit{adr}(i)$ and $\mathit{val}(i)$ are the address and data values of instruction i, seed is a partial function that maps loads to the store that sources their value, and stat is irrelevant for this chapter. A memory model is then simply a predicate on execution traces (that is, specifies which global traces are possible and which ones are not).

The axiomatic format of the definitions below foreshadows our generic specification format which lets the user specify the memory model directly, as we describe Section 3.3. To keep the definitions below as simple as possible, we omit atomic blocks and use a single type of fence only; we describe the full formalization later, in Sections 3.4 and 3.5.

3.2.2 Sequential Consistency (*SeqCons*)

A global trace $\boldsymbol{T} = (I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat})$ is sequentially consistent if and only if there exists a total order $<_M$ over $\mathit{accesses}(I)$, called the *memory order*, such that the following conditions hold:

For each load $l \in \mathit{loads}(I)$, let $S(l)$ be the set of stores that are "visible" to l:

$$S(l) = \{s \in \mathit{stores}(I) \mid \mathit{adr}(s) = \mathit{adr}(l) \wedge (s <_M l)\}$$

Then the following axioms must be satisfied:

(A1) if $x \prec y$, then $x <_M y$

(A2) if $\mathit{seed}(l) = s$, then $s \in S(l)$

(A3) if $s \in S(l)$ but not $\mathit{seed}(l) = s$, then there must exist an $s' \in \mathit{stores}(I)$ such that $\mathit{seed}(l) = s'$ and $s <_M s'$.

3.2.3 Relaxed

A global trace $\boldsymbol{T} = (I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat})$ is permitted by the memory model *Relaxed* if and only if there exists a total order $<_M$ over $\mathit{accesses}(I)$, called the *memory order*, such that the following conditions hold:

For each load $l \in \mathit{loads}(I)$, let $S(l)$ be the set of stores that are "visible" to l:

$$S(l) = \{s \in \mathit{stores}(I) \mid \mathit{adr}(s) = \mathit{adr}(l) \wedge ((s <_M l) \vee (s \prec l))\}$$

Then the following axioms must be satisfied:

(A1) if $x \prec y$, $adr(x) = adr(y)$, and $cmd(y) = store$, then $x <_M y$

(A2) if $seed(l) = s$, then $s \in S(l)$

(A3) if $s \in S(l)$ but not $seed(l) = s$, then there must exist a $s' \in stores(I)$ such that $seed(l) = s'$ and $s <_M s'$.

(A4) if $x, y \in accesses(I)$ and $f \in fences(I)$ and $x \prec f \prec y$, then $x <_M y$.

Discussion

The model *Relaxed* shows a structure that is very similar to sequential consistency (Section 3.2.2), and is best understood by comparing the differences:

- Axiom (A1) has been weakened to allow the memory order to be different from the program order. The program order is only enforced if both accesses go to the same address (we call such accesses *aliased* for short), and only if the second access is a store. These are the minimal guarantees necessary to maintain single-processor semantics: we may not reorder aliased stores, nor can we reorder a load past a subsequent aliased store. However, we *do* allow a store to be reordered past a subsequent aliased load, because the value will get forwarded from the store to the load (see next bullet). We also allow two aliased loads to get reordered.

- We enlarge the set $S(l)$ of stores that are visible to a load, in order to allow forwarding of values from stores sitting in a store buffer to subsequent aliased loads by the same processor. $S(l)$ now also contains stores that precede a load in program order ($s \prec l$) but are performed globally only after the load is performed ($l <_M s$).

- We added axiom (A4) has been added to describe the semantics of fences (there is no need for fences on sequentially consistent machines).

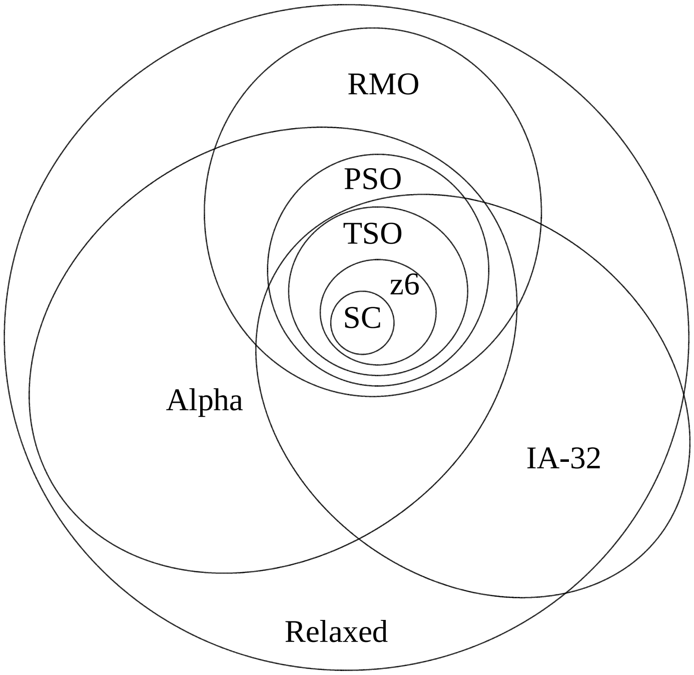


Figure 3.6: A comparison of various memory models.

3.2.4 Comparing Memory Models

Memory models can be directly compared in terms of the execution traces they allow. We call a model Y *stronger* than another model Y' if $Y \subset Y'$. For example, sequential consistency is stronger than *Relaxed* (because any sequentially consistent global trace satisfies the axioms for *Relaxed* as well).

We show a graphic representation that compares the sets of executions of *Relaxed* and other memory models in Fig. 3.6. The purpose of *Relaxed* is to provide a common, conservative approximation of several memory models (Sun SPARC v9 TSO/PSO/RMO [72], Alpha [13], and IBM 370/390/zArchitecture [41]). All of these models are stronger than *Relaxed*, which implies that once code runs correctly on *Relaxed*, it will run correctly on each of the former.

However, *Relaxed* is not strictly weaker than the official PowerPC [24], IA-64 [39] and IA-32 [38] models because it globally orders all stores (the execution in Fig. 3.5 illustrates this point). Even so, *Relaxed* still captures the most important relaxations of those models (store buffers and out-of-order execution) and is useful in determining where to place fences.

3.3 Specification Format

After we studied a number of relaxed memory models in various architecture manuals, it became clear that there are many subtle differences between them. Although our model *Relaxed* is a useful pragmatic approximation, we sometimes wish to specify the memory model more precisely or play with the details of the definitions. Of particular interest are support for ordering constraints created by data or control dependencies, and support for memory models that do not rely on a globally consistent store order.

We therefore developed a generic axiomatic format for specifying memory models that generalizes the format of the axiomatic definitions for sequential consistency and *Relaxed* we used in Section 3.2.1 and provides the desired flexibility.

3.3.1 Axiomatic Memory Model Specifications

We now describe our format for specifying memory models. Our memory model specifications use boolean formulae to constrain the global traces. Traditionally, this style of specification is called "axiomatic", and we call the formulae *axioms.*

First, in order to write axioms that express constraints over global traces, we introduce symbols that express properties of the global trace as defined in Section 2.1.4. We need predicates that capture the type of a given instruction, and relations to express the program order, whether two accesses are aliased (go to the same memory location), and the partial function *seed* which assigns to each load the store that sourced its value.

Definition 13 *Define a* memory model vocabulary *to be a vocabulary $\sigma = (\mathcal{S}, \mathcal{F}, \mathcal{V})$ (as defined in Section 2.2.1) that contains the following set and function symbols:*

$$\begin{aligned}\{\mathit{instr}\} &\subset \mathcal{S}\\ \{\mathit{load}, \mathit{store}, \mathit{fence}, \mathit{access}\} &\subset \mathcal{F}^{\mathit{instr}\to\mathit{bool}}\\ \{\mathit{progorder}, \mathit{aliased}, \mathit{seed}\} &\subset \mathcal{F}^{\mathit{instr}\times\mathit{instr}\to\mathit{bool}}\end{aligned}$$

Throughout the rest of this document, we assume that we are working with some fixed memory model vocabulary $\sigma = (\mathcal{S}, \mathcal{F}, \mathcal{V})$. We will thus not bother to mention it explicitly in every definition.

Axiomatic memory models may contain one or more existentially quantified second-order variables. For example, a global trace is defined to be sequentially consistent if and only if there exists a total order over the memory accesses, subject to some constraints (see Section 3.2.2). We formalize such ordering relations as relation variables over instructions. This leads to the following definition:

Definition 14 *An* axiomatic memory model specification Y *is a tuple* $Y = (\mathcal{R}, \phi_1, \ldots, \phi_k)$ *such that* ($k \geq 0$) *and*

- $\mathcal{R} \subset \mathcal{V}^{\mathit{instr}\times\cdots\times\mathit{instr}\to\mathit{bool}}$ *is a set of relation variables*
- $\phi_1, \ldots, \phi_k \in \mathcal{T}^{\mathit{bool}}$ *are formulae with syntax as shown in Fig. 3.7. We call them* axioms.
- $FV(\phi_1) \cup \cdots \cup FV(\phi_k) \subset \mathcal{R}$

We let Axm be the set of all axiomatic memory model specifications.

Note that the syntax for axioms as shown in Fig. 3.7 is a specialization of the general syntax for formulae and terms we defined in Fig. 2.3 (Section 2.2.2) with the following restrictions:

- We may use only the set and function symbols listed in Def. 13.

(predicate)	p	$\in$	$\mathcal{F}^{\text{instr}\rightarrow \mathit{bool}}$
(load)		$::=$	*load*
(store)			$\mid$ *store*
(access)			$\mid$ *access*
(fence)			$\mid$ *fence*
(binary relation)	b	$\in$	$\mathcal{F}^{\text{instr}\times \mathit{instr}\rightarrow \mathit{bool}}$
(program order)		$::=$	*progorder*
(aliased)			$\mid$ *aliased*
(seed)			$\mid$ *seed*
(relation variable)	R	$\in$	$\mathcal{V}^{\text{instr}\times\cdots\times \mathit{instr}\rightarrow \mathit{bool}}$
(instruction variable)	X	$\in$	$\mathcal{V}^{\mathit{instr}}$
(formula)	ϕ	$\in$	$\mathcal{T}^{\mathit{bool}}$
(false)		$::=$	*false*
(true)			$\mid$ *true*
(negation)			$\mid \neg\phi$
(conjunction)			$\mid \phi_1 \wedge \phi_2$
(disjunction)			$\mid \phi_1 \vee \phi_2$
(implication)			$\mid \phi_1 \Rightarrow \phi_2$
(equivalence)			$\mid \phi_1 \Leftrightarrow \phi_2$
(forall)			$\mid \forall\, p\, X : \phi$
(exists)			$\mid \exists\, p\, X : \phi$
(equality)			$\mid X_1 = X_2$
(predicate)			$\mid\ p(X)$
(binary relation)			$\mid\ b(X_1, X_2)$
(relation variable)			$\mid\ R(X_1, \ldots, X_k) \quad (k \geq 1)$

Figure 3.7: The abstract syntax for memory model axioms.

- We may use only variables of type *instr* and $\mathit{instr} \times \cdots \times \mathit{instr} \rightarrow \mathit{bool}$.
- All terms of type *instr* must be variables.

We use the same limited quantifier syntax $(\forall\ p\ X : \phi)$ and $(\exists\ p\ X : \phi)$, where p is a predicate that restricts the range over which the variable X is being quantified. This implies that the quantifiers may range only over subsets of instructions in the trace (which are finite).

Interpretation

We now define how to interpret an axiomatic specification. To do so, we need to define the function

$$E_G : \mathit{Axm} \rightarrow \mathcal{P}(\mathit{Gtr})$$

such that $E_G(Y)$ is the set of global traces that are allowed by the specification Y. First, we define how to interpret the symbols of the memory model signature for a given global trace $\boldsymbol{T}$. The symbol *instr* represents the the set of instructions in the trace, the predicates *load*, *store*, *fence*, *access* distinguish the type of an instruction, the relation *progorder* represents the program order, the relation *seed* represents the partial function *seed*, and the relation *aliased* represents whether two instructions are aliased memory accesses.

Definition 15 *For a global trace $\boldsymbol{T} = (I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat})$ and a memory model vocabulary σ, define the interpretation $[\![.]\!]_{\boldsymbol{T}}$ as follows:*

- $[\![\mathit{instr}]\!]_{\boldsymbol{T}} = I$
- $[\![\mathit{load}]\!]_{\boldsymbol{T}}$ *is the function that maps* $i \mapsto (i \in \mathit{loads}(I))$
- $[\![\mathit{store}]\!]_{\boldsymbol{T}}$ *is the function that maps* $i \mapsto (i \in \mathit{stores}(I))$
- $[\![\mathit{fence}]\!]_{\boldsymbol{T}}$ *is the function that maps* $i \mapsto (i \in \mathit{fences}(I))$
- $[\![\mathit{access}]\!]_{\boldsymbol{T}}$ *is the function that maps* $i \mapsto (i \in \mathit{accesses}(I))$
- $[\![\mathit{progorder}]\!]_{\boldsymbol{T}}$ *is the function that maps* $(i, j) \mapsto (i \prec j)$

- $[\![seed]\!]_{\boldsymbol{T}}$ *is the function that maps* $(i, j) \mapsto (seed(i) = j)$
- $[\![aliased]\!]_{\boldsymbol{T}}$ *is the function that maps*

$$(i, j) \mapsto \begin{cases} (adr(i) = adr(j)) & \text{if } i, j \in accesses(I) \\ false & otherwise \end{cases}$$

We now define the function E_G as follows:

Definition 16 *Given an axiomatic memory model specification* $Y = (\mathcal{R}, \phi_1, \ldots, \phi_k)$, *let* $E_G(Y)$ *be the set consisting of all global traces* $\boldsymbol{T} = (I, \prec, proc, adr, val, seed, stat)$ *for which there exists a valuation* ν *such that the following conditions hold:*

$$\operatorname{dom} \nu \;=\; \mathcal{R} \qquad and \qquad [\![\phi_1]\!]^{\nu}_{\boldsymbol{T}} = \cdots = [\![\phi_k]\!]^{\nu}_{\boldsymbol{T}} = true$$

3.3.2 Illustrative Example: Sequential Consistency

To illustrate how axiomatic specifications work, we now show how to represent sequential consistency (defined in Section 3.2.2) by an axiomatic memory model specification $(\mathcal{R}, \phi_1, \ldots, \phi_6)$. First, define the following variable symbols:

$$\begin{aligned} memorder &\in \mathcal{V}^{instr \times instr \rightarrow bool} \\ X, Y, Z &\in \mathcal{V}^{instr} \end{aligned}$$

Then we define $(\mathcal{R}, \phi_1, \ldots, \phi_6)$ as follows:

$$
\begin{aligned}
\mathcal{R} &= \{\mathit{memorder}\} \\
\phi_1 &= [\forall\ \mathit{access}\ X : \\
&\qquad \forall\ \mathit{access}\ Y : \\
&\qquad\qquad \forall\ \mathit{access}\ Z : \\
&\qquad\qquad\qquad (\mathit{memorder}(X,Y) \wedge \mathit{memorder}(Y,Z)) \Rightarrow \mathit{memorder}(X,Z)] \\
\phi_2 &= [\forall\ \mathit{access}\ X : \\
&\qquad \neg \mathit{memorder}(X,X)] \\
\phi_3 &= [\forall\ \mathit{access}\ X : \\
&\qquad \forall\ \mathit{access}\ Y : \\
&\qquad\qquad \mathit{memorder}(X,Y) \vee \mathit{memorder}(Y,X) \vee (X = Y)] \\
\phi_4 &= [\forall\ \mathit{access}\ X : \\
&\qquad \forall\ \mathit{access}\ Y : \\
&\qquad\qquad \mathit{progorder}(X,Y) \Rightarrow \mathit{memorder}(X,Y)] \\
\phi_5 &= [\forall\ \mathit{load}\ X : \\
&\qquad \forall\ \mathit{store}\ Y : \\
&\qquad\qquad \mathit{seed}(X,Y) \Rightarrow \mathit{memorder}(Y,X) \wedge \mathit{aliased}(Y,X)] \\
\phi_6 &= [\forall\ \mathit{load}\ X : \\
&\qquad \forall\ \mathit{store}\ Y : \\
&\qquad\qquad \mathit{aliased}(X,Y) \wedge \mathit{memorder}(Y,X) \wedge \neg \mathit{seed}(X,Y) \Rightarrow \\
&\qquad\qquad\qquad\qquad\qquad \exists\ \mathit{store}\ Z : (\mathit{seed}(X,Z) \wedge \mathit{memorder}(Y,Z))]
\end{aligned}
$$

We can easily relate these formulae to the earlier definition of sequential consistency in Section 3.2.2): the formulae ϕ_1, ϕ_2 and ϕ_3 express that the memory order is a total order over all accesses, while the formulae ϕ_4, ϕ_5 and ϕ_6 correspond to the axioms (A1), (A2) and (A3) respectively.

A valuation that satisfies all conditions thus exists if and only if a valuation for *memorder* can be found that satisfies all formulae, which is the case if and only if a total memory order $<_M$ (as defined in Section 3.2.2) exists.

3.4 Extensions

So far, we have used a very simple notion of trace that consists only of loads, stores and fences. While such simple traces are appropriate to explain the basic concepts, we need more structure to handle realistic memory models that contain synchronization operations, mention control- and data-dependencies, and contain various types of fences.

In this section, we describe how to extend the definitions of local traces, global traces and memory models to accommodate these concepts.

3.4.1 Atomic Blocks

To model synchronization operations such as compare-and-swap (CAS), we support *atomic blocks*. Fig. 3.8 shows how we conceptually model CAS using an atomic block. To express atomic blocks on the abstraction level of memory traces, we express atomicity as an equivalence relation $\sim_A$ on instructions in the trace. The idea is that two instructions are in the same equivalence class whenever they are produced by code that is enclosed in some atomic block. Instructions that are not contained in an atomic block go into an equivalence class of their own. Because atomic blocks never span multiple processors, each equivalence class is totally ordered by the program order.

Different memory models may implement slightly different semantic variations of atomicity. We can specify the meaning of atomic blocks by adding axioms to the memory model. In the case of sequential consistency or *Relaxed*, the case is relatively simple because it is straightforward to express atomicity by constraining the memory order, as follows:

Sequential Consistency. (Section 3.2.2) Add the axiom

(A4) if $x, y, z \in \mathit{accesses}(I)$ and $x \sim_A y$ and $x \not\sim_A z$ and $x \prec y$ and $x <_M z$, then $y <_M z$

```
bool cas(unsigned *loc,
         unsigned old, unsigned new) {
  atomic {
    if (*loc == old) {
      *loc = new;
      return true;
    } else {
      return false;
    }
  }
}
```

Figure 3.8: Pseudocode for the compare-and-swap (CAS) operation.

Relaxed. (Section 3.2.3) Add the axioms

(A5) if $x, y, z \in \mathit{accesses}(I)$ and $x \sim_A y$ and $x \not\sim_A z$ and $x \prec y$ and $x <_M z$, then $y <_M z$

(A6) if $x, y \in \mathit{accesses}(I)$ and $x \sim_A y$ and $x \prec y$, then $x <_M y$

Note that axiom (A6) is not just about atomicity, but also about ordering. We found that in all cases where we use atomic blocks, we implicitly also require that the ordering of the accesses within the block be maintained, so we chose to make this a part of the definition of atomic blocks.

In order to add the axioms above, we need to represent the atomic block structure within the definitions of local and global traces, as follows:

- Modify the definitions of local trace and global trace (Sections 2.1.2 and 2.1.4) by extending the trace tuples as follows:

  $$\boldsymbol{t} = (I, \prec, \mathit{adr}, \mathit{val}, \sim_A) \tag{3.1}$$

  $$\boldsymbol{T} = (I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat}, \sim_A) \tag{3.2}$$

  and add the requirement that $\sim_A$ be an equivalence relation over I such that $\prec$ is a total order on each partition of $\sim_A$.

- Extend Definition 13 (Section 3.3.1) to include the following second-order variable:

$$\mathit{atomic} \in \mathcal{F}^{\mathit{instr} \times \mathit{instr} \rightarrow \mathit{bool}}$$

- Extend Definition 15 (Section 3.3.1) with the following item:

$$[\![\mathit{atomic}]\!]_{\boldsymbol{T}} \text{ is the function that maps } (i, j) \mapsto (i \sim_A j)$$

3.4.2 Control and Data Dependencies

Most memory models make some intrinsic ordering guarantees for instructions that are control- or data-dependent. For example, consider the common code sequence

1. read a pointer `p` to some structure
2. read a field `p->f`

This sequence produces two loads, the second of which is data dependent on the first (the address used by the second load depends on the value loaded by the first). On most memory models, such a data dependence guarantees that the memory system will execute the accesses in order. However, this is not true on all systems: on some architectures (most notably Alpha [13]), the processor is allowed to perform the load of `p->f` before the load of `p` by speculating on the value of `p` and then confirming it afterward [51]. On a multiprocessor, this early load may read an incorrect value because `p->f` may not have been written yet at the time it is loaded (we give an example in Section 8.2.2).

Our memory model specifications must express what ordering (if any) is implied by data and control dependencies. To do so, we express them as binary relations on the instructions in the trace. We use the following definition for control and data dependency. Note that because we have not formalized the actual instruction set (to be introduced in Chapter 4), we use the terms "instruction" and "register" informally here.

Data Dependency. we say an instruction y is data-dependent on a load x (written $x <_D y$) if there exists a $k \geq 1$ and instructions $z_0, \ldots, z_k$ such that $z_0 = x$, $z_k = y$ and for all $1 \leq i \leq k$, instruction z_i reads a data register that is written by instruction z_{i-1}.

Control Dependency. we say an instruction y is control-dependent on a load x (written $x <_C y$) if there exists a conditional branch instruction b such that b precedes y in the program trace, and b is data dependent on x.

Neither of the memory models we introduced so far (sequential consistency and *Relaxed*) make reference to data- or control-dependencies. However, the *RMO* model (for which we give a definition later in this chapter) does define a special dependency order that restricts the reorderings a processor may perform. Moreover, in the next section, we add special fences to *Relaxed* that order data- or control-dependent instructions.

To represent control- and data-dependencies in our formalism, we modify the following definitions.

- Modify the definitions of local trace and global trace (Sections 2.1.2 and 2.1.4) by extending the trace tuples as follows:

$$\boldsymbol{t} = (I, \prec, \mathit{adr}, \mathit{val}, \sim_A, <_D, <_C) \tag{3.3}$$

$$\boldsymbol{T} = (I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat}, \sim_A, <_D, <_C) \tag{3.4}$$

  and add the requirements that

  - $<_D$ is a binary relation over $\mathit{loads}(I) \times \mathit{accesses}(I)$
  - $<_C$ is a binary relation over $\mathit{loads}(I) \times \mathit{accesses}(I)$
  - for all $i, j \in I$: if $i <_D j$ then $i \prec j$
  - for all $i, j \in I$: if $i <_C j$ then $i \prec j$

- Extend Definition 13 (Section 3.3.1) to include the following second-order variables:

$$\mathit{data\_dependent}, \mathit{control\_dependent} \quad \in \quad \mathcal{F}^{\mathit{instr} \times \mathit{instr} \to \mathit{bool}}$$

- Extend Definition 15 (Section 3.3.1) with the following item:

$$[\![\mathit{data\_dependent}]\!]_{\boldsymbol{T}} \text{ is the function that maps } (i, j) \mapsto (i <_D j)$$
$$[\![\mathit{control\_dependent}]\!]_{\boldsymbol{T}} \text{ is the function that maps } (i, j) \mapsto (i <_C j)$$

3.4.3 Fences

Most memory models support multiple weaker fence variations. The reason is that a full fence is quite expensive and rarely needed. Most of the time, light-weight fences are appropriate to achieve the necessary synchronization.

Many architectures provide fences that order specific types of accesses only. For example, a “load-load” fence enforces ordering of loads that occur prior to the fence relative to loads that occur after the fence. Similarly, a “load-store” fence enforces ordering of loads that occur prior to the fence relative to stores that occur after the fence. Fig. 3.9 shows an example (adapted from the SPARC manual [72]) that shows how to build a mutual exclusion lock with partial fences.

In addition to these four fences (*ll_fence*, *ls_fence*, *sl_fence*, *ss_fence*) we found it convenient to include special fences for data-dependent loads (*ddl_fence*) and for control-dependent accesses (*cd_fence*) to *Relaxed*. The reason is that when we studied some implementations of concurrent data types on *Relaxed* (see Chapter 8) we found failures caused by reordered data- or control-dependent loads. In those situations, we prefer to use a specialized fence (that is, place a *ddl_fence* between the data-dependent loads, or *cd_fence* between the control dependent accesses) rather than using a general load-load or load-store fence. This makes it easier to remember why a particular fence is needed, and allows us to drop those fences if we port the implementation to a stronger memory model that enforces them automatically.

```
typedef enum { free, held } lock_t;

void lock(lock_t *lock) {
  lock_t val;
  do {
    atomic {
      val = *lock;
      *lock = held;
    }
  } while (val != free);
  fence("load-load");
  fence("load-store");
}

void unlock(lock_t *lock) {
  fence("load-store");
  fence("store-store");
  atomic {
    assert(*lock == held);
    *lock = free;
  }
}
```

Figure 3.9: Pseudocode for the lock and unlock operations. The fences in the *lock* call prevent accesses in the critical section (which is below the call to *lock*) from drifting up and out of the critical section, past the synchronizing load. Symmetrically, the fences in the *unlock* call prevent accesses in the critical section (which is above the call to *unlock*) from drifting down and out of the critical section, past the synchronizing store.

For the same reasons, we added another fence *al_fence* to order aliased loads; by inserting this fence in between loads that target the same address, we guarantee that they will not be reordered (almost all memory models already enforce the relative order of such loads, but *Relaxed* and SPARC RMO do not).

To represent these various fence types in the the memory model definition of *Relaxed*, we make the following modifications to the formalism:

- Replace axiom (A4) in the definition of *Relaxed* (Section 3.2.3) by the following seven axioms

(A4.1) if $x \in \mathit{loads}(I)$ and $y \in \mathit{loads}(I)$ and $f \in \mathit{fences}(I)$ and $\mathit{ftyp}(f) = \mathit{ll\_fence}$ and $x \prec f \prec y$, then $x <_M y$.

(A4.2) if $x \in \mathit{loads}(I)$ and $y \in \mathit{stores}(I)$ and $f \in \mathit{fences}(I)$ and $\mathit{ftyp}(f) = \mathit{ls\_fence}$ and $x \prec f \prec y$, then $x <_M y$.

(A4.3) if $x \in \mathit{stores}(I)$ and $y \in \mathit{loads}(I)$ and $f \in \mathit{fences}(I)$ and $\mathit{ftyp}(f) = \mathit{sl\_fence}$ and $x \prec f \prec y$, then $x <_M y$.

(A4.4) if $x \in \mathit{stores}(I)$ and $y \in \mathit{stores}(I)$ and $f \in \mathit{fences}(I)$ and $\mathit{ftyp}(f) = \mathit{ss\_fence}$ and $x \prec f \prec y$, then $x <_M y$.

(A4.5) if $x \in \mathit{loads}(I)$ and $y \in \mathit{loads}(I)$ and $f \in \mathit{fences}(I)$ and $\mathit{ftyp}(f) = \mathit{al\_fence}$ and $\mathit{adr}(x) = \mathit{adr}(y)$ and $x \prec f \prec y$, then $x <_M y$.

(A4.6) if $x \in \mathit{loads}(I)$ and $y \in \mathit{loads}(I)$ and $f \in \mathit{fences}(I)$ and $\mathit{ftyp}(f) = \mathit{ddl\_fence}$ and $x <_D y$ and $x \prec f \prec y$, then $x <_M y$.

(A4.7) if $x \in \mathit{loads}(I)$ and $y \in \mathit{accesses}(I)$ and $f \in \mathit{fences}(I)$ and $\mathit{ftyp}(f) = \mathit{cd\_fence}$ and $x <_C y$ and $x \prec f \prec y$, then $x <_M y$.

- Modify the definitions of local traces and global traces (Sections 2.1.2 and 2.1.4) by extending the trace tuples as follows:

$$\boldsymbol{t} = (I, \prec, \mathit{adr}, \mathit{val}, \sim_A, <_D, <_C, \mathit{ftyp}) \tag{3.5}$$

$$\boldsymbol{T} = (I, \prec, \mathit{proc}, \mathit{adr}, \mathit{val}, \mathit{seed}, \mathit{stat}, \sim_A, <_D, <_C, \mathit{ftyp}) \tag{3.6}$$

  and add the requirement that *ftyp* be a function

$$\mathit{ftyp} : \mathit{fences}(I) \to \{\mathit{ll\_fence}, \mathit{ls\_fence}, \\ \mathit{sl\_fence}, \mathit{ss\_fence}, \mathit{al\_fence}, \mathit{ddl\_fence}, \mathit{cd\_fence}\}.$$

- Extend Definition 13 (Section 3.3.1) to include the following predicates:

$$\{\mathit{load}, \mathit{store}, \mathit{fence}, \mathit{access}, \mathit{ll\_fence}, \mathit{ls\_fence}, \\ \mathit{sl\_fence}, \mathit{ss\_fence}, \mathit{al\_fence}, \mathit{ddl\_fence}, \mathit{cd\_fence}\} \subset \mathcal{F}^{\mathit{instr} \to \mathit{bool}}$$

- Extend Definition 15 (Section 3.3.1) with the following items:

$$\begin{aligned}
[\![\mathit{ll\_fence}]\!]_{\boldsymbol{T}} &\text{ is the function that maps } i \mapsto (\mathit{ftyp}(i) = \mathit{ll\_fence})\\
[\![\mathit{ls\_fence}]\!]_{\boldsymbol{T}} &\text{ is the function that maps } i \mapsto (\mathit{ftyp}(i) = \mathit{ls\_fence})\\
[\![\mathit{sl\_fence}]\!]_{\boldsymbol{T}} &\text{ is the function that maps } i \mapsto (\mathit{ftyp}(i) = \mathit{sl\_fence})\\
[\![\mathit{ss\_fence}]\!]_{\boldsymbol{T}} &\text{ is the function that maps } i \mapsto (\mathit{ftyp}(i) = \mathit{ss\_fence})\\
[\![\mathit{al\_fence}]\!]_{\boldsymbol{T}} &\text{ is the function that maps } i \mapsto (\mathit{ftyp}(i) = \mathit{al\_fence})\\
[\![\mathit{ddl\_fence}]\!]_{\boldsymbol{T}} &\text{ is the function that maps } i \mapsto (\mathit{ftyp}(i) = \mathit{ddl\_fence})\\
[\![\mathit{cd\_fence}]\!]_{\boldsymbol{T}} &\text{ is the function that maps } i \mapsto (\mathit{ftyp}(i) = \mathit{cd\_fence})
\end{aligned}$$

3.5 Axiomatic Memory Model Specifications

In this section, we demonstrate the syntax we use to specify memory models. We start by giving a short description of the format (Section 3.5.1), to clarify how it expresses an axiomatic specification as defined earlier in this chapter. We then list the complete source file for sequential consistency (Section 3.5.2) and *Relaxed* (Section 3.5.3).

A third specification (for the SPARC RMO model) is presented in Section 3.6. We plan to specify more models in the future; we expect our format to handle all common hardware memory models and some software models. Of particular interest is the support of models that do not enforce a global store order. However, we have not finished those formalizations; it is likely that they require minor modifications to the format presented here, which we consider to be somewhat experimental still.

3.5.1 Short Description of the Format

Each model definition is prefixed by the keyword `model`, followed by the name of the model. The model definition ends with the keyword `end model`. In between, the following sections are specified:

Predefined section. In this section, we declare the symbols that have some predefined meaning. The symbols listed in this section may include (1) sets that are part of the standard vocabulary, such as *instr*, and (2) boolean-valued second-order variables (that is, predicates and relations) that represent properties of the instructions in the trace. The syntax indicates the arity (number and type of arguments) to each variable. We defined these predicates and relations in Definition 13 (Section 3.3.1) and in Section 3.4 (the names are slightly more verbose, but easy to match up).

Exists section. In this section, we declare second-order variable symbols for use in the axioms, using the same syntax. The variables in this section are existentially quantified (in the sense that a global trace is allowed by the memory model if and only if we can find a valuation for them, see Def. 16 in section 3.3.1). For sequential consistency and *Relaxed*, this section consists of a single variable, the memory order. For *RMO*, it contains both the memory order and a dependency order (as defined by the official *RMO* specification).

Forall section. In this section, we declare the type of the first-order variable symbols we wish to use in the axioms. To distinguish these first-order variables visually from the second-order variables declared in the previous sections, we require them to start with a capital letter.

Require section. In this section, we list the axioms. Each axiom is prefixed with a label enclosed in angle brackets $\langle\ \rangle$. The axioms are boolean formulae using standard syntax for negation, conjunction, disjunction, implication, and quantification. The atomic subformulae are either boolean constants ($\mathit{true}, \mathit{false}$), second-order variables applied to first-order variables ($\mathit{rel}(X, Y, Z)$), or equality expressions involving first-order variables ($X = Y$). If an axiom contains free variables, we implicitly replicate it for all possible valuations (or equivalently, wrap universal quantifiers around the axiom to obtain a closed formula). We then take the conjunction of all the axioms (in the sense that a global trace is allowed by the memory model if and only if it satisfies all axioms, see Def. 16 in Section 3.3.1).

3.5.2 Axiomatic Specification of Sequential Consistency

```
model sc

predefined
  set instruction
  predicate access(instruction)
  predicate load(access)
  predicate store(access)
  relation aliased(access,access)
  relation seed(load,store)
  predicate has_seed(load)
  relation program_order(instruction,instruction)
  relation atomic(instruction,instruction)

exists
  relation memory_order(access,access)

forall
  L     : load
  S,S’  : store
  X,Y,Z : access

require
```

```
  <T1> memory_order(X,Y) & memory_order(Y,Z) => memory_order(X,Z)
  <T2> ~memory_order(X,X)
  <T3> memory_order(X,Y) | memory_order(Y,X) | X = Y

  <M1> program_order(X,Y) => memory_order(X,Y)

  <v1> seed(L,S) => memory_order(S,L)
  <v2> seed(L,S) & aliased(L,S’) & memory_order(S’,L) & ~(S = S’)
                                             => memory_order(S’,S)
  <v3> aliased(L,S) & memory_order(S,L) => has_seed(L)

  <a1> atomic(X,Y) & program_order(X,Y) & ~atomic(Y,Z)
                          & memory_order(X,Z) => memory_order(Y,Z)

end model
```

3.5.3 Axiomatic Specification of *Relaxed*

```
model relaxed

predefined
  set instruction
  predicate access(instruction)
  predicate load(access)
  predicate store(access)
  relation program_order(instruction,instruction)
  relation aliased(access,access)
  relation seed(load,store)
  predicate has_seed(load)
  predicate fence(instruction)
  predicate ll_fence(fence)
  predicate ls_fence(fence)
  predicate sl_fence(fence)
  predicate ss_fence(fence)
  predicate al_fence(fence)
  predicate ddl_fence(fence)
  predicate cd_fence(fence)
  relation data_dependent(load,instruction)
  relation control_dependent(load,instruction)
  relation atomic(instruction,instruction)

exists
  relation memory_order(access,access)

forall
  L,L’  : load
  S,S’  : store
  X,Y,Z : access
  F     : fence

require

  <T1> memory_order(X,Y) & memory_order(Y,Z) => memory_order(X,Z)
  <T2> ~memory_order(X,X)
  <T3> memory_order(X,Y) | memory_order(Y,X) | X = Y

  <M1> program_order(X,S) & aliased(X,S) => memory_order(X,S)

  <ll> ll_fence(F) & program_order(L,F) & program_order(F,L’)
                                            => memory_order(L,L’)
  <ls> ls_fence(F) & program_order(L,F) & program_order(F,S’)
```

```
                                            => memory_order(L,S’)
  <sl> sl_fence(F) & program_order(S,F) & program_order(F,L’)
                                            => memory_order(S,L’)
  <ss> ss_fence(F) & program_order(S,F) & program_order(F,S’)
                                            => memory_order(S,S’)
  <al> al_fence(F) & program_order(L,F) & program_order(F,L’)
                            & aliased(L,L’) => memory_order(L,L’)
  <dd> ddl_fence(F) & program_order(L,F) & program_order(F,L’)
                     & data_dependent(L,L’) => memory_order(L,L’)
  <cd> cd_fence(F) & program_order(L,F) & program_order(F,X)
                    & control_dependent(L,X) => memory_order(L,X)

  <v1> seed(L,S) => memory_order(S,L) | program_order(S,L)
  <v2> seed(L,S) & aliased(L,S’)
           & (memory_order(S’,L) | program_order(S’,L))
                                & ~(S = S’) => memory_order(S’,S)
  <v3> aliased(L,S) & (memory_order(S,L) | program_order(S,L))
                                                   => has_seed(L)

  <a1> atomic(X,Y) & program_order(X,Y) & ~atomic(Y,Z)
                         & memory_order(X,Z) => memory_order(Y,Z)
  <a2> atomic(X,Y) & program_order(X,Y) => memory_order(X,Y)

end model
```

3.6 The *RMO* Model

In this section we show our formalization of the RMO memory model. The RMO model is officially defined by the SPARC v9 manual [72]. Out of the many relaxed models that exist, we chose RMO as an example because it features a realistic level of detail and the manual specifies it with reasonable precision.[1].

The most important difference between *RMO* and either sequential consistency or *Relaxed* is that *two* ordering relations are used. Apart from the memory order (which is a total order as usual), the manual defines a *dependence order*, a partial order between accesses by the same thread.

We first show the complete definition in our specification format (Section 3.6.1). Then we briefly discuss how the *RMO* model relates to *Relaxed*(Section 3.6.2), and show how it corresponds to the official specification [72] (Section 3.6.3).

For more information on the SPARC models, we recommend the architecture manual [72] and previous work on model checking relaxed models [16, 60].

