UNIVERSITY OF SOUTHAMPTON

Separating Computation from Communication: A Design Approach for Concurrent Bug Finding

by

Ermenegildo Tomasco

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

October 2017

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Ermenegildo Tomasco

With the spread of multi-core systems, the need to write concurrent programs in order to take advantage of their multi-core processors continues to grow. Developing concurrent programs free of errors is very difficult, due to their often non-deterministic nature. Further, *weak memory models* (WMMs) implemented in modern multi-core hardware architectures introduce additional executions that can lead to seemingly counter-intuitive results which confound the developers' reasoning. Although testing is a widely used approach to finding program errors, testing-only approaches, such as stress testing, remain highly ineffective for concurrency errors that rarely manifest themselves and are difficult to reproduce.

Despite the substantial advancements in the field of analysis and verification of concurrent programs assuming the classical *Sequential Consistency* (SC) memory model, the state-of-the-art technology for other WMMs is quite unsatisfactory.

The main goal of this thesis is to extend existing successful techniques that have been implemented for SC to more general WMMs. This work describes a general approach that allows to combine different verification techniques with different memory models in the style of a plug-and-play architecture. Its main idea is to introduce an abstraction that allows us to separate the computation and communication concerns of concurrent programs, without loosing the efficiency of existing approaches. We start by introducing an abstract data type, called *shared memory abstraction* (SMA), that encapsulates the semantics of the underlying memory model and implements it under the simpler SC and assume that all the standard concurrency operations in multi-threaded programs are performed by invoking the corresponding calls to *API operations* over the SMA.

Furthermore, we implement efficient SMAs for *Total Store Ordering* (TSO) and *Partial Store Ordering* (PSO) semantics in our tool *LazySMA* and we experimentally demonstrate that it is very effective in finding bugs on a large set of benchmarks from the literature.

Then, we formally characterize the concept of *thread-asynchronous transition systems* that allows us to optimize intra-thread analysis by rearranging the operations of the threads. this thesis also shows that several concurrent verification techniques from the literature can easily be recast in our setting and thus be extended to weak memory models. We give *thread-asynchronous* SMA implementations for the SC, TSO, and PSO memory models that are based on the idea of individual memory unwinding. Finally, we instantiate our approach by developing a new, efficient BMC-based bug finding tool for multi-threaded C programs under SC, TSO, or PSO memory models, and experimentally demonstrate on the same set of benchmarks used for *LazySMA* that it is competitive with existing tools.

Contents

List of Figures

List of Tables

To my wonderful daughter Desiree T. . . .

Chapter 1

Introduction

1.1 Problem Description

Modern computer architectures use several cores that work in parallel in order to increase their computational power. With the spread of these architectures, the need to write multithreaded programs in order to take advantage of their multi-core processors continues to grow.

Typically, **multithreaded programs** consist of several components, called threads, which run in parallel and communicate with each other through *shared memory*. A shared memory is a sequence of memory locations of fixed size. The content of each location can be read or written using an explicit memory operation. The semantics of read and write operations depend upon the adopted memory model[1]. In the classical Sequential Consistency (SC) memory model [Lam79] when a shared memory location is written by a thread, its new valuation is immediately visible to all the other threads. For performance reasons, due to the gap between CPU and memory speeds, modern multi-core hardware architectures implement *weak memory models* (WMMs), that allows optimizations such as *instruction reordering*, *store buffering* or *write atomicity relaxation* [AG96]. For example, the *total store ordering* (TSO) memory model [MA15, SSO+10a] uses buffering to speed up execution time of multi-threaded programs. Each thread t is equipped with a local *store buffer* that is used to cache the write operations performed by t. Updates to the shared memory occur nondeterministically along the computation. Before updating, the effect of a cached write is visible only to the thread that has performed it. For *partial store ordering* (PSO) each thread is endowed with a store buffer for each shared memory location. In the rest of this chapter and throughout the thesis we use the terms *shared-memory multithreaded programs*, *multithreaded programs* and *concurrent programs* interchangeably.

[1] To provide semantics to concurrent programs, we reason at the programming level (as in [SSO+10a]), by replacing each memory access with a composition of read-and-write operations that are executed atomically and whose semantics is given in Section 2.3.1.

Developing concurrent programs **free of errors** is already hard when assuming the SC memory model, due to the large number of possible ways in which the different elements of a concurrent program can interact with each other. The reordering allowed by WMMs has made this task even harder, because they introduce additional executions that can lead to seemingly counter-intuitive results that confound the developers' reasoning. Thus, automatic verification tools that enable detection of errors in a systematic and symbolic way are essential.

Testing remains the most widely used approach to finding program errors. It is useful when a high percentage of the selected executions lead to a violation of the program specification. However, testing-only approaches, such as stress testing, remain highly ineffective for concurrency errors that manifest themselves rarely and are difficult to reproduce and repair [TDB14]. Such "Heisenbugs" [MQB+08] have become more prevalent on modern hardware architectures that use weak memory models, because WMMs introduce additional non-determinism that remains outside the control of the testing environment. Consequently, testing needs to be complemented by automated verification techniques that can handle concurrency (and the non-determinism it introduces) symbolically.

However, it is difficult to build efficient symbolic verification tools for realistic programming languages like C, and harder yet to extend them to handle concurrency. Concurrent programs are very different in nature from sequential programs. The large number of possible ways in which the different components of a concurrent program can interact (e.g., the different interleavings of a multi-threaded program) makes it diffcult to scale up approaches that explicitly explore individual interactions (e.g., ESBMC [CF11] or CHESS [MQB07]). Therefore, symbolic approaches to handling concurrency, where all possible interactions are analyzed simultaneously, are desirable, but these approaches still face the problem of exponentially growing symbolic state spaces.

Tools often compromise generality to achieve efficiency, by focusing on a specific memory model, typically sequential consistency (SC), and by folding the concurrency handling deep into their general verification approaches (see [AAA+15, AKT13, ABP11, BCDM15, WT15, ZKW15]). This in turn introduces a strong coupling between the two, which makes it hard to reuse existing tools and to generalize solutions to other memory models.

Despite the substantial advancements in the field of analysis and verification of concurrent programs assuming the SC memory model, the state-of-the-art technology for other WMMs is quite unsatisfactory. The starting point of our research is thus to extend existing successful approaches that have been implemented for the simpler SC memory model to WMMs, without loosing their efficiency and preserving their correctness.

SAT/SMT-based *bounded model checking* (BMC) [CGP99] is a technique which, given a program, a property, and a bound k, translates the program into a SAT or SMT (quantified free) formula that is satisfiable if and only if the property has a counterexample whose size is less than k. It has been used successfully to discover subtle errors in sequential

software, even at large scale [KT14, SKB+15]. Sequential BMC tools can be extended symbolically to the concurrent case by conjoining the formulae representing the effect of each individual thread in isolation with an additional formula representing the possible interferences caused by concurrent accesses to the shared memory [SW11, AKT13]. Since this additional formula effectively includes an axiomatization of the underlying memory model, this approach can in principle work for both sequential consistency (SC) and different WMMs. However, embodying a memory model at the formula level requires extensive (and non-reusable) modifications of the underlying sequential BMC tool, and can affect scalability since the resulting expressions are large and complex.

An alternative approach is *sequentialization*, which translates concurrent programs into sequential programs with data non-determinism that (under certain assumptions) behave equivalently, so that the different interleavings do not need to be treated explicitly during the analysis. This allows the reuse of existing sequential BMC tools. The idea of sequentialization was proposed by Qadeer and Wu [QW04], but the most popular sequentialization that could handle an arbitrary but bounded number of context switches was proposed by Lal and Reps [LR09].

Sequentializations can be used as basis of very effective bug finding tools, such as our MU-CSeq [TIF+15b] and Lazy-CSeq [ITF+14a]. This is witnessed by our tools' top rankings in recent software verification competitions but is also borne out in practice: for example, using Lazy-CSeq we discovered in 30 minutes a bug in the safestack benchmark [Vyu10], while all other approaches, including testing, failed [TDB14]. However, to the best of our knowledge, sequentializations have been developed only for SC, and not for any of the WMMs that are prevalent in modern computer architectures.

1.2 Our Contributions

The main goal of this thesis is to extend existing successful bug finding techniques that have been implemented for SC to more general WMMs. We describe a general approach that allows to combine different verification techniques with different memory models in the style of a plug-and-play architecture. We also instantiate our approach by developing new efficient BMC-based bug finding tools for multi-threaded C programs under SC, TSO, or PSO memory models, and show experimentally that they are very effective in finding bugs on a large set of benchmarks from the literature.

Separating Computation from Communication. Our main idea is to introduce an abstraction that allows us to separate the computation and the communication concerns of concurrent programs, without loosing the efficiency of existing approaches.

Shared Memory Abstraction. We introduce an abstract data type that encapsulates the semantics of the underlying memory model and implements it under the simpler

SC. We call this abstract data type *shared memory abstraction* (SMA). In particular, we assume that all the standard concurrency operations in multi-threaded programs are performed by invoking the corresponding calls to *API operations* over a *shared memory abstraction*. Introducing the SMA provides a separation of concerns between the shared memory and the control-flow related aspects of concurrent programs, such that the verifier design can focus on each of these aspects in isolation. Our approach bears some similarity to the axiomatic representation of memory models [SW11, AKT13] but the fundamental difference is that we work at the code level. We replace all accesses to shared memory items (i.e., reads from and writes to shared memory locations, and synchronization primitives like lock and unlock) by explicit calls to *API operations* over the given *shared memory abstraction*. For example, if `x` and `y` are two shared scalar variables then the statement $\texttt{x} = \texttt{y} + \texttt{x} + 3$ is translated into $\texttt{write}(\texttt{x}, \texttt{read}(\texttt{y}) + \texttt{read}(\texttt{x}) + 3)$.

Extending Lazy-CSeq to handle TSO and PSO. Following this approach, we propose novel SMA implementations for TSO and PSO that are carefully designed to optimize some parameters that lead, in combination with a lazy sequentialization targeting BMC tools, to efficient SAT/SMT encodings. We implement this approach by extending our *Lazy-CSeq* tool, that implements our Lazy Sequentialization for SC [ITF+14a]. We compare our implementation, called *LazySMA*, with other verification tools for WMMs. The experiments show that our approach delivers a comparable performance on simple benchmarks, but outperforms these tools on more complex problems.

Our general approach. Although this approach has been successful in combination with Lazy-CSeq, extending verification methods to other memory models by simply replacing the SMA implementation might not result in scalable verification tools, for the following reason. To preserve the correct semantics of the memory operations, these must be invoked in the same order as they appear along the run, which may be a bottleneck when we explore the state space of the program, both in case of the analysis based on summaries (e.g., BDD-based model checking) or bounded model checking. In the former, we must keep a cross-product of the states of all threads in the configurations; this is a well-known problem that leads to state-space explosion. In the latter, since context-switches can happen at any point, we must encode into the SAT/SMT formula the code of all threads for each of the context-switch points in the underlying bounded multi-threaded program, which leads to large formulas.

Some approaches from the literature instead explore the program executions by rearranging the order in which the memory operations of the different threads are executed, e.g., by simulating each thread to completion [LMP10, LR09]. More generally, the approach of verifying each thread in isolation is also the essence of the compositional approaches based on assume-guarantee reasoning [MC81, Jon83].