3.6.1 Axiomatic Specification of *RMO*

```
model rmo


predefined
  set instruction
  predicate access(instruction)
  predicate load(access)
  predicate store(access)
  relation program_order(instruction,instruction)
  relation aliased(access,access)
  relation seed(load,store)
  predicate has_seed(load)
  relation data_dependent(load,instruction)
  relation control_dependent(load,instruction)
  predicate fence(instruction)
  predicate ll_fence(fence)
```

[1] Unfortunately, hardware memory models are often underspecified in the architecture manual and thus become a matter of discussion.

```
  predicate ls_fence(fence)
  predicate sl_fence(fence)
  predicate ss_fence(fence)
  relation atomic(instruction,instruction)

exists
  relation memory_order(access,access)
  relation dependence_order(access,access)

forall
  L,L’  : load
  S,S’  : store
  X,Y,Z : access
  F     : fence

require

  <T1> memory_order(X,Y) & memory_order(Y,Z) => memory_order(X,Z)
  <T2> ~memory_order(X,X)
  <T3> memory_order(X,Y) | memory_order(Y,X) | X = Y

  <A1> program_order(L,X) & dependence_order(L,X)
                                             => memory_order(L,X)



  <A2_ll> ll_fence(F) & program_order(L,F)
                      & program_order(F,L’) => memory_order(L,L’)
  <A2_ls> ls_fence(F) & program_order(L,F)
                      & program_order(F,S’) => memory_order(L,S’)
  <A2_sl> sl_fence(F) & program_order(S,F)
```

```
                      & program_order(F,L’) => memory_order(S,L’)
  <A2_ss> ss_fence(F) & program_order(S,F)
                      & program_order(F,S’) => memory_order(S,S’)

  <A3> program_order(X,S) & aliased(X,S) => memory_order(X,S)

  <v1> seed(L,S) => memory_order(S,L) | program_order(S,L)
  <v2> seed(L,S) & aliased(L,S’)
               & (memory_order(S’,L) | program_order(S’,L))
                                & ~(S = S’) => memory_order(S’,S)
  <v3> aliased(L,S) & (memory_order(S,L) | program_order(S,L))
                                                   => has_seed(L)

  <D1> program_order(X,S) & control_dependent(X,S)
                                         => dependence_order(X,S)
  <D2> program_order(X,Y) & data_dependent(X,Y)
                                         => dependence_order(X,Y)
  <D3> program_order(S,L) & aliased(S,L) => dependence_order(S,L)

  <Dt> program_order(X,Y) & program_order(Y,Z)
             & dependence_order(X,Y) & dependence_order(Y,Z)
                                         => dependence_order(X,Z)


  <a1> atomic(X,Y) & program_order(X,Y) & ~atomic(Y,Z)
                         & memory_order(X,Z) => memory_order(Y,Z)
  <a2> atomic(X,Y) & program_order(X,Y) => memory_order(X,Y)

end model
```

3.6.2 Comparison to *Relaxed*

We now compare the *RMO* axioms above with *Relaxed*. We claimed earlier that *Relaxed* conservatively approximates *RMO*. This claim can now be verified: any trace that satisfies the axioms for *RMO* must also satisfy the axioms for *Relaxed* (at least for code without special *Relaxed*-specific fence types) because the latter are a subset of the former:

- The axioms ⟨`T1`⟩, ⟨`T2`⟩, ⟨`T3`⟩, ⟨`v1`⟩, ⟨`v2`⟩, ⟨`v3`⟩, ⟨`a1`⟩, and ⟨`a2`⟩ are identical in both models.
- The axiom ⟨`M1`⟩ is identical to ⟨`A3`⟩.
- The axioms ⟨`ll`⟩, ⟨`ls`⟩, ⟨`sl`⟩, and ⟨`ss`⟩ are identical to ⟨`A2_ll`⟩, ⟨`A2_ls`⟩, ⟨`A2_sl`⟩, ⟨`A2_ss`⟩.
- The axioms ⟨`cd`⟩, ⟨`ddl`⟩ and ⟨`al`⟩ are vacuously satisfied for code that does not contain any of the respective fences.

Essentially, we are simply removing all references to the dependence order, thus allowing the processor to reorder dependent instructions at will.

3.6.3 Comparison to the SPARC Specification

We derived the *RMO* axioms above from the official SPARC v9 architecture manual [72], Appendix D (titled "Formal Specification of the Memory Models"). They correspond as follows:

- The axioms ⟨`T1`⟩, ⟨`T2`⟩, and ⟨`T3`⟩ encode that the memory order be a total order, as usual.
- The axioms ⟨`A1`⟩ and ⟨`A3`⟩ correspond to the axioms (1) and (3) in section D.4.4 of the SPARC manual, respectively. Furthermore, the axioms ⟨`A2_ll`⟩,

⟨A2_ls⟩, ⟨A2_sl⟩ and ⟨A2_ss⟩ all correspond to axiom (2), each treating one fence type.

- The axioms ⟨v1⟩, ⟨v2⟩ and ⟨v3⟩ define the value flow as stated in section D.4.5 (which allows store forwarding).

- The axioms ⟨D1⟩, ⟨D2⟩ and ⟨D3⟩ correspond to the axioms (1), (2) and (3) in section D.3.3, respectively, which define the dependence order.

- The axiom ⟨Dt⟩ defines transitivity of the dependence order. We use a little 'trick' here to improve the encoding. Because we know the dependence order applies to accesses in the same thread only, we prefix the axiom with conditions that make it vacuously satisfied if the accesses are not in the same thread. During our encoding, no constraints are thus generated to express transitivity across threads.

- The axioms ⟨a1⟩ and ⟨a2⟩ have no corresponding definitions in the SPARC manual, as they define atomic sections (which do not exist on the latter).

Chapter 4

Programs

In this section, we formalize our notion of program, by introducing an intermediate language called LSL (load-store language) with a formal syntax and semantics. Furthermore, we describe how to unroll programs to obtain finite, bounded versions that can be encoded into formulae.

4.1 Motivation for an Intermediate Language

The C language does not specify a memory model (standardization efforts for C/C++ are still under way). Therefore, executing memory-model-sensitive C code on a multiprocessor can have unpredictable effects [7] because the compiler may perform optimizations that alter the semantics on multiprocessors. On the machine language level, however, the memory model is officially defined by the hardware architecture. It is therefore possible to write C code for relaxed models by exerting direct control over the C compilation process to prevent optimizations that would alter the program semantics. Unfortunately, the details of this process are machine dependent and not portable.

To keep the focus on the key issues, we compile C into an an intermediate language LSL (load-store language) that resembles assembly language, but abstracts irrelevant machine details. We describe LSL and the translation in more detail later in this chapter. The design of LSL addresses the following requirements:

Trace Semantics. LSL programs have a well-defined behavior on multiprocessors, for a variety of memory models. Specifically, for each LSL program, we give a formal definition of what memory traces it may produce (in the sense of Section 2.1.3).

Conciseness. LSL is reasonably small and abstract to simplify reasoning and keep proofs manageable.

Translation. LSL has enough features to model programs in the target domain. Specifically, our *CheckFence* tool can automatically translate typical C implementation code for concurrent data types into LSL.

Encoding. We can encode the execution of unrolled LSL programs as formulae.

4.2 Syntax

Fig. 4.1 shows the abstract syntax of LSL, slightly simplified.[1] It lists the syntactic entities used to construct LSL programs. We discuss their meaning in the following sections.

4.2.1 Types

The C type system does not make strong guarantees and is therefore of little use to our encoding. We thus decided against using a static type system for LSL. We do however track the "runtime type" of each value: rather than encoding all values as "flat" integers as done by the underlying machine, we distinguish between undefined, pointer, or number values.

[1] To simplify the presentation, we assume that there is only a single type of fence. Adding more fence types (as discussed in Section 3.4.3) is technically straightforward. We also do not show the syntax for atomic blocks here, but defer it until Section 4.6.2.

(register)	r	$\in$	Reg	
(procedure)	p	$\in$	Proc	
(label)	l	$\in$	Lab	
(primitive op)	f	$\in$	Fun	
(value)	v	$\in$	Val	
(undefined)		$::=$	$\bot$	
(number)			$\mid\ n$	$n \in \mathbb{Z}$
(pointer)			$\mid\ [n_1 \ldots n_k]$	$(k \geq 1$ and $n_0, \ldots, n_k \in \mathbb{Z})$
(execution state)	e	$\in$	Exs	
(ok)		$::=$	ok	
(abort)			$\mid \mathit{abort}$	
(break)			$\mid \mathit{break}\ l$	
(continue)			$\mid \mathit{continue}\ l$	
(program statement)	s	$\in$	Stm	
(composition)		$::=$	$s\,;s$	
(label)			$\mid l : s$	
(throw)			$\mid$ **throw**(e)	$(e \neq \mathit{ok})$
(conditional)			$\mid$ **if** $(r)\ s$	
(procedure call)			$\mid p(r_1, \ldots, r_k)(r_1, \ldots, r_l)$	$(k, l \geq 0)$
(assign)			$\mid r := (f\ r_1\ \ldots\ r_k)$	$(k \geq 0)$
(store)			$\mid$ *****$r := r$	
(load)			$\mid r :=$ *****r	
(fence)			$\mid$ **fence**	
(procedure definition)	**def** $p(r_1, \ldots, r_k)(r_1, \ldots, r_l) = s$			$(k, l \geq 0)$

Figure 4.1: The abstract syntax of LSL

4.2.2 Values

We let v represent a value. We use the following values:

- We let $\perp$ represent an undefined value. It is the default value of any register or memory location before it is assigned or stored the first time.

- We let n represent a signed integer. We assume a two's complement bitvector representation for integers. However, unlike C, we do not model any overflow issues (assuming that the vectors are always wide enough to hold all values that appear in the executions under consideration).

- Pointers are represented as non-empty integer lists $[n_1 \ldots n_k]$; the first integer n_1 represents the base address, and the following integers $n_2, \ldots, n_k$ represent the sequence of offsets that are applied to the base pointer.

Using this extra structure on values has the following advantages:

- We can perform some automatic runtime type checks that help to catch programmer mistakes early (Section 7.1.1). For example, we flag it as an error if a programmer dereferences a non-pointer value.

- We can optimize the encoding by using type-specific encodings. For example, rather than calculating pointer offsets using multiplication and addition (which are expensive to encode), we use a simple list-append operation which we can encode very efficiently (shifting bits).

- We can recover some of the benefits of static type systems (in particular, achieve more succinct encodings) by using a light-weight flow-insensitive program analysis that recovers static type information (Section 7.3.2).

4.2.3 Identifiers

- We let r represent a register identifier. Registers are used to store intermediate computation values. We assume that there is an infinite supply of registers.

- We let p represent a procedure identifier. Procedure identifiers are used to connect procedure calls to the corresponding procedure definition.

- We let l represent a label. Labels are used to model control flow, including loops. Labels are not required to be unique.

- We let f represents a basic function identifier. We describe the supported functions in more detail in Section 4.2.8 below.

4.2.4 Execution States

We let e represent an execution state. Whenever a statement executes, it terminates with some execution state, which is one of the following:

- The state *ok* represents normal termination. It also means "proceed normally", in the following sense: If we execute a composed statement $(s\,;s)$, we start by executing the first one. If it terminates with execution state *ok*, the second statement is executed next.

- The state *abort* represents an error condition that causes a thread to abandon regular program execution immediately, such as a failing programmer assertion, or a runtime error (e.g. a division by zero or a null dereference).

- The state *break* l indicates that the program should immediately exit the nearest enclosing block with label l. It is always caused by a **throw**(*break* l) statement, and acts just like an exception (as defined by Java, for example). If there is no enclosing block with label l, the whole program terminates with execution state *break* l.

- The state *continue* l indicates that the program should immediately restart execution of the nearest enclosing block with label l. It is always caused by a statement **throw**(*continue* l), and acts like continue statements in C or Java.

The execution states *break* l and *continue* l are used in conjunction with labeled blocks to implement control structures such as switch statements or loops. We encourage the reader to look at the examples in Fig. 4.3, Fig. 4.4 and Fig. 4.6 now, which illustrate how LSL represents control flow.

4.2.5 Procedure Definitions

A procedure definition has the form **def** $p(r_1, \ldots, r_k)(r'_1, \ldots, r'_l) = s$. The registers $r_1, \ldots, r_k$ are called *input registers*, and the registers $r'_1, \ldots, r'_l$ are called *output registers*. The statement s represents the procedure body.

When a procedure is executed, the input registers contain the argument values of the call, and the procedure body assigns the return value(s) to the output registers.

4.2.6 Program Statements

$s \in \mathit{Stm}$ represents a program statement (or an entire program). The individual statements are as follows:

- Composition of statements $s_1 \,; s_2$. The statement s_1 is executed first; if it completes with an execution state other than *ok*, the statement s_2 is skipped.

- Labeled statement $l : s$. If the statement s completes with an execution state of *continue* l, then it is executed again. If the statement s completes with an execution state of *break* l, then $l : s$ completes with an execution state of *ok*. Otherwise, $l : s$ behaves the same as s.

- Throw statement **throw**(e). Completes with execution state e. It can therefore be used for aborting the thread, for continuing an enclosing loop, or for breaking out of an enclosing block.

- Conditional **if**$\,(r)\,s$. If register r is nonzero, s is executed, otherwise s is skipped.

- Procedure call $p(r_1, \ldots, r_k)(r'_1, \ldots, r'_l)$. All arguments are call-by-value. When a procedure is called, we execute the body of the procedure definition for p, after assigning the contents of registers $r_1, \ldots, r_k$ to the input registers of p. If the execution completes with state *ok*, the contents of the output registers of p are assigned to the registers $r'_1, \ldots, r'_l$. Otherwise the output registers retain the value they had before the procedure call.

- Assignment $r := (f\ r_1\ \ldots\ r_k)$. The function f is applied to the values contained in the registers $r_1, \ldots, r_k$, and the resulting value is assigned to the register r.

- Load $r := {*}r'$. The memory location whose address is contained in register r' is loaded, and the resulting value is assigned to register r.

- Store ${*}r' := r$. The value contained in register r is stored into the memory location whose address is contained in register r'.

- Fence **fence**. Produces a memory fence instruction.

4.2.7 Programs

With the syntactic entities thus defined, we are ready to define the set of programs.

Definition 17 *A* program *p is a tuple (s, D) where s is a statement and D is a set of procedure definitions such that there is a unique matching definition for each call that appears in s or D. Let Prog be the set of all such programs.*

4.2.8 Basic Functions

We now describe the basic functions available in LSL in more detail. For the most part, these functions are similar to the corresponding C operators, but they reside at a somewhat higher abstraction level, with the following differences:

- We ignore all effects that result from the bounded width of the underlying bitvector representation.
- We use special operators for pointer manipulation, instead of relying on normal integer multiplication and addition for offset calculations.
- We do not support floating-point operations or values.

The set *Fun* contains the following basic functions. For each basic function, we describe the arity, the domain, and the semantics. Conceptually, basic functions are partial functions only as they may not be well-defined on all inputs (for example, the division function is not defined for zero divisors). For convenient formalization, we specify that a basic function produce the special value $\perp$ if undefined.[2]

- For each value $v \in \mathit{Val}$, there is a corresponding zero-arity constant function $\mathit{constant}\langle v \rangle$. It produces the value v.
- The unary function *identity* returns the same value as given. Its domain is the entire set *Val*.
- The binary function *equals* is defined on $\mathit{Val} \times \mathit{Val}$ and returns 1 if its argument values are the same, or 0 otherwise. The binary function *notequals* is defined on $\mathit{Val} \times \mathit{Val}$ and returns 0 if its argument values are the same, or 1 otherwise.
- The unary function *bitnot* and the binary functions *bitand*, *bitor*, *bitxor* are defined on numbers and have the same semantics as the respective bitwise C operators `~`, `&`, `|`, and `^`.

[2] In practice, it makes more sense to abort execution and report the runtime error to the user, which is in fact what our CheckFence implementation does. We do not include this mechanism in the basic presentation, but it should be fairly clear how to support it by instrumenting the source code with assertions.

- The unary function *minus* and the binary functions *add*, *subtract*, *multiply* are defined on numbers and have the same semantics as the respective arithmetic C operators `-`, `+`, `-`, and `*`.
- The binary functions *div*, *mod* are defined on all pairs of numbers such that the second number is not zero. They have the same semantics as the respective arithmetic C operators `/` and `%`.
- The binary functions *lessthan*, *lesseq* are defined on all pairs of numbers. They have the same semantics as the respective C comparison operators `<` and `<=`.
- The binary functions *shiftleft*, *shiftright* are defined on all pairs of numbers and have the same semantics as the respective C operators `<<` and `>>`.
- The unary function *pointer* takes a numerical argument n and produces the pointer $[n]$. The binary function *offset* takes two arguments, a pointer $[n_1 \dots n_k]$ and a number n, and produces the pointer $[n_1 \dots n_k\ n]$. The unary function *getoffset* takes a pointer argument $[n_1 \dots n_k]$ and produces the number n_k. The unary function *getbase* takes a pointer argument $[n_1 \dots n_k]$ such that $k \geq 2$ and produces the pointer $[n_1 \dots n_{k-1}]$.

We do not need basic functions for every single C operator. For instance, to provide `>` or `>=`, we can simply swap the operands and use `<` or `<=` instead. We also do not directly implement the logic operators `!`, `&&` and `||`, but rather use *equals* 0/*notequals* 0 and the bitwise logical functions to implement the logical function. We also use explicit conditionals to express the implicit control flow aspect of `&&` and `||` (the second expression is only evaluated if the first does not already satisfy the logical connective).

4.3 Memory Trace Semantics

In this section, we give a formal semantics for LSL programs. Specifically, we define a local trace semantics (in the sense of Section 2.1.3) that defines the possible memory traces for a given LSL program.

We found that a big-step operational style fits our purpose best. We thus give a set of inference rules to derive sentences of the form

$$\boxed{\Gamma, \boldsymbol{t}, s \ \Downarrow_D \ \Gamma', \boldsymbol{t}', e}$$

with the following intuitive interpretation: "when starting with a register map Γ, a local trace $\boldsymbol{t}$, and a set D of procedure definitions, execution of the statement s completes with a register map Γ', a local trace $\boldsymbol{t}'$, and execution state e."

To prepare for the actual inference rules, we first need to define register maps and introduce syntactic shortcuts for constructing local traces.

Definition 18 *A* register map *is a function* $\Gamma : \mathit{Reg} \rightarrow \mathit{Val}$ *such that* $\Gamma(r) = \bot$ *for all but finitely many registers* r*. Define the initial register map* Γ^0 *to be the constant function such that* $\Gamma^0(r) = \bot$ *for all* r*.*

We construct the memory trace of a program by appending instructions one at a time. The following definition provides a notational shortcut to express this operation:

Definition 19 *For a local memory trace* $\boldsymbol{t} = (I, \prec, \mathit{adr}, \mathit{val})$ *and for elements* $i \in \mathcal{I} \setminus I$ *and* $a, v \in \mathit{Val}$ *we use the notation*

$$\boldsymbol{t}.\mathit{append}(i, a, v)$$

to denote a local memory trace $(I', \prec', \mathit{adr}', \mathit{val}')$ *where*

- $I' = I \cup \{i\}$
- $\prec' = \prec \cup (I \times \{i\})$
- $\mathit{adr}'(x) = \begin{cases} \mathit{adr}(x) & \mathit{if}\ x \neq i \\ a & \mathit{if}\ x = i \end{cases}$
- $\mathit{val}'(x) = \begin{cases} \mathit{val}(x) & \mathit{if}\ x \neq i \\ v & \mathit{if}\ x = i \end{cases}$

4.3.1 Inference Rules

With these definitions in place, we can now give the inference rules.

We use two rules for composition. Which one applied depends on whether the first statement terminates with execution state *ok*:

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', \mathit{ok} \qquad \Gamma', \boldsymbol{t}', s' \ \Downarrow_D\ \Gamma'', \boldsymbol{t}'', e}{\Gamma, \boldsymbol{t}, s\,;s' \ \Downarrow_D\ \Gamma'', \boldsymbol{t}'', e} \tag{\textsc{Comp-ok}}$$

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', e \qquad e \neq \mathit{ok}}{\Gamma, \boldsymbol{t}, s\,;s' \ \Downarrow_D\ \Gamma', \boldsymbol{t}', e} \tag{\textsc{Comp-skip}}$$

We use three rules for labeled blocks. They correspond to cases where the label has no effect, where the label catches a break, and where the label catches a continue:

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', e \qquad e \notin \{\mathit{break}\, l, \mathit{continue}\, l\}}{\Gamma, \boldsymbol{t}, l : s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', e} \tag{\textsc{Label}}$$

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', \mathit{break}\, l}{\Gamma, \boldsymbol{t}, l : s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', \mathit{ok}} \tag{\textsc{Label-break}}$$

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', \mathit{continue}\, l \qquad \Gamma', \boldsymbol{t}', l : s \ \Downarrow_D\ \Gamma'', \boldsymbol{t}'', e}{\Gamma, \boldsymbol{t}, l : s \ \Downarrow_D\ \Gamma'', \boldsymbol{t}'', e} \tag{\textsc{Label-cont}}$$

The throw statement simply changes the execution state:

$$\Gamma, \boldsymbol{t}, \mathbf{throw}(e) \ \Downarrow_D\ \Gamma, \boldsymbol{t}, e \tag{\textsc{Throw}}$$

We use two rules for conditionals, depending on the outcome of the test;

$$\frac{\Gamma(r) = 0}{\Gamma, \boldsymbol{t}, \mathbf{if}\,(r)\, s \ \Downarrow_D\ \Gamma, \boldsymbol{t}, \mathit{ok}} \tag{\textsc{If-not}}$$

$$\frac{\Gamma(r) \neq 0 \qquad \Gamma, \boldsymbol{t}, s \ \Downarrow\ \Gamma', \boldsymbol{t}', e}{\Gamma, \boldsymbol{t}, \mathbf{if}\,(r)\, s \ \Downarrow_D\ \Gamma', \boldsymbol{t}', e} \tag{\textsc{If-so}}$$

Assignments do modify the registers, but do not change the memory trace:

$$\frac{v = f(\Gamma(r_1), \ldots, \Gamma(r_k))}{\Gamma, \boldsymbol{t}, r := (f\ r_1\ \ldots\ r_k) \Downarrow_D\ \Gamma[r \mapsto v], \boldsymbol{t}, \mathit{ok}} \quad (\textsc{Assign})$$

The three rules for store, load, and fence instruction append a (nondeterministically chosen) fresh instruction identifier to the trace:

$$\frac{i \in (\mathit{cmd}^{-1}(\mathit{store}) \setminus \boldsymbol{t}.I) \qquad \boldsymbol{t'} = \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), \Gamma(r))}{\Gamma, \boldsymbol{t}, {*}r' := r\ \Downarrow_D\ \Gamma, \boldsymbol{t'}, \mathit{ok}} \quad (\textsc{Store})$$

$$\frac{v \in \mathit{Val} \qquad i \in (\mathit{cmd}^{-1}(\mathit{load}) \setminus \boldsymbol{t}.I) \qquad \boldsymbol{t'} = \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), v)}{\Gamma, \boldsymbol{t}, r := {*}r'\ \Downarrow_D\ \Gamma[r \mapsto v], \boldsymbol{t'}, \mathit{ok}} \quad (\textsc{Load})$$

$$\frac{i \in (\mathit{cmd}^{-1}(\mathit{fence}) \setminus \boldsymbol{t}.I) \qquad \boldsymbol{t'} = \boldsymbol{t}.\mathit{append}(i, \bot, \bot)}{\Gamma, \boldsymbol{t}, \textbf{fence}\ \Downarrow_D\ \Gamma, \boldsymbol{t'}, \mathit{ok}} \quad (\textsc{Fence})$$

The rules for procedure calls are a little more complicated. We use one of two rules, depending on whether the procedure body finishes with execution state *ok*. The registers $\hat{r}_i$ and $\hat{r}'_i$ represent the input and output registers of the procedure, respectively. The registers r_i contain the argument values, and the registers r'_i are the destination registers for the return values.

$$\frac{\begin{array}{c}\textbf{def}\, p(\hat{r}_1, \ldots, \hat{r}_k)(\hat{r}'_1, \ldots, \hat{r}'_l) = s \quad \in \quad D \\ [\hat{r}_1 \mapsto \Gamma(r_1)] \ldots [\hat{r}_k \mapsto \Gamma(r_k)], \boldsymbol{t}, s\ \Downarrow_D\ \Gamma', \boldsymbol{t'}, \mathit{ok}\end{array}}{\Gamma, \boldsymbol{t}, p(r_1, \ldots, r_k)(r'_1, \ldots, r'_l)\ \Downarrow_D\ \Gamma[r'_1 \mapsto \Gamma'(\hat{r}'_1)] \ldots [r'_l \mapsto \Gamma'(\hat{r}'_l)], \boldsymbol{t'}, \mathit{ok}} \quad (\textsc{Call-ok})$$

$$\frac{\begin{array}{c}\textbf{def}\, p(\hat{r}_1, \ldots, \hat{r}_k)(\hat{r}'_1, \ldots, \hat{r}'_l) = s \quad \in \quad D \\ [\hat{r}_1 \mapsto \Gamma(r_1)] \ldots [\hat{r}_k \mapsto \Gamma(r_k)], \boldsymbol{t}, s\ \Downarrow_D\ \Gamma', \boldsymbol{t'}, e \quad e \neq \mathit{ok}\end{array}}{\Gamma, \boldsymbol{t}, p(r_1, \ldots, r_k)(r'_1, \ldots, r'_l)\ \Downarrow_D\ \Gamma, \boldsymbol{t'}, e} \quad (\textsc{Call-ex})$$

We can now conclude this section with the definition of the function $E_L : \mathit{Prog} \rightarrow \mathcal{P}(\mathit{Ltr} \times \mathit{Exs})$ which describes the local traces that a given program may produce.

```
struct {
  long a;
  int b[3];
} x;
int y;
```

Pointer	C value	LSL value
`&(x)`	`0x000`	`[ 0 ]`
`&(x.a)`	`0x000`	`[ 0 0 ]`
`&(x.b)`	`0x008`	`[ 0 1 ]`
`&(x.b[0])`	`0x008`	`[ 0 1 0 ]`
`&(x.b[1])`	`0x00C`	`[ 0 1 1 ]`
`&(x.b[2])`	`0x010`	`[ 0 1 2 ]`
`&(y)`	`0x014`	`[ 1 ]`

Figure 4.2: Representing C pointers in LSL.

Definition 20 *For a given program $p = (s, D)$, let $E_L(p)$ be the set of all tuples $(\boldsymbol{t}, e)$ such that there exists a register map Γ and a derivation for* $\Gamma^0, \boldsymbol{t}^0, s \Downarrow_D \Gamma, \boldsymbol{t}, e$.

4.4 From C to LSL

An important design goal of LSL is to allow an automatic translation from C to LSL. Such a translation is quite similar to a standard compilation of C into machine language. However, because LSL sits at a somewhat higher abstraction level than machine language, there are some differences. We discuss the implementation issues we encountered, our solutions, and our compromises in more detail in Chapter 7. For now, let us just give a few brief examples to illustrate how to utilize LSL.

- Pointer values are represented as a sequence of natural numbers, representing the base address and sequence of offsets. The offsets may be field or array offsets, providing a unified way of handling arrays and structs. See Fig. 4.2 for an example of how C pointers may be represented.

- Two-armed conditionals can be modeled by labeled blocks and break statements. We show an example in Fig. 4.3 in the abstract syntax.

- Loops can be modeled by labeled blocks and continue statements (Fig. 4.4).

```
if (x.a) {
  y = 1;
} else {
  y = 2
};
int y;
```

```
pxa := (constant⟨[0 0]⟩);
r := *pxa;
cond : {
  if (r) {
    py := (constant⟨[1]⟩);
    one := (constant⟨1⟩);
    *py := one;
    throw(break cond)
  };
  py := (constant⟨[1]⟩);
  two := (constant⟨2⟩);
  *py := two
}
```

Figure 4.3: Example of how to translate a two-armed conditional from C to LSL.

```
int i = 0;
while (i < 2) {
  i = i + 1;
}
```

```
i := (constant⟨0⟩);
one := (constant⟨1⟩);
two := (constant⟨2⟩);
loop : {
  c := (lessthan i two);
  c := (bitnot c);
  if (c) throw(break loop);
  i := (add i one);
  throw(continue loop)
}
```

Figure 4.4: Example of how to translate a loop from C to LSL.

As apparent in the examples, the abstract syntax is quite cumbersome to write and read; for that reason, our tool introduces a concrete syntax for LSL (which we won't discuss in detail here) that leads to more streamlined code (for example, it does not require semicolons, allows direct use of constants, and provides syntactic sugar to simplify common use of control structures).

4.5 Unrolling Programs

In the next chapter (Chapter 5) we will present a way of encoding programs into a formula such that the solutions of the formula correspond to executions of the program. However, this encoding works only for unrolled programs. We now explain how we unroll programs, and how the executions of the unrolled program correspond to the original program.

First, we show how we unroll a given program, by explaining the various steps of the program transformation (Section 4.5.1). For the targeted domain, programs are naturally bounded and we can often find an unrolling that is sufficient for all executions. We then describe the syntax and semantics of the unrolled programs (Section 4.5.2). Finally, we clarify how the unrolled program relates to the original program (Section 4.5.3).

4.5.1 Unrolling Programs

When unrolling programs, we perform a number of transformation steps. In this section, we give a brief overview of this transformation and the resulting format.

As inputs to the unrolling procedure, we are given a concurrent program $P = (p_1, \ldots, p_n)$ and a *pruning specification.* A pruning specification consists of individual iteration bounds for each loop instance, and recursion bounds for recursive calls.

First, we perform the following steps on each program p_i:

- Unroll each loop as many times as indicated by the pruning specification, and replace the remaining iterations with a **throw**(*pruned*) statement (to catch the situation where our unrolling was not sufficient to cover all executions).

- Inline procedure calls. Roughly, we proceed as follows. First, we rename the registers in the body if there is any overlap with the calling context. Then, we replace the call with a composition of the following three steps: (1) assign the actual arguments to the input registers of the procedure, (2) execute the procedure body, and (3) assign the output registers of the procedure to the destination registers. If there is recursion, we handle it analogously to loop unrolling (that is, inline to the specified depth and replace the remaining calls with a **throw**(*pruned*) statement).

- Express all conditionals using conditional throws. This simplifies the encoding algorithm and does not reduce the expressive power (we can wrap labeled blocks around conditionally executed statements, and conditionally exit the block before executing them).

After these transformations, the code contains no more calls, and the throw statements may only cause *pruned*, *abort*, or *break* l. Therefore, each throw has the effect of a *forward jump*; speaking in terms of control flow graphs, our program is now a DAG (directed acyclic graph). The program is thus clearly bounded: the program source contains a finite number of loads, stores, and fences, and each statement can be executed at most once.

We now go through one final transformation step: we modify the syntax for loads, stores, and fences by affixing a unique instruction identifier i (chosen from the pool $\mathcal{I}$) such that $cmd(i)$ matches the command type. Note that the inference rules that describe evaluation of LSL programs choose a nondeterministic identifier; we now reduce this nondeterminism somewhat by choosing a fixed identifier at the time the unrolling is performed.

(register)	r	$\in$	Reg	
(procedure)	p	$\in$	Proc	
(label)	l	$\in$	Lab	
(primitive op)	f	$\in$	Fun	
(instruction id)	i	$\in$	$\mathcal{I}$	
(value)	v	$\in$	Val	
(undefined)		$::=$	$\bot$	
(number)			$\mid\ n$	$n \in \mathbb{Z}$
(pointer)			$\mid\ [n_1 \ldots n_k]$	$(k \geq 1$ and $n_0, \ldots, n_k \in \mathbb{Z})$
(execution state)	e	$\in$	Exs	
(ok)		$::=$	ok	
(abort)			$\mid \mathit{abort}$	
(break)			$\mid \mathit{break}\ l$	
(pruned)			$\mid \mathit{pruned}$	
(program statement)	s	$\in$	Stm	
(composition)		$::=$	$s\,;s$	
(label)			$\mid l : s$	
(conditional throw)			$\mid \mathbf{if}\,(r)\,\mathbf{throw}(e)$	$(e \neq \mathit{ok})$
(assign)			$\mid r := (f\ r_1\ \ldots\ r_k)$	$(k \geq 0)$
(store-id)			$\mid *r :=_i r$	
(load-id)			$\mid r :=_i *r$	
(fence-id)			$\mid \mathbf{fence}_i$	

Figure 4.5: The abstract syntax of unrolled LSL programs

4.5.2 Unrolled Programs

The result of the transformation is an unrolled program, which we formally define as follows.

Definition 21 *An* unrolled program *is a statement s formed according to the rules in Fig. 4.5, such that each identifier $i \in \mathcal{I}$ occurs at most once in s. An* unrolled concurrent program *is a tuple of statements $(s_1, \ldots, s_n)$ where each s_i is an unrolled program, and such that each identifier $i \in \mathcal{I}$ occurs at most once in $s_1, \ldots, s_n$.*

We continue to use the same big-step operational style the semantics (introduced in Section 4.3.1), with the following changes:

- We no longer need the rule (Label-cont) because the code contains no more loops.
- We no longer need the subscript D that represents procedure definitions in sentences of the form $(\Gamma, \boldsymbol{t}, s \ \Downarrow_D \ \Gamma', \boldsymbol{t}', e)$, and we no longer need the rules (Call-ex) and (Call-ok), because the code contains no more calls.
- For loads, stores, and fences, the identifier i is now part of the syntax.

The derived sentences are thus of the form

$$\boxed{\Gamma, \boldsymbol{t}, s \ \Downarrow \ \Gamma', \boldsymbol{t}', e}$$

with the following intuitive interpretation: "when starting with a register map Γ and a local trace $\boldsymbol{t}$, execution of the statement s completes with a register map Γ', a local trace $\boldsymbol{t}'$, and execution state e." The inference rules are as follows:

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow \ \Gamma', \boldsymbol{t}', ok \qquad \Gamma', \boldsymbol{t}', s' \ \Downarrow \ \Gamma'', \boldsymbol{t}'', e}{\Gamma, \boldsymbol{t}, s \,;\, s' \ \Downarrow \ \Gamma'', \boldsymbol{t}'', e} \qquad (\textsc{Comp-ok})$$

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow \ \Gamma', \boldsymbol{t}', e \qquad e \neq ok}{\Gamma, \boldsymbol{t}, s \,;\, s' \ \Downarrow \ \Gamma', \boldsymbol{t}', e} \qquad (\textsc{Comp-skip})$$

$$\frac{\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e \qquad e \notin \{\mathit{break}\, l, \mathit{continue}\, l\}}{\Gamma, \boldsymbol{t}, l : s \Downarrow \Gamma', \boldsymbol{t}', e} \tag{Label}$$

$$\frac{\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', \mathit{break}\, l}{\Gamma, \boldsymbol{t}, l : s \Downarrow \Gamma', \boldsymbol{t}', \mathit{ok}} \tag{Label-break}$$

$$\Gamma, \boldsymbol{t}, \mathbf{throw}(e) \Downarrow \Gamma, \boldsymbol{t}, e \tag{Throw}$$

$$\frac{\Gamma(r) = 0}{\Gamma, \boldsymbol{t}, \mathbf{if}\,(r)\, s \Downarrow \Gamma, \boldsymbol{t}, \mathit{ok}} \tag{If-not}$$

$$\frac{\Gamma(r) \neq 0 \qquad \Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e}{\Gamma, \boldsymbol{t}, \mathbf{if}\,(r)\, s \Downarrow \Gamma', \boldsymbol{t}', e} \tag{If-so}$$

$$\frac{v = f(\Gamma(r_1), \ldots, \Gamma(r_k))}{\Gamma, \boldsymbol{t}, r := (f\ r_1\ \ldots\ r_k) \Downarrow \Gamma[r \mapsto v], \boldsymbol{t}, \mathit{ok}} \tag{Assign}$$

$$\frac{\boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), \Gamma(r))}{\Gamma, \boldsymbol{t}, {*r'} :=_i r \Downarrow \Gamma, \boldsymbol{t}', \mathit{ok}} \tag{Store-id}$$

$$\frac{v \in \mathit{Val} \qquad \boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), v)}{\Gamma, \boldsymbol{t}, r :=_i {*r'} \Downarrow \Gamma[r \mapsto v], \boldsymbol{t}', \mathit{ok}} \tag{Load-id}$$

$$\frac{\boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \bot, \bot)}{\Gamma, \boldsymbol{t}, \mathbf{fence}_i \Downarrow \Gamma, \boldsymbol{t}', \mathit{ok}} \tag{Fence-id}$$

Just as for regular programs (Def. 20 in Section 4.3.1), we define the semantic function E_L on unrolled programs:

Definition 22 *For an unrolled program s let $E_L(s)$ be the set of all tuples $(\boldsymbol{t}, e)$ such that there exists a register map Γ and a derivation for $\Gamma^0, \boldsymbol{t}^0, s \Downarrow \Gamma, \boldsymbol{t}, e$.*

For notational convenience, define the following function:

Definition 23 *For an unrolled concurrent program $P = (s_1, \ldots, s_n)$, let instrs$(P) \subset \mathcal{I}$ be the set of all instruction identifiers that appear in the programs $s_1, \ldots, s_n$).*

4.5.3 Sound Unrollings

We have not formalized our unrolling algorithm at this point. This section provides a suggested starting point for characterizing soundness of unrollings. However, there remain some open questions about how to unroll concurrent programs soundly on memory models that allow circular dependencies (Section 4.5.4).

To clarify how the unrolled program relates to the original program, we introduce the concept of a *sound unrolling.* Roughly speaking, a sound unrolling of a program p is an unrolled program that (1) models a subset of the executions of p faithfully, and (2) models the remaining executions partially, by terminating with a special execution state *pruned*.

We show an example of a sound unrolling in Fig. 4.6. The right hand side shows how we unroll the loop two times, and replace the remaining iterations with a “throw pruned” statement. This is a sound unrolling; if the third loop iteration is unreachable, we know that unrolling the loop twice is sufficient. If it is reachable, then there exists an execution that terminates with *pruned* state.

Sound unrollings are useful for verification because we can start with heavily pruned program, and gradually relax the pruning as we discover which parts of the program are reachable. If a reachability analysis can prove that a sound unrolled program never terminates with the state *pruned*, we know that the unrolled program is equivalent to the original program and have thus successfully reduced the potentially unbounded verification problem for p to a bounded verification problem for p'.

We now define this idea more formally. First, we need to define prefixes and isomorphy of memory traces.

```
loop : {
  c := (lessthan i j);
  if (c) throw(break loop);
  i := (subtract i one);
  throw(continue loop)
}
```

```
loop : {
  c := (lessthan i j);
  if (c) throw(break loop);
  i := (subtract i one);
  c := (lessthan i j);
  if (c) throw(break loop);
  i := (subtract i one);
  c := (lessthan i j);
  if (c) throw(break loop);
  if (true) throw(pruned)
}
```

Figure 4.6: Unrolling Example: the unrolled LSL program on the right is a sound unrolling of the program on the left.

Definition 24 *A local memory trace* $\boldsymbol{t} = (I, \prec, adr, val)$ *is a* prefix *of the memory trace* $\boldsymbol{t}' = (I', \prec', adr', val')$ *(written* $\boldsymbol{t} \sqsubseteq \boldsymbol{t}'$*) if there exists a function* $\varphi : I \rightarrow I'$ *such that all of the following hold:*

- φ *is injective*
- *for all* $i \in I$ *and* $j \in (I \ \setminus \varphi(I))$*:* $\varphi(i) \prec' j$
- *for all* $i \in I : cmd(i) = cmd(\varphi(i))$
- *for all* $i, j \in I$*:* $i \prec j \Leftrightarrow \varphi(i) \prec' \varphi(j)$
- *for all* $i \in I$*:* $adr'(\varphi(i)) = adr(i)$
- *for all* $i \in I$*:* $val'(\varphi(i)) = val(i)$

Definition 25 *Two local memory traces* $\boldsymbol{t}$ *and* $\boldsymbol{t}'$ *are* isomorphic *(written* $\boldsymbol{t} \cong \boldsymbol{t}'$*) if* $\boldsymbol{t} \sqsubseteq \boldsymbol{t}'$ *and* $\boldsymbol{t}' \sqsubseteq \boldsymbol{t}$*.*

The following definition captures the requirement on unrollings.

Definition 26 *An unrolled program* s_u *is a sound unrolling for the program* $p = (s, D)$ *if both of the following conditions are satisfied:*

- *if* $\Gamma, \boldsymbol{t}, s_u \Downarrow \Gamma', \boldsymbol{t}', e$ *and* $e \neq$ *pruned, then* $\Gamma, \boldsymbol{t}, s \Downarrow_D \Gamma', \boldsymbol{t}', e$.
- *if* $\Gamma, \boldsymbol{t}, s \Downarrow_D \Gamma', \boldsymbol{t}', e$, *then one of the following must be true:*

  1. *There exists a local memory trace* $\boldsymbol{t}'' \cong \boldsymbol{t}'$ *such that* $\Gamma, \boldsymbol{t}, s_u \Downarrow \Gamma', \boldsymbol{t}'', e$.
  2. *There exists a local memory trace* $\boldsymbol{t}'' \sqsubseteq \boldsymbol{t}'$ *and a register map* Γ'' *such that* $\Gamma, \boldsymbol{t}, s_u \Downarrow \Gamma'', \boldsymbol{t}'', pruned$.

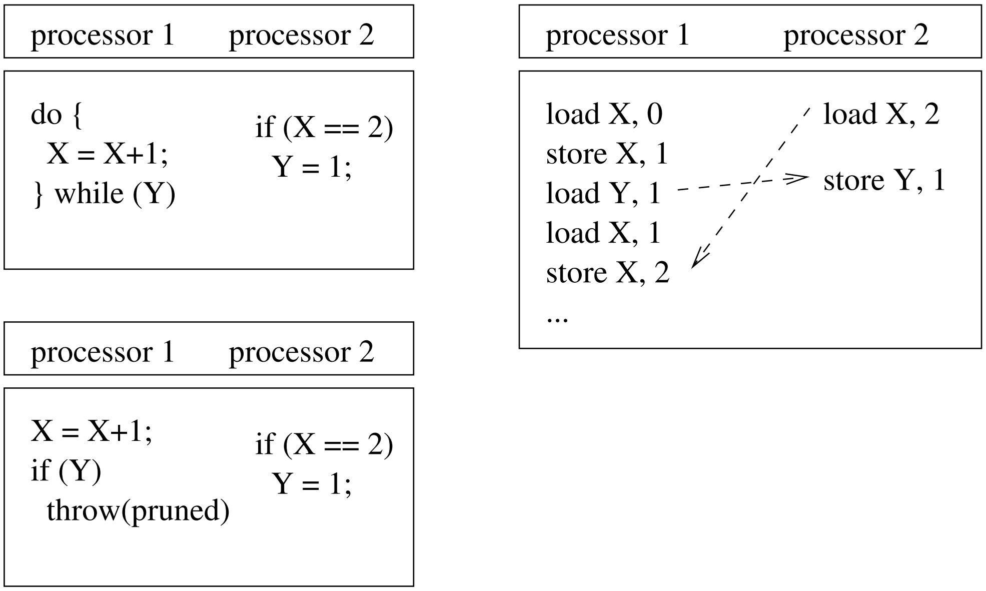
4.5.4 Circular Dependencies

While the definition of sound unrolling above seems reasonable when considering uniprocessor executions, some problems may arise on multiprocessors with pathological memory models that allow circular dependencies. Fig. 4.7 illustrates what we mean, by showing a relaxed trace of a concurrent program where the naive unrolling described in the previous section does not work.

We have neither investigated nor formalized this problem yet and leave it as future work to fix the corresponding issues (for example, by specifying restrictions on the memory model under which the unrolling is sound). None of the hardware memory models we studied would allow such a behavior. However, in the presence of speculative stores, such executions are possible and may need to be considered. Parts of the Java Memory Model [50] address similar problems, motivated by security concerns.

4.6 Extensions

We now define a few extensions to the basic LSL syntax and semantics that proved useful for our CheckFence implementation. We only briefly mention the aspects of the formalization here, and return to a discussion of how we use them in Chapter 7.



```
processor 1      processor 2

do {
  X = X+1;          if (X == 2)
} while (Y)           Y = 1;
```

```
processor 1      processor 2

X = X+1;            if (X == 2)
if (Y)                Y = 1;
  throw(pruned)
```

```
processor 1          processor 2

load X, 0            load X, 2
store X, 1
load Y, 1            store Y, 1
load X, 1
store X, 2
...
```

Figure 4.7: Informal Example: the concurrent program on the top left may produce the global memory trace on the right on a pathological memory model that allows circular dependencies. However, the naive unrolling on the bottom left does neither produce the trace on the right, nor is there an execution that throws *pruned*.

4.6.1 Nondeterministic Choice

Under certain circumstances, it is convenient to allow explicit nondeterminism in programs. Usually, this case arises if we wish to replace some system component with a nondeterministic abstraction. For example, instead of modeling a particular memory allocator, we can abstract it using nondeterministic choice. We return to this example in Section 7.2.4. Here, we simply give a technical description of how we support nondeterministic choice.

First, we add a new statement called choose that assigns a nondeterministically chosen value to the specified target register:

$$r := \textbf{choose}\ a \dots b \qquad\qquad a, b \in \mathbb{Z} \text{ and } a \leq b$$

The bounds a, b must be fixed numbers (they are not registers). Then, we add the following (nondeterministic) inference rule to the trace semantics:

$$\frac{v \in \mathit{Val} \qquad a \leq v \qquad v \leq b}{\Gamma, \boldsymbol{t}, r := \textbf{choose}\ a \dots b \ \Downarrow\ \Gamma[r \mapsto v], \boldsymbol{t}, \mathit{ok}} \quad (\textsc{Assign-choose})$$

4.6.2 Atomic Blocks

To model synchronization operations, we support *atomic blocks.* On the level of LSL programs, atomic blocks are statements with the following syntax:

$$\textbf{atomic}\ s$$

Atomic blocks may be nested; however, all but the outermost block are redundant. The semantic meaning of atomic blocks is not local to a thread (it involves the particulars of the memory model) and is described in Section 3.4.1.

4.6.3 Tagged Unions

During our implementation of the CheckFence tool, we found that it is desirable to add some limited form of user-defined types to LSL. Such types are useful in supporting packed words, or in implementing extended runtime type checking. Our tagged unions are not meant for supporting C unions and bear no similarity; they are more similar to tagged unions as used by OCaml.

We found that the following "simple-minded" version of tagged unions provides a reasonable compromise between expressive power and implementability:

- A program may declare a tagged union type using the following syntax (where $k \geq 0$ and $\mathit{tag}, \mathit{field}_1, \ldots, \mathit{field}_k$ are identifiers):

  $$\textbf{composite}\ \mathit{tag}\ (\mathit{field}_1,\ \ldots,\ \mathit{field}_k)$$

- For each tagged union declaration, we enlarge the set of syntactic values *Val* to include values of the form $\mathit{tag}(v_1, \ldots, v_k)$ where the v_i are regular (not composite) syntactic values.

- For each tagged union declaration, we add the following basic functions to *Fun*:

  - The k-ary function *make_tag* is defined on all k-tuples $(v_1, \ldots, v_k)$ such that each v_i is a non-composite value. The function produces the value $\mathit{tag}(v_1, \ldots, v_k)$.

  - For each $i \in \{1, \ldots, k\}$, the unary function $\mathit{get\_tag\_field}_i$ is defined on values $\mathit{tag}(v_1, \ldots, v_k)$ and returns the value v_i.

  - For each $i \in \{1, \ldots, k\}$, the binary function $\mathit{set\_tag\_field}_i$ is defined on pairs $(\mathit{tag}(v_1, \ldots, v_k), v)$ where v is a non-composite value, and it returns the value $\mathit{tag}(v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_k)$.

  Note that we do not need to provide a function corresponding to the `case` expression in OCaml, because C programs can not query the runtime type of a value.

We describe how we use tagged unions to support packed words in Section 7.2.5.

Chapter 5

Encoding

In this chapter, we describe how to construct, for a given unrolled concurrent program $P = (s_1, \ldots, s_n)$ and memory model Y, a propositional formula $\Phi_{P,Y}$ over a set of variables $V_{P,Y}$ such that each solution of $\Phi_{P,Y}$ corresponds to a concurrent execution of P on Y and vice versa:

$$\left[\begin{array}{c}\text{Concurrent Executions}\\ \boldsymbol{T} \in E(P,Y)\end{array}\right] \leftarrow \text{(correspond to)} \rightarrow \left[\begin{array}{c}\text{Valuations } \nu \text{ of } V_{P,Y}\\ \text{such that } [\![\Phi_{P,Y}]\!]^{\nu} = \textit{true}\end{array}\right]$$

This encoding lays the technical foundation for our verification method. In Chapter 6, we will show how to utilize $\Phi_{P,Y}$ to check the consistency of concurrent data type implementations, by expressing the correctness of executions $\boldsymbol{T}$ in such a way that we can carry it over to valuations ν under the above correspondence and then use a SAT solver to decide whether all executions are correct.

This chapter describes how to construct $\Phi = \Phi_{P,Y}$ (we assume fixed P, Y and thus drop the subscript) and proves that Φ indeed captures the semantics of the program P and memory model Y. We start by defining the vocabulary, the variables, and the correspondence between the variables and the executions:

- In Section 5.1, we describe the vocabulary we use for constructing the formula Φ. We give a list of the required set symbols $\mathcal{S}$ and typed function symbols $\mathcal{F}$, and define the set of variable symbols $V_{P,Y} \subset \mathcal{V}$ that we need to capture the particulars of executions of P on Y. Note that because we are operating on an unrolled program that produces a bounded number of instructions, a finite number of variables is sufficient.

- In Section 5.2, we show how valuations and executions correspond, by defining two maps (1) for each global trace $\boldsymbol{T} \in E(P, Y)$ a valuation $\nu(\boldsymbol{T})$, and (2) for each valuation ν a structure $\boldsymbol{T}(\nu)$.

Next, we break down the formula Φ into individual components

$$\Phi = \Psi \wedge \Theta \wedge \bigwedge_{k=1}^{n} \Pi_k$$

that capture the individual requirements on concurrent executions (as defined in Section 2.1.6):

- A concurrent execution is a global trace. The formula Ψ captures the corresponding constraints. We show how to construct Ψ in Section 5.3.

- The execution must be allowed by the memory model Y. The formula Θ encodes the corresponding constraints. We show how to construct Θ in Section 5.4.

- For each thread $k \in \{1, \ldots, n\}$, the execution must be consistent with the semantics of the program s_k. The formulae $\Pi_1, \ldots, \Pi_n$ capture the corresponding constraints. In Section 5.5, we show how we encode each Π_k. We introduce symbolic local traces and give a pseudocode algorithm for the encoding. We then prove that the encoding algorithm is correct. Because it is quite voluminous, the core of the proof is broken out into Appendix A.

Finally, in Section 5.6, we show that Φ is correct, in the following sense: for all valuations ν that satisfy Φ, $\boldsymbol{T}(\nu)$ is in $E(P, Y)$, and conversely, for all executions $\boldsymbol{T} \in E(P, Y)$, the valuation $\nu(\boldsymbol{T})$ satisfies Φ.

To simplify the notation throughout this chapter, we assume that we are working with a fixed unrolled concurrent program $P = (s_1, \ldots, s_n)$ (as defined in Section 4.5.2) and memory model $Y = (\mathcal{R}, \phi_1, \ldots, \phi_k)$ (as defined in Section 3.3.1). Moreover, we make the following definitions:

- $I_P \subset \mathcal{I}$ is the (finite) set of all instruction identifiers that occur in P.
- $\mathit{proc}_P : I_P \rightarrow \{1, \ldots, n\}$ is the function that maps an instruction i onto the number of the thread within which it occurs.
- $\prec_P$ is the partial order on I_P such that $i \prec_P j$ if $\mathit{proc}_P(i) = \mathit{proc}_P(j)$ and i precedes j in s_k (lexically).

5.1 Vocabulary for Formulae

We construct our formulae over an interpreted vocabulary which defines set symbols $\mathcal{S}$, typed function symbols $\mathcal{F}$ and typed variable symbols $\mathcal{V}$, as well as an interpretation of the symbols (to review the logic framework, refer to Section 2.2).

Our formula Φ (as well as its subformulae Ψ, Θ, and Π_k) are formed over the interpreted vocabulary $\sigma = (\mathcal{S}, \mathcal{F}, \mathcal{V})$ using a finite set of variables $V_{P,Y} \subset \mathcal{V}$. The variables in $V_{P,Y}$ represent the particulars of the execution (what instructions are executed, the values they assume, etc.). We now define these components in more detail.

5.1.1 Set Symbols

We let $\mathcal{S}$ contain the following set symbols

$$\{\mathit{instr}, \mathit{value}, \mathit{status}\} \subset \mathcal{S},$$

and we define the following interpretation

- $[\![\mathit{instr}]\!] = I_P$ is the set of instruction identifiers that appear in the program P.
- $[\![\mathit{value}]\!] = \mathit{Val}$ is the set of syntactic LSL values.
- $[\![\mathit{status}]\!] = \mathit{Exs}$ is the set of execution states.

The sets *Val* and *Exs* are defined in Fig. 4.5 in Section 4.5.2.

5.1.2 Function Symbols

We let $\mathcal{F}$ contain the following function symbols which represent the basic unary and binary functions used by LSL:

$$\begin{aligned}\{\mathit{identity}, \mathit{bitnot}, \mathit{minus}, \mathit{pointer}, \mathit{getbase}, \mathit{getoffset}\} &\in \mathcal{F}^{\mathit{value}\to\mathit{value}}\\ \{\mathit{bitand}, \mathit{bitor}, \mathit{bitxor}, \mathit{add}, \mathit{subtract}, \mathit{multiply}, \mathit{div}, \mathit{mod}, &\\ \mathit{equals}, \mathit{notequals}, \mathit{lessthan}, \mathit{lesseq}, \mathit{shiftleft}, \mathit{shiftright}, \mathit{offset}\} &\in \mathcal{F}^{\mathit{value}\times\mathit{value}\to\mathit{value}}\end{aligned}$$

We interpret these functions according to how they were defined in Section 4.2.8. Furthermore, we let $\mathcal{F}$ contain the constant functions

$$\begin{aligned}\{v \mid v \in \mathit{Val}\} &\subset \mathcal{F}^{\mathit{value}}\\ \{i \mid i \in I_P\} &\subset \mathcal{F}^{\mathit{instr}}\end{aligned}$$

with interpretation $[\![v]\!] = v$ and $[\![i]\!] = i$, respectively.

5.1.3 Variable Symbols

We let $V_{P,Y}$ be the disjoint union of the following variable sets:

$$V_{P,Y} = V_{\mathrm{D}} \cup V_{\mathrm{A}} \cup V_{\mathrm{G}} \cup V_{\mathrm{U}} \cup V_{\mathrm{S}} \cup V_M,$$

where the individual variable sets are defined as follows:

- We let $V_{\mathrm{D}} = \{D_i \mid i \in \mathit{accesses}(I_P)\} \subset \mathcal{V}^{\mathit{value}}$ be a set of first-order variable symbols that represent values loaded or stored by instructions. The type of these variables is *value*.

- We let $V_{\mathrm{A}} = \{A_i \mid i \in \mathit{accesses}(I_P)\} \subset \mathcal{V}^{\mathit{value}}$ be a set of first-order variable symbols that represent the address of a memory access. The type of these variables is *value*.

- We let $V_{\mathrm{G}} = \{G_i \mid i \in I_P\} \subset \mathcal{V}^{\mathit{bool}}$ be a set of first-order variable symbols used to represent whether the instruction i is executed or not. The type of these variables is *bool*.

- We let $V_{\mathrm{U}} = \{U_k \mid 1 \leq k \leq n\} \subset \mathcal{V}^{\mathit{status}}$ be a set of first-order variable symbols used to represent the execution state with which each thread terminates (where n is the number of threads in P). The type of these variables is *status*.

- We let $V_{\mathrm{S}} = \{S_{ij} \mid i \in \mathit{loads}(I_P) \wedge j \in \mathit{stores}(I_P)\} \subset \mathcal{V}^{\mathit{bool}}$ be a set of first-order variable symbols such that S_{ij} represents whether the seed of load i is the store j. The type of these variables is *bool*.

- We let V_M be a set of first-order variables that represent properties of the trace related to the memory model $Y = (\mathcal{R}, \phi_1, \ldots, \phi_k)$. To simplify the notation, let $|R|$ be the arity of each variable $R \in \mathcal{R}$. Then we define the variables

  $$V_M = \{M^R_{i_1 \ldots i_{|R|}} \mid R \in \mathcal{R} \text{ and } i_1, \ldots, i_{|R|} \in I_P\} \subset \mathcal{V}^{\mathit{bool}}$$

  to be a set of first-order boolean variables. The role of these variables is to represent the relational variables $\mathcal{R}$ used by the axiomatic memory model specification Y.

  For example, for sequential consistency, $\mathcal{R} = \{\mathit{memorder}\}$ and we define $V_M = \{M^{\mathit{memorder}}_{ij} \mid i, j \in I_P\}$. The variable $M^{\mathit{memorder}}_{ij}$ then represents $(i <_M j)$.

5.2 Valuations and Executions

To describe the correspondence between valuations and executions, we first define the execution $\boldsymbol{T}(\nu)$ that corresponds to a given valuation ν.

Definition 27 *For a valuation ν that assigns values to all variables in σ, define*

$$\boldsymbol{T}(\nu) = (I_\nu, \prec_\nu, proc_\nu, adr_\nu, val_\nu, seed_\nu, stat_\nu),$$

where

- $I_\nu = \{i \in I_P \mid [\![G_i]\!]^\nu = true\}$
- $proc_\nu = proc|_{I_\nu}$
- $\prec_\nu = \prec_P|_{I_\nu \times I_\nu}$
- *$adr_\nu : I_\nu \to Val$ is the function that maps $i \mapsto [\![A_i]\!]^\nu$*
- *$val_\nu : I_\nu \to Val$ is the function that maps $i \mapsto [\![D_i]\!]^\nu$*
- *$seed_\nu : I_\nu \rightharpoonup I_\nu$ is the partial function such that (1) $i \in \operatorname{dom} seed_\nu$ iff there is exactly one $j \in I_\nu$ such that $[\![S_{ij}]\!]^\nu =$ true, and (2) $seed_\nu(i) = j$ for the j such that $[\![S_{ij}]\!]^\nu =$ true.*
- *$stat_\nu : \mathbb{N} \to Exs$ is the function that maps $k \mapsto [\![U_k]\!]^\nu$*

We now proceed to the definition of the valuation $\nu(\boldsymbol{T})$ that corresponds to a given execution $\boldsymbol{T}$. The situation is slightly more complicated than before, because there is in general more than one valuation to describe the same execution.

Specifically, the memory model axioms state that for any valid execution, there exists a valuation of the memory model relation variables $\mathcal{R}$ that satisfies the axioms. However, there could be any number of such justifying valuations for the same global trace. For example, looking at sequential consistency, we see that the same global trace can be the result of more than one interleaving, because the global trace only captures the seeding store for each load, but not how accesses are globally interleaved.

To "determinize" the function $\nu(\boldsymbol{T})$, we pick an arbitrary but fixed valuation ν among all the possible ones; we achieve this by assuming some arbitrary total order over all valuations, and picking the minimal one.

Definition 28 *For an execution*

$$\boldsymbol{T} \in E(P, Y) \qquad \boldsymbol{T} = (I_{\boldsymbol{T}}, \prec_{\boldsymbol{T}}, proc_{\boldsymbol{T}}, adr_{\boldsymbol{T}}, val_{\boldsymbol{T}}, seed_{\boldsymbol{T}}, stat_{\boldsymbol{T}})$$

define the valuation $\nu(\boldsymbol{T})$ as follows

- *For $i \in accesses(I_P)$, we let $\nu(\boldsymbol{T})(D_i) = \begin{cases} val_{\boldsymbol{T}}(i) & \text{if } i \in I_{\boldsymbol{T}} \\ \perp & \text{otherwise} \end{cases}$*
- *For $i \in accesses(I_P)$, we let $\nu(\boldsymbol{T})(A_i) = \begin{cases} adr_{\boldsymbol{T}}(i) & \text{if } i \in I_{\boldsymbol{T}} \\ \perp & \text{otherwise} \end{cases}$*
- *For $i \in I_P$, we let $\nu(\boldsymbol{T})(G_i) = \begin{cases} true & \text{if } i \in I_{\boldsymbol{T}} \\ false & \text{otherwise} \end{cases}$*
- *For $k \in \{1, \ldots, n\}$, we let $\nu(\boldsymbol{T})(U_k) = stat_{\boldsymbol{T}}(k)$*
- *For $i \in loads(I_P)$ and $j \in stores(I_P)$, we let*
  $\nu(\boldsymbol{T})(S_{ij}) = \begin{cases} true & \text{if } i, j \in I_{\boldsymbol{T}} \text{ and } seed_{\boldsymbol{T}}(i) = j \\ false & \text{otherwise} \end{cases}$
- *Because $\boldsymbol{T} \in E(P, Y)$ implies $\boldsymbol{T} \in E_G(Y)$, we know (by Def. 16 in Section 3.3.1) that there exists a valuation ν of the variables $\mathcal{R}$ such that for all axioms ϕ of Y, $[\![\phi]\!]^{\nu}_{\boldsymbol{T}} =$ true. Now, pick a minimal such ν (according to some arbitrary, fixed order on valuations), and extend it by defining for each second-order variable $R \in \mathcal{R}$ and $i_1, \ldots, i_{|R|} \in I_P$*

$$\nu(\boldsymbol{T})(M^R_{i_1 \ldots i_{|R|}}) = [\![R]\!]^{\nu}(i_1, \ldots, i_{|R|}).$$

5.3 The Formula Ψ

We start by showing how we construct the formula Ψ, which captures the properties of global traces in the following sense:

- for each global trace $\boldsymbol{T}$, $[\![\Psi]\!]^{\nu(\boldsymbol{T})} = \mathit{true}$

- $\boldsymbol{T}(\nu)$ is a global trace for each valuation ν such that $[\![\Psi]\!]^{\nu} = \mathit{true}$

To simplify the notation, let $L = \mathit{loads}(I_P)$ and $S = \mathit{stores}(I_P)$. Then we define:

$$\begin{aligned}
\Psi \quad = \quad & \bigwedge_{l \in L} \bigwedge_{s \in S} (S_{ls} \;\Rightarrow\; G_l \wedge G_s \wedge (D_l = D_s) \wedge (A_l = A_s)) && (5.1)\\
& \wedge \bigwedge_{l \in L} \left(\left(G_l \wedge \bigwedge_{s \in S} \neg S_{ls} \right) \Rightarrow (D_l = \bot) \right) && (5.2)\\
& \wedge \bigwedge_{l \in L} \; \bigwedge_{s \in S} \; \bigwedge_{s' \in (S \setminus \{s\})} \neg (S_{ls} \wedge S_{ls'}) && (5.3)
\end{aligned}$$

Clearly, the formula reflects the conditions in the definition of global traces (Section 2.1.4) by requiring (1) that the variables S_{ls} encode a valid partial function *seed* from the set of executed loads to the set of executed stores, and (2) that the address and data values are consistent with that *seed* function.

The following two lemmas capture the properties of Ψ more formally. We use them in our final correctness theorem.

Lemma 29 *For all valuations ν such that* $\operatorname{dom} \nu \supset V_A \cup V_D \cup V_G \cup V_S$ *and* $[\![\Psi]\!]^{\nu} =$ *true, the structure $\boldsymbol{T}(\nu)$ is a global trace.*

Lemma 30 *For any execution $\boldsymbol{T} \in E(P, Y)$ and for all $k \in \{1, \ldots, n\}$, we have $[\![\Psi]\!]^{\nu(\boldsymbol{T})} = \mathit{true}$.*

The proofs are conceptually easy, but a little tedious. We include them for completeness; the reader may skim them and proceed to the next section.

PROOF. (Lemma 29) We have to show the items listed in Def. 2 (Section 2.1.4), assuming I_ν, $\prec_\nu$, proc_ν, adr_ν, val_ν, seed_ν, stat_ν are as defined in Def. 27) above. They are mostly straightforward, and we show only the slightly more interesting ones here:

- Show [seed_ν is a partial function $\mathit{loads}(I_\nu) \rightharpoonup \mathit{stores}(I_\nu)$].

  By construction, seed_ν always satisfies this.

- Show [if $seed_\nu(l) = s$, then $adr_\nu(l) = adr_\nu(s)$ and $val_\nu(l) = val_\nu(s)$].

  If $seed(l) = s$, then $[\![S_{ls}]\!]^\nu = true$. Therefore (because (5.1) evaluates to true) we know (1) $[\![G_l]\!]^\nu = [\![G_s]\!]^\nu = true$ and therefore s, l are elements of both $\mathrm{dom}\, adr_\nu$ and $\mathrm{dom}\, val_\nu$, and (2) $[\![D_l]\!]^\nu = [\![D_s]\!]^\nu$ and $[\![A_l]\!]^\nu = [\![A_s]\!]^\nu$, and thus the claim follows.

- Show [if $l \in (loads(I_\nu) \setminus \mathrm{dom}\, seed_\nu)$, then $val_\nu(l) = \bot$].

  $l \in loads(I_\nu)$ implies $[\![G_l]\!]^\nu = true$. Also, the fact that $l \notin \mathrm{dom}\, seed_\nu$ implies that there is no s such that $[\![S_{ls}]\!]^\nu = true$ (the only other possibility would be that $[\![S_{ls}]\!]^\nu = true$ for more than one j, which is ruled out by (5.3)). Therefore, (5.2) guarantees that $[\![D_l]\!]^\nu = \bot$ from which the claim follows.

□

Proof. (Lemma 30) Let $\boldsymbol{T} = (I_{\boldsymbol{T}}, \prec_{\boldsymbol{T}}, proc_{\boldsymbol{T}}, adr_{\boldsymbol{T}}, val_{\boldsymbol{T}}, seed_{\boldsymbol{T}}, stat_{\boldsymbol{T}})$ and $\nu = \nu(\boldsymbol{T})$ as defined in Def. 28. Then $I_{\boldsymbol{T}} \subset I_P$ because the executed instructions must be a subset of the instructions appearing in the concurrent program. Now, we know each line of Ψ must be satisfied by ν:

- Show (5.1). If $[\![S_{ls}]\!]^\nu = true$, then $seed_{\boldsymbol{T}}(l) = s$, which implies (1) $l, s \in I_{\boldsymbol{T}}$ and thus $[\![G_l]\!]^\nu = [\![G_s]\!]^\nu = true$, and (2) $adr_{\boldsymbol{T}}(l) = adr_{\boldsymbol{T}}(s)$ and thus $[\![A_l = A_s]\!]^\nu = true$, and (3) $val_{\boldsymbol{T}}(l) = val_{\boldsymbol{T}}(s)$ and thus $[\![D_l = D_s]\!]^\nu = true$. This implies that (5.1) is satisfied.

- Show (5.2). We consider each $l \in loads(I_P)$ of the outer conjunction separately. If $[\![G_l]\!]^\nu = false$ or $[\![D_l]\!]^\nu = \bot$, we are done for this l. Otherwise, we know $l \in I_{\boldsymbol{T}}$ and $val_{\boldsymbol{T}}(l) \neq \bot$. Because $\boldsymbol{T}$ is a global trace, this implies that there exists a $s \in stores(I_{\boldsymbol{T}})$ such that $seed_{\boldsymbol{T}}(l) = s$. Then $[\![S_{ls}]\!]^\nu = true$ and the clause is satisfied.

- Show (5.3). By definition of global traces, $seed_{\boldsymbol{T}}$ is a partial function and there can therefore be at most one s for any l such that $seed_{\boldsymbol{T}}(l) = s$. This implies that there is at most one s such that $[\![S_{ls}]\!]^\nu = \mathit{true}$ and the clause is thus always satisfied.

□

5.4 Encoding the Memory Model

In this section we describe how to encode the memory model constraints given by the axiomatic specification Y as a propositional formula Θ.

The specification $Y = (\mathcal{R}, \phi_1, \ldots, \phi_k)$ already contains formulae (the "axioms" $\phi_1, \ldots, \phi_k$). However, these axioms still need to be converted to propositional format. To do so, we perform the following steps:

- Eliminate quantifiers. Because we use a restricted form of quantifiers, we know each quantifier ranges over a subset of instructions. We can therefore replace quantifiers by finite disjunctions or conjunctions.
- Eliminate the second-order variables in $\mathcal{R}$. By definition (Def. 13 in Section 3.3.1), the variables in $\mathcal{R}$ are of the type

  $$\underbrace{\mathit{instr} \times \cdots \times \mathit{instr}}_{k} \to \mathit{bool}$$

  for some $k \geq 0$. We can thus represent each such variable by $|I_P|^k$ boolean first-order variables. Each first-order variable represents the value of the second order value for a specific instruction tuple.

We now describe this procedure with a little more detail.

5.4.1 Encoding procedure

For all axioms, we perform the following transformations.

1. First, we eliminate all quantifiers. Because of the syntax we use for axiomatic memory model specifications (see Fig. 3.7 in Section 3.3.1), the quantifiers occur in the form $(\forall\, p\; X : \phi)$ and $(\exists\, p\; X : \phi)$ where $p \in \{\mathit{load}, \mathit{store}, \mathit{fence}, \mathit{access}\}$ and $X \in \mathcal{V}^{\mathit{instr}}$. We use the following notations: (1) for a set of instructions $I \subset \mathcal{I}$ we let $p(I) \subset I$ be the subset described by p, (2) we let $[i/X]\phi$ denote the formula ϕ where each free occurrence of X has been replaced by i. Then we can describe the transformation step by the following rules:

$$\begin{array}{rcl} (\forall\, p\; X : \phi) & \mapsto & \bigwedge_{i \in p(I_P)} (\neg G_i) \vee [i/X]\phi \\ (\exists\, p\; X : \phi) & \mapsto & \bigvee_{i \in p(I_P)} G_i \wedge [i/X]\phi \end{array}$$

   As a result, we obtain transformed formulae $\phi'_1, \dots, \phi'_k$.

2. Now, for each occurrence of each relation variable $R \in \mathcal{R}$, we perform the following replacement

$$R(i_1, \dots, i_{|R|}) \quad \mapsto \quad M^R_{i_1 \dots i_{|R|}}$$

   (note that the i_k are constants because we replaced all occurrences of instruction variables by constants during the previous step). As a result, we obtain transformed formulae $\phi''_1, \dots, \phi''_k$.

3. Finally, we replace all occurrences of the following function symbols:

$$\begin{array}{rcl} \mathit{load}(i) & \mapsto & \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) = \mathit{load} \\ \mathit{false} & \text{otherwise} \end{cases} \\ \mathit{store}(i) & \mapsto & \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) = \mathit{store} \\ \mathit{false} & \text{otherwise} \end{cases} \\ \mathit{fence}(i) & \mapsto & \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) = \mathit{fence} \\ \mathit{false} & \text{otherwise} \end{cases} \end{array}$$

$$\begin{aligned}
\mathit{access}(i) &\mapsto \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) \in \{\mathit{load}, \mathit{store}\} \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathit{progorder}(i,j) &\mapsto \begin{cases} \mathit{true} & \text{if } i \prec_P j \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathit{aliased}(i,j) &\mapsto \begin{cases} (A_i = A_j) & \text{if } i,j \in \mathit{accesses}(I_P) \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathit{seed}(i,j) &\mapsto \begin{cases} S_{ij} & \text{if } i \in \mathit{loads}(I_P) \\ & \quad \text{and } j \in \mathit{stores}(I_P) \\ \mathit{false} & \text{otherwise} \end{cases}
\end{aligned}$$

As a result, we obtain transformed formulae $\phi_1''', \ldots, \phi_k'''$.

5.4.2 The Formula Θ

After performing the transformation steps described in the previous section, our axioms are now propositional formulae $\phi_1''', \ldots, \phi_k'''$ with variables in $V_{\mathrm{A}} \cup V_{\mathrm{G}} \cup V_{\mathrm{S}} \cup V_M$. We can then simply take the conjunction to obtain the desired formula

$$\Theta = \phi_1''' \wedge \cdots \wedge \phi_k'''.$$

The following two lemmas capture the properties of Θ more formally. We use them in our final correctness theorem.

Lemma 31 *If ν is a valuation of σ such that* $\operatorname{dom} \nu = V_{P,Y}$ *and* $[\![\Theta]\!]^\nu = \mathit{true}$ *and* $[\![\Psi]\!]^\nu = \mathit{true}$*, then* $\boldsymbol{T}(\nu) \in E_G(Y)$*.*

Lemma 32 $[\![\Theta]\!]^{\nu(\boldsymbol{T})} = \mathit{true}$ *for any execution* $\boldsymbol{T} \in E(P, Y)$*.*

Proof. (Lemma 31). First, let ν' be the valuation that extends ν to the relation variables $R \in \mathcal{R}$ by defining $\nu'(R)(i_1, \ldots, i_{|R|}) = [\![M^R_{i_1 \ldots i_{|R|}}]\!]^\nu$. Now we want to show that $[\![\phi_k]\!]^{\nu'}_{\boldsymbol{T}(\nu)} = \mathit{true}$ for all k, because that then implies $\boldsymbol{T}(\nu) \in E_G(Y)$ as desired. Assume we are given some fixed k.

Define the interpretation $[\![.]\!]_\nu$ as follows:

$$
\begin{aligned}
[\![\mathit{instr}]\!]_\nu &= I_P \\
[\![\mathit{load}]\!]_\nu(i) &= \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) = \mathit{load} \\ \mathit{false} & \text{otherwise} \end{cases} \\
[\![\mathit{store}]\!]_\nu(i) &= \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) = \mathit{store} \\ \mathit{false} & \text{otherwise} \end{cases} \\
[\![\mathit{fence}]\!]_\nu(i) &= \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) = \mathit{fence} \\ \mathit{false} & \text{otherwise} \end{cases} \\
[\![\mathit{access}]\!]_\nu(i) &= \begin{cases} \mathit{true} & \text{if } \mathit{cmd}(i) \in \{\mathit{load}, \mathit{store}\} \\ \mathit{false} & \text{otherwise} \end{cases} \\
[\![\mathit{progorder}]\!]_\nu(i,j) &= \begin{cases} \mathit{true} & \text{if } i \prec_P j \\ \mathit{false} & \text{otherwise} \end{cases} \\
[\![\mathit{aliased}]\!]_\nu(i,j) &= \begin{cases} \mathit{true} & \text{if } i,j \in \mathit{accesses}(I_P) \text{ and } [\![A_i]\!]^\nu = [\![A_j]\!]^\nu \\ \mathit{false} & \text{otherwise} \end{cases} \\
[\![\mathit{seed}]\!]_\nu(i,j) &= \begin{cases} \mathit{true} & \text{if } i \in \mathit{loads}(I_P),\ j \in \mathit{stores}(I_P) \text{ and } [\![S_{ij}]\!]^\nu = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Now, looking at transformation step 1, observe that

$$[\![\phi_k]\!]^{\nu'}_{\boldsymbol{T}(\nu)} = [\![\phi'_k]\!]^{\nu'}_{\nu}$$

because

- The conjunctions/disjunctions that we substituted in precisely capture the semantics of the quantification: the quantifier of the left goes over a smaller domain than the conjunction/disjunction on the right, because the latter includes instructions that are not executed. However, for such instructions, the guard evaluates to false and renders the corresponding subformula of the conjunction/disjunction irrelevant.

- On subformulae $u(i)$ or $b(i,j)$ where u, b are unary/binary relations and $[\![G_i]\!]^\nu = [\![G_j]\!]^\nu = \mathit{true}$, the evaluation functions $[\![.]\!]^{\nu'}_{\boldsymbol{T}(\nu)}$ and $[\![.]\!]^{\nu'}_{\nu}$ are the same:
  - For *load*, *store*, *access*, *fence*, *progorder*, and *aliased* it is clear directly from the definitions.
  - For *seed*, we need to use the assumption of the Lemma that states $[\![\Psi]\!]^\nu = \mathit{true}$, which guarantees that $[\![S_{ij}]\!]^{\nu'} = \mathit{true}$ if and only if $\mathit{seed}_{\boldsymbol{T}(\nu)}(i) = j$.
- Therefore, the evaluations of the formula are the same except on subformulae $u(i)$ or $b(i,j)$ where i, j are instruction constants and not both guards evaluate to true. Because the original formula ϕ_k does not contain instruction constants, such subformulae must have been substituted in during the transformation step. This implies that the value to which $u(i)$ or $b(i,j)$ evaluate is irrelevant because the subformula of the conjunction/disjunction that contains $u(i)$ or $b(i,j)$ already evaluates to true/false and has no bearing on the satisfaction of the containing conjunction/disjunction.

Next, looking at transformation step 2, observe that $[\![\phi'_k]\!]^{\nu'}_{\nu} = [\![\phi''_k]\!]^{\nu}_{\nu}$ because we defined ν' purposely so that the substituted variables evaluate to the same value. Looking at transformation step 3, observe that $[\![\phi''_k]\!]^{\nu}_{\nu} = [\![\phi'''_k]\!]^{\nu}$ because we constructed the interpretation $[\![.]\!]_{\nu}$ to match the substitutions performed in this step.

Putting the three observations together, we conclude

$$[\![\phi_k]\!]^{\nu'}_{\boldsymbol{T}(\nu)} = [\![\phi'_k]\!]^{\nu'}_{\nu} = [\![\phi''_k]\!]^{\nu}_{\nu} = [\![\phi'''_k]\!]^{\nu} = \mathit{true}.$$

□

Proof. (Lemma 32). We want to show $[\![\phi'''_k]\!]^{\nu(\boldsymbol{T})} = \mathit{true}$ for all k. Let k be fixed. First, define the interpretation $[\![.]\!]_{\nu(\boldsymbol{T})}$ the same way as we defined $[\![.]\!]_{\nu}$ in the proof of Lemma 31.

By definition of $\nu(\boldsymbol{T})$ we know that $[\![\phi]\!]^{\nu(\boldsymbol{T})}_{\boldsymbol{T}} = \mathit{true}$.