We generalize the ad-hoc approaches above, in our SMA framework where now intra-thread analysis can be optimized by rearranging the operations of the threads. This

requires that the used SMA implementation is *thread-asynchronous*, that is that its behaviours are insensitive to how the threads are interleaved. This allows us to transform freely the threads as long as we stay within the class of *thread-wise equivalent* programs, that is programs where the *intra-thread ordering* of the statements remains the same. We formalize the correctness of the derived design approach and also discuss how previous successful approaches from the literature fit into our setting, yielding correctness proofs for free; then we extend them to other more realistic memory models.

Thread-asynchronous SMA implementations. We also develop *thread-asynchronous* shared memory abstraction implementations for SC, TSO and PSO memory models, based on the idea of *individual shared memory location unwinding* (IMU for short), that extends the simpler concept of *memory unwinding* (MU for short), i.e. an explicit representation of the sequence of write operations into the shared memory, that we have introduced in [TIF+15b].

More specifically, we first give a novel implementation of *thread-asynchronous* shared memory abstraction based on the idea of *memory unwinding* for SC memory model. For the verification, we nondeterministically guess this unwinding and then explore all the executions of the program that respect this guess; each thread is simulated in isolation in according with the initially guessed memory unwinding. This approach is complementary to other approaches [FIP13a, LR09, LMP10] and explores an orthogonal dimension, i.e., the number of write operations of the shared memory. We follow up on the observation of Lu et al. [LPSZ08], that only a few memory accesses are relevant to finding concurrency bugs.

We implement our MU-based *thread-asynchronous* SMAs for SC in our prototype tool *MU-CSeq* and combine them with our eager sequentialization [TIF+15b]. We evaluate MU-CSeq over the set of concurrency benchmarks from the SV-COMP benchmark suite and we show that our tool compares well with existing tools with built-in concurrency handling as well as other sequentializations.

Finally, we extend the idea of *memory unwinding* to *individual shared memory location unwinding* and implement *thread-asynchronous* SMA implementations for the SC, TSO and PSO memory models that are based on this new concept. In particular, IMU splits the sequence of write operations of the shared memory into different sequences, one for each individual shared memory location. This simplifies the handling of store buffers in WMM and further leads to smaller formula sizes when used in combination with BMC verification tools.

We combine our IMU-based *thread-asynchronous* SMAs with our sequentialization [TIF+15b] in the protoype tool *IMU-CSeq*. We show experimentally, that this approach is very effective in finding bugs under TSO and PSO on the same set of benchmarks we use to evaluate *LazySMA*.

The material we use in this thesis is largely based on our published works [ITF+14a, ITF+14b, TNI+16a, TIF+14, TIF+15a, TNI+16b, TIF+15b, TNF+17]. With our tools, we have entered the last three editions of the Software Verification Competition SV-COMP in the concurrency category, where *MU-CSeq* [TIF+14, TIF+15a] has won two silver medals in the first two editions whereas *IMU-CSeq* [TNI+16b] has won the gold medal in the last edition.

1.3 Structure of the thesis

In Chapter 2, we provide a short background overview over the information useful to understand the material proposed in this thesis.

In Chapter 3, we describe our novel approach to design static analysis and bug finding tools for concurrent programs.

In Chapter 4, we give efficient SMA implementations for TSO and PSO and combine them with our Lazy sequentialization for SC memory model [ITF+14a]; in this way we obtain an efficient tool for the bug finding in concurrent programs under TSO and PSO memory model based on lazy sequentialization.

In Chapter 5, we give efficient *thread-asynchronous* SMA implementations for SC memory model, based on the idea of memory unwinding; we combine them with our eager sequentialization from [TIF+15b] for SC memory model in the tool MU-CSeq and we evaluate it over a set of concurrent benchmarks from SV-COMP.

In Chapter 6, we give efficient *thread-asynchronous* SMA implementations for SC, TSO and PSO memory model, based on the idea of individual shared memory location unwinding; we combine them with our eager sequentialization from [TIF+15b] for SC memory model in the tool IMU-CSeq and we evaluate it against the state of the art tool for concurrent programs under week memory models.

Chapter 2

Background

2.1 Alphabets, Words and Languages.

An alphabet Σ is a set of symbols. For an alphabet Σ, a word over Σ is a sequence of zero or more symbols from Σ. The empty word, denoted by ε, is the word formed of zero symbols. Recall that $w\varepsilon = \varepsilon w = w$ for any word w.

A *language* L over an alphabet Σ is a set of finite-length words over Σ.

2.2 Transition systems.

A *transition system* $\mathcal{A}$ is a tuple $(Q, \Sigma, \Delta, Q_0, F)$ where Q is a set of states, Σ is an alphabet, $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a transition relation, $Q_0 \subseteq Q$ is a set of *initial* states, and $F \subseteq Q$ is a set of final states.

A *run* π of $\mathcal{A}$ is a sequence $q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \ldots \xrightarrow{\sigma_d} q_d$ where $q_0 \in Q_0$ and $(q_{i-1}, \sigma_i, q_i) \in \Delta$ for each $i \in [1, d]$. Moreover, π is accepting if $q_d \in F$ and $\sigma_1 \ldots \sigma_d$ is the corresponding *word*. We denote by $L(\mathcal{A})$ the set of all words that correspond to accepting runs of $\mathcal{A}$.

Let $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, Q_{0,i}, F_i)$ be a transition system for $i \in \{1, 2\}$. The composition of A_1 and A_2, denoted $A_1|A_2$, is the standard cross product, i.e., $A_1|A_2$ is the transition system $(Q_1 \times Q_2, \Sigma, \Delta, Q_{0,1} \times Q_{0,2}, F_1 \times F_2)$ where Δ is the minimal set containing all tuples $((q_1, q_2), \sigma, (q_1', q_2'))$ such that either one of the following cases hold: 1. $\sigma = \varepsilon$, $(q_1, \varepsilon, q_1') \in \Delta_1$, $q_2 = q_2'$; or, 2. $\sigma = \varepsilon$, $q_1 = q_1'$, $(q_2, \varepsilon, q_2') \in \Delta_2$; or, 3. $\sigma \neq \varepsilon$, and $(q_i, \sigma, q_i') \in \Delta_i$ for $i \in \{1, 2\}$.

2.3 Multithreaded programs

A **multithreaded program** $\mathcal{P}$ consists of several components, called threads, which run in parallel and communicate with each other by using shared memory. A *shared memory* is a sequence of memory locations of fixed size. The content of each location can be read or written using an explicit memory operation. The semantics of read and write operations depend upon the adopted memory model.

Sequential Consistency (SC) is the classical memory model for concurrent programs with shared memory. However, modern multi-core hardware architectures do not implement the SC memory model; instead, they implement *weak memory models* (WMMs), that allow optimizations such as *instruction reordering*, *store buffering* or *write atomicity relaxation* [AG96]. Such optimizations introduce additional executions which are not captured by naively interleaving the sequential code. To describe what programmers can rely on with these architectures, vendors give architecture specifications which are full of hardware-level details and are difficult to understand at the programming language level. Further, some of these specifications are often ambiguous, and are sometimes incomplete and unsound, as observed by Sewl et al. [SSO+10a]. To provide semantics to concurrent programs, we then lift hardware models to the programming level, as in [SSO+10a], by replacing each memory access with a composition of read-and-write operations that are executed atomically and whose semantics is given below.

We consider multi-threaded programs with a simple C-`like` syntax including pointer arithmetics and dynamic memory allocation. We further consider POSIX-`like` threads with dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. Thread communication is implemented via shared memory in the form of global variables. The exact program syntax is defined in Section 2.3.2 by the grammar shown in Figure 2.2. We assume that all shared memory and synchronization operations are performed through an abstract data type, called *shared memory abstraction* (SMA for short), whose APIs are described in Section 3.2.2.

In this thesis, we describe a general approach that allows to combine different verification techniques with different memory models in the style of a plug-and-play architecture. We implemented our approach developing new efficient BMC-based bug finding tools for multi-threaded C programs under different memory models.

We remark that in our implementations we consider a simplified syntax and semantics of C. We allow for a restricted grammar with respect to the standard C [ISO11] and also introduce several assumptions; for instance, we assume atomicity of statements and non-deterministic values for initialized local variables. Our grammar and assumptions are even more restricted in comparison with multi-threaded C programs using POSIX threads [ISO09]. We examine only SC, TSO and PSO memory models, but there are many other more powerful memory models where the number of possible reorderings of

memory operations increase significantly. Further, we do not support the complex notion of sequence points in C and assume in-order instruction execution within threads.

These restrictions allow us to cover the Competition on Software Verification (SV-COMP)'s requirements, but exclude many subtle circumstances that actually do occur in practice [Boe05]. We could remove such limitations in order to design industrial-strength bug-finding tools, but it is to be left as future work.

2.3.1 Memory Models

In this thesis, beside SC, we also consider TSO and PSO; these models use buffering to speed up execution time of multi-threaded programs.

Sequential Consistency (SC). SC is the "standard model", where a write into the shared memory is performed directly on the memory location. This has the effect that the newly written value is instantaneously visible to all the other threads [Lam79]. A *configuration* of this model consists of a valuation of the shared locations, the addresses of such locations, the status of each created thread (i.e., if the thread is active, ready, suspended or terminated) and the status of each mutex (i.e., if it is locked or unlocked).

Total Store Ordering (TSO). The behaviour of the TSO memory model can be described using the simplified architecture shown in Figure 2.1 (cf. [SSO+10a]).

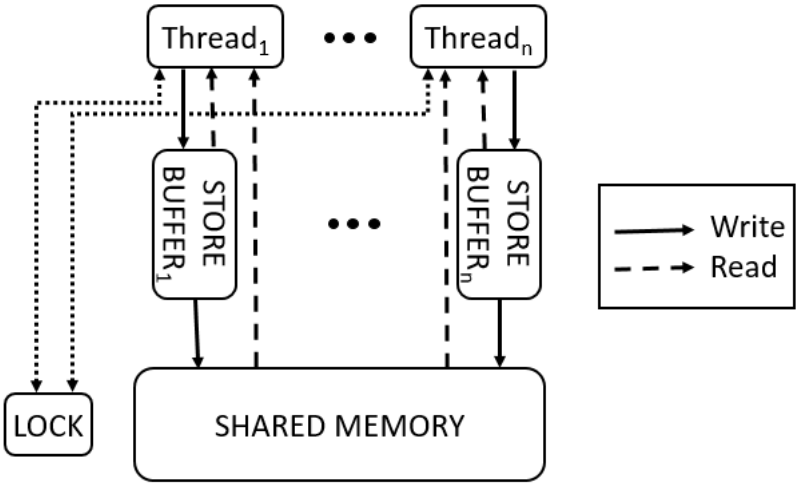

FIGURE 2.1: TSO architecture.

Each thread t is equipped with a local *store buffer* that is used to cache the write operations performed by t according to a FIFO policy. Updates to the shared memory occur nondeterministically along the computation, by selecting a thread, removing the oldest write operation from its store buffer, and then updating the shared memory

valuation accordingly. Before updating, the effect of a cached write is visible only to the thread that has performed it. A read by t of a variable y retrieves the value from the shared memory unless there is a cached write to y pending in its store buffer; in that case, the value of the *most recent* write in t's store buffer is returned. A thread can also execute a *fence*-operation to block its execution until its store buffer has been emptied.

Partial Store Ordering (PSO). The semantics of PSO is the same as for TSO except that each thread is endowed with a store buffer for each shared memory location.

We represent memory models as state transition systems whose configurations keep track of the valuation of the shared locations, the state of the store buffers (if any), the status of each created thread (i.e., active, ready, suspended or terminated) and the status of each mutex (i.e., locked or unlocked).