For the same reasons as explained in the proof of Lemma 31, this implies that $[\![\phi']\!]^{\nu(\boldsymbol{T})}_{\nu(\boldsymbol{T})} = \mathit{true}$. In transformation step 2, we substitute terms with identical values under $\nu(\boldsymbol{T})$ (by Definition of $\nu(\boldsymbol{T})$), and therefore we know $[\![\phi'']\!]^{\nu(\boldsymbol{T})}_{\nu(\boldsymbol{T})} = \mathit{true}$. Finally, in transformation step 3, observe that we constructed the interpretation $[\![.]\!]_{\nu(\boldsymbol{T})}$ to match the substitutions performed in this step, so $[\![\phi'''_k]\!]^{\nu(\boldsymbol{T})} = \mathit{true}$ as needed to conclude the proof. □

5.5 Encoding Local Traces

In this section, we describe how to encode the local program semantics of each thread $k \in \{1, \ldots, n\}$ as a formula Π_k such that each solution of Π_k corresponds to a local trace of the unrolled program s_k.

To represent the local program semantics, we first define *symbolic traces*. The purpose of symbolic traces is to concisely represent the highly nondeterministic semantics of unrolled programs (Section4.5.2) by representing values using variables, terms, and formulae.

We now give a rough overview of the structure of this section:

- In Section 5.5.1, we define symbolic traces. Our definition of a symbolic trace $\bar{\boldsymbol{t}}$ is similar to the definition of a local memory traces $\boldsymbol{t}$ (the overline serves as a visual reminder of the symbolic nature of the trace), but it uses variables, terms, and formulae rather than explicit values.

- In Section 5.5.2, we describe how to evaluate a symbolic trace $\bar{\boldsymbol{t}}$ for a given valuation ν of the variables in $\bar{\boldsymbol{t}}$, obtaining a local memory trace $[\![\bar{\boldsymbol{t}}]\!]^{\nu}$.

- Next, we describe our encoding algorithm, both with pseudocode and formal inference rules (Sections 5.5.4 and 5.5.5). Our encoding algorithm takes an unrolled program s and produces a symbolic trace.

- In Section 5.5.6, we show that the encoding is correct, in the sense that the satisfying valuations of the symbolic trace exactly correspond to the executions of the unrolled program. This proof is the key to to our encoding algorithm and its construction heavily influenced our formalization of concurrent executions and memory traces. Because it is quite voluminous, the core of the proof is broken out into Appendix A.

- Finally, in Section 5.5.7 we utilize our encoding algorithm to construct the formula Π_k, which is the primary goal of this section.

5.5.1 Symbolic Traces

Just like local memory traces, symbolic traces consist of a totally ordered set of instructions of a fixed type (load, store, or fence). However, a symbolic trace represents many actual memory traces with varying values. Therefore, a symbolic trace differs from a normal trace as follows:

- The values returned by loads are represented by variables of type *value*, rather than by explicit value.

- The address and data values of each instruction are represented by terms of type *value*, rather than by explicit values. These terms follow the syntax defined in Fig. 2.3 and use the vocabulary defined in Section 5.1.

- Some instructions may not always be executed (depending on the path taken through the program). We express this by defining a *guard* for each instruction; the guard is a formula representing the conditions under which this instruction gets executed.

We show an informal example of a symbolic trace that represents the traces of a program in Fig. 5.1.

Definition 33 *A* symbolic trace *is a tuple* $\bar{\boldsymbol{t}} = (I, \prec, V, \overline{adr}, \overline{val}, \overline{guard})$ *such that*

```
x = [0 1];
r = *x;
if (r != 0)
  r = *r;
*x = r
```

guard	command	address,value
true	*load*	$[0\ 1],\ D_1$
$(D_1 \neq 0)$	*load*	$D_1,\ D_2$
true	*store*	$[0\ 1],\ ((D_1 \neq 0)\ ?\ D_2 : D_1)$

Figure 5.1: Informal Example: the program on the left is represented by the symbolic trace on the right. The symbols D_1, D_2 represent the values loaded by the first and second load, and they appear in the formulae that represent subsequent guards, values, and addresses.

- *I is a set of instructions*
- *$\prec$ is a total order over I*
- *$V \subset \mathcal{V}^{\text{value}}$ is a finite set of first-order variables of type* value.
- *$\overline{adr}$ is a function $I \to \mathcal{T}^{\text{value}}$ that maps each instruction to a term of type* value *over the variables V.*
- *$\overline{val}$ is a function $I \to \mathcal{T}^{\text{value}}$ that maps each instruction to a term of type* value *over the variables V.*
- *$\overline{guard}$ is a function $I \to \mathcal{T}^{bool}$ that maps each instruction to a formula over the variables V.*

Let $\overline{Ltr}$ be the set of all symbolic traces. Let $\overline{\boldsymbol{t}}^0$ be the symbolic trace for which $I = V = \emptyset$.

5.5.2 Evaluating Symbolic Traces

If we assign values to the variables of a symbolic trace, we get a local memory trace. More formally, we extend the evaluation function $[\![.]\!]^\nu$ (defined for formulae in Section 2.2.3) to symbolic traces as follows.

Definition 34 *For a symbolic trace $\overline{\boldsymbol{t}} = (I, \prec, V, \overline{adr}, \overline{val}, \overline{guard})$ and valuation ν such that $V \subset \operatorname{dom} \nu$, define $[\![\overline{\boldsymbol{t}}]\!]^\nu = (I', \prec', adr, val)$ where*

- $I' = \{i \in I \mid [\![\overline{guard}(i)]\!]^\nu = true\}$
- $\prec' = \prec|_{(I' \times I')}$
- $adr(i) = [\![\overline{adr}(i)]\!]^\nu$
- $val(i) = [\![\overline{val}(i)]\!]^\nu$

It follows directly from the definitions that the evaluation $[\![\bar{\boldsymbol{t}}]\!]^\nu$ of a symbolic trace $\bar{\boldsymbol{t}}$ is a local trace.

5.5.3 Appending Instructions

During our encoding algorithm, we build symbolic traces by appending instructions one at a time. The following definition provides a notational shortcut to express this operation.

Definition 35 *For a symbolic trace $\bar{\boldsymbol{t}} = (I, \prec, V, \overline{adr}, \overline{val}, \overline{guard})$ and for an instruction $i \in (\mathcal{I} \setminus I)$, terms $a, v \in \mathcal{T}^{\text{value}}$ and a formula $g \in \mathcal{T}^{bool}$ we use the notation*

$$\boldsymbol{t}.append(i, a, v, g)$$

to denote the symbolic trace $(I', \prec', V', \overline{adr}', \overline{val}', \overline{guard}')$ where

- $I' = I \cup \{i\}$
- $\prec' = \prec \cup (I \times \{i\})$
- $V' = V \cup FV(a) \cup FV(v) \cup FV(g)$
- $\overline{adr}'(x) = \begin{cases} \overline{adr}(x) & \textit{if } x \neq i \\ a & \textit{if } x = i \end{cases}$
- $\overline{val}'(x) = \begin{cases} \overline{val}(x) & \textit{if } x \neq i \\ v & \textit{if } x = i \end{cases}$
- $\overline{guard}'(x) = \begin{cases} \overline{guard}(x) & \textit{if } x \neq i \\ g & \textit{if } x = i \end{cases}$

We now proceed to the description of the encoding algorithm for programs. The algorithm takes as input an unrolled program (as defined in Section 4.5.1), and returns a symbolic trace that represents all executions of the program. We start with a somewhat informal pseudo-code version, and then proceed to a formal presentation of the algorithm based on inference rules. Finally, we formulate and prove correctness of the algorithm.

5.5.4 Pseudocode Algorithm

We show a pseudocode version of the encoding algorithm below. The algorithm performs a forward symbolic simulation on the unrolled LSL program: we process the program in forward direction and record the trace and register values by building up terms and formulae. We track the program state individually for all possible current execution states.

There are three global variables:

- The variable `trace` stores the current symbolic trace. We assume that our implementation provides a suitable abstract data structure to represent symbolic traces as defined in Section 5.5.1, including an `append` operation as specified by Def. 35 in Section 5.5.3.

- The variable `guardmap` is an array that stores for each execution state a formula. The formula `guardmap[e]` captures the conditions under which execution is in state e, expressed as a function of the loaded values.

- The variable `regmap` is an array of arrays; for each execution state, it stores a map from register names to terms. The term `regmap[e][r]` represents the register content of register r if the execution state is e, as a function of the loaded values.

The variables are initialized to their default in the section marked `initially`. We then call the recursive procedure `encode(s)` with the unrolled LSL program s. After the call returns, the variable `trace` contains the resulting symbolic trace.

```
var
  trace : symbolic_trace;
  guardmap : array [Exs] of boolean_formula;
  regmap : array [Exs][Reg] of value_term;

initially {
  trace := empty_trace;
  foreach e in Exs do {
    guardmap[e] := (e = ok) ? true : false;
    foreach r in Reg do {
      regmap[e][r] := ⊥;
    }
  }
}

function encode(s : Stm) {
  match s with
  | (fence$_i$) ->
      trace := trace.append(i, ⊥, ⊥, guardmap[ok]);
  | (r := (f r1 ... rk))  ->
      regmap[ok][r] := f(regmap[ok][r1],...,regmap[ok][rk]);
  | (*r’ :=$_i$ r) ->
      trace := trace.append(i, regmap[ok][r’], regmap[ok][r],
                                                          guardmap[ok]);
  | (r :=$_i$ *r’) ->
      trace := trace.append(i, regmap[ok][r’], D$_i$, guardmap[ok]);
      regmap[ok][r] := D$_i$;
  | (if (r) throw e) ->
      var fla : boolean_formula;
      fla := ¬ (regmap[ok][r] = 0);
      guardmap[ok] := guardmap[ok] ∧ ¬ fla;
      guardmap[e] := guardmap[ok] ∧ fla;
      foreach r in Reg do
        regmap[e][r] := regmap[ok][r];
  | (l : s) ->
      encode(s);
      foreach r in Reg do
        regmap[ok][r] := (guardmap[break  l] ?
                           regmap[break  l][r] : regmap[ok][r]);
      guardmap[ok] := guardmap[ok] ∨ guardmap[break  l];
      guardmap[break  l] := false;
  | (s1 ; s2)  ->
      var gm : array [Exs] of boolean_formula;
      var rm : array [Exs][Reg] of value_term;
      encode(s1);
      gm := guardmap;  // make a copy
      rm := regmap;    // make a copy
      foreach e in Exs do
        if (e ≠ ok)
          guardmap[e] := false;
      encode(s2);
      foreach e in Exs do
        if (e ≠ ok) {
          guardmap[e] := gm[e] ∨ guardmap[e];
          foreach r in Reg do
            regmap[e][r] := (gm[e] ? rm[e][r] : regmap[e][r]);
        }
}
```

The encoding algorithm is deterministic and always terminates. Moreover, because the unrolled program is guaranteed to contain each instruction identifier at most once (Definition 21 in Section 4.5.2), all calls to `trace.append` are valid (that is, use fresh identifiers).

5.5.5 Formal Algorithm

The pseudocode algorithm in the previous section is somewhat informal, and its format is not proof-friendly. To clarify the technical details and prepare for proving correctness, we translate the pseudocode into a (logically equivalent) precise, formal notation that uses inference rules.

Given the combination of state and recursion of the algorithm, we choose a formalization that expresses how the program under translation "drives" the encoding algorithm through encoder states (similar to a word driving a finite automaton). We derive sentences of the form

$$\boxed{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \bar{\boldsymbol{t}}')}$$

with the following interpretation:

> if `encode(s)` is called with state `guardmap`$=\gamma$, `regmap`$=\Delta$ and `trace`$=\bar{\boldsymbol{t}}$,
> it returns with state `guardmap`$=\gamma'$, `regmap`$=\Delta'$ and `trace`$=\bar{\boldsymbol{t}}'$.

First, we need to define guard maps and symbolic register maps formally.

Definition 36 *A* guard map *is a function $\gamma : Exs \rightarrow \mathcal{T}^{bool}$. We use a subscript notation to indicate function application: γ_e denotes $\gamma(e)$. We define the guardmap γ^0 to be the function such that $\gamma^0_{ok} = true$ and $\gamma^0_e = false$ for all $e \neq ok$.*

Definition 37 *A* symbolic register map *is a function $\Delta : Exs \rightarrow (Reg \rightarrow \mathcal{T}^{value})$. We use a subscript notation to indicate function application of the first argument: Δ_e denotes $\Delta(e)$ (which is a function $Reg \rightarrow \mathcal{T}^{bool}$). We define the symbolic register map Δ^0 to be the function that such that $\Delta^0_e(r) = \bot$ for all e and r.*

To keep the inference rules from spilling over page boundaries, we use the following notational shortcuts.

Definition 38 *Given two functions $m, m' : Reg \to \mathcal{T}^{\text{value}}$ and a boolean formula ϕ we write*

$$(\phi \mathbin{?} m : m')$$

to represent the function $Reg \to \mathcal{T}^{\text{value}}$ that maps $r \mapsto (\phi \mathbin{?} m(r) : m'(r))$.

Definition 39 *Given a guard map γ and a syntactic expression expr, we let*

$$\gamma\llbracket e \mapsto \mathit{expr} \rrbracket_{e \neq \mathit{ok}}$$

denote the guardmap that maps ok $\mapsto \gamma_{\mathit{ok}}$ and maps $e \mapsto$ expr for all $e \neq$ ok. We also use the analogous notation $\Delta\llbracket e \mapsto \mathit{expr} \rrbracket_{e \neq \mathit{ok}}$ for symbolic register maps.

With these definitions in place, we can now give the inference rules.

$$\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \bot, \bot, \gamma_{\mathit{ok}})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{\textbf{fence}_i} (\gamma, \Delta, \overline{\boldsymbol{t}}')} \qquad (\text{E}\textsc{Fence})$$

$$\frac{\Delta' = \Delta[\mathit{ok} \mapsto \Delta_{\mathit{ok}}[r \mapsto f(\Delta_{\mathit{ok}}(r_1), \ldots, \Delta_{\mathit{ok}}(r_k))]]}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{r := (f\ r_1\ \ldots\ r_k)} (\gamma, \Delta', \overline{\boldsymbol{t}})} \qquad (\text{E}\textsc{Assign})$$

$$\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{\mathit{ok}}(r'), \Delta_{\mathit{ok}}(r), \gamma_{\mathit{ok}})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{{*}r' :=_i r} (\gamma, \Delta, \overline{\boldsymbol{t}}')} \qquad (\text{E}\textsc{Store})$$

$$\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{\mathit{ok}}(r'), D_i, \gamma_{\mathit{ok}})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{r :=_i {*}r'} (\gamma, \Delta[\mathit{ok} \mapsto \Delta_{\mathit{ok}}[r \mapsto D_i]], \overline{\boldsymbol{t}}')} \qquad (\text{E}\textsc{Load})$$

$$\frac{\gamma' = \gamma[\mathit{ok} \mapsto \gamma_{\mathit{ok}} \wedge \neg\phi][e \mapsto \gamma_{\mathit{ok}} \wedge \phi] \qquad \phi = \neg(\Delta_{\mathit{ok}}(r) = 0)}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{\textbf{if}\,(r)\,\textbf{throw}(e)} (\gamma', \Delta[e \mapsto \Delta_{\mathit{ok}}], \overline{\boldsymbol{t}})} \ (\text{E}\textsc{CondThrow})$$

$$\frac{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \overline{\boldsymbol{t}}') \qquad \gamma'' = \gamma'[\mathit{ok} \mapsto \gamma'_{\mathit{ok}} \vee \gamma'_{\mathit{break}\,l}][\mathit{break}\ l \mapsto \mathit{false}]}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{l:s} (\gamma'', \Delta'[\mathit{ok} \mapsto (\gamma'_{\mathit{break}\,l} \mathbin{?} \Delta'_{\mathit{break}\,l} : \Delta'_{\mathit{ok}})], \overline{\boldsymbol{t}}')} \ (\text{E}\textsc{Label})$$

$$\frac{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \bar{\boldsymbol{t}}') \qquad (\gamma'[\![e \mapsto \mathit{false}]\!]_{e \neq ok}, \Delta', \bar{\boldsymbol{t}}') \xrightarrow{s'} (\gamma'', \Delta'', \bar{\boldsymbol{t}}'')}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{s\,;s'} (\gamma''[\![e \mapsto \gamma'_e \vee \gamma''_e]\!]_{e \neq ok}, \Delta''[\![e \mapsto (\gamma'_e \mathrel{?} \Delta'_e : \Delta''_e)]\!]_{e \neq ok}, \bar{\boldsymbol{t}}'')} \text{(EComp)}$$

We verified the correspondence of the inference rules above with the pseudocode manually, by a careful inspection of each case. Because the pseudocode algorithm is deterministic and terminating, we can define the encoding function *enc* as follows:

Definition 40 *For an unrolled program s, define the* symbolic encoding *$enc(s)$ to be the triple $(\gamma, \Delta, \bar{\boldsymbol{t}})$ such that $(\gamma^0, \Delta^0, \bar{\boldsymbol{t}}^0) \xrightarrow{s} (\gamma, \Delta, \bar{\boldsymbol{t}})$.*

5.5.6 Correctness of the Algorithm

In this section, we formalize our expectations on the encoding algorithm for programs and prove that they are satisfied. This proof is the key to to our encoding algorithm and its construction heavily influenced our formalization of concurrent executions and memory traces. We moved the core of the proof to Appendix A to avoid overburdening this chapter.

The following theorem expresses the key property of our encoding: the correspondence between the trace semantics E_L and the valuations to the variables in the symbolic trace.

Theorem 41 *Let s be an unrolled program and let $enc(s) = (\gamma, \Delta, \bar{\boldsymbol{t}})$ be its symbolic encoding.*

1. *If ν is a valuation such that $\operatorname{dom} \nu \supset \bar{\boldsymbol{t}}.V$ and $[\![\gamma_e]\!]^\nu = true$, then*

$$([\![\bar{\boldsymbol{t}}]\!]^\nu, e) \in E_L(s).$$

2. *If $(\boldsymbol{t}, e) \in E_L(s)$, then there exists a valuation ν such that*

$$\operatorname{dom} \nu = \bar{\boldsymbol{t}}.V, \qquad [\![\gamma_e]\!]^\nu = true, \qquad \text{and} \qquad [\![\bar{\boldsymbol{t}}]\!]^\nu = \boldsymbol{t}.$$

Proof. The two claims follow directly from Lemmas 55 and 54 which are stated and proven in the appendix, in Section A.3. □

5.5.7 The Formulae Π_k

We now describe how we obtain the formulae $\Pi_1, \ldots, \Pi_n$ that represent the local semantics of the individual threads. Let $P = (s_1, \ldots, s_n)$ be an unrolled concurrent program, and let Y be a memory model. Furthermore, for $1 \leq k \leq n$, let

$$\mathit{enc}(s_k) = (\gamma_k, \Delta_k, \bar{\boldsymbol{t}}_k) \qquad \text{and} \qquad \bar{\boldsymbol{t}}_k = (I_k, \prec_k, V_k, \overline{\mathit{adr}}_k, \overline{\mathit{val}}_k, \overline{\mathit{guard}}_k).$$

To simplify the notation, let $L_k = \mathit{loads}(I_k)$ and $S_k = \mathit{stores}(I_k)$. Then define:

$$\begin{aligned} \Pi_k \quad = \quad & \bigwedge_{i \in I_k} G_i \Leftrightarrow \overline{\mathit{guard}}_k(i) \\ \wedge & \bigwedge_{i \in (L_k \cup S_k)} A_i = \overline{\mathit{adr}}_k(i) \\ \wedge & \bigwedge_{i \in (L_k \cup S_k)} D_i = \overline{\mathit{val}}_k(i) \\ \wedge & \bigwedge_{e \in \mathit{Exs}} (U_k = e) \Leftrightarrow \gamma_e \end{aligned}$$

The following two lemmas capture the properties of Θ formally. We use them in our final correctness theorem.

Lemma 42 *If ν is a valuation of $\sigma_{P,Y}$ such that* $\operatorname{dom}\nu \supset V_A \cup V_D \cup V_G \cup V_U$ *and* $[\![\Pi_k]\!]^\nu = \mathit{true}$*, then* $(\pi_k(\boldsymbol{T}(\nu)), [\![U_k]\!]^\nu) \in E_L(s_k)$.

Lemma 43 *For any execution* $\boldsymbol{T} \in E(P, Y)$ *and for all* $k \in \{1, \ldots, n\}$*, we have* $[\![\Pi_k]\!]^{\nu(\boldsymbol{T})} = \mathit{true}$.

We omit the proofs, which are straightforward (the heavy lifting being done by Theorem 41).

5.6 Correctness

We conclude by showing that the primary objective of this chapter is met, namely that each solution of Φ corresponds to a concurrent execution of P on Y.

Theorem 44 *Let P be an unrolled concurrent program, let Y be an axiomatic memory model specification, and let $V_{P,Y}$ and Φ be the set of variables and the formula constructed as described in this chapter, respectively. Then:*

1. *For all valuations ν such that $\operatorname{dom} \nu = V_{P,Y}$ and $[\![\Phi]\!]^{\nu} = true$, we have $\boldsymbol{T}(\nu) \in E(P, Y)$.*
2. *For all executions $\boldsymbol{T} \in E(P, Y)$, we have $[\![\Phi]\!]^{\nu(\boldsymbol{T})} = true$.*

The theorem is a quite direct consequence of the Lemmas we stated throughout this chapter, where the main work resided.

Proof. First, we examine the (easier) Claim 2. This claim follows directly from the definition of Φ and the combination of Lemma 30 (Section 5.3), Lemma 32 (Section 5.4.2), and Lemma 43 (Section 5.5.7).

Next, we examine Claim 1. By Lemma 29 (Section 5.3), we know that $\boldsymbol{T}(\nu)$ is a global trace. Furthermore, we can apply Lemma 31 (Section 5.4.2) to conclude that

$$\boldsymbol{T}(\nu) \in E_G(Y). \tag{5.4}$$

Now, by Lemma 42 (Section 5.5.7) we know for each k that $(\pi_k(\boldsymbol{T}(\nu)), [\![U_k]\!]^{\nu}) \in E_L(s_k)$. Because $[\![U_k]\!]^{\nu} = stat_{\nu}(k)$ this implies that

$$\forall k \in \{1, \ldots, n\} : (\pi_k(\boldsymbol{T}(\nu)), stat_{\nu}(k)) \in E_L(s_k). \tag{5.5}$$

Furthermore, from the way we constructed $\boldsymbol{T}(\nu)$ it is immediately clear that

$$\forall k \neq \{1, \ldots, n\} : \pi_k(\boldsymbol{T}(\nu)) = \boldsymbol{t}^0. \tag{5.6}$$

Finally, the combination of the facts (5.4), (5.5) and (5.6) implies that $\boldsymbol{T}(\nu) \in E(P, Y)$ as desired (Def. 2 in section 2.1.4). □

Chapter 6

Checking Consistency

In this chapter, we introduce concurrent data types and show how we define correctness and verify it by bounded model checking. The structure of the chapter is as follows:

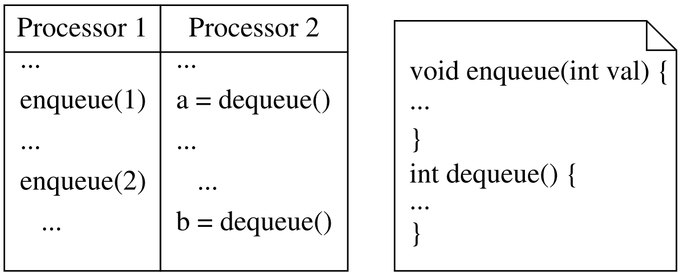
- In Section 6.1, we introduce concurrent data types and articulate why there is a need for automated verification.

- In Section 6.2, we describe the two most common correctness criteria, linearizability and sequential consistency. We also explain why we chose the latter for our purposes.

- In Section 6.3, we show how we bound the verification problem using *bounded test programs*.

- In Section 6.4, we show how we verify the correctness of an implementation for a bounded test program, using the encoding techniques from Chapter 5 and a solver.

- In Section 6.5, we show how we automate our method by mining the specification from the implementation automatically rather than asking the user to provide it.



Processor 1	Processor 2
...	...
enqueue(1)	a = dequeue()
...	...
enqueue(2)	...
...	b = dequeue()

```
void enqueue(int val) {
...
}
int dequeue() {
...
}
```

Figure 6.1: Illustration example: concurrent queue.

6.1 Introduction

Concurrent data types provide familiar data abstractions (such as queues, hash tables, or trees) to client programs that have concurrently executing threads. They can facilitate the development of safe and efficient multi-threaded programs in the form of concurrency libraries. Examples of such libraries include the java.util.concurrent package JSR-166 [46], the Intel Threading Building Blocks [40], or Keir Fraser's lock-free library [21].

Client programs with concurrent threads can benefit from implementations that are optimized for concurrency. Many sophisticated algorithms that use lock-free synchronization have been proposed for this purpose [29, 30, 52, 54, 55, 66, 23]. Such implementations are not race-free in the classic sense because they allow concurrent access to shared memory locations without using locks for mutual exclusion. They are thus sensitive to relaxations in the memory model, and typically need additional memory ordering fences to work correctly on relaxed memory models.

Fig. 6.1 shows an illustrative example: a client program executes concurrently on a multiprocessor (left) and makes invocations to a queue implementation (right). The implementation may guarantee that the operations never block on locks, by using lock-free synchronization (we show such a queue implementation in Section 8.2.1).

Algorithms with lock-free synchronization are notoriously hard to verify, as witnessed by many formal verification efforts [12, 25, 70, 74] and by bugs found in published algorithms [17, 53]. Many more interleavings need to be considered than for implementations that follow a strict locking discipline. Moreover, the deliberate use of races prohibits the use of classic race-avoiding design methodologies and race-detecting tools [1, 20, 31, 43, 57, 61, 63, 73].

In the remainder of this section, we provide some additional background on concurrent data types and lock-free synchronization. The material is intended to serve as background information and motivation only, and is not required to understand the following sections; the reader may choose to proceed to Section 6.2 directly.

6.1.1 Lock-based Implementations

The most common strategy for implementing concurrent data types is to use mutual exclusion locks. Typically (but not always), implementors use such locks to construct *critical sections*. Each process must acquire a lock before executing its critical section and must release it afterwards. Because this prevents more than one thread from being within the critical section, competing accesses to the data are serialized and appear to execute atomically.

Despite their apparent simplicity, experience has shown that there are many problems with lock-based solutions; the symptoms range from relatively harmless performance issues to more serious correctness problems such as deadlocks or missed deadlines in a real-time system.

6.1.2 Lock-free Synchronization

To overcome the drawbacks of locks, many *lock-free* implementations of concurrent data types have been proposed [29, 30, 52, 54, 55, 66, 23]. An implementation is called lock-free if global progress is guaranteed regardless of how threads are scheduled. Lock-free implementations are called *wait-free* if they further guarantee that each operation completes within a bounded number of steps by the calling process.

To write lock- and wait-free implementations, the hardware needs to support non-blocking synchronization primitives such as compare-and-swap (CAS), load-linked / store-conditional (LL/SC), or transactional memory [34]. Multiprocessor architectures commonly used today do not support anything stronger than a 64-bit compare-and-swap and a restricted form of LL/SC (sometimes called RLL/RSC), however, and we focus our attention on implementations designed for those.

Lock- and wait-free implementations have been the subject of much theoretical and practical research. On the more theoretical end, we find existential results [32] or universal constructions for wait-free implementations [33, 4]. In practice, such universal constructions have inferior performance characteristics when compared to simple lock-based alternatives, however. Therefore, many researchers have presented more specialized algorithms that directly implement specific data types such as queues, lists or sets, or provide universal building blocks such as multiword-CAS, multiword-LL/SC, or software transactional memory.

6.1.3 Correctness Proofs

Because programming with lock-free synchronization is well recognized to be difficult, most publications of lock- or wait-free algorithms include some sort of correctness argument, ranging from a few informally described invariants to more formally structured proof sketches or full proofs. In some cases [12, 74], researchers have even undertaken the considerable effort of developing fully machine-checked proofs, using interactive theorem provers such as PVS [59] or shape analysis.

Such proofs (whether fully formal or not) are important to establish that the basic algorithms are correct, as it is only too easy to miss corner cases (as witnessed by the bug in the original pseudocode of the snark algorithm [14, 17]). Unfortunately, proofs are labor-intensive and often operate on a level of abstraction that is well above the architectural details. The actual implementations may thus contain errors even if the high-level algorithm has been proven correct. Our experiments confirmed this as we found a bug in a formally verified implementation (see Section 8.2).

Of particular concern is the fact that almost all published algorithms (and all published correctness proofs) assume a sequentially consistent multiprocessor. Such implementations do not work correctly on a relaxed memory model unless memory ordering fences are added (see Section 3.1). Publications on lock-free algorithms usually ignore this problem, or at best, mention that it remains to be addressed.[1]

6.2 Correctness Criteria

The two most common correctness criteria for concurrent data types are sequential consistency [44] and linearizability [35]. We discuss them only briefly in this section. For a more detailed discussion we recommend the original paper [35].

6.2.1 Linearizability

Briefly stated, linearizability requires that each operation must appear to execute atomically at some point of time between its call and its return. More specifically, it is defined as follows:

- We assume that we have a *serial specification* of the data type, which describes the intended semantics (for example, whether it is a queue, a set, etc.) The specification describes the possible return values if given a linear sequence of operation calls and input arguments.

- We describe concurrent executions as a single sequence of events (the *global history*) where the events are the calls and returns of data type operations, along with the corresponding input arguments and return values.

[1] The notable exceptions are Keir Fraser's lock-free library [21] and Michael's lock-free memory allocator [52] which do discuss the fence issue informally and indicate where fences may be required in the pseudocode.

- We define a global history to be linearizable if it is possible to extend the global history by inserting linearization points for each operation such that (1) the linearization point for an operation is ordered between its call and return events, and (2) the observed return values are consistent with what the serial specification allows for the operations if they are executed in the order of their linearization points.

6.2.2 Sequential Consistency

Sequential consistency predates linearizability and is slightly weaker. It does not require that the operations happen somewhere in between call and return, only that the relative order within each thread be respected.

Sequential consistency is defined as follows:

- Just as for linearizability, we assume that we have a *serial specification* of the data type, which describes the intended semantics.
- We describe concurrent executions as tuples of n operation sequences which describe the sequence of operation calls made by each of the n threads along with the input arguments and return values.
- We define a tuple of n operation sequences to be sequentially consistent if they can be interleaved to form a single sequence of calls that is consistent with the serial specification.

Note that this definition of sequential consistency of concurrent data types coincides with the definition of sequential consistency as a memory model (Section 3.2.2) if we interpret the shared memory itself as a concurrent data type that supports the operations “load” and “store”.

6.2.3 Our Choice

Linearizability has become the de-facto correctness standard and is used pervasively by the concurrent data type community. We believe its success is largely due to its intuitive nature and its attractive compositionality property. However, while perfectly sensible for sequentially consistent multiprocessors, we face some issues when we try to apply this definition to relaxed memory models:

- Executions as defined for general memory models (Section 2.1.4) do not provide a global history of events; the loads and stores that occur in different threads are related by the *seed* function only which matches loads to the stores that source their value. There is no obvious way of recovering a global history from this limited information.

- Under limited circumstances, reordering of memory accesses across operation boundaries should be permitted for performance reasons. For example, it may be desirable for an enqueue call to a concurrent queue to return before the enqueue becomes globally visible to all processors. For a quantitative comparison between sequential consistency and linearizability, see [5].

Because of these issues, we chose to use the sequential consistency criterion, which is well-defined on relaxed memory models, but weaker than linearizability and not compositional (as explained in the original paper [35]). Finding a suitable generalization for linearizability on relaxed memory models remains an open challenge that we may address in future work.

6.3 Bounded Verification

In this section, we describe how we restrict the verification problem (checking sequential consistency of concurrent data types) to bounded instances, which is crucial to allow automatic verification. It is known [3] that the sequential consistency of concurrent data types (for unbounded client programs) is an undecidable problem even if the implementations are finite-state.

To bound the verification problem, we represent the possible client programs using a suite of *bounded test programs* written by the user. For each individual test program, we then try to verify the data type implementation or produce a counterexample if it is incorrect. Of course, we may miss bugs if the test suite does not contain the tests required to expose them.

6.3.1 Bounded Test Programs

A bounded test program specifies a finite sequence of operation calls for each thread. It may choose to use nondeterministic argument values, which conveniently allows us to cover many scenarios with a single test. It may also specify initialization code to be performed prior to the concurrent execution of the threads. Fig. 6.2 shows an example of a bounded test program for a concurrent queue. The input values v_1 and v_2 are nondeterministic.

For a given test T, implementation I and memory model Y, we let $E_{T,I,Y}$ denote the set of concurrent executions of T and I on Y (where executions are defined as in Section 2.1.6).

6.3.2 Correctness Condition

For a test to succeed, all executions of the test must be sequentially consistent in the sense of the general definition of sequential consistency[2] (Section 6.2). Applying

[2] Note that we are talking about sequential consistency on the level of the data type operations here, not on the level of memory accesses as in Section 3.2.2. The two are independent: a data type

thread 1:	thread 2:
enqueue(v_1) *enqueue*(v_2)	$(v_3, v_4) = \mathit{dequeue}()$

Meaning of the operations:

— *enqueue*(v)
adds value v to the queue

— *dequeue*() returns values (r, v)
if queue is empty, returns $r = 0$;
otherwise, returns $r = 1$ and the
dequeued value v

Figure 6.2: Example: A bounded test.

the general definition to the special case of bounded testcases, we can simplify it by (1) narrowly defining the parts of an execution that need to be observed to decide whether it is sequentially consistent, and (2) by finding a simple way to specify the serial behavior of the data type.

- Sequential consistency defines the observables of a concurrent execution to be the sequence of calls made by each thread, along with the argument and return values. All executions of a test T use the same sequence of calls. However, the argument and return values may vary. We define the *observation vector* of an execution to be the tuple consisting of all argument and return values appearing in T, and we define the set V_T to be the set of all observation vectors:

  $$\mathit{obs} : E_{T,I,Y} \to V_T$$

  For example, for the test T in Fig. 6.2, the set V_T is the set of all valuations $(v_1, v_2, v_3, v_3) \in \mathit{Val} \times \mathit{Val} \times \mathit{Val} \times \mathit{Val}$.

- For a fixed test, we can decide whether an execution is sequentially consistent by looking only at the observation vector. We can thus specify the serial semantics of the data type (for example, whether it is a queue, a set, or so on) by specifying the subset $S \subset V_T$ of the observation vectors that are consistent

implementation may be sequentially consistent on the operation level even if the memory model is not sequential consistency. In fact, we take the perspective that a correct implementation *hides* the underlying memory model from the programmer and provides the illusion of sequential consistency to the client program.

with the serial semantics. For example, for the test T in Fig. 6.2, we would specify the intended queue behavior as follows:

$$S = \{(v_1, v_2, v_3, v_4) \mid (v_3 = 1 \land v_4 = v_1) \lor (v_3 = 0)\}$$

6.3.3 Checking Tests

To decide whether all executions of T, I on Y are sequentially consistent, we perform the following two steps independently:

1. We perform a *specification mining* that can extract a finite specification S automatically from the serial executions of an implementation. We describe this procedure in Section 6.5.

2. We perform a *consistency check* that verifies whether all executions of T, I on Y satisfy the specification S in the sense of Def. 45 below. We describe this procedure in Section 6.4.

Definition 45 *The test T and implementation I on memory model Y* satisfy *the specification S if and only if* $\{obs(\boldsymbol{T}) \mid \boldsymbol{T} \in E_{T,I,Y})\} \subset S$.

6.4 Consistency Check

In this section, we show how we can decide whether T, I on Y satisfy a finite specification S by encoding executions as formulae and using a solver to determine if they are satisfiable or not. We call this step the *consistency check*.

We perform the consistency check by constructing a formula Λ of boolean type such that Λ has a satisfying solution if and only if I is *not* sequentially consistent. Then, we call a solver to determine if Λ has a satisfying valuation: if not, I is sequentially consistent. Otherwise, we can construct a counterexample trace from the satisfying valuation.

To start with, we represent the bounded test program T and implementation I by an unrolled concurrent program $P_{T,I} = (s_1, \dots, s_n)$ where each s_k is an unrolled program (as defined in Section 4.5.1) that makes the sequence of operation calls as specified by thread k of I and uses nondeterministic input arguments. For now, we simply assume that the unrolled program is a sound pruning (Section 4.5.3); in practice, we check this separately, and gradually unroll the program until it is. Although the unrolling may not always terminate, it did so for all the implementations we studied because of their global progress guarantees.

By using the the techniques described in Chapter 5, we can then construct the following formulae:

- a formula $\Phi_{T,I,Y}$ such that for all total valuations ν,

$$[\![\Phi_{T,I,Y}]\!]^{\nu} = \mathit{true} \quad \Leftrightarrow \quad \boldsymbol{T}(\nu) \in E_{T,I,Y}.$$

- a formula $\Xi_{T,I,Y}$ such that for all total valuations ν,

$$[\![\Xi_{T,I,Y}]\!]^{\nu} = \mathit{obs}(\boldsymbol{T}(\nu)).$$

Then we define the formula

$$\Lambda = \Phi_{T,I,Y} \ \wedge \ \bigwedge_{o \in S} \Xi_{T,I,Y} \neq o.$$

Now we claim that Λ has the desired property: Λ has a satisfying valuation if and only if I is not sequentially consistent on memory model Y for a given test T with specification S:

- If $E_{T,I,Y}$ is not contained in S, there exists an execution $\boldsymbol{T}$ such that $\mathit{obs}(\boldsymbol{T}) \neq o$ for all $o \in S$, and the valuation $\nu(\boldsymbol{T})$ must thus satisfy Λ.

- If ν is a valuation such that $[\![\Lambda]\!]^{\nu} = \mathit{true}$, then $\boldsymbol{T}(\nu)$ is an execution in $E_{T,I,Y}$ such that $\mathit{obs}(\boldsymbol{T}(\nu)) \neq o$ for all $o \in S$, which implies that $E_{T,I,Y}$ is not contained in S. The execution $\boldsymbol{T}(\nu)$ serves as a counterexample.

6.5 Specification Mining

In this section, we show how we can further automate the verification by mining the specification automatically rather than requiring the user to specify the set S. The basic idea is that we use the serial executions of the implementation as the specification for the concurrent executions.

For a test T and I, we extract the *observation set* $S_{T,I}$ as

$$S_{T,I} = \{obs(\boldsymbol{T}) \mid \boldsymbol{T} \in E_{T,I,\mathrm{Serial}}\},$$

where $E_{T,I,\mathit{Serial}}$ is defined to be the set of *serial executions* of T and I, consisting of all executions $\boldsymbol{T} \in E_{T,I,\mathit{SeqCons}}$ such that the operations are treated as atomic in $\boldsymbol{T}$; that is, the execution never switches threads in the middle of an operation, but at operation boundaries only.

We can make use of a mined specification $S_{T,I}$ in two somewhat different ways. The first one is slightly more automatic, while the second one can potentially find more bugs.

1. We can extract the observation set $S_{T,I}$ directly from the implementation I that we are verifying, and then check whether I is sequentially consistent with respect to $S_{T,I}$.

   While fully automatic (the user need not specify anything beyond the test T), this method may miss some bugs:

   - We may miss bugs that manifest identically in all serial and concurrent executions. For example, if a dequeue() call always returns zero by mistake, our method would not detect it. However, such bugs are easy to find using classic sequential verification approaches; it is thus reasonable to exclude them from our observational seriality check and then check separately whether the serial executions are correct.

- This method may fail to work if the serial executions are more constrained than the specification. Although certainly imaginable (a specification for a data type may allow nondeterministic failure of operations, while such failures do not occur in the implementation if it is restricted to serial executions) the implementations we studied did not exhibit this problem.

2. We can extract the observation set $S_{T,I'}$ from a reference implementation I' that is specifically written to serve as a semantic reference, and then check whether I is sequentially consistent with respect to $S_{T,I'}$.

   In essence, this means that we are reverting back to the user providing a formal specification. However, the specification is written in the form of regular implementation code, rather than some formal specification language. Note that I' need not be concurrent and is thus simple to write correctly.

6.5.1 Algorithm

In this section we give a more technical description of how we perform the specification mining by encoding executions as formulae and then calling a solver.

As in Section 6.4, we start by

- representing the bounded test program T and implementation I by an unrolled concurrent program

$$P_{T,I} = (s_1, \ldots, s_n).$$

- constructing a formula $\Phi_{T,I,Y}$ such that for all total valuations ν,

$$[\![\Phi_{T,I,Y}]\!]^{\nu} = \mathit{true} \quad \Leftrightarrow \quad \boldsymbol{T}(\nu) \in E_{T,I,Y}.$$

- constructing a formula $\Xi_{T,I,Y}$ such that for all total valuations ν,

$$[\![\Xi_{T,I,Y}]\!]^{\nu} = \mathit{obs}(\boldsymbol{T}(\nu)).$$

To construct the observation set $S_{T,I}$ we then use the following iterative procedure. Basically, we keep solving for serial executions that give us fresh observations (by adding clauses to the formula to exclude observations already found). When the formula becomes unsatisfiable, we know we have found all observations.

```
function mine(T : test; I : implementation) : set of observations
  var
    obsset : set of observations;
    fla    : boolean_formula;
{
  obsset  := empty_set;
  fla := $\Phi_{T,I,\mathtt{Serial}}$;
  while (has_solution(fla)) {
    var valuation := solve(fla);
    var observation := $[\![\Xi_{T,I,Y}]\!]^{\mathtt{valuation}}$;
    obsset := obsset $\cup$ { observation };
    fla := fla $\wedge$ $\neg$ ( $\Xi_{T,I,Y}$ = observation );
  }
  return obsset;
}
```

Our practical experience suggests that even though the set of serial executions $E_{T,I,Serial}$ can be quite large (due to nondeterministic memory layout and interleavings), the observation set $S_{T,I}$ contains no more than a few thousand elements for the testcases we used (Section 8.3.2). Therefore, the iterative procedure described above is sufficiently fast, especially when used with a SAT solver that supports incremental solving.

Chapter 7

The *CheckFence* Implementation

In this section, we describe the tool *CheckFence*, which implements our verification method, and discuss a few of the implementation challenges. The chapter is structured as follows:

- Section 7.1 gives a general description of the tool (the “black-box” view).
- Section 7.2 describes the front end that translates C to LSL.
- Section 7.3 describes the back end that unrolls loops, encodes executions as SAT formulae, and makes calls to the SAT solver.

7.1 General Description

Our *CheckFence* tool works as follows (see also the example in Fig. 7.1). The user supplies (1) an implementation for a concurrent data type such as a queue or a set (written in C, possibly with memory ordering fences), (2) a symbolic test program (a finite list of the operation calls made by each thread, with nondeterministic argument values), and (3) a choice of memory model. The tool then verifies that all executions of this test on the chosen memory model meet the following correctness criteria:

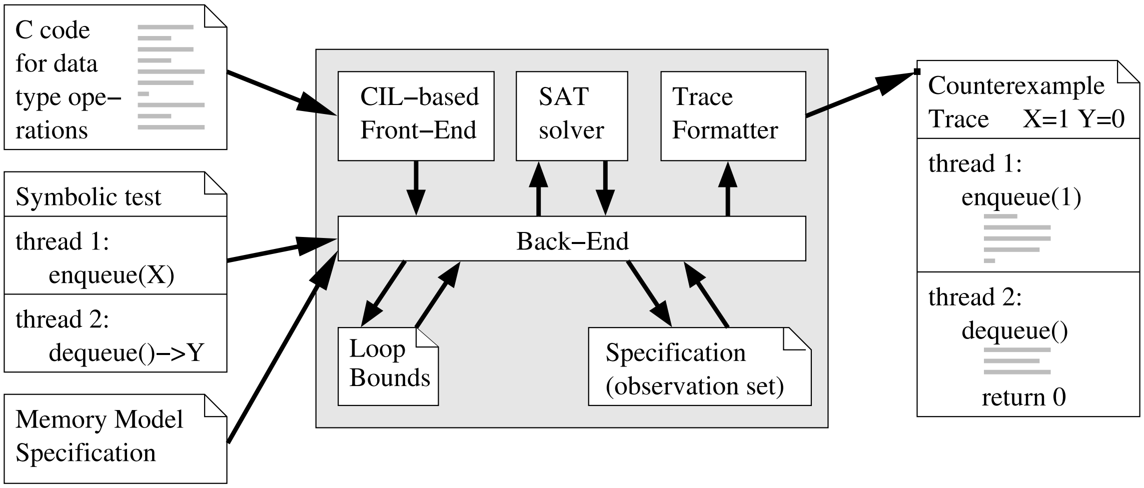


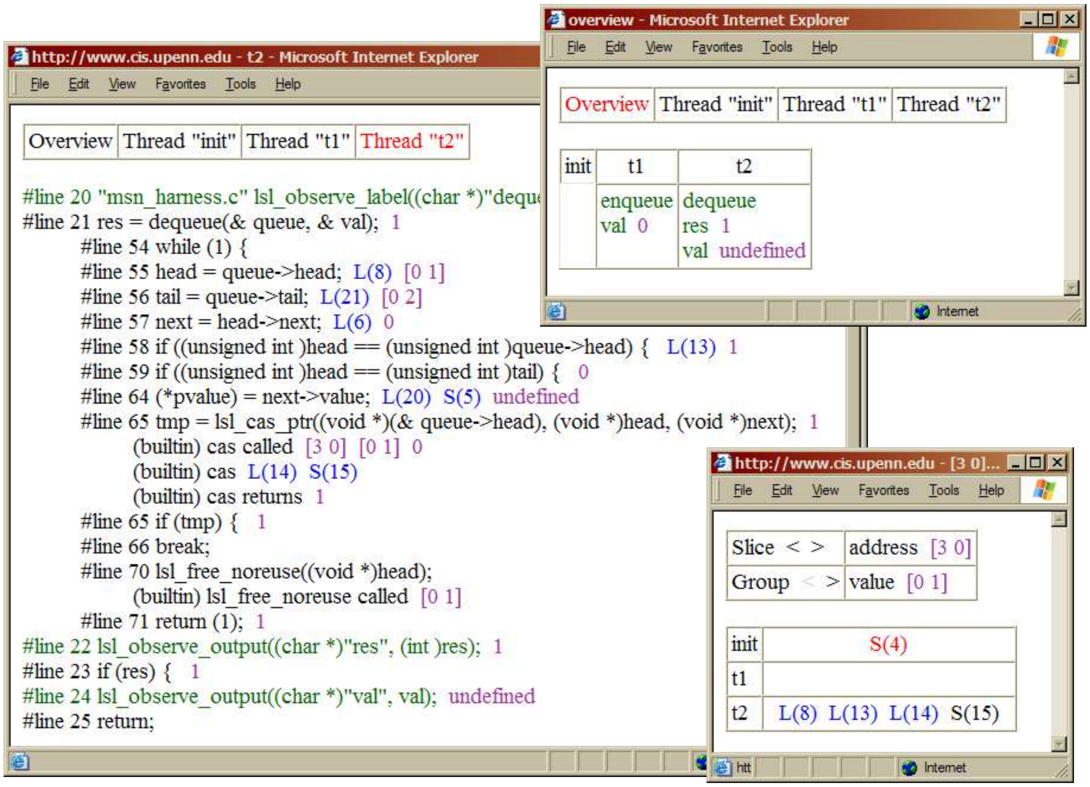
Figure 7.1: An example run of the tool. The counterexample reveals an execution for which the return values of the operations are not consistent with any serial execution of the test.

- the values returned by the operations of the data type are consistent with some serial execution of the operations (that is, an execution where the threads are interleaved along operation boundaries only).

- all runtime checks are satisfied (e.g. no null pointers are dereferenced). We describe the checks below in Section 7.1.1.

If there is an incorrect execution, the tool creates a counterexample trace. Otherwise, it produces a witness (an arbitrarily selected valid execution). The execution traces are formatted in HTML and can be inspected using a browser. We show a screenshot in Fig. 7.2.

CheckFence is sound in the sense that all counterexamples point out real code problems. It is complete in the sense that it exhaustively checks all executions of the given symbolic test. Of course, bugs may be missed because the test suite (which is manually written by the user) does not cover them. However, our experience with actual examples (Chapter 8) suggests that a few simple tests are often sufficient to find memory-model related errors. A single test covers many executions


http://www.cis.upenn.edu - t2 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
overview - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Internet
http://www.cis.upenn.edu - [3 0]...
File Edit View Favorites Tools Help
htt
Internet

Overview	Thread "init"	Thread "t1"	Thread "t2"

```
#line 20 "msn_harness.c" lsl_observe_label((char *)"deque
#line 21 res = dequeue(& queue, & val);  1
        #line 54 while (1) {
        #line 55 head = queue->head;  L(8)  [0 1]
        #line 56 tail = queue->tail;  L(21)  [0 2]
        #line 57 next = head->next;  L(6)  0
        #line 58 if ((unsigned int )head == (unsigned int )queue->head) {   L(13)  1
        #line 59 if ((unsigned int )head == (unsigned int )tail) {   0
        #line 64 (*pvalue) = next->value;  L(20)  S(5)  undefined
        #line 65 tmp = lsl_cas_ptr((void *)(& queue->head), (void *)head, (void *)next);  1
               (builtin) cas called  [3 0]  [0 1]  0
               (builtin) cas  L(14)  S(15)
               (builtin) cas returns  1
        #line 65 if (tmp) {   1
        #line 66 break;
        #line 70 lsl_free_noreuse((void *)head);
               (builtin) lsl_free_noreuse called  [0 1]
        #line 71 return (1);  1
#line 22 lsl_observe_output((char *)"res", (int )res);  1
#line 23 if (res) {   1
#line 24 lsl_observe_output((char *)"val", val);  undefined
#line 25 return;
```

Overview	Thread "init"	Thread "t1"	Thread "t2"

init	t1	t2
	enqueue val 0	dequeue res 1 val undefined

Slice < >	address [3 0]
Group < >	value [0 1]

init	S(4)
t1	
t2	L(8) L(13) L(14) S(15)

Figure 7.2: A screenshot showing a counterexample that is viewed with a standard HTML browser

because all interleavings, all instruction reorderings, and all choices for input values are considered.

7.1.1 Runtime Type Checks

Before performing the consistency check (which determines whether all concurrent executions are observationally equivalent to a serial execution), it makes sense to check a few simpler 'sanity' conditions. The reason is that we would like to reduce the time required by the user to understand the counterexample. A symptom such as a null pointer dereference is much easier to debug if the tool directly reports it as such.

For this reason, we distinguish the type of a value (one of undefined, pointer, or number) at runtime, and check the following conditions for all executions (we call them runtime type checks).

- All loads and stores must use pointers for accessing memory. Using an undefined value or a number fails the test.

- All equality tests must be either between two numbers, between two pointers, or between a pointer and the number 0 (which is used in C to denote an invalid pointer). Comparing an undefined value to any other value or comparing a pointer to a nonzero number fails the test.

- All basic functions are applied to arguments for which they are defined (as described in Section 4.2.8). For example, arithmetic operations require numbers, division or modulus operators require nonzero divisors, and pointer manipulation functions require pointers.

7.2 Front End

The translation from C to LSL is in many ways similar to a standard compilation of C into assembly language. However, because LSL sits at a somewhat higher abstraction level than assembly language, there are some differences.

7.2.1 General Philosophy

To limit the complexity of the translation, we take advantage of the domain-specific nature of the code we are targeting. Furthermore, we assume a cooperating programmer who is willing to perform some minor rewriting of the C code for verification purposes. However, we are careful to avoid any negative implications on the performance of the resulting libraries; we do not disallow any of the optimizations programmers of concurrent data types tend to use, such as packing several fields into a single machine word.

Where implementations require the use of special low-level C idioms, we provide library functions that let the programmer express her intentions on the correct abstraction level. For example, if there is a need to apply negative offsets (say we have a pointer to a field and wish to get a pointer to the containing structure), we provide a library function

```
lsl_get_struct_pointer(structtype, fieldname, fieldpointer)
```

which is more readable and easier to translate than cryptic pointer arithmetic.

7.2.2 Parsing

We chose to use the OCaml language for implementing our front end, because it offers powerful pattern matching that is very convenient for transforming programs.

Our front end relies heavily on the CIL (C Intermediate Language) component [58]. Using CIL frees us from many of the cumbersome details associated with C programs: it parses the C source and provides us with a cleaned up and simplified program representation with the following properties:

- Rather than using a generic control flow graph, the block structure of the program is preserved. For instance, loops are directly represented as such, and need not be recovered from the CFG.
- Pointer offset arithmetic is recognized as such, and represented by different operators than integer arithmetic.

We then transform the program we get from CIL into LSL. Here are some of the design choices we made:

Local Variables. If a local variable is scalar (not an array or struct), and its address is not taken, we represent it as a LSL register (simulating register allocation by the compiler). Otherwise, we reserve space in main memory.

Structs and Arrays. We reserve space for structures and arrays in the "address space". Because our addresses are simply lists of integers, our job is much easier than that of an actual C compiler: we need not know the size of the allocated structures, but can simply use consecutive numbers for all offsets, regardless of whether they are arrays or structs (see Fig. 4.2 in Section 4.4 for an illustration of this concept).

Gotos. We do not currently support arbitrary goto statements, but only allow forward-directed gotos that do not jump into bodies of loops and conditionals (which we can easily model in LSL by exiting an enclosing block using a break statement).

Memory Access Alignment. We currently assume that the memory is accessed at a uniform granularity. Specifically, any two memory accesses in the program must go to either disjoint or identical memory regions. This implies that structs may not be copied as entire blocks (note that such an operation has unspecified multiprocessor semantics).

Unsupported Features. We do not currently model unions, function pointers, floating point operations, string operations, or standard library calls.

We used the implementations as a guide to decide what features to include, and will add more features as needed when studying more implementations.

7.2.3 Locks

Some of the implementations we studied use locks. To implement locks, we provide functions

```
void lsl_initlock(lsl_lock_t *lock);
void lsl_lock(lsl_lock_t *lock);
void lsl_unlock(lsl_lock_t *lock);
```

The front end replaces calls to these functions with the equivalent of a spinlock implementation (Fig. 3.9 in Section 3.4.3) that we adapted from the SPARC manual [72]. It uses an atomic load-store primitive and (partial) memory ordering fences.

To deal with the spinloop in the back end (note that we can not meaningfully unroll a spin loop) we use a special LSL construct for side-effect-free spinloops. Our encoding then (1) models the last iteration of the spin loop only (all others being redundant), and (2) checks specifically for deadlocks. We did not formalize this part of the algorithm yet, but may do so in the future.

7.2.4 Dynamic Memory Allocation

To simulate dynamic allocation, we provide two variants of memory allocation for the user to choose from. Both follow the same syntax as the standard C calls:

```
void *lsl_malloc_noreuse(size_t size);
void lsl_free_noreuse(void *ptr);

void *lsl_malloc(size_t size);
void lsl_free(void *ptr);
```

1. The lsl_malloc_noreuse function simply returns a fresh memory location every time it is called. We can encode this mechanism easily and efficiently, but it may miss bugs in the data type implementation that are caused by reallocations of previously freed memory.

2. The lsl_malloc and lsl_free functions model dynamic memory allocation more accurately. To do so, we create an array of blocks, each with its own lock. We make the array contain as many blocks as there are calls to lsl_malloc in the unrolled code. A call to lsl_malloc then nondeterministically selects a free block in the array and locks it. A call to lsl_free unlocks it again (or flags an error if it is already unlocked).

By using the lsl_malloc and lsl_free we can detect ABA problems[1] that are common for implementations based on compare-and-swap. Our simplified nonblocking queue implementation (Section 8.1) is susceptible to the ABA problem because we removed the counter from the original implementation. We confirmed experimentally that it fails when we use the lsl_malloc and lsl_free.

7.2.5 Packed Words

Many lock-free implementations use a compare-and-swap primitive to achieve nonblocking synchronization. Such primitives are available on all major hardware platforms. However, the hardware implementations operate on limited word widths only,

[1] The ABA problem occurs if a compare-and-swap succeeds when it should fail, because it sees a new value that coincidentally matches an old value (as could happen with a freed and reallocated pointer) and wrongly assumes that the value was never modified.

typically 32 or 64 bit. As a result, algorithm designers must often resort to packing structures into a single word or doubleword.

Since LSL does support structures (by using pointer offsets, see Fig. 4.2) it may seem at first that there is no need for adding direct support for packed words. However, we found that for the practice of lock-free programming, packed words have a fairly different flavor than structures and deserve special treatment. For one, packed words are not nested (because they are intended to be used with a fixed-width CAS). Moreover, programs cannot access individual fields of the packed word but are forced to load and/or store the entire word, which has implications on the multiprocessor semantics.

To support packed words, we have the front end recognize functions that create or modify packed words by means of a special naming convention. For example, to work with a "marked pointer" type `mpointer_t` which combines a pointer and a boolean into a single word, the programmer can simply declare and use the following functions (as done by the set implementation *harris* listed in Section B.4):

```
mpointer_t lsl_make_mpointer(node_t *ptr, boolean_t marked)
node_t *lsl_get_mpointer_ptr(mpointer_t mpointer)
boolean_t lsl_get_mpointer_marked(mpointer_t mpointer)
mpointer_t lsl_set_mpointer_ptr(mpointer_t mpointer, node_t *ptr)
mpointer_t lsl_set_mpointer_marked(mpointer_t mpointer,boolean_t marked)
```

Internally, the front end ignores the C definitions of these functions (if any are supplied) and substitutes them with basic functions that construct and access a LSL tagged union type

composite mpointer (ptr, marked)

(as defined in Section 4.6.3).

7.3 Back End

In the back end of our *CheckFence* tool, we unroll the LSL program and encode it into one or more formulae that are then handed to the SAT solver. In the following sections, we discuss some of the technical issues associated with this step.

7.3.1 Lazy Loop Unrolling

For the implementations and tests we studied, all loops are statically bounded. However, this bound is not necessarily known in advance. We therefore unroll loops *lazily* as follows. For the first run, we unroll each loop exactly once. We then run our regular checking, but restrict it to executions that stay within the bounds. If an error is found, a counterexample is produced (the loop bounds are irrelevant in that case). If no error is found, we run our tool again, solving specifically for executions that exceed the loop bounds. If none is found, we know the bounds to be sufficient. If one is found, we increment the bounds for the affected loop instances and repeat the procedure.

7.3.2 Range Analysis

To reduce the number of boolean variables and determine the required bitwidth to represent values, we perform a range analysis before encoding the concurrent execution formula. Specifically, we use a simple lightweight flow-insensitive analysis to calculate for each assignment to an SSA register r and each memory location m, sets S_r, S_m that conservatively approximate the values that r or m may contain during a valid execution. We can sketch the basic idea as follows. First, initialize S_r and S_m to be the empty set. Then, keep propagating values as follows until a fixpoint is reached:

- constant assignments of the form $r = c$ propagate the value c to the set S_r.

- assignments of the form $r = f(r_1, \ldots, r_k)$ propagate values from the sets $S_{r_1}, \ldots, S_{r_k}$ to the set S_r (applying the function).
- stores of the form $*r' = r$ propagate values from the set S_r to the sets $\{S_m \mid m \in S_{r'}\}$.
- loads of the form $r = *r'$ propagate values from the sets $\{S_m \mid m \in S_{r'}\}$ to the set S_r.

This analysis is sound for executions that do not have circular value dependencies.[2] To ensure termination, we need an additional mechanism. First, we count the number of assignments in the test that have unbounded range. That number is finite because we are operating on the unrolled, finite test program. During the propagation of values, we tag each value with the number of such functions it has traversed. If that number ever exceeds the total number of such functions in the test, we can discard the value.

We use the sets S_r for four purposes: (1) to determine a bitwidth that is sufficient to encode all integer values that can possibly occur in an execution, (2) to determine a maximal depth of pointers, (3) to fix individual bits of the bitvector representation (such as leading zeros), and (4) to rule out as many aliasing relationships as possible, thus reducing the size of the memory model formula.

7.3.3 Using a SAT Solver

Using the method described in Chapter 6, we can decide whether a test is observationally serial or not by using a solver that takes some propositional formula Ψ and either finds a valuation ν such that $[\![\Psi]\!]^\nu = \mathit{true}$, or proves that no such valuation exists. We now show how we "dumb" this problem down to the level of a SAT solver (which is a well-known standard procedure). Essentially, we need to (1) convert the

[2] We do not define formally what we mean by "circular dependency". For an informal example, see Fig. 4.7 which shows an execution with circular dependencies that may be allowed by some pathological memory model. None of the hardware models we studied would allow such a behavior.

formula to CNF (conjunctive normal form), and (2) express all values using boolean variables only.

Note that our actual implementation does not closely follow the procedure described below (much of our conversion takes place as a byproduct of the translation of the program into instruction streams). Nevertheless, the description we include below gives a fair impression of the concept.

Assume that we are given some propositional formula Ψ over a vocabulary σ. Then, we perform the following steps.

- We introduce an auxiliary variable F_ψ for each non-atomic subformula ψ of Ψ, of the same type as ψ. We let σ' denote this new, larger vocabulary that includes the auxiliary variables.

- For a given formula ψ with non-atomic subformulae $\psi_1, \ldots, \psi_k$, let ψ' be the formula that results from substituting F_{ψ_i} for ψ_i in ψ.

- Define the following formula over σ':

$$\Psi' = F_\Psi \wedge \bigwedge_{\psi \text{ is a non-atomic subformula of } \Psi} (F_\Psi = \psi')$$

Now, it is easy to see that Ψ and Ψ' are equivalent in the sense that (1) any valuation ν satisfying Ψ can be extended to a valuation ν' that satisfies Ψ', and (2) any valuation ν' satisfying Ψ' restricts to a valuation ν that satisfies Ψ.

To get Ψ' down to CNF form suitable for use with a SAT solver, we represent each boolean variable by a SAT variable, and each value variable by a vector of SAT variables (our range analysis helps us to determine a sufficient width for the vectors). We can then encode the equations $(F_\Psi = \psi')$ directly using SAT clauses.

For some functions (such as the elementary logical functions) the resulting clauses are very easy; for some they are more complex and may require additional auxiliary variables. For example, to express signed integer additions, we require carry bits. Multiplication and division also require a large number of auxiliary bits.

7.3.4 Optimized Memory Models

Before we added support for user-specified axiomatic memory models, our tool used handwritten memory model encoding procedures for *SeqCons* and *Relaxed*. These encoding procedures are optimized for the specific properties of the memory model, and differ from the generic memory model encoding (Section 5.4) as follows:

- Rather than representing the memory order by $|\mathit{accesses}(I)|^2$ variables $\{M_{xy} \mid x, y \in \mathit{accesses}(I)\}$ such that $M_{xy} = \mathit{true}$ if and only if both x and y are executed and $x <_M y$, we "internalize" the fact that the memory order is a total order and use only $|\mathit{accesses}(I)| * |\mathit{accesses}(I) - 1|/2$ literals $\{M_{xy} \mid x, y \in \mathit{accesses}(I) \wedge x < y\}$ for some arbitrary total order $<$ over the accesses.

- Many axioms contain redundant guard conditions when expanded automatically; our handwritten encoding does not have this drawback.

- We statically establish the connection between the program order (which is statically known) and the memory order. That is, if $x \prec y$, we statically fix the variable $M_{xy} = 1$; where necessary, we also adjust all axioms that use these variables to account for the fact that the guards of x and y are no longer implicit in M_{xy}.

These optimizations lead to much smaller encodings and significantly faster solving time, as our measurements demonstrate (Section 8.3.4). Of course, it should be possible (and interesting) to apply these optimizations in a generic way, that is, to arbitrary memory model specifications. We have not investigated this yet, however, and leave it as future research.

Chapter 8

Experiments

In this chapter, we describe the experiments we performed with *CheckFence* and analyze the results. The primary goal of the experiments is to answer the following questions:

- How well does *CheckFence* achieve the stated goal of supporting the design and implementation of concurrent data types?
- How scalable is *CheckFence*? Is it reasonably efficient at finding bugs and/or missing fences?
- How does the choice of memory model and encoding impact the tool performance?

To answer these questions, we studied the five implementations shown in Fig. 8.1. All of them make deliberate use of data races. Although the original publications contain detailed pseudocode, they do not indicate where to place memory ordering fences. Thus, we set out to (1) verify whether the algorithm functions correctly on a sequentially consistent memory model, (2) find out what fails on the relaxed model and (3) add memory fences to the code as required.

First we wrote symbolic tests (Fig. 8.2). To keep the counterexamples small, we started with small and simple tests, say, two to four threads with one operation each. We then gradually added larger tests until we reached the limits of the tool.

Using *CheckFence*, we found a number of bugs in the implementations, both algorithmic bugs that are not related to the memory model, and failures that are caused by missing fences. The performance results confirm that our method provides an efficient way to check bounded executions of concurrent C programs with up to about 200 memory accesses.

8.1 Implementation Examples

Fig. 8.1 lists the implementations we used for our experiments. The implementations provide commonly used abstract data types (queues, sets, and deques). All of the implementations were taken from the cited publication, where they appear in the form of C-like pseudocode. To subject them to our experiments, we translated the pseudocode to C (making a few modifications as listed in the next paragraph). In the rightmost column, we show the (approximate) number of lines of the C programs.

With respect to the original algorithms, we simplified the code in one case: the original code for the nonblocking queue stores a counter along with each pointer to avoid the so-called ABA problem.[1] To put more focus on the core algorithm, we decided to remove the counter and instruct the memory allocator not to reuse memory locations.[2]

[1] The ABA problem occurs if a compare-and-swap succeeds when it should fail, for instance, if a dynamically allocated pointer was freed and re-allocated, thus appearing to be the same pointer and making the CAS wrongly assume that the list was not modified.

[2] We confirmed that without counters and with reallocation of freed memory locations (see Section 7.2.4), the code fails, exhibiting an ABA problem on test Ti2.

ms2	Two-lock queue [54] Queue is represented as a linked list, with two independent locks for the head and tail.	54 loc
msn	Nonblocking queue [54] Similar to ms2, but uses compare-and-swap for synchronization instead of locks (Section 8.2.1).	74 loc
lazylist	Lazy list-based set [12, 30] Set is represented as a sorted linked list. Per-node locks are used during insertion and deletion, but the list supports a lock-free membership test.	111 loc
harris	Nonblocking set [28] Set is represented as a sorted linked list. Compare-and-swap is used instead of locks.	140 loc
snark	Nonblocking deque [14, 17] Deque is represented as linked list. Uses double-compare-and-swap.	141 loc

Figure 8.1: The implementations we studied. We use the mnemonics on the left for quick reference.

Queue tests: (e,d for enqueue, dequeue)

T0 = (e | d)
T1 = (e | e | d | d)
Tpc2 = (ee | dd)
Tpc3 = (eee | ddd)
Tpc4 = (eeee | dddd)
Tpc5 = (eeeee | ddddd)
Tpc6 = (eeeeee | dddddd)

Ti2 = e (ed | de)
Ti3 = e (de | dde)
T53 = (eeee | d | d)
T54 = (eee | e | d | d)
T55 = (ee | e | e | d | d)
T56 = (e | e | e | e | d | d)

Set tests: (a, c, r for add, contains, remove)

Sac = (a | c)
Sacr = (a | c | r)
Sacr2 = aar (a | c | r)
S1 = (a' | a' | c' | c' | r' | r')

Sar = (a | r)
Saacr = a (a | c | r)
Saaarr = aaa (r | rc)
Sarr = (a | r | r)

Deque tests: (a_l, a_r, r_l, r_r for add/remove left/right)

D0 = $(a_l \; r_r \mid a_r \; r_l)$
Da = $a_l \; a_l \; (r_r \; r_r \mid r_l \; r_l)$
Dm = $(a'_l \; a'_l \; a'_l \mid r'_r \; r'_r \; r'_r \mid r'_l \mid a'_r)$
Dq = $(a'_l \mid a'_l \mid a'_r \mid a'_r \mid r'_l \mid r'_l \mid r'_r \mid r'_r \;)$

Db = $(r_r \; r_l \mid a_r \mid a_l)$

Figure 8.2: The tests we used. We show the invocation sequence for each thread in parentheses, separating the threads by a vertical line. Some tests include an initialization sequence which appears before the parentheses. If operations need an input argument, it is chosen nondeterministically out of $\{0, 1\}$. Primed versions of the operations are restricted forms that assume no retries (that is, retry loops are restricted to a single iteration).

Data Type	Description	Bugs found	Fences inserted			
			LL	SS	CL	DL
ms2	Two-lock queue	0	0	1	0	1
msn	Nonblocking queue	0	4	2	2	1
lazylist	Lazy list-based set	1 (new)	0	1	0	3
harris	Nonblocking list-based set	0	0	1	3	2
snark(buggy)	Buggy “snark” algorithm	2 (prev. known)				
snark(fixed)	Fixed “snark” algorithm	0	2	4	6	4

Figure 8.3: The issues we found using our *CheckFence* tool.

8.2 Bugs Found

Fig. 8.3 summarizes the issues we found with the code. We discuss these results in the remainder of this section.

We found several bugs that are not related to relaxations in the memory model. The snark algorithm has two known bugs [17, 45]. We found the first one quickly on test D0. Finding the other one requires a fairly deep execution. We found it with the test Dq, which took about an hour.

We also found a not-previously-known bug in the lazy list-based set: the pseudocode fails to properly initialize the ‘marked’ field when a new node is added to the list. This simple bug went undetected by a formal correctness proof [12] because the PVS source code did not match the pseudocode in the paper precisely [49]. This confirms the importance of using actual code (rather than pseudocode and manual modeling) for formal verification.

8.2.1 Missing Fences

As expected, our tests revealed that all five implementations require memory fences to function correctly on relaxed memory models (the original algorithms are designed for sequentially consistent architectures and do therefore not need any fences). To our knowledge, there is no established, sound methodology of determining where fences are required in such implementations. The following passage from Keir Fraser's dissertation *Practical Lock-Freedom* [22] describes the state of the art:

> Unfortunately, there is no automatic method for determining the optimal placement of memory barriers. [...] In the absence of an automated method, the accepted technique is to determine manually, and in an ad hoc fashion, where barriers need to be placed. I base these decisions on analysis of the pseudocode [...]. This is backed up with extensive testing on real hardware [...].

Nonblocking Queue with Memory Fences

To give a concrete example, we show the source code for the non-blocking queue with appropriate fences below. The algorithm uses a list-based representation of the queue, with individual pointers to the head and tail. It enqueues elements at the tail and dequeues them at the head. The actual modification of the list is carried out by a compare-and-swap operation that operates on pointers.

Without fences, the algorithm does not work correctly on a relaxed memory model, for a number of reasons that we discuss in Section 8.2.2.

To our knowledge, this is the first published version of Michael and Scott's non-blocking queue that includes memory ordering fences (besides the one appearing in our paper [9]). We verified that on *Relaxed* these fences are sufficient and necessary for the tests in Fig. 8.2. Of course, our method may miss some fences if the tests do not cover the scenarios for which they are needed. An interesting observation is that the implementations we studied required only load-load and store-store fences. On some architectures (such as Sun TSO or IBM zSeries), these fences are automatic and the algorithm therefore works without inserting any fences on these architectures.

```
typedef struct node {
  struct node *next;
  value_t value;
```

```
} node_t;
typedef struct queue {
  node_t *head;
  node_t *tail;
} queue_t;

extern void assert(bool expr);
extern void fence(char *type);
extern int cas(void *loc,
               unsigned old, unsigned new);
extern node_t *new_node();
extern void delete_node(node_t *node);

void init_queue(queue_t *queue)
{
  node_t *node = new_node();
  node->next = 0;
  queue->head = queue->tail = node;
}
void enqueue(queue_t *queue, value_t value)
{
  node_t *node, *tail, *next;
  node = new_node();
  node->value = value;
  node->next = 0;
  fence("store-store");
  while (true) {
    tail = queue->tail;
    fence("load-load");
    next = tail->next;
    fence("load-load");
    if (tail == queue->tail)
      if (next == 0) {
        if (cas(&tail->next,
                (unsigned) next, (unsigned) node))
          break;
      } else
        cas(&queue->tail,
            (unsigned) tail, (unsigned) next);
  }
  fence("store-store");
  cas(&queue->tail,
      (unsigned) tail, (unsigned) node);
}
bool dequeue(queue_t *queue, value_t *pvalue)
{
  node_t *head, *tail, *next;
  while (true) {
    head = queue->head;
    fence("load-load");
    tail = queue->tail;
    fence("load-load");
    next = head->next;
    fence("load-load");
    if (head == queue->head) {
      if (head == tail) {
        if (next == 0)
          return false;
        cas(&queue->tail,
            (unsigned) tail, (unsigned) next);
64    } else {
65     *pvalue = next->value;
66     if (cas(&queue->head,
67         (unsigned) head, (unsigned) next))
68      break;
69    }
70   }
71  }
72  delete_node(head);
73  return true;
74 }
```

8.2.2 Description of Typical Failures

We now describe the different types of failures that we found to occur commonly on relaxed memory models.

Incomplete initialization. A common failure occurs with code sequences that (1) allocate a new node, (2) set its fields to some value and (3) link it into the list. On relaxed memory models, the stores to the fields (in step 2) may be delayed past the pointer store (in step 3). If so, operations by other threads can read the node fields before they contain the correct values, with fatal results. All five implementations showed this behavior. The fix is the same in all cases: adding a store-store fence between steps (2) and (3). In our example, the store-store barrier on line 29 was added for this reason.

Reordering of value-dependent instructions. Some weak architectures (such as Alpha [13]) allow loads to be reordered even if they are value dependent. For example, the common code sequence (1) read a pointer `p` to some structure and (2) read a field `p->f` is (somewhat surprisingly) susceptible to out-of-order execution:

the processor may perform the load of `p->f` before the load of `p` by speculating on the value of `p` and then confirming it afterward [51]. We found this behavior to cause problems in all five implementations. To avoid it, we add a load-load fence between the two instructions. In our example, the load-load fence on line 32 was inserted for this reason.

Reordering of CAS operations. We model the compare-and-swap operation without any implied fences (Fig. 3.8). As a result, two CAS instructions to different addresses may be reordered. We observed this behavior only for the nonblocking queue, where it causes problems in the dequeue operation if the tail is advanced (line 45) before the node is linked into the list (line 37). To fix this problem, we added a store-store fence on line 44.

Reordering of load sequences. The nonblocking queue uses simple load sequences to achieve some synchronization effects. For example, `queue->tail` is loaded a first time on line 31; next, `tail->next` is loaded (line 33); then, `queue->tail` is loaded a second time (line 35) and the value is compared to the previously loaded value. If the values are the same, the implementation infers that the values that were loaded for `queue->tail` and `tail->next` are consistent (that is, can be considered to have been loaded atomically). A similar load sequence is used in the enqueue operation (lines 52 and 58). For this mechanism to work, we found that the loads in the sequence must not be reordered, and we added a number of load-load fences to achieve this effect (lines 32, 34, 53, 55, 57). The other implementations did not exhibit this behavior.

8.3 Quantitative Results

Our experiments confirm that the *CheckFence* method is very efficient at verifying or falsifying small tests. As small tests appear to be sufficient to find memory-model-related bugs (all of the tests that we required to expose missing fences contained fewer than 200 memory accesses and took less than 10 minutes to verify), *CheckFence* achieves its main goal.

Furthermore, *CheckFence* proved to be useful to find algorithmic bugs that are not related to the memory model. In general, however, However, some of those bugs require deeper executions which challenge the scalability of the tool. For example, the test Dq (see Fig. 8.2) required about 1 hour of time to expose the second snark bug. In general, we found that the tool does not scale for tests that contain more than a few hundred memory accesses.

We first describe the performance numbers of the consistency check in more detail (Section 8.3.1). Then we show data on the observation mining (Section 8.3.2). Next, we discuss the results on how the range analysis and the memory model encoding algorithm affect the tool performance (Sections 8.3.3 and 8.3.4).

8.3.1 Consistency Check Statistics

To illustrate the performance of the consistency checks, we show statistics in Fig. 8.4. We plot the resource requirements in Fig. 8.5. The dots correspond to individual tests, and we use separate marks to show which implementation is being tested. The placement of the dots shows the correlation between the number of memory accesses in the test (X axis, linear scale) and the resource requirement (Y axis, logarithmic scale).

As described in Section 2.1.6, *CheckFence* encodes the consistency check as a CNF formula which is then refuted by the zChaff SAT solver [56] (version 2004/11/15).