2.3.2 Syntax of programs.

The syntax of programs is defined by the grammar shown in Figure 2.2. Terminal symbols are set in typewriter font. $\langle n\ \texttt{t}\rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals `t`; x denotes a local variable, y an identifier of a shared variable, p an identifier of a pointer variable, m a mutex identifier, t a thread identifier and f a function name. We assume expressions e to be local variables, pointer values (returned by a read of a pointer variable), or integer constants that can be combined using mathematical operators. Boolean expressions b comprise the constants `true`, `false`, and Boolean variables, and can be combined using standard Boolean operations.

$$
\begin{array}{lll}
P & ::= & \texttt{init}(\,);\ (\mathit{type}\ f\ (\langle \mathit{dec},\rangle^*)\texttt{\{}(\mathit{dec};)^*\,\mathit{stm}\texttt{\}})^* \\
\mathit{dec} & ::= & \mathit{type}\ z \mid \mathit{type}*\ p \\
\mathit{type} & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
\mathit{stm} & ::= & \mathit{seq} \mid \mathit{conc} \mid \texttt{\{}\langle \mathit{stm};\rangle^*\texttt{\}} \\
\mathit{seq} & ::= & \texttt{assume}(b) \mid \texttt{assert}(b) \mid x = e \mid x = f(\langle e,\rangle^*) \mid \texttt{return}\ e \\
 & & \mid \texttt{if}(b)\ \ \mathit{stm}\ \texttt{else}\ \mathit{stm} \mid \texttt{while}(b)\ \ \mathit{stm} \\
\mathit{conc} & ::= & p = \texttt{address}(y,t) \mid p = \texttt{malloc}(e,t) \\
 & & \mid x = \texttt{read}(y,t) \mid x = \texttt{ind\_read}(p,t) \mid \texttt{write}(y,x,t) \mid \texttt{ind\_write}(p,x,t) \\
 & & \mid t = \texttt{create}(f,pt) \mid \texttt{join}(t_1,t_2) \mid \texttt{terminate}(t) \\
 & & \mid \texttt{fence}(t) \mid \texttt{lock}(m,t) \mid \texttt{unlock}(m,t)
\end{array}
$$

FIGURE 2.2: Syntax of multi-threaded programs.

A *multi-threaded* program consists of an `init()` invocation followed by a list of functions. `init()` instantiates a shared memory abstraction that captures a number of *shared* locations. Each function has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement.

A statement is either a sequential or a concurrent statement, or a sequence of statements enclosed in braces (*compound statement*).

A *sequential statement* can be an `assume`- or `assert`-statement, an assignment, a call to a function that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional statement, or a loop. All variables involved in a sequential statement are local. Local variables are considered uninitialised right after their declaration, which means that they can take any value from their domains. Therefore, until not explicitly set by an appropriate assignment statement, they can nondeterministically assume any value allowed by their type.

A *concurrent statement* involves an interaction with the shared memory abstraction and thus we have a different concurrent statement for each of the functions of the SMA API (other than `init` that is invoked only in the beginning).

We assume that a valid program P satisfies the usual well-formedness and type-correctness conditions. We also assume that P contains a function `main`, which is the starting function of the only thread that exists in the beginning. We call this the *main thread*. We further assume that there are no calls to `main` in P and that no other thread can be created that uses `main` as starting function.

2.3.3 Semantics of programs.

We adopt an interleaving semantics where only one of the *enabled* threads can be *active* at any given time. A computation of a *multi-threaded* program $\mathcal{P}$ is obtained by interleaving the computations of its threads.

Initially, only the `main` thread is active; new threads can be spawned from any thread by a call to the `create` function. Once created, a thread is added to the pool of inactive threads. A step is either the execution of a step of the active thread or is a *context-switch* that replaces the active thread with one of the inactive threads. At a context switch the current thread is suspended and becomes inactive, and one of the inactive threads is resumed and becomes the new active thread. When a thread is resumed its execution continues either from the point where it was suspended or, if it becomes active for the first time, from the beginning. A thread can pause its execution until another thread terminates, or it can acquire and release a mutex. Since threads can allocate memory dynamically using `malloc`, different threads can simultaneously access and alter shared dynamic data structures. A thread will no longer be available when its execution is terminated, i.e., there are no more steps that it can take.

For ease of presentation, we assume that each statement is atomic. Note that, this is not a restriction, in fact it is always possible to decompose a statement in a sequence of statements, each involving at most one shared variable [Mül06].

A *thread configuration* is a triple $\langle locals, pc, stack \rangle$, where *locals* is a valuation of the local variables, *pc* is the *program counter* that tracks the currently executing statement, and *stack* is a stack of function calls that works as usual. In the SC memory model a *configuration* of a multithreaded program is a tuple of thread configurations along with valuation of the global variables that are shared by all threads. In the TSO and PSO memory models the notion of configuration is extended to include the valuations of the store buffers.

In Section 3.2, we give a formal semantics of multi-threaded programs by the composition of two transition systems $\mathcal{C}|\mathcal{M}$, where $\mathcal{C}$ is the *control-flow transition system* that captures the control flow of the program and $\mathcal{M}$ is the *shared memory abstraction transition system* that implements the behaviours of the SMA.

2.3.4 Reachability problems

Let P be a concurrent program. The *reachability problem* asks whether there is a reachable assertion-failure configuration of P. For concurrent programs, reachability is undecidable [Ram00]. By restricting to only computations with a bounded number of context switches, not only the problem becomes decidable but it has shown to be an effective method to finding bugs in concurrent programs.

2.4 Propositional Logic

The alphabet of propositional logic consists of:

- a countably-infinite set $Lp = (x_1, x_2, \dots)$ of *propositional variables*
- the *logical connectives* NOT ($\neg$), OR ($\vee$)
- parentheses (and).

Further logical connectives, such as AND ($\wedge$), IMPLIES ($\rightarrow$) and EQUIVALENT TO ($\leftrightarrow$), are derived from the standard connectives ($\neg$,$\vee$) in the usual way.

The set WFF of *well-formed formulae of propositional logic* is inductively defined as follows:

- $\forall\ x$, if $x \in Lp \implies x \in WFF$
- if $\alpha, \beta \in WFF \implies \alpha \vee \beta \in WFF$
- if $\alpha \in WFF \implies \neg\alpha \in WFF$

- if $\alpha \in WFF \implies (\alpha) \in WFF$
- nothing else $\in WFF$

An *interpretation* is a function $Int : WFF \to \{\bot, \top\}$ that assigns either TRUE ($\top$) or FALSE ($\bot$) to every propositional symbol in WFF.

Given a well-formed formula φ and an interpretation Int, either Int satisfies φ or not. This is indicated with $Int \models \varphi$ or $Int \not\models \varphi$, respectively, and inductively defined by the following rules:

- if $\varphi = x$, with $x \in Lp$, then $Int \models \varphi$ if and only if $Int(x) = \top$
- if $\varphi = \neg\psi$, with $\psi \in WFF$, then $Int \models \varphi$ if and only if $Int \not\models \psi$
- if $\varphi = \psi_1 \vee \psi_2$, with $\psi_1, \psi_2 \in WFF$, then $Int \models \varphi$ if and only if $Int \models \psi_1$ or $Int \models \psi_2$.

A formula φ is *satisfiable* if there exists an interpretation Int under which the formula is true; if no such interpretation exists, φ is *unsatisfiable*, or a *contradiction*. A formula that evaluates to true under all possible assignments is *valid*, or a *tautology*.

2.4.1 Boolean Satisfiability

The *boolean satisfiability problem*, also known as *SAT* (for short), consists in determining whether a given propositional formula is satisfiable.

A *SAT solver* is a decision procedure that takes as input a given formula φ and returns as output an assignment Int of the variables of φ that satisfies φ, or FALSE if the φ is unsatisfiable.

In general, SAT is an NP-complete problem [Coo71] and no one knows a polynomial algorithm that could solve it. However, modern SAT solvers use sophisticated heuristics developed in the last 40 years that are very effective on several big real world instances. In general, it is hard to predict what a good SAT encoding is [KSMS11]. However, as a rule of thumb, it is desirable to produce SAT formulas of small sizes as it often leads to better performance in modern SAT solvers.

2.5 Bounded Model Checking

Bounded model-checking (BMC) is a technique where a program is first transformed into a *bounded program* by unrolling all loops up to a given bound, and inlining functions

(a) input program

```
x:=x-y;

if(x==1) then
   x:=2;
else
   x:=x+1;

assert(x==6);
```

(b) SSA form

```
x1:=x0-y0;



x2:=2;
x3=x1+1;
x4=(x1==1)?x2:x3

assert(x4==6);
```

(c) verification condition

$$\varphi_C := \quad x_1 = x_0 - y_0 \wedge$$
$$x_2 = 2 \wedge$$
$$x_3 = (x_1 + 1) \wedge$$
$$x_4 = (x_1 = 1)?x_2 : x_3)$$

$$\varphi_P := \quad x_4 = 6$$

FIGURE 2.3: Bounded model-checking: (a) simple input program, (b) static single assignment (SSA) form and (c) verification condition

(if a function is recursive, it is inlined up to the given bound). Conceptually, a loop statement is removed by replicating its body k times, where each such copy is guarded using an if-statement that uses the loop condition, and a loop unwinding assertion with the negated loop condition is added after the last copy (*loop unwinding*). A function call is removed by inlining its code, and replacing the return statements with an assignment for the return value (if any) and a goto to the end of the function. Recursive calls are handled similarly to the case of loop unwinding by asserting in the end that the recursion does not pass the bound (*function-call unwinding*).

Thus, the resulting program captures all possible computations of the original program for a bounded number of steps. Then the program is transformed into SSA (static single assignment) form [AKT13]; this guarantees that each variable is assigned at most once (see Figure 2.3(b)).

Then the intermediate program in SSA form is translated into a SAT or SMT (quantified free) formula (see Figure 2.3(c)). This formula is checked for satisfiability. A satisfiability assignment for variables is then used to build a counter-example of the program. If the formula is not satisfiable, it means that no error is found in the given bound. Therefore, BMC techniques are appealing only for discovering shallow bugs (as is often the case for concurrent programs [MQB+08]). The success of BMC techniques is borrowed by efficient SAT/SMT solvers that can nowadays handle formulas with several millions of variables [JLBRS12, CGBD12].

BMC is an under-approximation technique and can thus be used only to find violations of the property up to the bound k (i.e., bug finding). In order to *prove* properties we can either select a bound k that is "big enough" (i.e., exceeds the *completeness threshold* [Bie09]) or uses techniques like k-induction [SSS00] that build on top of BMC.

2.6 Bounded Model Checking of Concurrent Programs

We recall that a **Concurrent program** $\mathcal{P}$ consists of several components, called threads, which run in parallel and communicate with each other by using shared memory. A computation of $\mathcal{P}$ is obtained by interleaving the computations of its threads. The number of possible interleavings grows exponentially with the number of threads and statements in the program.

Attempts to extend BMC to the analysis of concurrent programs face the problem of state space explosion. There are two main approaches to address this problem: *context bounding* limits the analysis to a given number of context switches and *partial-order reduction* prunes the search space by avoiding the exploration of multiple executions leading to the same state.

2.6.1 Context-Bounded Analysis

State space explosion can be reduced by limiting the number of interactions that are analyzed by a verification tool. Context-bounded analysis (CBA) is a technique that bounds the number of context switches [QR05, LR09, LMP09a]. In [MQ07, QW04], it was empirically shown that most of the errors in concurrent programs manifested themselves in a few context-switches.

A context-bounded analysis can be implemented as SAT/SMT-based *bounded model checking* (BMC), which has been used successfully to discover subtle errors in sequential software, even at large scale [KT14, SKB+15]. Sequential BMC tools can be extended symbolically to the concurrent case by conjoining the formula representing the effect of each individual thread in isolation with a second formula representing the possible interferences caused by concurrent accesses to the shared memory [SW11, AKT13]. Since this second formula effectively includes an axiomatization of the underlying memory model, this approach can in principle handle sequential consistency (SC) as well as different WMMs.

An alternative approach is based on *sequentialization*, which translates concurrent programs into sequential programs with data non-determinism that (under certain assumptions) behave equivalently, so that the different interleavings do not need to be treated explicitly during the analysis. This allows the reuse of existing sequential BMC tools.

2.6.2 Sequentialization

A sequentialization translates a concurrent program P into a non-deterministic sequential program P_{seq} that (under certain assumptions) behaves equivalently. Sequentialization

can be seen as a kind of pre-processing for existing mature sequential program verification tools.

A first attempt of sequentialization could be to implement it as an interpreter which simulates the concurrent program tracking at all times the cross product of the local states of each thread. For theoretical reasons, this kind of solution is ineffective due to the state-space explosion. To make the approach effective, P_{seq} should not track the whole state-space of the concurrent program, as in the cross product of the thread states. All sequentializations that have been proposed in literature only track one local state at a time and use $2k$ copies of the shared variables [LR09, LMP09a, LMP12], for a given parameter k. Under these restrictions, such approaches can only cover a subset of computations in which each thread can at most interact k times with the other threads. By increasing the parameter k, we can capture more computations, but this of course comes with a cost in terms of computational resources.

The first sequentialization of concurrent programs was proposed in the KISS paper (*Keep It Simple and Sequential*) by Qadeer and Wu in 2004 [QW04]. The sequentialization is bounded on a finite number of threads and two context-switches, while the length of the runs is unbounded. The resulting sequential program P_{seq} uses only one stack which is partitioned into contiguous blocks where each of them corresponds to the local stack of one thread. On the top of the stack we have the local stack of the thread that is executing currently. The scheduler non-deterministically can do, at any time, one of the following options:

1. terminate the thread currently executing and popping its contiguous block; then the thread whose block is just below is resumed;
2. schedule another thread by calling its starting function.

The authors have also implemented their technique in a tool called KISS which is a pre-processing for the SLAM [BR] model checker for sequential C programs. They used KISS to finding several bugs in Windows NT device drivers.

In 2007, Qadeer and Masuvathi empirically showed that most of the errors in concurrent programs manifest themselves in a few preempting context-switches [MQ07]. In the same paper, the authors showed that a fair percentage of state-space is reached through executions with a few preemptions.

Sequentializations can be classified in *eager* and *lazy*. Eager sequentializations guess the different values of the shared memory before the verification, which means that they can explore infeasible computations that need to be pruned away afterwards. In contrast, lazy sequentializations [LMP09a] only explore feasible computations.

2.6.2.1 Eager Sequentialization

Lal and Reps in 2009 proposed a sequentialization (LR) [LR09] which allows the handling of a fixed number of context-switches between a statically-created set of threads.

The LR sequentialization translates a concurrent program P into a sequential program P_{seq}, which non-deterministically simulates a *round-robin schedule*. P_{seq} simulates each thread to completion and uses a copy of the shared global memory for each round. The initial values of all memory copies are nondeterministically guessed in the beginning (*eager* exploration). At the end a checker prunes away all infeasible runs where the initial values guessed for one round do not match the values computed at the end of the previous round.

Lal and Reps' technique has been implemented in several tools. Rek [CGS11] is targeted to real-time systems. STORM [LQR09], Poirot [Qad11] and Corral first translate concurrent C programs into Boogie and then implement Lal and Reps at the Boogie level. Corral also integrates Lal and Reps into a CEGAR approach. CSeq [FIP13b, FIP13a] implements Lal and Reps directly on concurrent C programs and it also handles dynamic thread creation. CSeq can use several BMC back-ends, like CBMC, ESMBC and LLBMC.

In 2012, Bouajjani and Emmi proposed a sequentialization for distributed applications where threads communicate through FIFO channels [BE12].

MU-CSeq. In our paper [TIF+15b], we describe a new sequentialization-based approach to the symbolic verification of multithreaded programs with shared memory and dynamic thread creation, whose main novelty is the idea of *memory unwinding* (MU), i.e., an explicit representation of the sequence of write operations into the shared memory.

Recent empirical studies have pointed out other common features for concurrency errors, but these have not yet been exploited for practical verification algorithms. In particular, Lu et al. [LPSZ08] observed that "almost all [...] concurrency bugs are guaranteed to manifest if certain partial order among *no more than 4 memory accesses* is enforced."

In [TIF+15b], we follow up on their observation that only a few memory accesses are relevant, and propose a corresponding new approach to the automated bug finding in concurrent programs. The proposed approach simulates the executions of a multithreaded program but bounds the total number of write operations into the shared memory that can be read by threads other than the one performing the writing. It is related to context-bounded analyses [QR05, LR09, LMP09a] but the bounding parameter is different, which allows an orthogonal exploration of the search space.

Figure 2.4(a) shows the C-code of a multi-threaded implementation of a nondeterministic Fibonacci-function. This example from the SV-COMP benchmark suite uses two threads

```
int i = 1, j = 1;


void f1() {
  i = i + j;
  i = i + j;
  i = i + j;
  i = i + j;
  i = i + j;
}

void f2() {
  j = j + i;
  j = j + i;
  j = j + i;
  j = j + i;
  j = j + i;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, ((void*)0), f1, ((void*)0));
  pthread_create(&t2, ((void*)0), f2, ((void*)0));
  assert(j < 144);

}
```

(a)

pos	*thr*	*var*	*val*
0	2	i	2
1	2	i	3
2	2	i	4
3	2	i	5
4	3	j	5
5	3	j	10
6	3	j	15
7	2	i	15
8	3	j	30
9	3	j	45

(c)

```
int i = 1, j = 1;
int ct;  bool _error= F;
int th_pos[3];

void f1_S() {
 write(⟨i⟩, read(⟨i⟩, ct)+read(⟨j⟩, ct) , ct);
 write(⟨i⟩, read(⟨i⟩, ct)+read(⟨j⟩, ct) , ct);
 write(⟨i⟩, read(⟨i⟩, ct)+read(⟨j⟩, ct) , ct);
 write(⟨i⟩, read(⟨i⟩, ct)+read(⟨j⟩, ct) , ct);
 write(⟨i⟩, read(⟨i⟩, ct)+read(⟨j⟩, ct) , ct);
}

void f2_S() {
 write(⟨j⟩, read(⟨j⟩, ct)+read(⟨i⟩, ct) , ct);
 write(⟨j⟩, read(⟨j⟩, ct)+read(⟨i⟩, ct) , ct);
 write(⟨j⟩, read(⟨j⟩, ct)+read(⟨i⟩, ct) , ct);
 write(⟨j⟩, read(⟨j⟩, ct)+read(⟨i⟩, ct) , ct);
 write(⟨j⟩, read(⟨j⟩, ct)+read(⟨i⟩, ct) , ct);

void main_S() {
  int t1, t2;
  t1 =CREATE(f1_S, ct);
  t2 =CREATE(f2_S, ct);
  check_return;
  if(read(⟨j⟩, ct) >= 144) {
    _error:= T;  return 0;
  }
}

void main(void) {...}
```

(b)

pos	*threads*	*exposed writes*
0	$\{2\}$	$\{(i = 4)\}$
1	$\{2, 3\}$	$\{(i = 5), (j = 10)\}$
2	$\{3\}$	$\{(j = 15)\}$
3	$\{2, 3\}$	$\{(i = 15), (j = 45)\}$

(d)

FIGURE 2.4: Multi-threaded Fibonacci: (a) C-code for 5 iterations, (b) translated code, and sample memory unwindings: (c) fine-grained and (d) coarse-grained.

$f1$ and $f2$ to repeatedly increment the shared variables i and j by j and i, respectively. With a *perfect* interleaving (i.e., a round-robin schedule with context switches after each assignment) the variables i and j take on the consecutive values from the Fibonacci-series, and the program terminates with $i = \mathit{fib}(11) = 89$ and $j = \mathit{fib}(12) = 144$ if this interleaving starts with $f1$. Any other schedule will lead to smaller values for i or j. The `main` function first creates the two threads $f1$ and $f2$, then uses two join statements to ensure that both threads run to completion, and finally checks the outcome of the choosen interleaving.

Here, we first give a general overview of the main concepts of our approach, namely the memory unwinding and the simulation of the read and write operations, before

we describe the sequentialization translation. We use the Fibonacci-example above to illustrate the concepts.

High-level description. Our approach is based on a code-to-code translation of any concurrent program P into a corresponding sequential program $P_{\mathtt{W},\mathtt{T},\mathtt{V}}$ that captures all the executions of P involving at most $\mathtt{W}$ write operations in the shared memory, $\mathtt{T}$ threads and $\mathtt{V}$ memory locations. We denote the original concurrent program with P and the sequentialized program $P_{\mathtt{W},\mathtt{T},\mathtt{V}}$ with P_S in the rest of this section.

A core concept in this translation is the *memory unwinding*. A $\mathtt{W}$*-memory unwinding* $\mathcal{M}$ of P is a sequence of writes $w_0 \dots w_{\mathtt{W}-1}$ of P's shared variables; each w_i is a triple (t_i, var_i, val_i) where t_i is the identifier of the thread that has performed the write operation, var_i is the name of the written variable and val_i is the new value of var_i. A *position* in an $\mathtt{W}$-memory unwinding $\mathcal{M}$ is an index in the interval $[0, \mathtt{W}-1]$. An execution of P *conforms to* a memory unwinding $\mathcal{M}$ if the sequence of its writes in the shared memory exactly matches $\mathcal{M}$.

The approach allows us to vary which write operations are represented and thus exposed to the other threads. This leads to different strategies with different performance characteristics. In a *fine-grained* MU (see Figure 2.4(c)) *every* write operation is represented explicitly and individually while *coarse-grained* MU only represents a subset of the writes, but group together multiple writes. In a coarse-grained MU (see Figure 2.4(d)) we store at each position a partial mapping from the shared variables to values, with the meaning that the variables in the domain of the mapping are modified from the previous position and the value given by the mapping is their value at this position.

Fig. 2.4(c) gives a fine-grained memory unwinding that conforms to an execution of the multithreaded program given in Fig. 2.4(a), namely the execution given by (we omit the main thread): the first three assignments of $f1$, followed by a read of i by $f2$, then the fourth assignment of $f1$, the completion of the first two assignments of $f2$, the read of j by $f1$, the third assignment of $f2$, the last assignment of $f1$ and the remaining assignments of $f2$. Note that a memory unwinding can be *unfeasible* for a program, in the sense that no execution of the program conforms to it. Conversely, multiple executions can also conform to one memory unwinding, although this is not the case for the Fibonacci-example. Fig. 2.4(d) gives a coarse-grained memory unwinding that conforms to the same execution described above.