	Test	Unrolled code			Encoding	CNF formula		Zchaff		Total
	Name	instrs	loads	stores	time [s]	vars	clauses	[MB]	time [s]	time [s]
ms2	T0	142	13	20	0.1	433	5,077	<1	<0.1	0.1
	T1	256	25	33	0.6	1,266	40,454	4	0.2	0.8
	T53	346	33	47	2.0	2,445	125,831	12	1.9	3.9
	T54	346	33	47	1.8	2,378	125,745	12	5.1	6.9
	T55	346	33	47	1.6	2,331	125,788	12	7.2	8.7
	T56	346	33	47	1.6	2,304	123,660	12	3.3	4.9
	Ti2	301	29	40	0.8	1,409	42,418	4	0.2	1.0
	Ti3	370	37	46	1.7	2,116	90,066	12	0.7	2.4
	Tpc2	256	25	33	0.7	1,294	41,553	4	0.1	0.8
	Tpc3	370	37	46	2.4	2,591	142,742	15	1.6	4.0
	Tpc4	484	49	59	7.4	4,324	342,446	47	12.0	19.4
	Tpc5	598	61	72	16.4	6,493	677,390	94	91.6	108.0
	Tpc6	712	73	85	35.0	9,098	1,178,842	186	367.0	402.0
msn	T0	214	22	14	0.2	1,004	12,151	1	<0.1	0.2
	T1	1000	115	55	22.0	14,848	1,597,115	189	314.0	336.0
	T53	966	107	57	22.0	14,536	1,426,104	188	105.0	127.0
	Ti2	843	93	48	8.5	10,407	709,416	94	7.2	15.7
	Ti3	1086	122	60	25.6	16,344	1,633,713	190	35.4	61.0
	Tpc2	454	48	27	1.4	3,496	126,013	12	0.7	2.0
	Tpc3	694	74	40	5.4	7,638	468,274	48	6.5	11.9
	Tpc4	934	100	53	16.4	12,979	1,165,993	186	74.0	90.4
	Tpc5	1174	126	66	42.0	19,679	2,347,472	372	455.0	497.0
	Tpc6	1414	152	79	130.0	27,747	4,133,783	610	1930.0	2060.0
lazylist	Sac	254	29	23	0.5	1,396	24,658	2	<0.1	0.5
	Sar	435	56	39	4.0	4,521	210,799	24	0.8	4.8
	Sacr	505	65	39	5.2	5,435	280,223	47	0.7	5.9
	Saa	543	69	48	8.7	7,120	424,579	80	2.1	10.8
	Saacr	747	97	58	11.7	9,233	504,304	81	3.5	15.2
	Sacr2	1071	141	81	16.3	14,364	555,692	93	11.6	27.9
	Sarr	842	114	67	103.0	15,941	1,731,774	318	47.3	150.3
	S1	821	107	56	18.2	13,610	1,201,727	186	66.4	84.6
	Saaarr	1183	158	93	93.6	23,474	1,945,051	321	86.4	180.0
harris	Sac	406	34	14	0.4	1,882	25,456	2	<0.1	0.5
	Sar	575	51	18	1.4	3,670	85,824	12	0.1	1.6
	Saa	896	77	28	5.2	8,629	333,128	48	1.0	6.3
	Sacr	1349	125	32	28.0	16,411	1,157,264	187	6.9	34.9
snark	Da	760	77	51	4.1	5,229	230,292	24	0.8	4.9
	D0	810	89	65	19.9	9,254	1,075,792	121	9.3	29.2
	Db	980	107	75	47.4	12,278	1,815,494	191	49.6	97.0
	Dm	748	77	57	8.1	8,086	698,752	62	28.7	36.8
	Dq	748	77	57	7.0	8,015	710,252	62	123.0	130.0

Figure 8.4: Statistics about the consistency checks. For a given implementation (listed in Fig. 8.1) and test (listed in Fig. 8.2), we show (from left to right): the size of the unrolled code, the time required to create the SAT instance, the size of the SAT instance, the resources required by the SAT solver to refute the SAT instance, and the overall time required. All measurements were taken on a 3 GHz Pentium 4 desktop PC with 1GB of RAM, using zchaff [56] version 2004/11/15.

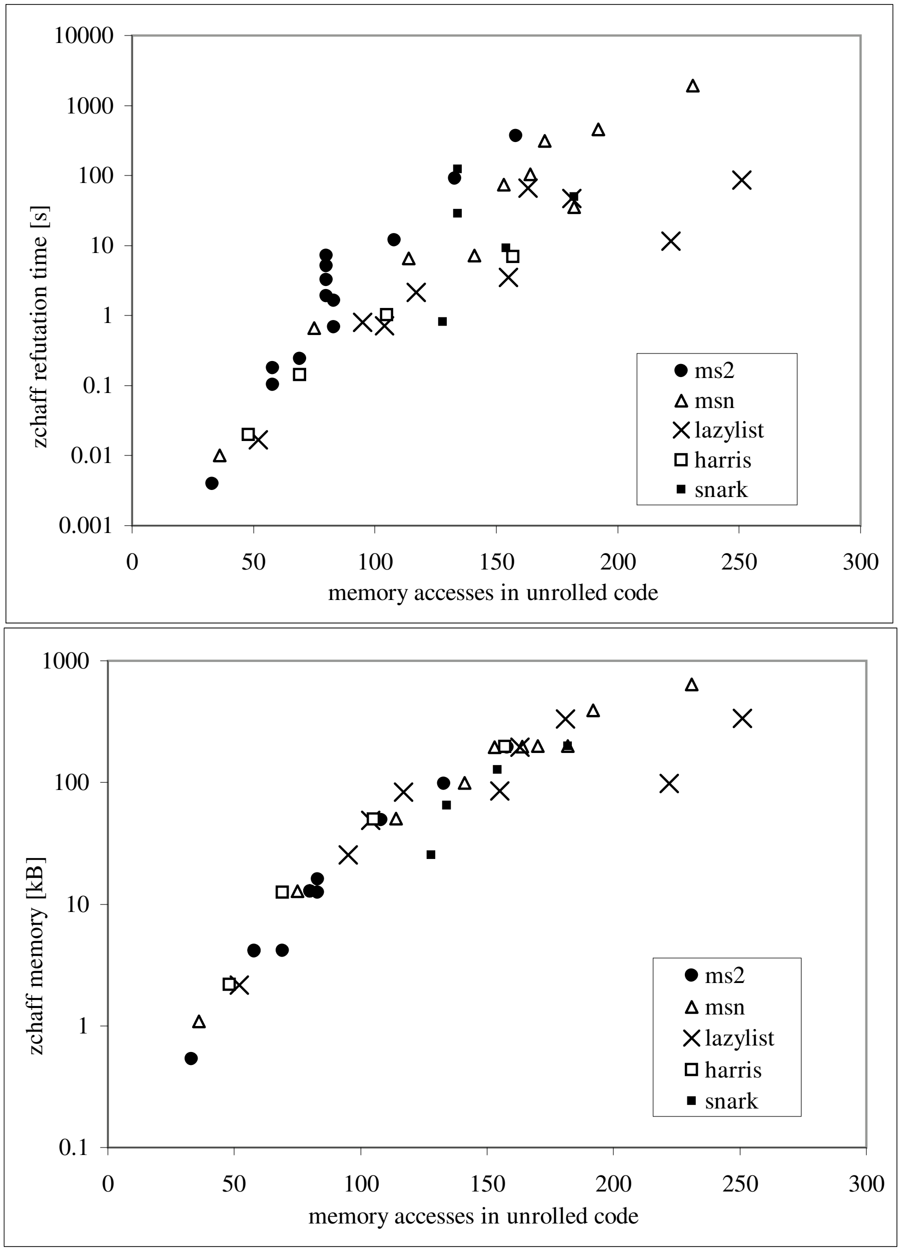


Figure 8.5: Time/Memory required by SAT solver (on Y axis, logarithmic scale) increases sharply with the number of memory accesses in the unrolled code (on X axis, linear scale). The data points represent the individual tests, grouped by implementation.

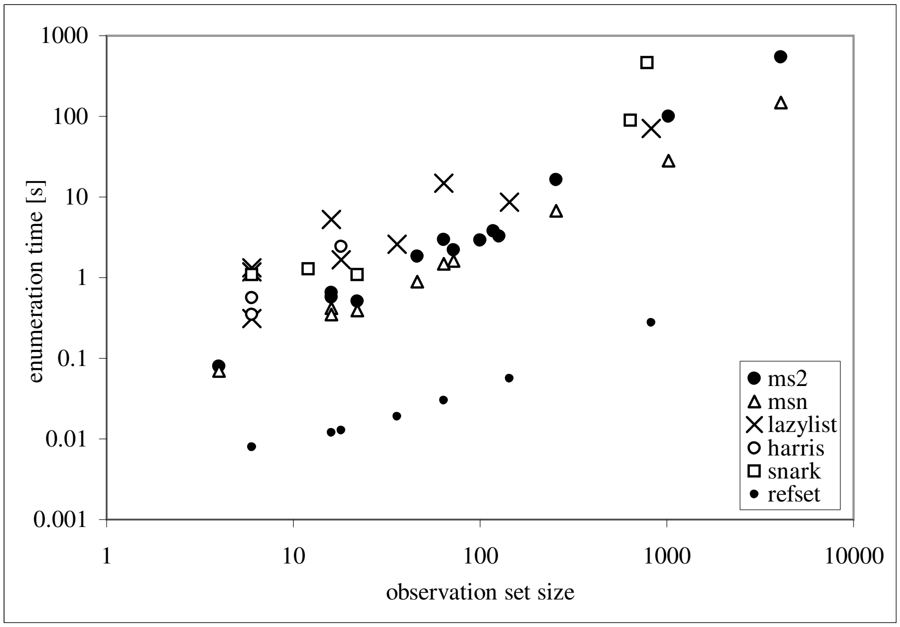


Figure 8.6: Time required to enumerate the observation set (Y axis), and the number of elements in the observation set (X axis). The data points represent the individual tests (Fig. 8.2), grouped by implementation.

To keep the trends visible, we do not include the time required for the lazy loop unrolling because it varies greatly between individual tests and implementations.

8.3.2 Specification Mining Statistics

We show information about the specification mining in Fig. 8.6. Most observation sets were quite small (less than 200 elements). The time spent for the specification mining averaged about a third of the total runtime (Fig. 8.7a). However, in practice, much less time is spent for observation set enumeration because (1) observation sets need not be recomputed after each change to the implementation, and (2) we can often compute observation sets much more efficiently by using a small, fast reference implementation (as shown by the data points for "refset").

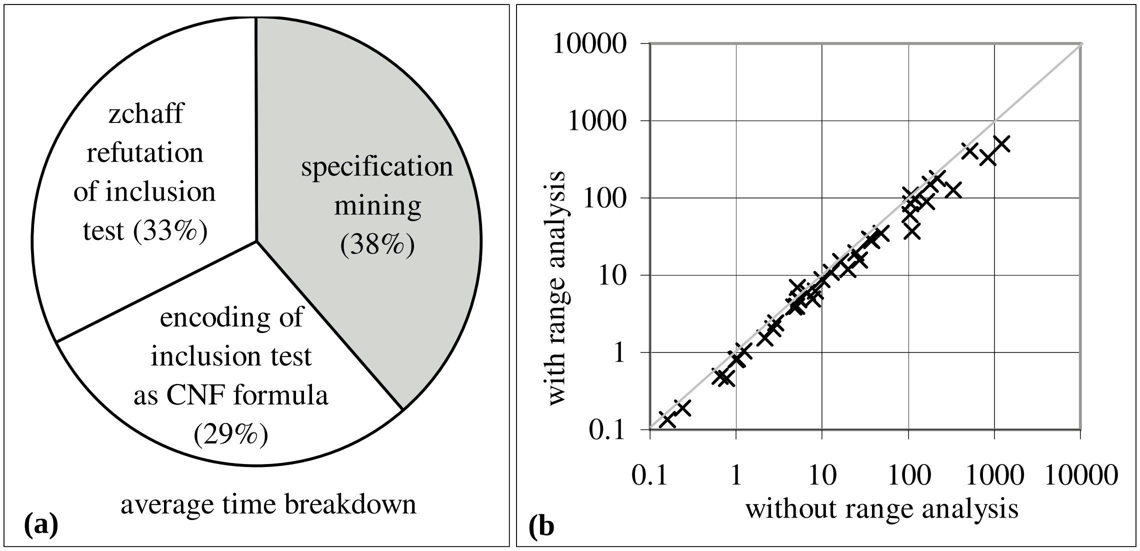


Figure 8.7: **(a)** average breakdown of total runtime. **(b)** impact of range analysis on runtime. The data points represent the individual tests.

8.3.3 Impact of Range Analysis

As described in Section 7.3.2, we perform a range analysis prior to the encoding to obtain data bounds, alias analysis, and range information. This information is used to improve the encoding by reducing the number of boolean variables. Fig. 8.7b shows the effect of the range analysis on runtime. On average, the performance improvement was about 42%. On larger tests (where we are most concerned), the positive impact is more pronounced (the tool finished up to 3× faster).

Although the performance improvement certainly justifies the use of a range analysis, it is less pronounced than we expected. A possible reason is that the SAT solver algorithm already produces effects that approximate a range analysis.

8.3.4 Memory Model Encoding

We compared the performance of different memory model encodings. Fig. 8.8 shows the SAT solving time and memory requirements as a function of the chosen tests

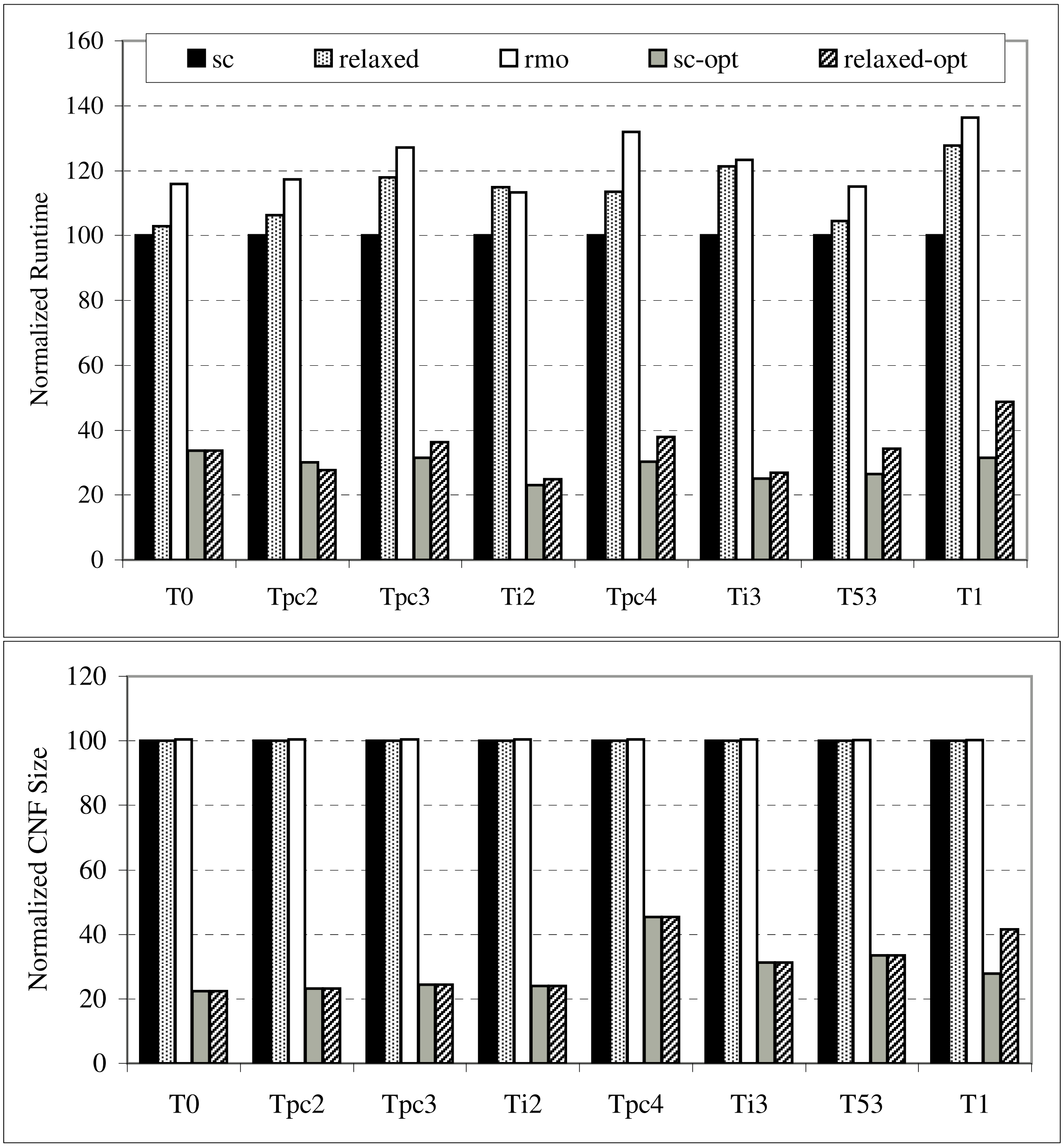


Figure 8.8: SAT solving time and memory requirements for various memory model encodings. The Y axis shows normalized time/space requirements; the X axis shows the tests, in order of increasing runtime.

and memory model encoding. To make it easier to spot trends, all tests are for the same implementation (the nonblocking queue). The Y axis shows the normalized SAT runtime (top) and memory footprint (bottom) for each test. The X axis shows the test used, in order of increasing runtime.

The first three of the five encodings shown are the models *SeqCons*, *Relaxed* and *RMO*, which we showed in Section 3.5, encoded automatically by the procedure described in Section 5.4. The last two encodings are optimized CNF encodings as we described in Section 7.3.4.

We make two observations:

- The three models *SeqCons*, *Relaxed* and *RMO* show significantly different performance. However, it appears that the relative speed is less a function of the weakness of the model (*Relaxed* is the weakest model, yet performs better than *RMO*) than a result of the complexity of the specification.

- The optimized encoding performs significantly better. Much of this improvement is probably due to the smaller number of memory order variables which leads to speedier SAT solving. Future work may address how to generalize these optimizations to arbitrary memory model specifications.

Chapter 9

Conclusions

Verifying concurrent data type implementations that make deliberate use of data races and memory ordering fences is challenging because of the many interleavings and counterintuitive instruction reorderings that need to be considered. Conventional automated verification tools for multithreaded programs are not sufficient because they make assumptions on the programming style (race-free programs) or the memory model (sequential consistency).

Our CheckFence verification method and tool address this challenge and provide a valuable aid to algorithm designers and implementors.

Our experiments confirm that the *CheckFence* method is very efficient at finding memory-model related bugs. All of the testcases we required to expose missing fences contained fewer than 200 memory accesses and took less than 10 minutes to verify. Furthermore, *CheckFence* proved to be useful to find algorithmic bugs that are not related to the memory model.

9.1 Contributions

Our main contributions are as follows:

- Our *CheckFence* implementation is the first model checker for C code that supports relaxed memory models.
- The *CheckFence* method does not require formal specifications or annotations, but mines a specification directly from the C code (either from the implementation under test, or from a reference implementation).
- *CheckFence* supports a reasonable subset of C, as required for typical implementations. This subset includes conditionals, loops, pointers, arrays, structures, function calls, locks, and dynamic memory allocation.

In addition to achieving our overall goal of applying automated model checking to the verification of concurrent data types, we solved a number of problems during our implementation of *CheckFence* whose solutions may be of interest in their own right:

- To work around the lack of precision and completeness of typical hardware memory model specifications, we developed a specification method for memory models and used it to formalize a number of memory models.
- To separate issues related to the C language semantics from the fundamental problem of encoding concurrent executions, we developed an intermediate language with formal syntax and semantics.
- To model executions of concurrent programs on axiomatically specified memory models, we developed a general encoding algorithm that expresses the executions as solutions to a formula, and provided a correctness proof.

9.2 Future Work

As we developed our *CheckFence* method and tool, we naturally discovered many new leads for future research. We distinguish two categories: Promising directions for research in the general area of concurrent data types and relaxed memory models, and research that may directly improve the *CheckFence* tool.

9.2.1 General Research Directions

- It seems that axiomatic encodings provide a novel, state-free way of modeling concurrent systems which may be of use for general asynchronous verification problems (such as protocol verification). We would like to compare such solutions to state-of-the art explicit or symbolic model checking.

- General reasoning techniques for relaxed memory models (such as a suitable generalization of linearizability) could allow us to go beyond testcases and prove sufficiency of fences for some programs.

- We would like to develop an algorithm that can insert fences automatically.

9.2.2 Improving *CheckFence*

- Recent progress in the area of theorem proving, SMT solving and SAT solving may be leveraged to improve the scalability of *CheckFence*. In particular, the use of SMT or SAT solvers with special support for relational variables may reduce the time and/or memory requirements.

- Our experimental results (Section 8.3.4) indicate that the hand-optimized encodings we use for sequential consistency and *Relaxed* can greatly improve the solving time. We would like to generalize these optimizations to arbitrary memory model specifications.

- We would like to explore how our specification format could be extended to cover applications beyond hardware-level models, such as the Java Language Memory Model [50].

Last but not least, we are curious to see how our *CheckFence* tool (which is freely available for download and includes the full source code) will be received and utilized by the community.

Appendix A

Correctness of Encoding

In this chapter, we prove that the encoding algorithm for unrolled LSL programs (Chapter 5) correctly captures the trace semantics of the program (Chapter 4): for each execution, there must be a solution to the formula, and vice versa. We prove each direction separately (Lemmas 52 and 53). The proofs are based on structural induction, using the inference rules for evaluation and for encoding.

The chapter is structured as follows:

- Because the technical details can get complicated if we try to fit them all into a single proof, we establish some basic invariants of the encoding algorithm and state some invariance lemmas in Section A.1.
- In Section A.2 we describe how a state of the encoder corresponds to an execution, by defining a correspondence relation (similar to a simulation relation).
- In Section A.3 we state the core lemmas.
- In Section A.4 we prove the core lemmas.

A.1 Basic Invariants

Our encoding algorithm is designed to maintain a number of invariants. We capture them in this section, in the form of the following technical lemma.

Definition 46 *We call a triple* $(\gamma, \Delta, \overline{\boldsymbol{t}})$ *a* valid encoder state *if it satisfies all of the following conditions*

1. *for all* e *we have* $FV(\gamma_e) \subset \overline{\boldsymbol{t}}.V$
2. *for all* e *and* r, $FV(\Delta_e(r)) \subset \overline{\boldsymbol{t}}.V$
3. *for all valuations* ν *satisfying* $\operatorname{dom}\nu \supset \overline{\boldsymbol{t}}.V$, *there is at most one* e *such that* $[\![\gamma_e]\!]^\nu = \mathit{true}$

Definition 47 *A valid encoder state* $(\gamma, \Delta, \overline{\boldsymbol{t}})$ *is called* void *for a valuation* ν *if*

1. $\operatorname{dom}\nu \supset \overline{\boldsymbol{t}}.V$ *and*
2. $[\![\gamma_e]\!]^\nu = \mathit{false}$ *for all* e

Lemma 48 *If we have a derivation*

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \overline{\boldsymbol{t}}'),$$

and $(\gamma, \Delta, \overline{\boldsymbol{t}})$ *is a valid encoder state, then all of the following hold*

(i) $(\gamma', \Delta', \overline{\boldsymbol{t}}')$ *is a valid encoder state*

(ii) $\overline{\boldsymbol{t}}.V \subset \overline{\boldsymbol{t}}'.V$

(iii) for all e *we have* $FV(\gamma'_e) \subset \overline{\boldsymbol{t}}'.V$

(iv) for all e *and* r *we have* $FV(\Delta'_e(r)) \subset \overline{\boldsymbol{t}}'.V$

(v) if ν *is a valuation with* $\operatorname{dom}\nu \supset \overline{\boldsymbol{t}}'.V$ *and* $(\gamma, \Delta, \overline{\boldsymbol{t}})$ *is void for* ν, *then* $(\gamma', \Delta', \overline{\boldsymbol{t}}')$ *is void for* ν.

(vi) if ν *is a valuation and* $\operatorname{dom}\nu \supset \overline{\boldsymbol{t}}'.V$, *then* $[\![\gamma_e]\!]^\nu = \mathit{true}$ *for at most one* e

(vii) if ν is a valuation with $\operatorname{dom}\nu \supset \overline{\boldsymbol{t}}'.V$ *and* $(\gamma, \Delta, \overline{\boldsymbol{t}})$ *is void for* ν*, then* $[\![\overline{\boldsymbol{t}}']\!]^\nu = [\![\overline{\boldsymbol{t}}]\!]^\nu$

Proof. By structural induction. The proof is thus a collection of induction cases, one for each inference rule. The cases (ECondThrow), (EAssign), (EStore), (ELoad) and (EFence) are axioms of the inference rule system, and need not use the induction hypothesis. (ELabel) uses the induction hypothesis once, and can do so directly. For (EComp), we need to apply induction twice; the first time works directly, but the second application requires a little more consideration. What makes it work is that the first induction tells us that $(\gamma', \Delta', \overline{\boldsymbol{t}}')$ is a valid encoding state, which in turn implies (by Def. 46) that $(\gamma'[\![e \mapsto \mathit{false}]\!]_{e \neq ok}, \Delta', \overline{\boldsymbol{t}}')$ is a valid encoding state also and lets us apply the induction to the premise on the right.

For each inference rule, we need to prove each individual claim (i), (ii), (iii), (iv), (v), (vi), (vii). For easier readability, we organize these proof components by the individual claims, rather than by the inference rules.

Claim (i). Follows from (iii), (iv), and (vi) which are proven below.

Claim (ii). (ECondThrow), (EAssign), (EStore) and (EFence) do not modify $\overline{\boldsymbol{t}}.V$. (ELoad) actually modifies $\overline{\boldsymbol{t}}.V$, adding a fresh variable. In (ELabel) we need to apply the induction hypothesis, which directly tells us that $\overline{\boldsymbol{t}}.V \subset \overline{\boldsymbol{t}}'.V$. For (EComp), we need to apply induction twice; this works because the first induction tells us that $(\gamma', \Delta', \overline{\boldsymbol{t}}')$ is a valid encoding state, which in turn implies (by Def. 46) that $(\gamma'[\![e \mapsto \mathit{false}]\!]_{e \neq ok}, \Delta', \overline{\boldsymbol{t}}')$ is a valid encoding state also and lets us apply the induction to the premise on the right.

Claim (iii). (EAssign),(EStore), (ELoad) and (EFence) do not modify γ and may only increase $\overline{\boldsymbol{t}}.V$ (as we know by (ii)), so the claim follows immediately. (ECondThrow) modifies γ, but the new formulae are built from subformulae whose free variables are contained in $\overline{\boldsymbol{t}}.V$, so the claim follows with (ii). For (ELabel) we first apply induction to the premise on the left, then use the same argument.

For (EComp) we apply induction twice (as discussed before under the proof of (ii)). Then we can use the same argument.

Claim (iv). (EStore) and (EFence) do not modify γ or $\bar{\boldsymbol{t}}.V$ so the claim follows immediately. (EAssign) constructs formulae from subformulae whose variables are contained in $\bar{\boldsymbol{t}}.V$. (ELoad) constructs formulae from subformulae whose variables are contained in $\bar{\boldsymbol{t}}.V$, as well as a new variable that it specifically adds. The other cases are similar to the proof of (iii)(use induction once or twice, apply the same argument).

Claim (v). (EAssign),(EStore), (ELoad) and (EFence) do not modify γ so the claim follows easily. In (ECondThrow), the property follows because $\llbracket \gamma_{ok} \rrbracket^{\nu} = \mathit{false}$. The same argument holds for (ELabel) after applying induction, which guarantees $\llbracket \gamma'_{ok} \rrbracket^{\nu} = \mathit{false}$. In (EComp), we use induction twice (as usual), checking that the encoder state remains void. We then know that for all e, both $\llbracket \gamma'_{e} \rrbracket^{\nu} = \mathit{false}$ and $\llbracket \gamma''_{e} \rrbracket^{\nu} = \mathit{false}$ from which we get the desired property.

Claim (vi). (EAssign),(EStore), (ELoad) and (EFence) do not modify γ so the claim follows easily. In (ECondThrow), the fields *ok* and e are modified, but in a way that guarantees the desired property. The same argument holds for (ELabel). In (EComp), the situation is not as obvious. To see why the property is guaranteed, we make a case distinction based on the value of $\llbracket \gamma'_{ok} \rrbracket^{\nu}$. If it is true, $\llbracket \gamma'_{e} \rrbracket^{\nu} = \mathit{false}$ for all $e \neq \mathit{ok}$ and we can deduce $\llbracket \gamma'''_{ok} \rrbracket^{\nu} = \llbracket \gamma''_{ok} \rrbracket^{\nu}$ for all e, and the property follows because we know γ'' has the desired property. If it is false, then we can apply induction claim (v): we know $(\gamma'\llbracket e \mapsto \mathit{false} \rrbracket_{e \neq ok}, \Delta', \bar{\boldsymbol{t}}')$ is void for ν, implying $\llbracket \Delta''_{e} \rrbracket^{\nu} = \mathit{false}$ for all e. From that, the desired property follows quite directly.

Claim (vii). (EAssign) and (ECondThrow) do not modify the trace, so the claim is immediate. For (EStore), (ELoad) and (EFence), we observe that all instructions added to the trace are guarded by γ_{ok}, so the claim follows with Lemma 49. For (ELabel), we use simple induction. For (EComp), we use induction claims (vii) and (v) on the left, and (vii)on the right.

□

We use the following lemma during the induction proof. It expresses how the *append* operation and the $[\![.]\!]^\nu$ commute.

Lemma 49 *Let* $\bar{\boldsymbol{t}} = (I, \prec, V, \overline{adr}, \overline{val}, \overline{guard})$ *be a symbolic trace, let* $i \in (\mathcal{I} \setminus I)$*, let* $a, v \in \mathcal{T}^{\text{value}}$ *and let* $g \in \mathcal{T}^{bool}$*. Then, for any valuation* ν *such that* $\operatorname{dom} \nu \supset V$*, the following is true:*

$$[\![\bar{\boldsymbol{t}}.append(i, a, v, g)]\!]^\nu = \begin{cases} [\![\bar{\boldsymbol{t}}]\!]^\nu.append(i, [\![a]\!]^\nu, [\![v]\!]^\nu) & \textit{if } [\![g]\!]^\nu = \textit{true} \\ [\![\bar{\boldsymbol{t}}]\!]^\nu & \textit{if } [\![g]\!]^\nu = \textit{false} \end{cases}$$

Proof. Directly from the definitions. □

A.2 Correspondence

We now describe how a state of the encoder corresponds to a trace, by defining a correspondence relation (similar to a simulation relation).

Definition 50 *For an encoder state* $(\gamma,\ \Delta,\ \bar{\boldsymbol{t}})$*, a trace* $\boldsymbol{t}$*, a register map* Γ*, an execution state* e*, and a valuation* ν*, we write*

$$(\gamma, \Delta, \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, e)$$

to express the conjunction of the following properties:

(a) $(\gamma,\ \Delta,\ \bar{\boldsymbol{t}})$ *is a valid encoder state*

(b) $\operatorname{dom} \nu \supset \bar{\boldsymbol{t}}.V$

(c) $[\![\bar{\boldsymbol{t}}]\!]^\nu = \boldsymbol{t}$

(d) $[\![\gamma_e]\!]^\nu = \textit{true}$

(e) for all $r \in \textit{Reg}$*, we have* $[\![\Delta_e(r)]\!]^\nu = \Gamma(r)$

The initial encoder state corresponds to the empty trace.

Lemma 51 *For all valuations ν, $(\gamma^0, \Delta^0, \overline{\boldsymbol{t}}^0) \rhd_\nu (\Gamma^0, \boldsymbol{t}^0, ok)$*

Proof. Directly from the definitions. □

A.3 Core Lemmas

We now state the core lemmas.

Lemma 52 *If the following condition holds*

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, ok) \tag{A.1}$$

and for some statement s, we have derivations for

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \overline{\boldsymbol{t}}') \qquad \textit{and} \qquad \Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e \tag{A.2}$$

then there exists a ν' that extends ν and satisfies $(\gamma', \Delta', \overline{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma', \boldsymbol{t}', e)$.

Proof. See Section A.4.2. □

Lemma 53 *If the following condition holds*

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, ok) \tag{A.3}$$

and for some statement s we have

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \overline{\boldsymbol{t}}') \tag{A.4}$$

$$\operatorname{dom} \nu \supset \overline{\boldsymbol{t}}'.V \tag{A.5}$$

then there exists a register map Γ', a trace $\boldsymbol{t}'$ and an execution state e such that

$$\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e \tag{A.6}$$

$$(\gamma', \Delta', \overline{\boldsymbol{t}}') \rhd_\nu (\Gamma', \boldsymbol{t}', e) \tag{A.7}$$

Proof. See Section A.4.1. □

Lemma 54 *If s is an unrolled program, and*

$$(\gamma^0, \Delta^0, \bar{\boldsymbol{t}}^0) \xrightarrow{s} (\gamma, \Delta, \bar{\boldsymbol{t}}), \qquad \textit{and} \qquad \Gamma^0, \boldsymbol{t}^0, s \Downarrow \Gamma, \boldsymbol{t}, e,$$

then there exists a valuation ν such that

$$\operatorname{dom} \nu = \bar{\boldsymbol{t}}.V, \qquad \textit{and} \qquad [\![\gamma_e]\!]^\nu = \textit{true}, \qquad \textit{and} \qquad [\![\bar{\boldsymbol{t}}]\!]^\nu = \boldsymbol{t}$$

Proof. The combination of Lemma 51 and Lemma 52 tells us that there exists a valuation ν such that

$$(\gamma, \Delta, \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, e) \tag{A.8}$$

This directly implies (by Def. 50) that $[\![\gamma_e]\!]^\nu = \textit{true}$ and $[\![\bar{\boldsymbol{t}}]\!]^\nu = \boldsymbol{t}$ as required. It also implies that $\operatorname{dom} \nu \supset \bar{\boldsymbol{t}}.V$. Because we know that γ, Δ and $\bar{\boldsymbol{t}}$ contain only variables in $\bar{\boldsymbol{t}}.V$ (by Lemma 48), we can restrict ν if needed such that $\operatorname{dom} \nu = \bar{\boldsymbol{t}}.V$, without affecting the validity of the other two claims. □

Lemma 55 *If s is an unrolled program and*

$$(\gamma^0, \Delta^0, \bar{\boldsymbol{t}}^0) \xrightarrow{s} (\gamma, \Delta, \bar{\boldsymbol{t}})$$

and ν is a valuation such that $\operatorname{dom} \nu \supset \bar{\boldsymbol{t}}.V$ and $[\![\gamma_e]\!]^\nu = \textit{true}$, then

$$\Gamma^0, \boldsymbol{t}^0, s \Downarrow \Gamma, [\![\bar{\boldsymbol{t}}]\!]^\nu, e$$

Proof. The combination of Lemma 51 and Lemma 53 tells us that there exists a register map Γ, a trace $\boldsymbol{t}$ and an execution state e' such that

$$\Gamma^0, \boldsymbol{t}^0, s \Downarrow \Gamma, \boldsymbol{t}, e' \tag{A.9}$$

$$(\gamma, \Delta, \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, e') \tag{A.10}$$

Now, by (A.10)(c) (see Def. 50), this implies

$$[\![\bar{\boldsymbol{t}}]\!]^\nu = \boldsymbol{t} \tag{A.11}$$

Moreover, by (A.10)(d), we know that $[\![\gamma_{e'}]\!]^{\nu} = \mathit{true}$. Applying Lemma 48(vi), we can deduce that

$$e = e' \tag{A.12}$$

The claim then follows from (A.9), (A.11) and (A.12). □

A.4 Core Proofs

A.4.1 Proof of Lemma 52

We perform a simultaneous structural induction over the derivations (A.2). Each case treats a combination of the last respective inference rules used, shown in a box. Following each box, we give the proof of the corresponding case.

$$\frac{(\gamma,\Delta,\bar{\boldsymbol{t}}) \xrightarrow{s} (\gamma',\Delta',\bar{\boldsymbol{t}}') \qquad (\gamma'[\![e \mapsto \mathit{false}]\!]_{e\neq ok},\Delta',\bar{\boldsymbol{t}}') \xrightarrow{s'} (\gamma'',\Delta'',\bar{\boldsymbol{t}}'')}{(\gamma,\Delta,\bar{\boldsymbol{t}}) \xrightarrow{s\,;s'} (\gamma''[\![e\mapsto \gamma'_e \vee \gamma''_e]\!]_{e\neq ok}, \Delta''[\![e\mapsto (\gamma'_e \;?\; \Delta'_e : \Delta''_e)]\!]_{e\neq ok}, \bar{\boldsymbol{t}}'')}\;(\text{E}\textsc{Comp})$$

$$\frac{\Gamma,\boldsymbol{t},s \Downarrow \Gamma',\boldsymbol{t}',ok \qquad \Gamma',\boldsymbol{t}',s' \Downarrow \Gamma'',\boldsymbol{t}'',e}{\Gamma,\boldsymbol{t},s\,;s' \Downarrow \Gamma'',\boldsymbol{t}'',e}\;(\textsc{Comp-ok})$$

For briefer reference, let us name the following expressions:

$$\gamma''' = \gamma''[\![e\mapsto \gamma'_e \vee \gamma''_e]\!]_{e\neq ok}$$
$$\Delta''' = \Delta''[\![e\mapsto (\gamma'_e \;?\; \Delta'_e : \Delta''_e)]\!]_{e\neq ok}$$

We now show how to find a valuation ν'' that extends ν from (A.1) and satisfies $(\gamma''',\Delta''',\bar{\boldsymbol{t}}'') \rhd_{\nu''} (\Gamma'',\boldsymbol{t}'',e)$ as claimed by the lemma.

First, we apply the induction to the hypotheses appearing on the left of the inference rules which gives us a valuation ν' that extends ν and satisfies

$$(\gamma',\Delta',\bar{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma',\boldsymbol{t}',ok) \tag{A.13}$$

This same valuation ν' also satisfies $(\gamma'\llbracket e \mapsto \mathit{false}\rrbracket_{e\neq \mathit{ok}}, \Delta', \bar{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma', \boldsymbol{t}', \mathit{ok})$ (directly from Def. 50). Therefore, we can apply the induction hypothesis once more, this time to the hypotheses appearing on the right. This gives us a valuation ν'' that extends ν' (and therefore ν) and satisfies

$$(\gamma'', \Delta'', \bar{\boldsymbol{t}}'') \rhd_{\nu''} (\Gamma'', \boldsymbol{t}'', e) \tag{A.14}$$

Now we can prove the individual pieces of $(\gamma''', \Delta''', \bar{\boldsymbol{t}}'') \rhd_{\nu''} (\Gamma'', \boldsymbol{t}'', e)$ as follows to complete this case:

(a) using Lemma 48(i)

(b) directly from (A.14)(b)

(c) $\llbracket \bar{\boldsymbol{t}}'' \rrbracket^{\nu''} = \boldsymbol{t}''$ by (A.14)(c)

(d) $\llbracket \gamma'''_e \rrbracket^{\nu''} = \begin{cases} \llbracket \gamma''_e \rrbracket^{\nu''} & \text{if } e = \mathit{ok} \\ \llbracket \gamma'_e \vee \gamma''_e \rrbracket^{\nu''} & \text{if } e \neq \mathit{ok} \end{cases}$

In either case, we can easily conclude $\llbracket \gamma'''_e \rrbracket^{\nu''} = \mathit{true}$ because we know that $\llbracket \gamma''_e \rrbracket^{\nu''} = \mathit{true}$ by (A.14)(d).