We use a memory unwinding $\mathcal{M}$ to explore the runs of P that conform to it by running each thread t separately. The idea is to use the memory unwinding for the concurrent statements (which involve the shared memory) and execute the sequential statements directly. In particular, when we execute a *write* of t in the shared memory, we check that it matches the next write of t in $\mathcal{M}$. However, a *read* of t in the shared memory is more involved, since the sequence of reads is not explicitly stored in the memory unwinding. We therefore need to guess nondeterministically the position in the unwinding from which

we read. Admissible values are all the positions that are in the range from the current position (determined by previous operations on the shared memory) to the position of t's next write in $\mathcal{M}$. The nondeterministic guess ensures that we are accounting for all possible interleavings of thread executions that conform to the memory unwinding.

For example, consider again the 10-memory unwinding of Fig. 2.4(c). The execution of $f1$ is simulated over this as follows. The first four writes are matched with the first four positions of the unwinding; moreover, the related reads are positioned at the current index since they are each followed by the write which is at the next position in the memory unwinding. The fifth write is matched with position 7. The corresponding read operations can be assigned nondeterministically to any position from 3 to 6. However, in order to match the value 15 with the write, the read of j must be positioned at 5. Note that the read of i can be positioned anywhere in this range since it was written last time at position 3.

When simulating one thread, we assume that all the writes executed by the other threads, and stored in the memory unwinding, indeed all occur and, moreover, in the ordering shown in $\mathcal{M}$. Thus, for the correctness of the simulation, for each thread t we must ensure not only that each of its writes involving the shared variables conforms to the write sequence in the guessed memory unwinding, but also that all the writes claimed in the unwinding are actually executed. Further, t should not contribute to the computation before the statement that creates it has been simulated. This can be easily enforced by making the starting position of the child thread to coincide with the current position in $\mathcal{M}$ of the parent thread when its creation is simulated.

Construction of P_S. The program P_S first guesses an `W`-memory unwinding $\mathcal{M}$ and then simulates a run of P that conforms to $\mathcal{M}$. The simulation starts from the main thread (which is the only active thread when P starts) and then calls the other threads one by one, as soon as their thread creation statements are reached. Thus, the execution of the parent thread is suspended and then resumed after the simulation of the child thread has completed. Essentially, dynamic thread creation in P is modeled by function calls in P_S. P_S is formed by a main, and a new procedure p_S for each procedure p of P. It uses some additional global variables: *_error* is initialized to false and stores whether an assertion failure in P has occurred; *ct* stores the current thread; the array `th_pos` stores the current position in the memory unwinding for each thread. The main procedure of P_S is given in Fig. 2.5. First, we call `init(V,W,T)` that guesses an `W`-memory unwinding with `V` variables and `T` threads, and then `create`$(main_S, 0)$ that registers the main thread and returns its *id*. Note that we encode each of P's shared variables y with a different integer $\langle y \rangle$ in the interval $[0, \mathtt{V}-1]$ and each thread with a different integer $\langle t \rangle$ in $[0, \mathtt{T}-1]$; once `T` threads are created `create` returns -1 (an

```
void main(void) {
 init(V,W,T);
 ct := create(main_S,0);
 main_S(x_1,...,x_k);
 terminate(ct);
 finish();
 assert(_error ≠ 1)  }
```

FIGURE 2.5: P_S: `Mu main()`.

invalid *id*) that causes the thread not to be executed. The parameter passed to `create` is the *id* of the thread that is invoking the thread creation. For the creation of the main thread we thus pass 0 to denote that this is the first created thread. The call to $\texttt{main}_S$ starts the simulation of the main thread. Then, we check that all the write operations guessed for the main thread have been executed (by `terminate`), and all the threads involved in the guessed writes have been indeed simulated (by `finish`). If either one of the above checks fails, the simulation is considered infeasible and thus aborted. The global variable `_error` is used to check whether an assertion has been violated. It is set to T in the simulation of the threads of P whenever an assertion gets violated and is never reset.

Each p_S is obtained from p according to the transformation function $[\cdot]$ defined inductively over the programs syntax by the rules given in Fig. 2.6. For example, Fig. 2.4(b) gives the transformations for the functions of the Fibonacci program from Fig. 2.4(a). There we use the macro `CREATE` as a shorthand for the code given in the translation rules for the create statement. Also, we have omitted the declaration of *tmp* in the functions $f1_S$ and $f2_S$ since it is not used there, and reported the translation of the assignments in a compact form.

1. $[type\ p\,(par^*)\{dec^*\ stm\}]\ ::=\ type\ p_S\,(par^*)\{dec^*;\ \texttt{uint}\ tmp;\ [stm]\}$
2. $[\{\langle stm;\rangle^*\}]\ ::=\ \{\langle [stm];\rangle^*\}$

Sequential statements: (x is local)

3. $[\texttt{assume}(b)]\ ::=\ \texttt{check\_return; assume}(b)$
4. $[\texttt{assert}(b)]\ ::=\ \texttt{check\_return; if}(\neg b)\ \{\texttt{\_error}:= T;\ \texttt{return}\ 0\}$
5. $[\texttt{return}\ e]\ ::=\ \texttt{return}\ e$
6. $[x = e]\ ::=\ x = e$
7. $[x = f(e_1, \ldots, e_n)]\ ::=\ x = f_S(e_1, \ldots, e_n);\ \texttt{check\_return;}$
8. $[\texttt{if}\,(b)\ stm\ \texttt{else}\ stm]\ ::=\ \texttt{if}\,(b)\ [stm]\ \texttt{else}\ [stm]$
9. $[\texttt{while}\,(b)\,\{\ stm\ \}\ ::=\ \texttt{while}\,(b)\,\{\ \texttt{check\_return;}\ [stm]\ \}$

Concurrent statements: (x is local, y is shared, ct contains the current tread id)

10. $[p := address(y,t)]\ ::=\ p := address(\langle y\rangle, t);$
11. $[p := malloc(e,t)]\ ::=\ p := malloc(e,t);$
12. $[x = read(y,t)]\ ::=\ x = read(\langle y\rangle, t);$
13. $[x = ind\_read(p,t)]\ ::=\ x = ind\_read(\langle p\rangle, t);$
14. $[write(y,x,t)]\ ::=\ write(\langle y\rangle, x, t);$
15. $[ind\_write(p,x,t)]\ ::=\ write(\langle p\rangle, x, t);$
16. $[t = \texttt{create}\ (p(e_1, \ldots, e_n), pt)]\ ::=\ \texttt{check\_return;}\ tmp = ct;\quad t:= \texttt{create}(p_S, ct);$
    $\texttt{if}(t > 0)\ \{\ ct = t;\ \ p_S(e_1, \ldots, e_n);\ \ \};$
    $ct = tmp;$
17. $[\texttt{join}\ (ct,t)]\ ::=\ \texttt{join}(ct,t);$
18. $[\texttt{terminate}\ (t)]\ ::=\ \texttt{terminate}(t);$
19. $[\texttt{fence}\ (t)]\ ::=\ \texttt{fence}(t);$
20. $[\texttt{lock}\ (v,ct)]\ ::=\ \texttt{lock}(\langle v\rangle, ct);$
    $[\texttt{unlock}\ (v,ct)]\ ::=\ \texttt{unlock}(\langle v\rangle, ct);$

FIGURE 2.6: Mu-CSeq Rewriting rules.

The transformation adds a local variable *tmp* that is used to store the current thread id when a newly created thread is simulated. The sequential statements are left unchanged except for the injection of the macro `check_return` that is defined as "`if(is_th_terminated(t)) then return 0;`". Function `is_th_terminated`(t) returns true either when the thread t has previously been interrupted, or it is non-deterministically interrupted now. The macro is injected after each function call, as a first statement of a

loop, and before any `assume`- and `assert`-statement; in this way, when the simulation of the current thread has finished or is aborted, we resume the simulation of the parent thread.

The concurrent statements are transformed as follows. Each of them, excluding `terminate`, begins with the macro `check_return`, i.e., they are executed only if the thread simulation is not terminated yet. A *write* of v into a shared variable x in thread t is simulated by a call to `write` that checks that the next write operation of t in the guessed memory unwinding (starting from the current position) writes x and the guessed value coincides with v. Otherwise, the simulation of all threads must be aborted as the current execution does not conform to the memory unwinding. If t has already terminated its writes in the memory unwinding, we return immediately; otherwise we update t's current position to the index of its next write operation. A *read* of a shared variable x in thread t is simulated by a call to `read`. The read value is determined by nondeterministically guessing a position i between the current position for ct and the position prior to the next write operation of t in the memory unwinding. Thus, we return the value of the write operation involving x that is at the largest position $j \leq i$ and then update the stored position of t to i.

As above, we use `create` and `terminate` for the translation of thread creations, but we check whether the thread creation was successful before calling the simulation function. Also, we save the current thread id in a temporary variable during the execution of the newly created thread.

The remaining concurrent statements are simulated by corresponding functions. A call `join`(ct, t) returns if and only if t is terminated at the current position of thread ct in the memory unwinding. Otherwise, the simulation is aborted. A call `lock`$(ct, \langle v \rangle)$ gives exclusive usage of the mutex $\langle v \rangle$ to thread ct. If the mutex is already locked, the whole simulation is aborted. The unlocking is done by calling `unlock`.

Note that for join, lock, and unlock operations we choose to abort also computations that are still feasible, i.e., the lock could still be acquired later or we can wait for another thread to terminate. This is indeed correct for our purposes, in the sense that we are not missing bugs for this. In fact, we can capture those computations by scheduling the request to acquire the lock or to join exactly at the time when this will be possible, and by maintaining the rest of the computation unchanged.

2.6.2.2 Lazy Sequentialization

The set of reachable states of a concurrent program is often much smaller than the whole state space. *Lazy* exploration is a technique that explores only the reachable states. In [LMP09b], the authors give a fixed-point algorithm for the verification of concurrent

Boolean programs; while in [LMP09a] (LMP), the same authors gave a sequentialization algorithm.

In [LMP10, LMP12], the authors have extended LMP to concurrent Boolean programs with an unbounded number of threads .

Like Lal and Reps, LMP also uses several copies of the shared memory, but these copies are always computed and not guessed. However, since the local state of a thread is not stored on context switches, when a thread is resumed, it is necessary to recompute the values of its thread-local variables from scratch. This recomputation poses no problem for tools that compute function summaries [LMP09a, LMP10] since they can re-use the summaries from previous rounds. However, for BMC, since this recomputation need to be simulated at each shared memory access, this leads to exponentially growing formula sizes [GHR10].

In [ITF+14b], our solution is to keep the values of the thread-local variables between the different function activations (by turning them into `static` variables), thus avoiding their recomputation and then the exponential growth of the formula size [GHR10], as observed above. The size of the formulas is instead proportional to the product of the size of the original program, the number of threads and the number of rounds.

Lazy-CSeq. Lazy-CSeq's sequentialization [ITF+14b] schema aggressively exploits the structure of bounded programs and works well with BMC-based backends [ITF+14a]. Given a multi-threaded program P, the sequentialized program $P_{\mathtt{seq}}$ simulates all bounded executions of the original program for a bounded number of rounds, where in each round each thread is scheduled at most once. It is composed of a main driver and an individual simulation function for each thread, where function calls and loops of the input program are inlined and unrolled, respectively, and each *visible*[1] statement is denoted by an integer label and guarded by some light-weight context switch simulation code.

```
int m;
int c=0;
void P(void *b) {
  int tmp=(*b);
  lock(&m);
  if(c>0)  c++;
  else{
    c=0;
    while(tmp>0){
      c++;
     tmp--;
    }
  }
  unlock(&m);
}
```

```
void C(){
  assume(c>0);
  c--;
  assert(c>=0);
}
int main(void) {
  int x=1,y=5;
  int p0,p1,c0,c1;
  create(&p0,P,&x);
  create(&p1,P,&y);
  create(&c0,C,0);
  create(&c1,C,0);
  return 0;
}
```

FIGURE 2.7: Producer/Consumer Example

[1] We say a statement is *visible* if its execution involves either a read or a write operation of a shared variable, and *invisible* otherwise.