(e) for all $r \in \mathit{Reg}$, we have

$\llbracket \Delta'''_e(r) \rrbracket^{\nu''} = \begin{cases} \llbracket \Delta''_e(r) \rrbracket^{\nu''} & \text{if } e = \mathit{ok} \\ \llbracket \gamma'_e \mathbin{?} \Delta'_e(r) : \Delta''_e(r) \rrbracket^{\nu''} & \text{if } e \neq \mathit{ok} \end{cases}$

By (A.13)(d), we know that $\llbracket \gamma'_{\mathit{ok}} \rrbracket^{\nu'}$ is true. Moreover, because of (A.13)(a), we know $(\gamma', \Delta', \bar{\boldsymbol{t}}')$ is a valid encoder state and therefore $\llbracket \gamma'_e \rrbracket^{\nu'} = \mathit{false}$ for all $e \neq \mathit{ok}$ (by Lemma 48(vi)). This implies $\llbracket \gamma'_e \rrbracket^{\nu''} = \mathit{false}$ for all $e \neq \mathit{ok}$ (because ν'' extends ν'). Therefore, we can see that in either case ($e = \mathit{ok}$ or $e \neq \mathit{ok}$) we have $\llbracket \Delta'''_e(r) \rrbracket^{\nu''} = \llbracket \Delta''_e(r) \rrbracket^{\nu''} \overset{(A.14)}{=} \Gamma''(r)$

$$\frac{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \bar{\boldsymbol{t}}') \qquad (\gamma'\llbracket e \mapsto \mathit{false}\rrbracket_{e\neq \mathit{ok}}, \Delta', \bar{\boldsymbol{t}}') \xrightarrow{s'} (\gamma'', \Delta'', \bar{\boldsymbol{t}}'')}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{s\,;s'} (\gamma''\llbracket e \mapsto \gamma'_e \vee \gamma''_e\rrbracket_{e\neq \mathit{ok}}, \Delta''\llbracket e \mapsto (\gamma'_e \mathbin{?} \Delta'_e : \Delta''_e)\rrbracket_{e\neq \mathit{ok}}, \bar{\boldsymbol{t}}'')} \ (\textsc{EComp})$$

$$\frac{\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e \qquad e \neq \mathit{ok}}{\Gamma, \boldsymbol{t}, s\,; s' \Downarrow \Gamma', \boldsymbol{t}', e} \ (\textsc{Comp-skip})$$

For briefer reference, let us name the following expressions:

$$\gamma''' = \gamma''[\![e \mapsto \gamma'_e \vee \gamma''_e]\!]_{e \neq ok}$$
$$\Delta''' = \Delta''[\![e \mapsto (\gamma'_e \; ? \; \Delta'_e : \Delta''_e)]\!]_{e \neq ok}$$

We now show how to find a valuation ν'' that extends ν from (A.1) and satisfies $(\gamma''', \Delta''', \bar{\boldsymbol{t}}'') \rhd_{\nu''} (\Gamma', \boldsymbol{t}', e)$ as claimed by the lemma.

First, we apply the induction to the hypotheses appearing on the left of the inference rules which gives us a valuation ν' extending ν such that

$$(\gamma', \Delta', \bar{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma', \boldsymbol{t}', e) \tag{A.15}$$

This implies that $[\![\gamma'_e]\!]^{\nu'} = \mathit{true}$ (by (A.15)(d)) and therefore $[\![\gamma'_{ok}]\!]^{\nu'} = \mathit{false}$ (by Lemma 48(vi)). Thus, $(\gamma'[\![e \mapsto \mathit{false}]\!]_{e \neq ok}, \Delta', \bar{\boldsymbol{t}}')$ is void for ν. Now, by assigning arbitrary values to the variables in $(\bar{\boldsymbol{t}}''.V \setminus \operatorname{dom} \nu')$, we can extend ν' to a valuation ν'' such that

$$\operatorname{dom} \nu'' \supset \bar{\boldsymbol{t}}''.V \tag{A.16}$$

Then by Lemma 48(vii),

$$[\![\bar{\boldsymbol{t}}'']\!]^{\nu''} = [\![\bar{\boldsymbol{t}}']\!]^{\nu'} \tag{A.17}$$

Now we can prove the individual pieces of $(\gamma''', \Delta''', \bar{\boldsymbol{t}}'') \rhd_{\nu''} (\Gamma', \boldsymbol{t}', e)$ as follows to complete this case:

(a) using Lemma 48(i)

(b) by (A.16)

(c) $[\![\bar{\boldsymbol{t}}'']\!]^{\nu''} \overset{(A.17)}{=} [\![\bar{\boldsymbol{t}}']\!]^{\nu'} \overset{(A.15)}{=} \boldsymbol{t}'$

(d) $[\![\gamma'''_e]\!]^{\nu''} = [\![\gamma'_e \vee \gamma''_e]\!]^{\nu''} = \underbrace{[\![\gamma'_e]\!]^{\nu''}}_{=[\![\gamma'_e]\!]^{\nu'}=\mathit{true}} \vee [\![\gamma''_e]\!]^{\nu''} = \mathit{true}$

(e) for all $r \in \mathit{Reg}$, we have $[\![\Delta'''_e(r)]\!]^{\nu''} = [\![(\gamma'_e \; ? \; \Delta'_e : \Delta''_e)(r)]\!]^{\nu''}$
Now, we know $[\![\gamma'_e]\!]^{\nu''} = [\![\gamma'_e]\!]^{\nu'} = \mathit{true}$ and therefore
$[\![\Delta'''_e(r)]\!]^{\nu''} = [\![\Delta'_e(r)]\!]^{\nu''} \overset{(A.15)}{=} \Gamma'(r)$

$$\frac{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \bar{\boldsymbol{t}}') \qquad \gamma'' = \gamma'[\mathit{ok} \mapsto \gamma'_{\mathit{ok}} \vee \gamma'_{\mathit{break}\,l}][\mathit{break}\,l \mapsto \mathit{false}]}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{l:s} (\gamma'', \Delta'[\mathit{ok} \mapsto (\gamma'_{\mathit{break}\,l} \;?\; \Delta'_{\mathit{break}\,l} : \Delta'_{\mathit{ok}})], \bar{\boldsymbol{t}}')} \ (\text{E}\textsc{Label})$$

$$\frac{\Gamma, \boldsymbol{t}, s \ \Downarrow\ \Gamma', \boldsymbol{t}', e \qquad e \notin \{\mathit{break}\,l, \mathit{continue}\,l\}}{\Gamma, \boldsymbol{t}, l : s \ \Downarrow\ \Gamma', \boldsymbol{t}', e} \ (\textsc{Label})$$

For briefer reference, let us name the following expression:

$$\Delta'' = \Delta'[\mathit{ok} \mapsto (\gamma'_{\mathit{break}\,l} \;?\; \Delta'_{\mathit{break}\,l} : \Delta'_{\mathit{ok}})]$$

We now show how to find a valuation ν' that extends ν from (A.1) and satisfies $(\gamma'', \Delta'', \bar{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma', \boldsymbol{t}', e)$ as claimed by the lemma.

First, we apply the induction to the hypotheses appearing in the inference rules which gives us a valuation ν' extending ν such that

$$(\gamma', \Delta', \bar{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma', \boldsymbol{t}', e) \tag{A.18}$$

Now we can prove the individual pieces of $(\gamma'', \Delta'', \bar{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma', \boldsymbol{t}', e)$ as follows to complete this case:

(a) using Lemma 48(i)

(b) by (A.18)(b)

(c) by (A.18)(c)

(d) $[\![\gamma''_e]\!]^{\nu'} = \begin{cases} [\![\gamma'_{\mathit{ok}} \vee \gamma'_{\mathit{break}\,l}]\!]^{\nu'} & \text{if } e = \mathit{ok} \\ [\![\gamma'_e]\!]^{\nu'} & \text{otherwise} \end{cases}$

In either case, we can easily conclude $[\![\gamma''_e]\!]^{\nu'} = \mathit{true}$ because we know that $[\![\gamma'_e]\!]^{\nu'} = \mathit{true}$ by (A.18)(d).

(e) We do a case distinction on $e = \mathit{ok}$. If $e \neq \mathit{ok}$, then $\Delta''_e = \Delta'_e$ and the claim follows from (A.18)(e). If $e = \mathit{ok}$, then $[\![\gamma_{\mathit{ok}}]\!]^{\nu'} = \mathit{true}$ by (A.18)(d), and thus $[\![\gamma'_{\mathit{break}\,l}]\!]^{\nu'} = \mathit{false}$ by Lemma 48(vi). This implies that $[\![\Delta''_e(r)]\!]^{\nu'} = [\![\Delta'_{\mathit{ok}}(r)]\!]^{\nu'}$ for all r and the claim follows with (A.18)(e).

$$\frac{(\gamma,\Delta,\overline{\boldsymbol{t}}) \xrightarrow{s} (\gamma',\Delta',\overline{\boldsymbol{t}}') \qquad \gamma''=\gamma'[\mathit{ok}\mapsto \gamma'_{\mathit{ok}}\vee\gamma'_{\mathit{break}\,l}][\mathit{break}\;l\mapsto \mathit{false}]}{(\gamma,\Delta,\overline{\boldsymbol{t}}) \xrightarrow{l:s} (\gamma'',\Delta'[\mathit{ok}\mapsto(\gamma'_{\mathit{break}\,l}\;?\;\Delta'_{\mathit{break}\,l}:\Delta'_{\mathit{ok}})],\overline{\boldsymbol{t}}')}\;(\text{E}\textsc{Label})$$

$$\frac{\Gamma,\boldsymbol{t},s \;\Downarrow\; \Gamma',\boldsymbol{t}',\mathit{break}\;l}{\Gamma,\boldsymbol{t},l:s \;\Downarrow\; \Gamma',\boldsymbol{t}',\mathit{ok}}\;(\textsc{Label-break})$$

For briefer reference, let us name the following expression:

$$\Delta''=\Delta'[\mathit{ok}\mapsto(\gamma'_{\mathit{break}\,l}\;?\;\Delta'_{\mathit{break}\,l}:\Delta'_{\mathit{ok}})]$$

We now show how to find a valuation ν' that extends ν from (A.1) and satisfies $(\gamma'',\Delta'',\overline{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma',\boldsymbol{t}',\mathit{ok})$ as claimed by the lemma.

First, we apply the induction to the hypotheses appearing in the inference rules which gives us a valuation ν' extending ν such that

$$(\gamma',\Delta',\overline{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma',\boldsymbol{t}',\mathit{break}\;l) \tag{A.19}$$

Now we can prove the individual pieces of $(\gamma'',\Delta'',\overline{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma',\boldsymbol{t}',\mathit{ok})$ as follows to complete this case:

(a) using Lemma 48(i)

(b) by (A.19)(b)

(c) by (A.19)(c)

(d) $[\![\gamma''_{\mathit{ok}}]\!]^{\nu'}=[\![\gamma'_{\mathit{ok}}\vee\gamma'_{\mathit{break}\,l}]\!]^{\nu'}=\mathit{true}$
(because we know $[\![\gamma'_{\mathit{break}\,l}]\!]^{\nu'}=\mathit{true}$ by (A.19)(d)).

(e) $[\![\Delta''_{\mathit{ok}}(r)]\!]^{\nu'}=[\![(\gamma'_{\mathit{break}\,l}\;?\;\Delta'_{\mathit{break}\,l}:\Delta'_{\mathit{ok}})(r)]\!]^{\nu'}=[\![\Delta'_{\mathit{break}\,l}(r)]\!]^{\nu'}$ and the claim follows from (A.19)(e).

$$\frac{\gamma' = \gamma[ok \mapsto \gamma_{ok} \wedge \neg\phi][e \mapsto \gamma_{ok} \wedge \phi] \qquad \phi = \neg(\Delta_{ok}(r) = 0)}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{\textbf{if}\,(r)\,\textbf{throw}(e)} (\gamma', \Delta[e \mapsto \Delta_{ok}], \bar{\boldsymbol{t}})}\ (\textsc{ECondThrow})$$

$$\frac{\Gamma(r) \neq 0 \qquad \Gamma, \boldsymbol{t}, \textbf{throw}(e) \Downarrow \Gamma, \boldsymbol{t}, e}{\Gamma, \boldsymbol{t}, \textbf{if}\,(r)\,\textbf{throw}(e) \Downarrow \Gamma, \boldsymbol{t}, e}\ (\textsc{If-so})(\textsc{throw})$$

By assumption, we know that

$$(\gamma, \Delta, \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, ok) \tag{A.20}$$

To prove this case, we show that for the same ν,

$$(\gamma', \Delta[e \mapsto \Delta_{ok}], \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, e)$$

because each individual claim is satisfied:

(a) using Lemma 48(i)

(b) by (A.20)(b)

(c) by (A.20)(c)

(d) First, observe that $[\![\phi]\!]^\nu = [\![\neg(\Delta_{ok}(r) = 0)]\!]^\nu = \mathit{true}$ because $[\![\Delta_{ok}(r)]\!]^\nu = \Gamma(r)$ by (A.20)(e), and $\Gamma(r) \neq 0$ by the first prerequisite of (If-so).
Now, $[\![\gamma'_e]\!]^\nu = [\![\gamma_{ok} \wedge \phi]\!]^\nu = [\![\gamma_{ok}]\!]^\nu = \mathit{true}$ because of (A.20)(d).
Note that $e \neq ok$ because **throw**(ok) is not allowed (Table 4.1).

(e) $[\![(\Delta[e \mapsto \Delta_{ok}])_e(r)]\!]^\nu = [\![\Delta_{ok}]\!]^\nu(r) = \Gamma(r)$ because of (A.20)(e).

$$\frac{\gamma' = \gamma[ok \mapsto \gamma_{ok} \wedge \neg\phi][e \mapsto \gamma_{ok} \wedge \phi] \qquad \phi = \neg(\Delta_{ok}(r) = 0)}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{\textbf{if}\,(r)\,\textbf{throw}(e)} (\gamma', \Delta[e \mapsto \Delta_{ok}], \bar{\boldsymbol{t}})}\ (\textsc{ECondThrow})$$

$$\frac{\Gamma(r) = 0}{\Gamma, \boldsymbol{t}, \textbf{if}\,(r)\,\textbf{throw}(e) \Downarrow \Gamma, \boldsymbol{t}, ok}\ (\textsc{If-not})$$

By assumption, we know that

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, \mathit{ok}) \tag{A.21}$$

To prove this case, we show that for the same ν,

$$(\gamma', \Delta[e \mapsto \Delta_{\mathit{ok}}], \overline{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, \mathit{ok})$$

because each individual claim is satisfied:

(a) using Lemma 48(i)

(b) by (A.21)(b)

(c) by (A.21)(c)

(d) First, observe that $[\![\phi]\!]^\nu = [\![\neg(\Delta_{\mathit{ok}}(r) = 0)]\!]^\nu = \mathit{false}$ because $[\![\Delta_{\mathit{ok}}(r)]\!]^\nu = \Gamma(r)$ by (A.21)(e), and $\Gamma(r) = 0$ by the prerequisite of (If-not).
Now, $[\![\gamma'_{\mathit{ok}}]\!]^\nu = [\![\gamma_{\mathit{ok}} \wedge \neg\phi]\!]^\nu = [\![\gamma_{\mathit{ok}}]\!]^\nu = \mathit{true}$ because of (A.21)(d).
Note that $e \neq \mathit{ok}$ because **throw**(ok) is not allowed (Table 4.1).

(e) $[\![(\Delta[e \mapsto \Delta_{\mathit{ok}}])_{\mathit{ok}}(r)]\!]^\nu = [\![\Delta_{\mathit{ok}}]\!]^\nu(r) = \Gamma(r)$ because of (A.21)(e).

$$\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \bot, \bot, \gamma_{\mathit{ok}})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{\textbf{fence}_i} (\gamma, \Delta, \overline{\boldsymbol{t}}')} \quad (\text{EFence})$$

$$\frac{\boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \bot, \bot)}{\Gamma, \boldsymbol{t}, \textbf{fence}_i \Downarrow \Gamma, \boldsymbol{t}', \mathit{ok}} \quad (\text{Fence-id})$$

By assumption, we know that

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, \mathit{ok}) \tag{A.22}$$

To prove this case, we show that for the same ν,

$$(\gamma, \Delta, \overline{\boldsymbol{t}}') \rhd_\nu (\Gamma, \boldsymbol{t}', \mathit{ok})$$

because each individual claim is satisfied:

(a) using Lemma 48(i)

(b) By Def. 35,

$$\overline{\boldsymbol{t}}'.V = \overline{\boldsymbol{t}}.V \cup FV(i) \cup FV(\bot) \cup FV(\bot) \cup FV(\gamma_{ok})$$

By Lemma 48(iii), $FV(\gamma_{ok}) \subset \boldsymbol{t}.V$ and thus $\boldsymbol{t}'.V = \boldsymbol{t}.V$. Therefore, the claim follows from (A.22)(b).

(c) First, note that $[\![\gamma_{ok}]\!]^{\nu} = \mathit{true}$ by (A.22)(d). Now

$$[\![\overline{\boldsymbol{t}}']\!]^{\nu} = [\![\overline{\boldsymbol{t}}.\mathit{append}(i, \bot, \bot, \gamma_{ok})]\!]^{\nu} \overset{\text{Lemma 49}}{=} [\![\overline{\boldsymbol{t}}]\!]^{\nu}.\mathit{append}(i, [\![\bot]\!]^{\nu}, [\![\bot]\!]^{\nu})$$

Because we know that $[\![\overline{\boldsymbol{t}}]\!]^{\nu} = \boldsymbol{t}$ by (A.22)(c), the latter is equal to

$$= \boldsymbol{t}.\mathit{append}(i, \bot, \bot) = \boldsymbol{t}'$$

(d) by (A.22)(d)

(e) by (A.22)(e)

$$\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{ok}(r'), \Delta_{ok}(r), \gamma_{ok})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{*r' :=_i r} (\gamma, \Delta, \overline{\boldsymbol{t}}')} \quad (\text{EStore})$$

$$\frac{\boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), \Gamma(r))}{\Gamma, \boldsymbol{t}, *r' :=_i r \;\Downarrow\; \Gamma, \boldsymbol{t}', \mathit{ok}} \quad (\text{Store-id})$$

By assumption, we know that

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_{\nu} (\Gamma, \boldsymbol{t}, \mathit{ok}) \tag{A.23}$$

To prove this case, we show that for the same ν,

$$(\gamma, \Delta, \overline{\boldsymbol{t}}') \rhd_{\nu} (\Gamma, \boldsymbol{t}', \mathit{ok})$$

because each individual claim is satisfied:

(a) using Lemma 48(i)

(b) By Def. 35,

$$\overline{\boldsymbol{t}}'.V = \overline{\boldsymbol{t}}.V \cup FV(i) \cup FV(\Delta_{ok}(r')) \cup FV(\Delta_{ok}(r)) \cup FV(\gamma_{ok})$$

By Lemma 48(iii) and Lemma 48(iv), we can conclude $\boldsymbol{t}'.V = \boldsymbol{t}.V$. Therefore, the claim follows from (A.23)(b).

(c) First, note that $\llbracket \gamma_{ok} \rrbracket^{\nu} = \mathit{true}$ by (A.23)(d). Now

$$\begin{aligned} \llbracket \overline{\boldsymbol{t}}' \rrbracket^{\nu} &= \llbracket \overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{ok}(r'), \Delta_{ok}(r), \gamma_{ok}) \rrbracket^{\nu} \\ &\overset{\text{Lemma 49}}{=} \llbracket \overline{\boldsymbol{t}} \rrbracket^{\nu}.\mathit{append}(i, \llbracket \Delta_{ok}(r') \rrbracket^{\nu}, \llbracket \Delta_{ok}(r) \rrbracket^{\nu}) \end{aligned}$$

Because we know that $\llbracket \overline{\boldsymbol{t}} \rrbracket^{\nu} = \boldsymbol{t}$ by (A.23)(c), and because $\llbracket \Delta_{ok}(r') \rrbracket^{\nu} = \Gamma(r')$ and $\llbracket \Delta_{ok}(r) \rrbracket^{\nu} = \Gamma(r)$ (by (A.23)(e)), the latter is equal to

$$= \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), \Gamma(r)) = \boldsymbol{t}'$$

(d) by (A.23)(d)

(e) by (A.23)(e)

$$\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{ok}(r'), D_i, \gamma_{ok})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{r:=_i *r'} (\gamma, \Delta[ok \mapsto \Delta_{ok}[r \mapsto D_i]], \overline{\boldsymbol{t}}')} \quad (\text{ELoad})$$

$$\frac{v \in \mathit{Val} \qquad \boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), v)}{\Gamma, \boldsymbol{t}, r :=_i *r' \Downarrow \Gamma[r \mapsto v], \boldsymbol{t}', ok} \quad (\text{Load-id})$$

For briefer reference, let us name the following expressions:

$$\Delta' = \Delta[ok \mapsto \Delta_{ok}[r \mapsto D_i]]$$

$$\Gamma' = \Gamma[r \mapsto v]$$

By assumption, we know that

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_{\nu} (\Gamma, \boldsymbol{t}, \mathit{ok}) \tag{A.24}$$

To prove this case, we show that for the valuation $\nu' = \nu[D_i \mapsto v]$,

$$(\gamma, \Delta', \overline{\boldsymbol{t}}') \rhd_{\nu'} (\Gamma', \boldsymbol{t}', \mathit{ok})$$

because each individual claim is satisfied:

(a) using Lemma 48(i)

(b) By Def. 35,

$$\overline{\boldsymbol{t}}'.V = \overline{\boldsymbol{t}}.V \cup FV(i) \cup FV(\Delta_{\mathit{ok}}(r')) \cup FV(D_i) \cup FV(\gamma_{\mathit{ok}})$$

By Lemma 48(iii) and Lemma 48(iv), we can conclude $\boldsymbol{t}'.V = \boldsymbol{t}.V \cup \{\gamma_{\mathit{ok}}\}$. Therefore, the claim follows from (A.24)(b) and the fact that we explicitly assign γ_{ok} in ν'

(c) First, note that $[\![\gamma_{\mathit{ok}}]\!]^{\nu'} = [\![\gamma_{\mathit{ok}}]\!]^{\nu} = \mathit{true}$ by (A.24)(d). Now

$$[\![\overline{\boldsymbol{t}}']\!]^{\nu'} = [\![\overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{\mathit{ok}}(r'), D_i, \gamma_{\mathit{ok}})]\!]^{\nu'}$$
$$\overset{\text{Lemma 49}}{=} [\![\overline{\boldsymbol{t}}]\!]^{\nu'}.\mathit{append}(i, [\![\Delta_{\mathit{ok}}(r')]\!]^{\nu'}, [\![D_i]\!]^{\nu'})$$

Because we know that $[\![\overline{\boldsymbol{t}}]\!]^{\nu} = \boldsymbol{t}$ by (A.24)(c), and because $[\![\Delta_{\mathit{ok}}(r')]\!]^{\nu} = \Gamma(r')$ (by (A.24)(e)) and $[\![D_i]\!]^{\nu'} = v$, the latter is equal to

$$= \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), v) = \boldsymbol{t}'$$

(d) by (A.24)(d)

(e) for an arbitrary register r', do a case distinction on $r' = r$:
(1) If $r' \neq r$, then $[\![\Delta'_{\mathit{ok}}(r')]\!]^{\nu'} = [\![\Delta_{\mathit{ok}}(r')]\!]^{\nu'} = \Gamma(r) = \Gamma'(r)$ by (A.24)(e).
(2) If $r = r'$, then $[\![\Delta'_{\mathit{ok}}(r)]\!]^{\nu'} = [\![D_i]\!]^{\nu'} = v = \Gamma'(r)$.

$$\frac{\Delta' = \Delta[ok \mapsto \Delta_{ok}[r \mapsto f(\Delta_{ok}(r_1), \ldots, \Delta_{ok}(r_k))]]}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{r:=(f\ r_1\ \ldots\ r_k)} (\gamma, \Delta', \bar{\boldsymbol{t}})} \quad (\text{EAssign})$$

$$\frac{v = f(\Gamma(r_1), \ldots, \Gamma(r_k))}{\Gamma, \boldsymbol{t}, r := (f\ r_1\ \ldots\ r_k) \Downarrow \Gamma[r \mapsto v], \boldsymbol{t}, ok} \quad (\text{Assign})$$

For briefer reference, let us name the following expression:

$$\Gamma' = \Gamma[r \mapsto v]$$

By assumption, we know that

$$(\gamma, \Delta, \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, ok) \tag{A.25}$$

To prove this case, we show that for the same valuation ν,

$$(\gamma, \Delta', \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma', \boldsymbol{t}, ok)$$

because each individual claim is satisfied:

(a) using Lemma 48(i)

(b) by (A.25)(b)

(c) by (A.25)(c)

(d) by (A.25)(d)

(e) for an arbitrary register r', do a case distinction on $r' = r$:

(1) If $r' \neq r$, then $[\![\Delta'_{ok}(r')]\!]^\nu = [\![\Delta_{ok}(r')]\!]^\nu = \Gamma(r) = \Gamma'(r)$ by (A.25)(e).

(2) If $r = r'$, then

$$[\![\Delta'_{ok}(r)]\!]^\nu = [\![f(\Delta_{ok}(r_1), \ldots, \Delta_{ok}(r_k))]\!]^\nu = f([\![\Delta_{ok}(r_1)]\!]^\nu, \ldots, [\![\Delta_{ok}(r_k)]\!]^\nu)$$

Because we know that $[\![\Delta_{ok}(r_i)]\!]^\nu = \Gamma(r_i)$ by (A.23)(e), the latter is equal to

$$= f(\Gamma(r_1), \ldots, \Gamma(r_k)) = \Gamma'(r)$$

A.4.2 Proof of Lemma 53

We perform a structural induction over the derivation for (A.4). We start each case by showing the last used inference rule in a box.

$$\boxed{\dfrac{(\gamma,\Delta,\overline{\boldsymbol{t}}) \xrightarrow{s} (\gamma',\Delta',\overline{\boldsymbol{t}}') \qquad (\gamma'[\![e \mapsto \mathit{false}]\!]_{e\neq ok},\Delta',\overline{\boldsymbol{t}}') \xrightarrow{s'} (\gamma'',\Delta'',\overline{\boldsymbol{t}}'')}{(\gamma,\Delta,\overline{\boldsymbol{t}}) \xrightarrow{s\,;s'} (\gamma''[\![e \mapsto \gamma'_e \vee \gamma''_e]\!]_{e\neq ok},\Delta''[\![e \mapsto (\gamma'_e \mathrel{?} \Delta'_e : \Delta''_e)]\!]_{e\neq ok},\overline{\boldsymbol{t}}'')}\ (\textsc{EComp})}$$

For briefer reference, let us introduce explicit names for the guard and register maps appearing on the second line:

$$\gamma''' = \gamma''[\![e \mapsto \gamma'_e \vee \gamma''_e]\!]_{e\neq ok}$$

$$\Delta''' = \Delta''[\![e \mapsto (\gamma'_e \mathrel{?} \Delta'_e : \Delta''_e)]\!]_{e\neq ok}$$

We now show how to find e, Γ'' and $\boldsymbol{t}''$ such that

$$\Gamma,\boldsymbol{t},s\,;s' \ \Downarrow\ \Gamma'',\boldsymbol{t}'',e \tag{A.26}$$

$$(\gamma''',\Delta''',\overline{\boldsymbol{t}}'') \rhd_\nu (\Gamma'',\boldsymbol{t}'',e) \tag{A.27}$$

First, we apply induction on the left (note that $\overline{\boldsymbol{t}}'.V \subset \overline{\boldsymbol{t}}''.V$ by Lemma 48(ii) and thus $\operatorname{dom}\nu \supset \overline{\boldsymbol{t}}'.V$) which gives us e', Γ', and $\boldsymbol{t}'$ such that

$$\Gamma,\boldsymbol{t},s \ \Downarrow\ \Gamma',\boldsymbol{t}',e' \tag{A.28}$$

$$(\gamma',\Delta',\overline{\boldsymbol{t}}') \rhd_\nu (\Gamma',\boldsymbol{t}',e') \tag{A.29}$$

We now distinguish two cases.

Case ($e' = ok$). From (A.29) we get $(\gamma'[\![e \mapsto \mathit{false}]\!]_{e\neq ok},\Delta',\overline{\boldsymbol{t}}') \rhd_\nu (\Gamma',\boldsymbol{t}',ok)$ (directly from Def. 50, exploiting $e' = ok$). We can thus apply induction on the right, which gives us Γ'', $\boldsymbol{t}''$ and e such that

$$\Gamma',\boldsymbol{t}',s' \ \Downarrow\ \Gamma'',\boldsymbol{t}'',e \tag{A.30}$$

$$(\gamma'',\Delta'',\overline{\boldsymbol{t}}'') \rhd_\nu (\Gamma'',\boldsymbol{t}'',e) \tag{A.31}$$

We now use rule

$$\frac{\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', \mathit{ok} \qquad \Gamma', \boldsymbol{t}', s' \Downarrow \Gamma'', \boldsymbol{t}'', e}{\Gamma, \boldsymbol{t}, s\,;s' \Downarrow \Gamma'', \boldsymbol{t}'', e} \qquad (\textsc{Comp-ok})$$

to get (A.26) from (A.28) and (A.30). To get (A.27) (and conclude this case), we need the following components:

(a) using Lemma 48(i)

(b) from (A.5)

(c) by (A.31)(c)

(d) observe that γ''' evaluates the same as γ'' (because $[\![\gamma'_x]\!]^\nu = \mathit{false}$ for all $x \neq \mathit{ok}$, because of Lemma 48(vi) and $[\![\gamma'_{\mathit{ok}}]\!]^\nu = [\![\gamma'_{e'}]\!]^\nu = \mathit{true}$ by (A.29)(d)) and then use (A.31)(d)

(e) for the same reasons, observe that Δ''' evaluates the same as Δ'' and then use (A.31)(e)

Case ($e' \neq \mathit{ok}$). We set $\Gamma'' = \Gamma'$ and $\boldsymbol{t}'' = \boldsymbol{t}'$ and $e = e'$. Now we can use the inference rule

$$\frac{\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e \qquad e \neq \mathit{ok}}{\Gamma, \boldsymbol{t}, s\,;s' \Downarrow \Gamma', \boldsymbol{t}', e} \qquad (\textsc{Comp-skip})$$

to get (A.26) from (A.28). To get (A.27) (and conclude this case), we show

$$(\gamma''', \Delta''', \bar{\boldsymbol{t}}'') \rhd_\nu (\Gamma', \boldsymbol{t}', e)$$

which has the following components:

(a) using Lemma 48(i)

(b) from (A.5)

(c) By (A.29)(d) we know $[\![\gamma'_{e'}]\!]^\nu = \mathit{true}$. Because $e' \neq \mathit{ok}$ and because of Lemma 48(vi), this implies $[\![\gamma'_{\mathit{ok}}]\!]^\nu = \mathit{false}$. Therefore, $(\gamma'[\![e \mapsto \mathit{false}]\!]_{e \neq \mathit{ok}}, \Delta', \bar{\boldsymbol{t}}')$ is void for ν, and by Lemma 48(vii) we know $[\![\bar{\boldsymbol{t}}'']\!]^\nu = [\![\bar{\boldsymbol{t}}']\!]^\nu$. Therefore, we get the claim directly from by (A.29)(c).

(d) because $e' \neq \mathit{ok}$, $[\![\gamma'''_{e'}]\!]^\nu = [\![\gamma'_{e'} \vee \gamma''_{e'}]\!]^\nu = \mathit{true}$ because $[\![\gamma'_{e'}]\!]^\nu = \mathit{true}$ by (A.29)(d)

(e) $[\![\Delta'''_e(r)]\!]^\nu = [\![\Delta'_e(r)]\!]^\nu$ because $e' \neq \mathit{ok}$ and $[\![\gamma'_e]\!]^\nu = \mathit{true}$. Therefore, the claim follows from (A.29)(e).

$$\boxed{\dfrac{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{s} (\gamma', \Delta', \bar{\boldsymbol{t}}') \qquad \gamma'' = \gamma'[\mathit{ok} \mapsto \gamma'_{\mathit{ok}} \vee \gamma'_{\mathit{break}\,l}][\mathit{break}\,l \mapsto \mathit{false}]}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{l:s} (\gamma'', \Delta'[\mathit{ok} \mapsto (\gamma'_{\mathit{break}\,l}\ ?\ \Delta'_{\mathit{break}\,l} : \Delta'_{\mathit{ok}})], \bar{\boldsymbol{t}}')}\ (\textsc{ELabel})}$$

For briefer reference, let us name the following expression:

$$\Delta'' = \Delta'[\mathit{ok} \mapsto (\gamma'_{\mathit{break}\,l}\ ?\ \Delta'_{\mathit{break}\,l} : \Delta'_{\mathit{ok}})]$$

We are now showing that there exist e', Γ' and $\boldsymbol{t}'$ such that

$$\Gamma, \boldsymbol{t}, l : s \Downarrow \Gamma', \boldsymbol{t}', e' \tag{A.32}$$

$$(\gamma'', \Delta'', \bar{\boldsymbol{t}}') \rhd_\nu (\Gamma', \boldsymbol{t}', e') \tag{A.33}$$

First, we apply induction to the left premise. That gives us Γ', $\boldsymbol{t}'$ and e satisfying

$$\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e \tag{A.34}$$

$$(\gamma', \Delta', \bar{\boldsymbol{t}}') \rhd_\nu (\Gamma', \boldsymbol{t}', e) \tag{A.35}$$

We now distinguish two cases.

Case ($e \neq \mathit{break}\,l$). Because we are running on unrolled programs, we know that $e \neq \mathit{continue}\,l$ also and can therefore soundly apply rule

$$\dfrac{\Gamma, \boldsymbol{t}, s \Downarrow \Gamma', \boldsymbol{t}', e \qquad e \notin \{\mathit{break}\,l, \mathit{continue}\,l\}}{\Gamma, \boldsymbol{t}, l : s \Downarrow \Gamma', \boldsymbol{t}', e} \quad (\textsc{Label})$$

to get (A.32) from (A.34), by setting $e' = e$. To get (A.33) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) by (A.35)(c)

(d) we know $[\![\gamma'_e]\!]^\nu = \mathit{true}$ (because of (A.35)(d)) and can use Lemma 48(vi) to conclude $[\![\gamma'_{\mathit{break}\,l}]\!]^\nu = \mathit{false}$. Thus, $[\![\gamma''_e]\!]^\nu = [\![\gamma'_e]\!]^\nu = \mathit{true}$ by (A.35)(d).

(e) because $[\![\gamma'_{\mathit{break}\,l}]\!]^\nu = \mathit{false}$ we can see that Δ'' evaluates the same as Δ', so the claim folllows from (A.35)(e)

Case ($e = \mathit{break}\,l$). We apply rule

$$\frac{\Gamma, \boldsymbol{t}, s \;\Downarrow\; \Gamma', \boldsymbol{t}', \mathit{break}\,l}{\Gamma, \boldsymbol{t}, l : s \;\Downarrow\; \Gamma', \boldsymbol{t}', \mathit{ok}} \qquad (\textsc{Label-break})$$

to get (A.32) from (A.34), by setting $e' = \mathit{ok}$. To get (A.33) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) by (A.35)(c)

(d) we know $[\![\gamma'_{\mathit{break}\,l}]\!]^\nu = \mathit{true}$ (because of (A.35)(d)), therefore $[\![\gamma''_{\mathit{ok}}]\!]^\nu = \mathit{true}$

(e) because $[\![\gamma'_{\mathit{break}\,l}]\!]^\nu = \mathit{true}$ we can see that for all r, $[\![\Delta''_{\mathit{ok}}]\!]^\nu$ is equal to $[\![\Delta'_{\mathit{break}\,l}]\!]^\nu$ which is in turn equal $\Gamma'(r)$ by (A.35)(e)

$$\frac{\gamma' = \gamma[\mathit{ok} \mapsto \gamma_{\mathit{ok}} \wedge \neg\phi][e \mapsto \gamma_{\mathit{ok}} \wedge \phi] \qquad \phi = \neg(\Delta_{\mathit{ok}}(r) = 0)}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{\mathbf{if}\,(r)\,\mathbf{throw}(e)} (\gamma', \Delta[e \mapsto \Delta_{\mathit{ok}}], \bar{\boldsymbol{t}})} \; (\textsc{ECondThrow})$$

For briefer reference, let us name the register map appearing on the second line:

$$\Delta' = \Delta[e \mapsto \Delta_{\mathit{ok}}]$$

By assumption of the lemma, we know

$$(\gamma, \Delta, \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, \mathit{ok}) \tag{A.36}$$

We are now showing that there exists an e' such that

$$\Gamma, \boldsymbol{t}, \mathbf{if}\,(r)\,\mathbf{throw}(e) \;\Downarrow\; \Gamma, \boldsymbol{t}, e' \tag{A.37}$$

$$(\gamma', \Delta', \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, e') \tag{A.38}$$

To do so, we distinguish two cases.

Case $[\![\phi]\!]^\nu = \mathit{false}$. This implies $[\![\Delta_{\mathit{ok}}(r)]\!]^\nu = 0$ and thus $\Gamma(r) = 0$ by (A.36)(e). By setting $e' = \mathit{ok}$ we can therefore use rule

$$\frac{\Gamma(r) = 0}{\Gamma, \boldsymbol{t}, \mathbf{if}\,(r)\,s \;\Downarrow\; \Gamma, \boldsymbol{t}, \mathit{ok}} \tag{\textsc{If-not}}$$

to get (A.37). To get (A.38) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) by (A.36)(c)

(d) $[\![\gamma''_{\mathit{ok}}]\!]^\nu = [\![\gamma_{\mathit{ok}} \wedge \neg\phi]\!]^\nu$ which is true because $[\![\phi]\!]^\nu = \mathit{false}$ and $[\![\gamma_{\mathit{ok}}]\!]^\nu = \mathit{true}$ by (A.36)(d).

(e) for all r we have $[\![\Delta'_{\mathit{ok}}(r)]\!]^\nu = [\![\Delta_{\mathit{ok}}(r)]\!]^\nu$ and the latter is equal to $\Gamma(r)$ by (A.36)(e).

Case $[\![\phi]\!]^\nu = \mathit{true}$. This implies $[\![\Delta_{\mathit{ok}}(r)]\!]^\nu \neq 0$ and thus $\Gamma(r) \neq 0$ by (A.36)(e). By setting $e' = e$, $\Gamma' = \Gamma$, $\boldsymbol{t}' = \boldsymbol{t}$ and $s = \mathbf{throw}(e)$ we can combine the rules

$$\Gamma, \boldsymbol{t}, \mathbf{throw}(e) \;\Downarrow\; \Gamma, \boldsymbol{t}, e \tag{\textsc{Throw}}$$

$$\frac{\Gamma(r) \neq 0 \qquad \Gamma, \boldsymbol{t}, s \;\Downarrow\; \Gamma', \boldsymbol{t}', e}{\Gamma, \boldsymbol{t}, \mathbf{if}\,(r)\,s \;\Downarrow\; \Gamma', \boldsymbol{t}', e} \tag{\textsc{If-so}}$$

to get (A.37). To get (A.38) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) by (A.36)(c)

(d) $[\![\gamma''_e]\!]^\nu = [\![\gamma_{ok} \wedge \phi]\!]^\nu$ which is true because $[\![\phi]\!]^\nu = \mathit{true}$ and $[\![\gamma_{ok}]\!]^\nu = \mathit{true}$ by (A.36)(d).