For the ease of presentation, we can assume that P indeed already consists of just $n+1$ functions $f_0, \ldots, f_n$ (where f_0 denotes the `main` function) and creates at most n threads respectively with start functions $f_1, \ldots, f_n$, respectively (we can clone the start functions of each thread if necessary). We denote with $f_{\texttt{seq}_i}$ the function of $P_{\texttt{seq}}$ corresponding to thread f_i in P, for $i = 1, \ldots, n$. In each round, the main driver calls each such $f_{\texttt{seq}_i}$; however, the context switch simulation code ensures that their execution does not repeat all the steps done in the previous rounds but instead jumps (in multiple hops) back to the stored program location where the previous round has finished, executes the number of steps it is allowed in the corresponding round (which is guessed by the driver) and then jumps (again in multiple hops) to the end.

While simulating P, the sequentialized program P_{seq} maintains the data structures below; here `T` is a symbolic constant denoting the maximal number of threads in the program, i.e., $n+1$.

- `bool active[T];` tracks whether a thread is active, i.e., has been created but not yet terminated. Initially, only `active[0]` is `true` since f_0 simulates the `main` function of P.
- `void* arg[T];` stores the argument used for thread creation.
- `int size[T];` stores the largest label used as jump target in the thread simulation functions.
- `int pc[T];` stores the label of the last context switch point for each thread simulation function.
- `int ct;` tracks the index of the thread currently under simulation.
- `int cs;` contains the label at which the next context switch will happen.

```
void main(void) {
  for (r=1; r<=K; r++){
    ct=0;
    //only active threads
    if(active[ct]){
      //next context switch
      cs=pc[ct]+nondet_uint();
      //appropriate value?
      assume(cs<=size[ct]);
      //thread simulation
      fseq_0(arg[ct]);
      //store context switch
      pc[ct]=cs;
    }
    .........
    .........

    ct=n;
    if(active[ct]){
      cs=pc[ct]+nondet_uint();
      assume(cs<=size[ct]);
      fseq_n(arg[ct]);
      pc[ct]=cs;
    }
  }
}
```

FIGURE 2.8: Lazy-CSeq main driver.

Fig. 2.8 shows the new function `main` in $P_{\texttt{seq}}$, which drives the simulation. Each iteration of the loop simulates one entire round of a computation of P. The simulation of

each thread $f_{\texttt{ct}}$ invokes the corresponding simulation function $f_{seq_{ct}}$ with the argument `arg[ct]` that was originally used to create the thread. The order in which the functions are called corresponds to the round-robin schedule ρ, here $0, \ldots, n$. For each active thread the driver thus executes the following steps: (i) nondeterministically guess the label for next context switch and store it in `cs`, (ii) check that the value is appropriate, (iii) simulate the thread from `pc[ct]` through to `cs`, and (iv) store `cs` in `pc[ct]`, since in the next round the computation must restart from this label.

Crucially, the values of the thread-local variables are kept between the different function activations (by turning them into static variables), which avoids their recomputation and thus the exponentially growing formula sizes observed by Ghafari et al. [GHR10]. The translation also has very small memory overheads and very few sources of nondeterminism, so that it produces simple formulas, and is thus very effective in practice [Bey15].

```
bool active[T]={1,0,0,0,0};
int cs,ct,pc[T],size[T]={5,8,8,2,2};
#define GUARD(L) assume(cs>=L);
#define J(A,B) \
  if(pc[ct]>A||A>=cs) goto B;
pthread_mutex_t m;
int c=0;
void P0(void *b) {
 0:J(0,1) static int tmp=(*b);
 1:J(1,2) lock(&m);
 2:J(2,3) if(c>0)
 3:J(3,4)   c++;
          else{ GUARD(4)
 4:J(4,5)   c=0;
            if(!(tmp>0)) goto _exit_l1;
 5:J(5,6)   c++;
            tmp--;
            if(!(tmp>0)) goto _exit_l1;
 6:J(6,7)   c++;
            tmp--;
            assume(!(tmp>0));
            _exit_l1:  GUARD(7);
          } GUARD(7)
 7:J(7,8) unlock(&m);
           goto _exit_P0;
          _exit_P0:  GUARD(8)
 8:       return;
}
```

```
void P1(void *b) {...}
void C0() {
 0:J(0,1) assume(c>0);
 1:J(1,2) c--;
          assert(c>=0);
          goto _exit_C0:
          _exit_C0:  GUARD(2)
 2:       return;
}
void C1() {...}
int main0() {
          static int x=1,y=5;
          pthread_t p0,p1,c0,c1;
 1:J(1,2) create(&p0,P0,&x,1);
 1:J(2,3) create(&p1,P1,&y,2);
 2:J(3,4) create(&c0,C0,0,3);
 3:J(4,5) create(&c1,C1,0,4);
          goto _exit_main;
          _exit_main:  GUARD(4)
 5:       return 0;
}
```

FIGURE 2.9: Lazy-CSeq translation of example program in Figure 2.7.

A core feature of our code-to-code translation that significantly impacts its effectiveness is that it just injects light-weight, non-invasive control code into the input program. Essentially, the control code is composed of few lines of guarded `goto` statements and, within the added function `main`, also very few assignments.

Figure 2.7 shows a producer/consumer system, while Figure 2.9 shows the transformation result from Lazy-CSeq for the producer/consumer example (illustrated in Figure 2.7), obtained using an unwind bound of 2. The figure 2.9 shows (in black) the code of the multi-threaded program after the rewriting of memory statements with the operations from the API of the SMA and (in gray) the extra code fragments injected by Lazy-CSeq.

Laziness allows us to avoid handling all spurious errors that can occur in an eager exploration. Thus, we can inherit from the backend tool all checks for sequential C programs such as array-bounds-check, division-by-zero, pointer-checks, overflow-checks, reachability of error labels and assertion failures, etc.

2.6.3 CSeq framework

Our sequentializations have been implemented on prototype tools that are built on the CSeq framework [Inv15, INF+15]. It is a framework for developing new source-to-source transformation-based verification tools for program analysis of multithreaded C programs using POSIX threads [ISO09]. To date, the framework has been used to implement Lazy-CSeq [ITF+14a, ITF+14b, INF+15, TNI+16a], MU-CSeq [TIF+14, TIF+15a, TIF+15b] and UL-CSeq [NFLP15, NFLP16] tools.

A verification tool is composed of several modules that are executed in a sequence, called *configuration*; and each module can be either a *translator* that implements a source-to-source transformation of C programs (e.g. loop unrolling, function inlining, etc.), or a *wrapper* that is used for general-purpose tasks (e.g. back-end invocation and user report generation).

The architecture of CSeq is shown in Figure 2.10 (cfr. [Inv15]). The *front-end* is the main driver. The user provides to the front-end through the command-line (*i*) the name of one or more input files, (*ii*) the name of the configuration definition file that contains the list of the modules to execute in sequence, and (*iii*) a list of parameters used by the modules. Each module takes as input a string and zero or more parameters and produces

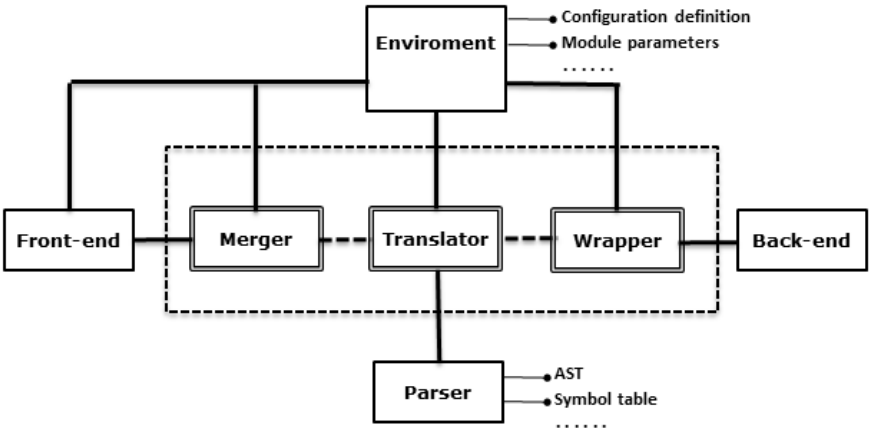

FIGURE 2.10: Architecture of the CSeq framework; the dashed and the double-framed rectangles indicate the configuration and the modules, respectively.

as output a modified string, and zero or more output parameters. Output parameters can be used to transfer information across modules.

```
import core.module

class test(core.module.Translator):
   def visit_UnaryOp(self,n):
      if n.op == "p++":
         return "%s = %s + 1" % (n.expr,n.expr)

      return super(test,self).visit_UnaryOp(n)
```

FIGURE 2.11: Source transformation module: from `x++` to `x=x+1`.

The *Merger* is the first module to be executed by the *front-end* and it takes as input the source code of one or multiple C programs and produces as output a single merged C program. The input of each module is the output of the previous module. The output of the last module is the output shown on the standard output at the end of the execution.

The *Environment* object contains shared information that the front-end and the modules can access during an execution of a configuration.

When a *Translator* is executed by the front-end, the input source C code is automatically parsed, using the *Parser* object, to build the abstract syntax tree (AST), the symbol table, and other data structures; then the AST is recursively visited and unparsed back into a string that corresponds to the output C code. *Translators* are built on top of `pycparser`[2], an open-source C parser that uses `PLY`[3], an implementation of Lex-Yacc [LS90, Joh75]. Source transformation is obtained by modifying the standard AST visit by conveniently overriding pycparser's AST-based pretty-printer. Figure 2.11 shows an example of a source transformation module that replaces each unary post-increment operator on a variable, say x, with a statement that assigns to x the value of x incremented by 1.

The transformation is implemented by overriding the `visit_UnaryOp` method that generates the source code from AST-subtrees representing unary operations.

CSeq framework provides a few built-in modules for standard code transformations, commonly used in program analysis, such as function inlining and loop unrolling, as well as modules for generic program instrumentation and backend invocation.

[2]pycparser, https://github.com/eliben/pycparser
[3]PLY, http://www.dabeaz.com/ply/

Chapter 3

Separating Computation from Communication

In this chapter, we describe an approach to design static analysis and verification tools for concurrent programs that separates intra-thread computation from inter-thread communication by means of a shared memory abstraction (SMA). We formally characterize the concept of thread-asynchronous transition systems that underpins our approach and that allows us to design tools as two independent components, the intra-thread analysis, which can be optimized separately, and the implementation of the SMA itself, which can be exchanged easily (e.g., from the SC to the TSO memory model). We describe the SMA's API and show that several concurrent verification techniques from the literature can easily be recast in our setting and thus be extended to weak memory models. This chapter is largely based on our paper [TNF+17].

The chapter starts by giving a short introduction in Section 3.1. In Section 3.2.1, we describe the *shared memory abstraction* interface, that captures the standard concurrency operations in multi-threaded programs. In Section 3.2.2, we give the semantics of multi-threaded programs by the composition of two transition systems; the first one that captures the control flow of the program and the second one that implements the behaviours of the SMA. We formalize our approach in Section 3.3 and introduce the main concepts of *thread-wise equivalent programs*, that is programs where the *intra-thread ordering* of the statements remains the same, and *thread-asynchronous SMA*, that is that its behaviors are insensitive to how the threads are interleaved. We further show how some algorithms from the literature can be recast in our setting, yielding correctness proofs for free. We conclude this chapter with Section 3.3.1.2, where we explain how to extend these verification algorithms to handle programs under weak memory model semantics by giving the corresponding shared memory abstractions.

3.1 Introduction

Developing correct concurrent programs is a complex and difficult task, due to the large number of possible concurrent executions that must be considered. The advent of modern multi-core hardware architectures that implement *weak memory models* (WMMs) has made this task even harder, because they introduce additional executions that can lead to seemingly counter-intuitive results that confound the developers' reasoning.

Testing remains the most widely used approach to finding bugs. It can be effective if the fraction of buggy executions is high, but it remains highly ineffective for bugs that manifest themselves only rarely and are difficult to reproduce [TDB14]; however, such "Heisenbugs" are unfortunately more prevalent with WMMs. Since other verification approaches that explore executions *explicitly* face the same problems as testing, even with optimizations such as partial order reduction that eliminate redundant executions, we need approaches that can handle multiple concurrent executions *symbolically*.

However, it is difficult to build efficient symbolic verification tools for realistic programming languages like C, and harder yet to extend them to handle concurrency. Tools thus often compromise generality to achieve efficiency, by focusing on a specific memory model, typically sequential consistency (SC), and by folding the concurrency handling deep into their general verification approaches (see [AAA+15, AKT13, ABP11, BCDM15, WT15, ZKW15]). This in turn introduces a strong coupling between the two, which makes it hard to reuse existing tools and to generalize solutions to other memory models. Our goal here is to break this coupling and to separate the computation (i.e., individual threads) and the communication (i.e., shared memory) concerns of concurrent programs, without loosing the efficiency of existing approaches.

More specifically, we develop an approach that allows us to combine different concurrent verification techniques with different memory models in the style of a plug-and-play architecture. For this, we define and describe in Section 3.2 an interface that we call *shared memory abstraction* (SMA). The SMA captures the standard concurrency operations in multi-threaded programs such as shared memory reads, writes, and allocations, thread creation and termination, and synchronization operations such as thread join and mutex locking and unlocking. We then assume that all operations involving concurrency are performed by invoking the corresponding SMA operations (which can easily be achieved by rewriting non-conforming programs). In this way, we achieve the desired separation of concerns—in fact, we can even view a multi-threaded program as the composition of two independent *sub-systems*, one comprising all threads and one capturing the concurrency (including the memory model), which synchronize using the APIs provided by the SMA.

We formalize this view in Section 3.3.1 and introduce the concepts of *thread-wise equivalence* and *thread-asynchronous closure* of transition systems. We show that reachability is preserved if we exchange the transition system of a program for a thread-wise equivalent

one (assuming the SMA is thread-asynchronous) or an SMA for its thread-asynchronous closure. This has two important consequences. First, it allows us to generalize existing concurrent verification approaches to different memory models simply by implementing the corresponding different SMAs. Second, it gives us a degree of freedom in designing concurrent verification algorithms, since it allows us to rearrange the order in which the verifier explores the execution of the statements among different threads. This is implicitly exploited by some algorithms from the literature, such as the sequenzialization by Lal and Reps [LR09] and the fixed-point algorithm [LMP10] where bounded-round computations are explored by executing each individual thread to completion. We further show in Section 3.3.1.1 how these algorithms can be recast in our setting, yielding correctness proofs for free.

However, the way the computation and communication concerns are combined affects the scalability of the resulting verification tool. In Chapters 5 and 6, we therefore instantiate our general approach to achieve an efficient BMC-based bug-finding tool. We describe efficient SMA-implementations for sequential consistency (SC), total store ordering (TSO), and partial store ordering (PSO) that are based on the idea of memory unwinding, and we show through experiments that our SMA implementations used in combination with the eager sequentialization from [TIF+15b] compares well with existing tools.

In Chapter 4, we also describe efficient *thread-synchronous* SMA-implementations[1] for total store ordering (TSO), and partial store ordering (PSO) that work well in combination with existing lazy sequentialization tool, and in particular we combine them with our tool Lazy-CSeq [ITF+14a].

3.2 Multi-Threaded Programms over Shared Memory Abstractions

Here, we consider multi-threaded programs with a C-like syntax including pointer arithmetics and dynamic memory allocation. We further consider POSIX-like threads with dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization, but no thread communication primitives: threads communicate only via the shared memory. We also assume a `fence`-statement that flushes all store buffers of a thread. The exact program syntax is defined in Chapter 2 by the grammar shown in Figure 2.2. The semantics of multi-threaded programs is given in Section 3.2.2.

[1] We say that an SMA is *thread-synchronous* if its behaviors depend on the order in which the statements among different threads are executed.

```
 1:  void init();
 2:  uint address(uint vid, uint tid);
 3:  uint malloc(uint expr, uint tid);
 4:  int read(uint vid, uint tid);
 5:  int ind_read(uint addr, uint tid);
 6:  void write(uint vid, int val, uint tid);
 7:  void ind_write(uint addr, int val, uint tid);
 8:  void lock(uint mut, uint tid);
 9:  void unlock(uint mut, uint tid);
10:  void fence(uint tid);
11:  uint create(f, tid);
12:  void join(tid, tid);
13:  void terminate(tid);
```

FIGURE 3.1: Shared Memory Abstraction APIs.

3.2.1 Shared Memory Abstractions

The semantics of multi-threaded programs ultimately depends on the underlying memory model. In order to combine existing concurrent verification techniques with different memory models we define a "concurrency interface" or *shared memory abstraction* (SMA) that abstracts away the shared memory operations in the syntax of multi-threaded programs. The intended meaning of the SMA's functions is standard; note that most functions carry the calling thread `t` as an extra argument to allow the SMA to update its internal state.

In detail, the SMA API is formed of the following functions:

- `init()` initializes the SMA and the shared variables; this must be the first statement in the program;
- `address(v,t)` returns the memory address of the shared variable `v`;
- `malloc(n,t)` allocates a continuous block of `n` memory locations and returns the base address of the block;
- `read(v,t)` (resp. `ind_read(a,t)`) returns the valuation of the shared variable `v` (resp. memory location with address `a`) as seen by `t`;
- `write(v,val,t)` (resp. `ind_write(a,val,t)`) sets the valuation of the shared variable `v` (resp. memory location with address `a`) to the value `val`;
- `fence(t)` flushes all store buffers of `t` and updates the shared memory;
- `lock(m,t)` and `unlock(m,t)` are the standard thread synchronization primitives that acquire and release a mutex `m` for `t`; if `m` is currently acquired, the `lock` operation is blocking for `t`, i.e., `t` is suspended until `m` is released and then acquired;

- `create(f,t)` spawns a new thread that starts from function `f`, and returns a fresh thread identifier for this thread;
- `terminate(t)` terminates the execution of `t`; each thread must explicitly call it at the end;
- `join(t',t)` pauses the execution of `t` until `t'` has terminated its execution.

3.2.2 Multi-threaded programs as composition of transition systems

The formal semantics of multi-threaded programs is often given by a transition system that captures the program computations by interleaving the computations of each thread. For our class of programs we exploit the separation between the control flow and the shared memory aspects introduced with the notion of SMA. Thus, the semantics of a multi-threaded program is given as the composition $\mathcal{C}|\mathcal{M}$ of the *control-flow transition system* $\mathcal{C}$ that captures the control flow of the program and the *shared memory abstraction transition system* $\mathcal{M}$ that implements the behaviors of the SMA. This allows us to keep the semantics of the sequential part and re-interpret it in different ways with different WMMs; it also aligns nicely with different SMA implementations.

These two transition systems are synchronized over the SMA API that defines the alphabet that labels the transitions of $\mathcal{C}$ and $\mathcal{M}$. More precisely, the alphabet consists of the calls to the SMA API functions that do not return values, and the calls augmented with a parameter denoting the returned value for the others. For example, read(3,v,t) is the letter corresponding to a call `read(v,t)` that returns value 3. We denote this alphabet with Σ_{SMA}.

Control-flow transition system. The states of the control flow transition system $\mathcal{C}$ are the set of tuples of thread configurations. A thread configuration consists of a program counter, an evaluation of the global variables and a call stack, as usual. $\mathcal{C}$ has a unique initial state that corresponds to the empty configuration (i.e., no threads are active in the beginning) and all states are final.

The transitions correspond to the execution of any of the statements. Transitions corresponding to invocations of API functions of SMA are labeled with the corresponding letter from Σ_{SMA}. In particular, transitions from the initial state are labeled with init() and enter a state with the starting configuration of the main thread. No other transitions are labeled with init(). Transitions corresponding to SMA functions that return a value are handled as assignments of the corresponding variables with the returned values. Additionally, on a thread creation the tuple of thread configurations is augmented with the starting configuration of the newly created thread. Similarly, the effect of a transition on terminate(t) is to delete the configuration of the terminated thread. The remaining transitions over Σ_{SMA} letters just update the program counter. Transitions corresponding

to all other (i.e., sequential) statements are labeled with the empty word ε and update the configuration of the issuing thread as usual.

Shared memory abstraction transition system. In general, an SMA transition system $\mathcal{M}$ has an initial state and a state for each possible configuration of the corresponding memory model. The transitions update memory configurations to capture the memory model's intended meaning. Note that from the initial state there are only outgoing transitions, which are all labeled with init(), and no other transitions have this label.

For SC, the system $\mathcal{M}_{sc}$ can enter from the initial state any state that has only one thread (which must be active), has any number of shared locations (which must all have the value of zero), and has any number of mutexes (which must all be unlocked). All other transitions update the state of $\mathcal{M}_{sc}$ according to the meaning given in Section 2.3.1. Since there are no store buffers in SC, there are no fence-transitions. Further, in a transition on terminate(t), $\mathcal{M}_{sc}$ enters a state where the status of t is terminated. From any such state only states where the t status remains terminated can be reached, and no other transitions corresponding to invocations of API functions from t are allowed. The final states are all the states where all the threads are terminated.

For the WMMs we denote the corresponding SMA transition system with $\mathcal{M}_{tso}$ and $\mathcal{M}_{pso}$, respectively. The states of both systems also account for the content of the thread store buffers, the transitions on reads and writes reflect the corresponding semantics as described in Section 2.3.1, and there are fence(t)-transitions on calls to `fence` by t and ε-transitions for store buffer updates.

3.3 Verification with thread-asynchronous SMAs

In this section, we discuss our design approach for the verification of multi-threaded programs. Its basis is the separation between the intra-thread control-flows and the SMA already discussed in Section 3.2. In this view, a verification tool is composed of an SMA implementation and a search algorithm that explores the program executions. This by itself allows for a convenient way to extend verification methods to other memory models by simply replacing the SMA implementation. However, this might not result in scalable verification tools, for the following reasons.

First, to preserve the correct semantics of the memory operations, these must be invoked in the same order as they appear along the run, which may be a bottleneck when we explore the state space of the program, both in case of the analysis based on summaries (e.g., BDD-based model checking) or bounded model checking. In the former, we must keep a cross-product of the states of all threads in the configurations; this is a well-known problem that leads to state-space explosion. In the latter, since context-switches can happen at any point, we must encode into the SAT/SMT formula the code of all threads

for each of the context-switch points in the underlying bounded multi-threaded program, which leads to large formulas.