(e) for all r we have $[\![\Delta'_e(r)]\!]^\nu = [\![\Delta_{ok}(r)]\!]^\nu$ and the latter is equal to $\Gamma(r)$ by (A.36)(e).

$$\boxed{\frac{\Delta' = \Delta[ok \mapsto \Delta_{ok}[r \mapsto f(\Delta_{ok}(r_1), \ldots, \Delta_{ok}(r_k))]]}{(\gamma, \Delta, \bar{\boldsymbol{t}}) \xrightarrow{r:=(f\ r_1\ \ldots\ r_k)} (\gamma, \Delta', \bar{\boldsymbol{t}})} \quad \text{(EAssign)}}$$

By assumption of the lemma, we know

$$(\gamma, \Delta, \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, ok) \tag{A.39}$$

We are showing that there exists a Γ' such that

$$\Gamma, \boldsymbol{t}, r := (f\ r_1 \ldots r_k) \Downarrow \Gamma', \boldsymbol{t}, ok \tag{A.40}$$

$$(\gamma, \Delta', \bar{\boldsymbol{t}}) \rhd_\nu (\Gamma', \boldsymbol{t}, ok) \tag{A.41}$$

To do so, set $\Gamma' = \Gamma[r \mapsto f(\Gamma(r_1), \ldots, \Gamma(r_k))]$ and use the inference rule

$$\frac{v = f(\Gamma(r_1), \ldots, \Gamma(r_k))}{\Gamma, \boldsymbol{t}, r := (f\ r_1\ \ldots\ r_k) \Downarrow \Gamma[r \mapsto v], \boldsymbol{t}, ok} \quad \text{(Assign)}$$

to get (A.40). To get (A.41) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) by (A.39)(c)

(d) by (A.39)(d).

(e) For a given r', look at $[\![\Delta'_{ok}(r')]\!]^{\nu}$ and distinguish two cases. If $r' \neq r$, then it is equal to $[\![\Delta_{ok}(r')]\!]^{\nu}$ which is equal to $\Gamma(r')$ by (A.39)(e). If $r' = r$, then it is equal to $[\![f(\Delta_{ok}(r_1), \ldots, \Delta_{ok}(r_k))]\!]^{\nu}$. Now, by (A.39)(e), we know that for all i, $[\![\Delta_{ok}(r_i)]\!]^{\nu} = \Gamma(r_i)$ and the claim follows.

$$\boxed{\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{ok}(r'), \Delta_{ok}(r), \gamma_{ok})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{*r' :=_i r} (\gamma, \Delta, \overline{\boldsymbol{t}}')} \quad (\text{ES}\textsc{tore})}$$

By assumption of the lemma, we know

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_{\nu} (\Gamma, \boldsymbol{t}, \mathit{ok}) \tag{A.42}$$

We are showing that there exists a $\boldsymbol{t}'$ such that

$$\Gamma, \boldsymbol{t}, *r' :=_i r \ \Downarrow\ \Gamma, \boldsymbol{t}', \mathit{ok} \tag{A.43}$$

$$(\gamma, \Delta, \overline{\boldsymbol{t}}') \rhd_{\nu} (\Gamma, \boldsymbol{t}', \mathit{ok}) \tag{A.44}$$

To do so, use the inference rule

$$\frac{\boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \Gamma(r'), \Gamma(r))}{\Gamma, \boldsymbol{t}, *r' :=_i r \ \Downarrow\ \Gamma, \boldsymbol{t}', \mathit{ok}} \quad (\textsc{Store-id})$$

to get (A.43). To get (A.44) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) We need to show that $[\![\overline{\boldsymbol{t}}.\mathit{append}(i, \Delta_{ok}(r'), \Delta_{ok}(r), \gamma_{ok})]\!]^{\nu} = \boldsymbol{t}'$. This follows from Lemma 49 and the following individual facts:

- $[\![\gamma_{ok}]\!]^{\nu} = \mathit{true}$ by (A.42)(d)
- $[\![\Delta_{ok}(r')]\!]^{\nu} = \Gamma(r')$ by (A.42)(e)

- $[\![\Delta_{ok}(r)]\!]^{\nu} = \Gamma(r)$ by (A.42)(e)
- $[\![\overline{\boldsymbol{t}}]\!]^{\nu} = \boldsymbol{t}$ by (A.42)(c)

(d) by (A.42)(d).

(e) by (A.42)(e).

$$\boxed{\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.append(i, \Delta_{ok}(r'), D_i, \gamma_{ok})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{r:=_i *r'} (\gamma, \Delta[ok \mapsto \Delta_{ok}[r \mapsto D_i]], \overline{\boldsymbol{t}}')} \quad (\text{ELoad})}$$

For briefer reference, let us name the following expression:

$$\Delta' = \Delta[ok \mapsto \Delta_{ok}[r \mapsto v]]$$

By assumption of the lemma, we know

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_{\nu} (\Gamma, \boldsymbol{t}, ok) \tag{A.45}$$

We are showing that there exists a Γ' and $\boldsymbol{t}'$ such that

$$\Gamma, \boldsymbol{t}, r :=_i *r' \Downarrow \Gamma', \boldsymbol{t}', ok \tag{A.46}$$

$$(\gamma, \Delta', \overline{\boldsymbol{t}}') \rhd_{\nu} (\Gamma', \boldsymbol{t}', ok) \tag{A.47}$$

First, let $v = [\![D_i]\!]^{\nu}$. Now we set $\Gamma' = \Gamma[r \mapsto v]$ and use the inference rule

$$\frac{v \in Val \qquad \boldsymbol{t}' = \boldsymbol{t}.append(i, \Gamma(r'), v)}{\Gamma, \boldsymbol{t}, r :=_i *r' \Downarrow \Gamma[r \mapsto v], \boldsymbol{t}', ok} \quad (\text{Load-id})$$

to get (A.46). To get (A.47) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) We need to show that $[\![\overline{\boldsymbol{t}}.append(\Delta_{ok}(r'), D_i, \gamma_{ok})]\!]^{\nu} = \boldsymbol{t}'$. This follows from Lemma 49 and the following individual facts:

- $[\![\gamma_{ok}]\!]^\nu = \mathit{true}$ by (A.45)(d)
- $[\![\Delta_{ok}(r')]\!]^\nu = \Gamma(r')$ by (A.45)(e)
- $[\![D_i]\!]^\nu = v$
- $[\![\overline{\boldsymbol{t}}]\!]^\nu = \boldsymbol{t}$ by (A.45)(c)

(d) by (A.45)(d).

(e) For a given r'', look at $[\![\Delta'_{ok}(r'')]\!]^\nu$ and distinguish two cases. If $r'' \neq r$, then it is equal to $[\![\Delta_{ok}(r'')]\!]^\nu$ which is equal to $\Gamma(r'')$ by (A.45)(e), and thus also equal to $\Gamma'(r'')$. If $r'' = r$, then it is equal to $[\![D_i]\!]^\nu$ which is equal to v and thus also equal to $\Gamma'(r'')$ as required.

$$\boxed{\frac{\overline{\boldsymbol{t}}' = \overline{\boldsymbol{t}}.\mathit{append}(i, \bot, \bot, \gamma_{ok})}{(\gamma, \Delta, \overline{\boldsymbol{t}}) \xrightarrow{\textbf{fence}_i} (\gamma, \Delta, \overline{\boldsymbol{t}}')} \qquad (\text{EF}\textsc{ence})}$$

By assumption of the lemma, we know

$$(\gamma, \Delta, \overline{\boldsymbol{t}}) \rhd_\nu (\Gamma, \boldsymbol{t}, ok) \tag{A.48}$$

We are showing that there exists a $\boldsymbol{t}'$ such that

$$\Gamma, \boldsymbol{t}, \textbf{fence}_i \Downarrow \Gamma, \boldsymbol{t}', ok \tag{A.49}$$

$$(\gamma, \Delta, \overline{\boldsymbol{t}}') \rhd_\nu (\Gamma, \boldsymbol{t}', ok) \tag{A.50}$$

To do so, use the inference rule

$$\frac{\boldsymbol{t}' = \boldsymbol{t}.\mathit{append}(i, \bot, \bot)}{\Gamma, \boldsymbol{t}, \textbf{fence}_i \Downarrow \Gamma, \boldsymbol{t}', ok} \qquad (\textsc{Fence-id})$$

to get (A.49). To get (A.50) (and conclude this case), we show the following claims:

(a) using Lemma 48(i)

(b) from (A.5)

(c) We need to show that $[\![\bar{\boldsymbol{t}}.\mathit{append}(i, \bot, \bot, \gamma_{ok})]\!]^{\nu} = \boldsymbol{t}'$. This follows from Lemma 49 and the following individual facts:

- $[\![\gamma_{ok}]\!]^{\nu} = \mathit{true}$ by (A.45)(d)
- $[\![\bar{\boldsymbol{t}}]\!]^{\nu} = \boldsymbol{t}$ by (A.45)(c)

(d) by (A.48)(d).

(e) by (A.48)(e).

Appendix B

C Implementation Sources

In this chapter, we show the C code for the implementations we studied (see Table 8.1 in Chapter 8), including the memory ordering fences we placed. We verified that these fences are sufficient for the testcases shown in Table 8.2, using our *CheckFence* tool.

B.1 Prototypes

The file `lsl_protos.h` captures (1) common, useful definitions, and (2) function prototypes for externally defined functions that have a special meaning and are substituted for appropriate LSL stubs by our frontend.

```
/* general */
enum boolean { false=0, true=1 };
typedef enum boolean boolean_t;

/* unsigned types */
typedef unsigned uint32;
typedef unsigned long long uint64;

/* dynamic memory allocation */
extern void *lsl_malloc(size_t size);
extern void lsl_free(void *ptr);

/* dynamic memory allocation without pointer reuse */
extern void *lsl_malloc_noreuse(size_t size);
extern void lsl_free_noreuse(void *ptr);

/* verification directives */
extern void lsl_assert(boolean_t expr);
extern void lsl_assume(boolean_t expr);

/* simple lock */
typedef unsigned lsl_lock_t;
extern void lsl_initlock(lsl_lock_t *lock);
extern void lsl_lock(lsl_lock_t *lock);
extern void lsl_unlock(lsl_lock_t *lock);

/* fences */
extern void lsl_fence(char *fencetype);

/* thread id */
extern int lsl_get_number_threads();
extern int lsl_get_thread_id();

/* observables */
extern void lsl_observe_label(char *opname);
extern void lsl_observe_input(char *name, int val);
extern void lsl_observe_output(char *name, int val);

/* nondeterministic values */
extern int lsl_nondet(int min, int max);
extern int lsl_choose();

/* compare-and-swap */
extern boolean_t lsl_cas_32(void *loc, uint32 old, uint32 new);
extern boolean_t lsl_cas_64(void *loc, uint64 old, uint64 new);
extern boolean_t lsl_cas_ptr(void *loc, void *old, void *new);

/* double compare-and-swap */
extern boolean_t lsl_dcas_32(void *loc1, void *loc2,
           uint32 old1, uint32 old2, uint32 new1, uint32 new2);
extern boolean_t lsl_dcas_64(void *loc1, void *loc2,
           uint64 old1, uint64 old2, uint64 new1, uint64 new2);
extern boolean_t lsl_dcas_ptr(void *loc1, void *loc2,
               void *old1, void *old2, void *new1, void *new2);

```

B.2 Two-Lock Queue (ms2)

```
#include "lsl_protos.h"

/* ---- data types  ---- */

typedef int value_t;

typedef struct node {
  struct node *next;
  value_t value;
} node_t;

typedef struct queue {
  node_t *head;
  node_t *tail;
  lsl_lock_t headlock;
  lsl_lock_t taillock;
} queue_t;

/* ---- operations ---- */

```

```
void init_queue(queue_t *queue)
{
  node_t *dummy = lsl_malloc(sizeof(node_t));
  dummy->next = 0;
  dummy->value = 0;
  queue->head = dummy;
  queue->tail = dummy;
  lsl_initlock(&queue->headlock);
  lsl_initlock(&queue->taillock);
}

void enqueue(queue_t *queue, value_t val)
{
  node_t *node = lsl_malloc(sizeof(node_t));
  node->value = val;
  node->next = 0;
  lsl_lock(&queue->taillock);
  lsl_fence("store-store"); /* trace1 */
  queue->tail->next = node;
  queue->tail = node;
  lsl_unlock(&queue->taillock);
}

boolean_t dequeue(queue_t *queue, value_t *retvalue)
{
  node_t *node;
  node_t *new_head;
  lsl_lock(&queue->headlock);
  node = queue->head;
  new_head = node->next;
  if (new_head == 0) {
    lsl_unlock(&queue->headlock);
    return false;
  }
  lsl_fence("data-dependent-loads"); /* trace2 */
  *retvalue = new_head->value;
  queue->head = new_head;
  lsl_unlock(&queue->headlock);
  lsl_free(node);
  return true;
}

```

B.3 Nonblocking Queue (msn)

```
#include "lsl_protos.h"

/* ---- data types  ---- */

typedef int value_t;

typedef struct node {
  struct node *next;
  value_t value;
} node_t;

typedef struct queue {
  node_t *head;
  node_t *tail;
} queue_t;

/* ---- operations  ---- */

void init_queue(queue_t *queue)
{
  node_t *node = lsl_malloc_noreuse(sizeof(node_t));
  node->next = 0;
  queue->head = queue->tail = node;
```

```
}

void enqueue(queue_t *queue, value_t value)
{
  node_t *node;
  node_t *tail;
  node_t *next;

  node = lsl_malloc_noreuse(sizeof(node_t));
  node->value = value;
  node->next = 0;
  lsl_fence("store-store"); /* trace3 */
  while (true) {
    tail = queue->tail;
    lsl_fence("data-dependent-loads"); /* trace9 */
    next = tail->next;
    lsl_fence("load-load"); /* trace7 */
    if (tail == queue->tail)
      if (next == 0) {
        if (lsl_cas_64(&tail->next, next, node))
          break;
      } else
        lsl_cas_ptr(&queue->tail, tail, next);
  }
  lsl_fence("store-store"); /* trace5 */
  lsl_cas_ptr(&queue->tail, tail, node);
}

boolean_t dequeue(queue_t *queue, value_t *pvalue)
{
  node_t *head;
  node_t *tail;
  node_t *next;

```

```
 while (true) {
  head = queue->head;
  lsl_fence("load-load"); /* trace6 */
  tail = queue->tail;
  lsl_fence("load-load"); /* trace1 */
  next = head->next;
  lsl_fence("load-load"); /* trace8 */
  if (head == queue->head) {
   if (head == tail) {
    if (next == 0)
     return false;
    lsl_cas_ptr(&queue->tail, tail, next);
   } else {
    *pvalue = next->value;
    if (lsl_cas_ptr(&queue->head, head, next))
     break;
   }
  }
 }
 lsl_free_noreuse(head);
 return true;
}
```

B.4 Nonblocking List-Based Set (harris)

```
#include "lsl_protos.h"

/* ---- data types ---- */

/* data values */
typedef int value_t;

/* marked pointers */
typedef void *mpointer_t;

typedef struct node {
  mpointer_t next;
  value_t key;
} node_t;

typedef struct list {
  mpointer_t head;
  mpointer_t tail;
} list_t;


/* ---- constructor & accessors for marked pointers---- */

inline mpointer_t lsl_make_mpointer(node_t *ptr, boolean_t marked) {
  return (void*) (((ptrdiff_t) ptr) | marked);
}
inline node_t *lsl_get_mpointer_ptr(mpointer_t mpointer) {
  return (node_t *) ((((ptrdiff_t) mpointer) >> 1) << 1);
}
inline boolean_t lsl_get_mpointer_marked(mpointer_t mpointer) {
  return ((ptrdiff_t) mpointer) & 0x1;
}
inline mpointer_t lsl_set_mpointer_ptr(mpointer_t mpointer, node_t *ptr) {
  return (void*) (((ptrdiff_t) ptr) | (((ptrdiff_t) mpointer) & 0x1));
}
inline mpointer_t lsl_set_mpointer_marked(mpointer_t mpointer,
                                                          boolean_t marked){
  return (void*) ((((ptrdiff_t) mpointer >> 1) << 1) | marked);
}


/* ---- operations  ---- */

```

```
void init(list_t *list, value_t sentinel_min_value,
                                                value_t sentinel_max_value)
{
  node_t *node;

  node = lsl_malloc_noreuse(sizeof(node_t));
  node->next = lsl_make_mpointer(0, false);
  node->key = sentinel_max_value;
  list->tail = lsl_make_mpointer(node, false);
  node = lsl_malloc_noreuse(sizeof(node_t));
  node->next = list->tail;
  node->key = sentinel_min_value;
  list->head = lsl_make_mpointer(node, false);
}

mpointer_t search(list_t *list, value_t search_key,
                  mpointer_t *l_node, boolean_t allow_retries)
{
  mpointer_t l_node_next, r_node, t, t_next;

  do {
    t = list->head;
    t_next = lsl_get_mpointer_ptr(list->head)->next;

    /* 1 : find l_node and r_node */
    do {
      if (! lsl_get_mpointer_marked(t_next)) {
        *l_node = t;
        l_node_next = t_next;
      }
      t = lsl_set_mpointer_marked(t_next, false);
      if (t == list->tail)
        break;
      lsl_fence("data-dependent-loads"); /* trace2 */
      t_next = lsl_get_mpointer_ptr(t)->next;
    } while ( lsl_get_mpointer_marked(t_next) ||
              (lsl_get_mpointer_ptr(t)->key < search_key)); /* B1 */
    r_node = t;
    lsl_fence("data-dependent-loads"); /* trace1 */

    /* 2 : check nodes are adjacent */
    if (l_node_next == r_node)
      if ((r_node != list->tail) && lsl_get_mpointer_marked(
                                   lsl_get_mpointer_ptr(r_node)->next)) {
        lsl_assume(allow_retries);
        continue; /* G1 */
      }
      else
        return r_node; /* R1 */

    /* 3 : Remove one or more marked nodes */
    if (lsl_cas_ptr (&(lsl_get_mpointer_ptr(*l_node)->next),
                                           l_node_next, r_node)) /* C1 */
      if ((r_node != list->tail) && lsl_get_mpointer_marked(
                                   lsl_get_mpointer_ptr(r_node)->next)) {
        lsl_fence("aliased-loads"); /* trace4 */
        continue; /* G2 */
      }
      else
        return r_node; /* R2 */

  } while (true);  /* B2 */
}

boolean_t contains(list_t *list,
                             value_t search_key, boolean_t allow_retries)
{
  mpointer_t r_node, l_node;

  r_node = search(list, search_key, &l_node, allow_retries);
  if ((r_node == list->tail)
      || (lsl_get_mpointer_ptr(r_node)->key != search_key))
    return false;
  else
    return true;
}

boolean_t add(list_t *list, value_t key, boolean_t allow_retries)
{
  mpointer_t n_node, r_node, l_node;

  n_node = lsl_make_mpointer(lsl_malloc_noreuse(sizeof(node_t)), false);
  (lsl_get_mpointer_ptr(n_node))->key = key;

  do {
    r_node = search(list, key, &l_node, allow_retries);
    if ((r_node != list->tail)
        && (lsl_get_mpointer_ptr(r_node)->key == key)) /*T1*/
      return false;
    lsl_get_mpointer_ptr(n_node)->next = r_node;
    lsl_fence("store-store"); /* trace3 */
    if (lsl_cas_ptr(
             &(lsl_get_mpointer_ptr(l_node)->next), r_node, n_node)) /*C2*/
      return true;
    lsl_assume(allow_retries);
    lsl_fence("aliased-loads"); /* trace4 */
  } while(true); /*B3*/
}

boolean_t remove(list_t *list, value_t search_key, boolean_t allow_retries)
{
  mpointer_t r_node, r_node_next, l_node;

  do {
    r_node = search(list, search_key, &l_node, allow_retries);
    if ((r_node == list->tail)
        || (lsl_get_mpointer_ptr(r_node)->key != search_key)) /*T1*/
      return false;
    r_node_next = lsl_get_mpointer_ptr(r_node)->next;
    if (! lsl_get_mpointer_marked(r_node_next))
      if (lsl_cas_ptr( &(lsl_get_mpointer_ptr(r_node)->next),
           r_node_next, lsl_set_mpointer_marked(r_node_next, true))) /*C3*/
        break;
    lsl_assume(allow_retries);
    lsl_fence("aliased-loads"); /* trace4 */
  } while (true);  /*B4*/

  if (allow_retries) {
    if (! lsl_cas_ptr(&(lsl_get_mpointer_ptr(l_node)->next),
                                               r_node, r_node_next)) /*C4*/
      r_node = search (list, lsl_get_mpointer_ptr(r_node)->key,
                                                   &l_node, allow_retries);
  }
  return true;
}


```

B.5 Lazy List-Based Set (lazylist)

```
#include "lsl_protos.h"

/* ---- data types ---- */

/* data values */
typedef int value_t;

typedef struct node {
  struct node *next;
  value_t key;
  lsl_lock_t lock;
  boolean_t marked;
} node_t;

typedef struct list {
  node_t *head;
  node_t *tail;
} list_t;

/* ---- operations for list ---- */

void init(list_t *list, int sentinel_min_value, int sentinel_max_value)
{
  node_t *node;
  node = lsl_malloc_noreuse(sizeof(node_t));
  node->next = 0;
  node->key = sentinel_max_value;
  node->marked = false; /* trace1 */
  lsl_initlock(&node->lock);
  list->tail = node;
  node = lsl_malloc_noreuse(sizeof(node_t));
  node->next = list->tail;
  node->key = sentinel_min_value;
  node->marked = false; /* trace1 */
  lsl_initlock(&node->lock);
  list->head = node;
}

boolean_t contains(list_t *list, value_t key, boolean_t allow_retries) {
  node_t *curr;

  curr = list->head;
  while(curr->key < key) {
    curr = curr->next;
    lsl_fence("data-dependent-loads"); /* trace5 */
  }
  if (curr->key == key && !curr->marked)
    return true;
  else
    return false;
}

void locate(list_t *list, value_t key, node_t **left,
            node_t **right, boolean_t allow_retries) {
  node_t *pred;
  node_t *curr;

  while (1) {
    pred = list->head;
    curr = pred->next;
    lsl_fence("data-dependent-loads"); /* trace2 */
    while(curr->key < key) {
      pred = curr;
      curr = curr->next;
      lsl_fence("data-dependent-loads"); /* trace6 */
    }
    lsl_lock(&pred->lock);
    lsl_lock(&curr->lock);
    if ( ! pred->marked
         && (! curr->marked)
         && (pred->next == curr)) {
      *left = pred;
      *right = curr;
      return;
    }
    lsl_assume(allow_retries);
    lsl_unlock(&pred->lock);
    lsl_unlock(&curr->lock);
  }
}

boolean_t add(list_t *list, value_t key, boolean_t allow_retries) {
  node_t *pred;
  node_t *curr;
  node_t *entry;
  boolean_t res;

  locate(list, key, &pred, &curr, allow_retries);
  if (curr->key != key) {
    entry = lsl_malloc_noreuse(sizeof(node_t));
    entry->key = key;
    entry->next = curr;
    entry->marked = false; /* trace1 */
    lsl_initlock(&entry->lock); /* trace4 */
    lsl_fence("store-store"); /* trace3 */
    pred->next = entry;
    res = true;
  } else
    res = false;
  lsl_unlock(&pred->lock);
  lsl_unlock(&curr->lock);
  return res;
}

boolean_t remove(list_t *list, value_t key, boolean_t allow_retries) {
  node_t *pred;
  node_t *curr;
  node_t *entry;
  boolean_t res;

  locate(list, key, &pred, &curr, allow_retries);
  if (curr->key == key) {
    curr->marked = true;
    entry = curr->next;
    pred->next = entry;
    res = true;
  } else
    res = false;
  lsl_unlock(&pred->lock);
  lsl_unlock(&curr->lock);
  return res;
}
```

B.6 Snark Deque (snark)

This is the code for the (corrected) snark deque [17].

```
#include "lsl_protos.h"

/* data values */
typedef int value_t; // positive number (-1 reserved for special value)
#define CLAIMED -1

/* node objects */
typedef struct node {
  struct node *R;
  struct node *L;
  value_t V;
} node_t;

/* deque objects */
typedef struct deque {
  node_t *Dummy;
  node_t *LeftHat;
  node_t *RightHat;
} deque_t;

/* ---- operations for deque ---- */

void init_deque(deque_t *deque)
{
  node_t *dummy = lsl_malloc_noreuse(sizeof(node_t));
  dummy->L = dummy->R = dummy;
  deque->Dummy = dummy;
  deque->LeftHat = dummy;
  deque->RightHat = dummy;
}

void push_right(deque_t *deque, value_t value, boolean_t allow_retries)
{
  node_t *nd, *rh, *rhR, *lh;

  nd = lsl_malloc_noreuse(sizeof(node_t));
  nd->R = deque->Dummy;
  nd->V = value;
  while (true) {
    rh = deque->RightHat;
    lsl_fence("data-dependent-loads"); /* trace3 */
    rhR = rh->R;
    if (rhR == rh) {
      nd->L = deque->Dummy;
      lh = deque->LeftHat;
      lsl_fence("store-store"); /* trace4 */
      lsl_fence("aliased-loads"); /* trace10  */
      if (lsl_dcas_ptr(&(deque->RightHat),
                                &(deque->LeftHat), rh, lh, nd, nd)) /* A */
```

```
        return;
    } else {
      nd->L = rh;
      lsl_fence("store-store"); /* trace6 */
      if (lsl_dcas_ptr(&(deque->RightHat),
                                        &(rh->R), rh, rhR, nd, nd)) /* B */
        return;
    }
    lsl_assume(allow_retries);
    lsl_fence("aliased-loads"); /* trace9  */
  }
}

void push_left(deque_t *deque, value_t value, boolean_t allow_retries)
{
  node_t *nd, *lh, *lhL, *rh;

  nd = lsl_malloc_noreuse(sizeof(node_t));
  nd->L = deque->Dummy;
  nd->V = value;
  while (true) {
    lh = deque->LeftHat;
    lsl_fence("data-dependent-loads"); /* trace3 */
    lhL = lh->L;
    if (lhL == lh) {
      nd->R = deque->Dummy;
      rh = deque->RightHat;
      lsl_fence("store-store"); /* trace4 */
      lsl_fence("aliased-loads"); /* trace10  */
      if (lsl_dcas_ptr(&(deque->LeftHat),
                              &(deque->RightHat), lh, rh, nd, nd)) /* A’ */
        return;
    } else {
      nd->R = lh;
      lsl_fence("store-store"); /* trace6 */
      if (lsl_dcas_ptr(&(deque->LeftHat),
                                       &(lh->L), lh, lhL, nd, nd)) /* B’ */
        return;
    }
    lsl_assume(allow_retries);
    lsl_fence("aliased-loads"); /* trace9  */
  }
}

boolean_t pop_right(deque_t *deque, value_t *value,
                                                 boolean_t allow_retries) {
  node_t *rh, *rhL;
  value_t result;
  while(true) {
    rh = deque->RightHat;
    lsl_fence("data-dependent-loads"); /* trace5 */
    rhL = rh->L;
    if (rh->R == rh) {
      lsl_fence("load-load"); /* trace7 */
      if (deque->RightHat == rh)
        return false;
    } else {
      if (lsl_dcas_ptr(&(deque->RightHat),
                                         &(rh->L), rh, rhL, rhL, rh)) {/*D*/
        result = rh->V;
        if (result != CLAIMED) {
          if (lsl_cas_ptr(&(rh->V), result, CLAIMED)) {
            rh->R = deque->Dummy;
            *value = result;
            return true;
          } else
            return false;
        } else
```

```
          return false;
      }
    }
    lsl_assume(allow_retries);
    lsl_fence("aliased-loads"); /* trace8  */
  }
}

boolean_t pop_left(deque_t *deque,
                          value_t *value, boolean_t allow_retries) {
  node_t *lh, *lhR;
  value_t result;
  while(true) {
    lh = deque->LeftHat;
    lsl_fence("data-dependent-loads"); /* trace5 */
    lhR = lh->R;
    if (lh->L == lh) {
      lsl_fence("load-load"); /* trace7 */
      if (deque->LeftHat == lh)
        return false;
    } else {
      if (lsl_dcas_ptr(&(deque->LeftHat),
                                     &(lh->R), lh, lhR, lhR, lh)) {/*D*/
        result = lh->V;
        if (result != CLAIMED) {
          if (lsl_cas_ptr(&(lh->V), result, CLAIMED)) {
            lh->L = deque->Dummy;
            *value = result;
            return true;
          } else
            return false;
        } else
          return false;
      }
    }
    lsl_assume(allow_retries);
    lsl_fence("aliased-loads"); /* trace8  */
  }
}


```

B.7 Harnesses

These files serve to connect the test definitions (Table 8.2) to the source code that implements the operations.

```
/* ---- harness for queue ---- */

queue_t queue;

void i()
{
  init_queue(&queue);
}
void e()
{
  value_t val = lsl_nondet(0, 1);
  lsl_observe_label("enqueue");
  lsl_observe_input("val", val);
  enqueue(&queue, val);
}
void d()
{
  boolean_t res;
  value_t val;
  lsl_observe_label("dequeue");
  res = dequeue(&queue, &val);
  lsl_observe_output("res", res);
  if (res)
    lsl_observe_output("val", val);
}
```

```
/* ---- harness for set ---- */

list_t list; /* use global variable for list object */

#define sentinel_min_value 0
#define min_value          2
#define max_value          3
#define sentinel_max_value 4

void i() {
  init(&list, sentinel_min_value, sentinel_max_value);
}

void c()
{
  value_t val;
  boolean_t result;
  val = lsl_nondet(min_value, max_value);
  lsl_observe_label("contains");
  lsl_observe_input("val", val);
  result = contains(&list, val, true);
  lsl_observe_output("result", result);
}
void a()
{
  value_t val;
  boolean_t result;
  val = lsl_nondet(min_value, max_value);
  lsl_observe_label("add");
  lsl_observe_input("val", val);
  result = add(&list, val, true);
  lsl_observe_output("result", result);
}
void r()
{
  value_t val;
  boolean_t result;
  val = lsl_nondet(min_value, max_value);
  lsl_observe_label("remove");
  lsl_observe_input("val", val);
  result = remove(&list, val, true);
  lsl_observe_output("result", result);
}
void c_n()
{
  value_t val;
  boolean_t result;
  val = lsl_nondet(min_value, max_value);
  lsl_observe_label("contains");
  lsl_observe_input("val", val);
  result = contains(&list, val, false);
  lsl_observe_output("result", result);
}
void a_n()
{
  value_t val;
  boolean_t result;
  val = lsl_nondet(min_value, max_value);
  lsl_observe_label("add");
  lsl_observe_input("val", val);
  result = add(&list, val, false);
  lsl_observe_output("result", result);
}
void r_n()
```

```
{
  value_t val;
  boolean_t result;
  val = lsl_nondet(min_value, max_value);
  lsl_observe_label("remove");
  lsl_observe_input("val", val);
  result = remove(&list, val, false);
  lsl_observe_output("result", result);
}
```

```
/* ---- harness for deque ---- */

deque_t deque; /* use global variable for deque object */

void init()
{
  init_deque(&deque);
}
void pushL()
{
  value_t val = lsl_nondet(0, 1);
  lsl_observe_label("pushleft");
  lsl_observe_input("value", val);
  push_left(&deque, val, true);
}
void pushR()
{
  value_t val = lsl_nondet(0, 1);
  lsl_observe_label("pushright");
  lsl_observe_input("value", val);
  push_right(&deque, val, true);
}
void popL()
{
  boolean_t res;
  value_t val;
  lsl_observe_label("popleft");
  res = pop_left(&deque, &val, true);
  lsl_observe_output("nonempty", res);
  if (res)
    lsl_observe_output("value", val);
}
void popR()
{
  boolean_t res;
  value_t val;
  lsl_observe_label("popright");
  res = pop_right(&deque, &val, true);
  lsl_observe_output("nonempty", res);
  if (res)
    lsl_observe_output("value", val);
}
void pushL_n()
{
  value_t val = lsl_nondet(0, 1);
  lsl_observe_label("pushleft");
  lsl_observe_input("value", val);
  push_left(&deque, val, false);
}
void pushR_n()
{
  value_t val = lsl_nondet(0, 1);
  lsl_observe_label("pushright");
  lsl_observe_input("value", val);
  push_right(&deque, val, false);
}
void popL_n()
{
  boolean_t res;
  value_t val;
  lsl_observe_label("popleft");
  res = pop_left(&deque, &val, false);
  lsl_observe_output("nonempty", res);
  if (res)
    lsl_observe_output("value", val);
}
void popR_n()
{
  boolean_t res;
  value_t val;
  lsl_observe_label("popright");
  res = pop_right(&deque, &val, false);
  lsl_observe_output("nonempty", res);
  if (res)
    lsl_observe_output("value", val);
}

```

Bibliography

[1] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[2] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[3] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Logic in Computer Science (LICS)*, pages 219–228, 1996.

[4] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Principles of Distributed Computing (PODC)*, pages 184–193, New York, NY, USA, 1995. ACM Press.

[5] H. Attiya and J. Welch. Sequential consistency versus linearizability (extended abstract). In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 304–315, 1991.

[6] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a c compiler front-end. In *Formal Methods (FM)*, LNCS 4085, pages 460–475. Springer, 2006.

[7] H.-J. Boehm. Threads cannot be implemented as a library. In *Programming Language Design and Implementation (PLDI)*, pages 261–268, 2005.

[8] S. Burckhardt, R. Alur, and M. Martin. Bounded verification of concurrent data types on relaxed memory models: A case study. In *Computer-Aided Verification (CAV)*, LNCS 4144, pages 489–502. Springer, 2006.

[9] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[10] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2988, pages 168–176. Springer, 2004.

[11] W. W. Collier. *Reasoning about parallel architectures.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[12] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer-Aided Verification (CAV)*, LNCS 4144, pages 475–488. Springer, 2006.

[13] Compaq Computer Corporation. *Alpha Architecture Reference Manual*, 4th edition, January 2002.

[14] D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Conference on Distributed Computing (DISC)*, LNCS 1914, pages 59–73. Springer, 2000.

[15] D. Detlefs, R. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.

[16] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[17] S. Doherty, D. Detlefs, L. Grove, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS is not a silver bullet for nonblocking algorithm design. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 216–224, 2004.

[18] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: verifying concurrent programs by runtime refinement-violation detection. In *Programming Language Design and Implementation (PLDI)*, pages 27–37, 2005.

[19] X. Fang, J. Lee, and S. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *International Conference on Supercomputing (ICS)*, pages 285–294, 2003.

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

[21] K. Fraser. lock-free-lib.tar.gz. `http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free/`.

[22] K. Fraser. *Practical Lock-Freedom.* PhD thesis, University of Cambridge, 2004.

[23] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[24] B. Frey. *PowerPC Architecture Book v2.02.* International Business Machines Corporation, 2005.

[25] H. Gao and W. Hesslink. A formal reduction for lock-free parallel algorithms. In *Computer-Aided Verification (CAV)*, LNCS 3114, pages 44–56. Springer, 2004.

[26] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, University of Utah, 2005.

[27] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Computer-Aided Verification (CAV)*, LNCS 3114, pages 401–413, 2004.

[28] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *Conference on Distributed Computing (DISC)*, LNCS 2180, pages 300–314. Springer, 2001.

[29] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *Conference on Distributed Computing (DISC)*, LNCS 2508, pages 265–279. Springer, 2002.

[30] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems (OPODIS)*, 2005.

[31] T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Programming language design and implementation (PLDI)*, pages 1–13, 2004.

[32] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[33] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[34] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)*, pages 289–300, New York, NY, USA, 1993.

[35] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[36] T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Formal Methods (FM)*, LNCS 4085, pages 476–491. Springer, 2006.

[37] Intel Corporation. *A Formal Specification of the Intel Itanium Processor Family Memory Ordering*, October 2002.

[38] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, November 2006.

[39] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual, Book 2, rev. 2.2*, January 2006.

[40] Intel Corporation. *Intel Threading Building Blocks*, September 2006.

[41] International Business Machines Corporation. *z/Architecture Principles of Operation*, first edition, December 2000.

[42] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis*, pages 14–25, 2000.

[43] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. In *TV'06 Workshop, Federated Logic Conference (FLoC)*, pages 66–77, 2006.

[44] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

[45] L. Lamport. Checking a multithreaded algorithm with +CAL. In *Conference on Distributed Computing (DISC)*, LNCS 4167, pages 151–163. Springer, 2006.

[46] D. Lea. The java.util.concurrent synchronizer framework. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, 2004.

[47] R. Leino, J. Saxe, and R. Stata. Checking java programs via guarded commands. Technical Report 1999-002, Compaq Systems Research Center, 1999.

[48] X. Leroy. Formal certification of a compiler back-end. In *Principles of Programming Languages (POPL)*, pages 42–54, 2006.

[49] V. Luchangco. Personal communications, October 2006.

[50] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, pages 378–391, 2005.

[51] M. Martin, D. Sorin, H. Cain, M. Hill, and M. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *International Symposium on Microarchitecture (MICRO)*, pages 328–337, 2001.

[52] M. Michael. Scalable lock-free dynamic memory allocation. In *Programming Language Design and Implementation (PLDI)*, pages 35–46, 2004.

[53] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[54] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[55] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Principles of distributed computing (PODC)*, pages 219–228, 1997.

[56] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535, 2001.

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

[58] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conf. on Compiler Constr. (CC)*, 2002.

[59] S. Owre, J. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[60] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 34–41, 1995.

[61] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Programming Language Design and Implementation (PLDI)*, pages 320–331, 2006.

[62] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Computer-Aided Verification (CAV)*, LNCS 3576, pages 82–97. Springer, 2005.

[63] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comp. Sys.*, 15(4):391–411, 1997.

[64] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[65] R. Steinke and G. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

[66] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[67] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[68] S. Tasiran and S. Qadeer. Runtime refinement verification of concurrent data structures. In *Runtime Verification*, Electronic Notes in Theoretical Computer Science, 2004.

[69] O. Trachsel, C. von Praun, and T. Gross. On the effectiveness of speculative and selective memory fences. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[70] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 129–136, 2006.

[71] C. von Praun, T. Cain, J. Choi, and K. Ryu. Conditional memory ordering. In *International Symposium on Computer Architecture (ISCA)*, 2006.

[72] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

[73] M. Xu, R. Bodik, and M. Hill. A serializability violation detector for shared-memory server programs. In *Programming Language Design and Implementation (PLDI)*, 2005.

[74] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[75] J. Y. Yang. *Formalizing Shared Memory Consistency Models for Program Analysis*. PhD thesis, University of Utah, 2005.

[76] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. In *International Conference on Formal Engineering Methods (ICFEM)*, LNCS 3308, pages 30–45. Springer, 2004.

[77] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Rigorous concurrency analysis of multithreaded programs. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, 2004.

[78] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.