Some approaches from the literature instead explore the program executions by rearranging the order in which the memory operations of the different threads are executed, e.g., by simulating each thread to completion [LMP10, LR09]. In Chapter 5 and 6, we propose a verification approach where each thread is executed in isolation with respect to a memory unwinding (i.e., a sequence of writes that is guessed at the beginning). More generally, the approach of verifying each thread in isolation is also the essence of the compositional approaches based on assume-guarantee reasoning [MC81, Jon83].

We generalize the ad-hoc approaches above, and present a general framework in which to design concurrent program verification approaches. This requires that the used SMA implementation is *thread-asynchronous*, that is that its behaviors are insensitive to how the threads are interleaved. This allows us to freely transform the threads as long as we stay within the class of *thread-wise equivalent* programs, that is programs where the *intra-thread ordering* of the statements remains the same. This, along with the correctness of the derived design approach, will be formalized below.

We conclude this chapter by discussing how previous successful approaches from the literature fit into our setting. In Chapters 5 and 6, we will then give efficient implementations of thread-asynchronous SMAs and show that these can be combined with the sequentialization transformation from [TIF+15b] to achieve a competitive verification tool.

3.3.1 Thread-asynchronous SMAs

For a thread t, we denote with Σ^t_{SMA} the maximal subset of Σ_{SMA} containing only letters that are issued by t. Clearly, for threads t and t' with $t \neq t'$, Σ^t_{SMA} and $\Sigma^{t'}_{SMA}$ are disjoint. For a thread t and a word α over Σ_{SMA}, let $\alpha_{|t}$ be the projection of α onto Σ^t_{SMA}, i.e., the word obtained from α by deleting all the letters that do not belong to Σ^t_{SMA}. If $t_1, \ldots, t_h$ are all the threads that issue at least a letter in α, we define $\pi(\alpha)$ as the map $\pi(\alpha)(t_i) = \alpha_{|t_i}$ for $i \in [1, h]$.

A language L of words over Σ_{SMA} is *thread-asynchronous* if for any $\alpha \in L$ and for each α' starting with $\mathsf{init}()$ s.t. $\pi(\alpha) = \pi(\alpha')$, also $\alpha' \in L$. The *thread-asynchronous closure* of a language L, denoted by $L^{\#}$, is the smallest thread-asynchronous language such that $L \subseteq L^{\#}$.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two transition systems over the alphabet Σ_{SMA}. We say that $\mathcal{A}_1$ and $\mathcal{A}_2$ are *thread-wise equivalent* if for each word α accepted by one of them there is a word α' accepted by the other one such that $\pi(\alpha) = \pi(\alpha')$.

A standard analysis for multi-threaded programs is to search for the reachability of an error program counter of a given thread (*local error state*), often denoted by an error label or a `false`-assertion. In the following, we give two theorems stating sufficient conditions under which the reachability of local error states is preserved.

The first theorem states that if the SMA is thread-asynchronous we can transform a program P_1 into a thread-wise equivalent program P_2 such that a local error state is reachable in the resulting program P_2 if and only if it is reachable in P_1.

Theorem 3.1. *Let $\mathcal{C}_i$ be a control-flow transition system for $i = 1, 2$ and $\mathcal{M}$ be an SMA transition system. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are thread-wise equivalent, and $\mathcal{M}$ is thread-asynchronous, then a local error state is reachable in $P_1 = \mathcal{C}_1|\mathcal{M}$ iff it is reachable in $P_2 = \mathcal{C}_2|\mathcal{M}$.*

Proof. Since the SMA transition system $\mathcal{M}$ is thread-asynchronous, we have that the interaction of each thread with the SMA is independent of how threads are interleaved; in particular, by fixing a run ρ, the values returned by the read operations performed by a thread are ensured to be the same in all the possible interleaving of the projections of ρ onto each thread. Since we assume that $\mathcal{C}_1$ and $\mathcal{C}_2$ are thread-wise equivalent, the sequences of SMA operations issued along the runs of P_1 and P_2, may differ only by interleavings of the threads, we get that the reachability is preserved. □

Theorem 3.1 states a crucial property for our approach: we can implement a thread-asynchronous SMA, and combine it with any transformation of the program that rearranges the interleaving among threads and still get a correct verification approach. In the next subsection, we discuss how to implement thread-asynchronous SMAs that are suitable to recast known verification approaches for SC and extend them to WMMs.

The second theorem shows that we can replace an SMA $\mathcal{M}_1$ with another SMA $\mathcal{M}_2$ that captures its thread-asynchronous closure, and still preserve reachability of local error states.

Theorem 3.2. *Let $\mathcal{C}$ be a control-flow transition system and $\mathcal{M}_i$ be an SMA transition system for $i = 1, 2$. If $L(\mathcal{M}_2) = (L(\mathcal{M}_1))^{\#}$, then a local error state is reachable in $\mathcal{C}|\mathcal{M}_1$ iff it is reachable in $\mathcal{C}|\mathcal{M}_2$.*

Proof. Since $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$, we have that each sequence α accepted by $\mathcal{M}_1$ is also accepted by $\mathcal{M}_2$. The interesting case of the proof is when a sequence α is accepted by $\mathcal{M}_2$ but not by $\mathcal{M}_1$. In this case, since the returned values are visible in Σ_{SMA} letters and there must be a sequence α' that is accepted by $\mathcal{M}_1$ such that $\pi(\alpha) = \pi(\alpha')$, we get that the sequence of local states that are visited by any thread of any program P are the same for both sequences α and α'. Therefore, Theorem 3.2 holds. □

By combining both theorems, we can easily show the correctness of WMM extensions of correct verification methods that transform programs by keeping the ordering of the sequence of the operations within each thread. In fact, we just need to provide an SMA that captures the thread-asynchronous closure of the memory model.

3.3.1.1 Thread-asynchronous SMAs for thread interfaces

We briefly recall the notions of thread interface [LMP10] and discuss how to recast some approaches from the literature in our setting by means of the SMAs derived from these notions.

Thread interface. A *thread interface* for a thread t summarizes computations of t across a bounded number of context-switches. Formally, it is a sequence of pairs $(r_1, s_1), \ldots, (r_k, s_k)$ where r_i, s_i for $i \in [1, k]$ are valuations of the shared locations. The intended meaning is that there is a computation of t such that t starts with r_1 as valuation of the shared locations and reaches s_1, is suspended and then reactivated with shared valuation r_2, and reaches s_2, and so on.

In a bounded context switch analysis we can assume that computations of programs are arranged in k rounds where threads are always scheduled according to the same fixed round-robin schedule $t_1, \ldots, t_n$. Thus, exploring the computations of a multi-threaded program up to k rounds corresponds to computing thread interfaces and composing them [LMP10]. We start with thread t_1 and guess the in-valuations at rounds $2, \ldots, k$ (i.e., the valuations $r_2, \ldots, r_k$; note that r_1 is the initial valuation of the program and thus known); we then compute the out-valuations (i.e., $s_1, \ldots, s_k$) for thread t_1 and take them as the in-valuations of the next thread t_2, and so on. In the end, in order to establish that the computed thread interfaces form a computation of the program we just need to check that the out-valuation of thread t_n at round $i \in [1, k-1]$ equals the (guessed) in-valuation of thread t_1 at round $i+1$.

This is the essence of the well-known sequentialization algorithm by Lal and Reps [LR09] and the fixed-point algorithm given in [LMP10]. We can recast these two algorithms in our setting by means of an SMA that extends the standard SMA for SC by thread interfaces. The resulting transition system $\mathcal{M}_{sc}^{ti}$ is as follows. On the $\mathsf{init}()$-transition, $\mathcal{M}_{sc}^{ti}$ guesses a round schedule $t_1, \ldots, t_n$, a bound k, and for each thread t_i an interface $I^i = (r_1^i, s_1^i) \ldots (r_k^i, s_k^i)$ such that $r_j^i = s_j^i$ for $j \in [1, k]$. $\mathcal{M}_{sc}^{ti}$ keeps for each thread the current round in the corresponding thread interface. If the current round of a thread is less than the round bound k, it can be increased by one by an ε-transition (i.e., it is nonderministically either increased or left unmodified). Further, for any input sequence α, $\mathcal{M}_{sc}^{ti}$ ensures that:

- on $\mathsf{write(v,val,t)}$ (resp. $\mathsf{ind\_write(a,val,t)}$), the out-valuation of the current round of thread t is updated according to the write;

- on read(val,v,t) (resp. ind_read(val,a,t)), the out-valuation of the current round of thread t must evaluate v (resp. a) as *val*.

In order to accept α, create(t,f,t') must occur in α for each thread t with a guessed interface, and the computed interfaces form a computation in the sense described above.

The transition system $\mathcal{M}_{sc}^{ti}$ is thread-wise equivalent to $\mathcal{M}_{sc}$, and, moreover, it can execute all the computations of $\mathcal{M}_{sc}$ by advancing each involved thread in any order. The proof of the following lemma is a consequence of the results from [LMP10].

Lemma 3.3. $L(\mathcal{M}_{sc}^{ti}) = (L(\mathcal{M}_{sc}))^{\#}$.

Proof. We start showing that $L(\mathcal{M}_{sc}^{ti}) \supseteq (L(\mathcal{M}_{sc}))^{\#}$. To do this we prove that for any $\alpha \in L(\mathcal{M}_{sc})$ and $\alpha' \in \{\alpha\}^{\#}$, $\alpha' \in L(\mathcal{M}_{sc}^{ti})$. From [LMP10], observe that for every $\mathcal{M}_{sc}$ run ρ of finite length and round schedule of the threads involved in ρ, there exists a round number k such that ρ is a k-round execution. Now, let ρ be an accepting run of $\mathcal{M}_{sc}$ over α, $t_1, \ldots, t_n$ be a round schedule of all threads involved in ρ and I be the set of thread interfaces $I^i = (r_1^i, s_1^i) \ldots (r_k^i, s_k^i)$ corresponding to ρ, for each thread t_i and $i \in [1, n]$. We recall that $\mathcal{M}_{sc}^{ti}$ on the `init` transition can guess any set of thread interfaces, any round schedule and any round bound. Thus, $\mathcal{M}_{sc}^{ti}$ on the initial transition can enter a state with set of thread interfaces I, round schedule $t_1, \ldots, t_n$ and round bound k for ρ. Since I fully captures the thread interfaces of ρ, it is straightforward from the definition of $\mathcal{M}_{sc}^{ti}$ that it can simulate the execution of each thread in ρ independently by interleaving the transitions of these threads in any given order. Thus, $\mathcal{M}_{sc}^{ti}$ accepts all words in $\{\alpha\}^{\#}$ and therefore, $L(\mathcal{M}_{sc}^{ti}) \supseteq (L(\mathcal{M}_{sc}))^{\#}$.

We now prove that $L(\mathcal{M}_{sc}^{ti}) \subseteq (L(\mathcal{M}_{sc}))^{\#}$. For $\beta \in L(\mathcal{M}_{sc}^{ti})$, denote with γ an accepting run of $\mathcal{M}_{sc}^{ti}$ over β. From [LR09], it is always possible to rearrange the order in which the operations of different threads in γ, by preserving the order of the operations of each thread, to form an accepting run ρ of $\mathcal{M}_{sc}$. Let α be the word accepted by $\mathcal{M}_{sc}$ along ρ. Since $\alpha \in L(\mathcal{M}_{sc})$ and $\alpha \in \{\beta\}^{\#}$, also $\beta \in (L(\mathcal{M}_{sc}))^{\#}$. Therefore, $L(\mathcal{M}_{sc}^{ti}) \subseteq (L(\mathcal{M}_{sc}))^{\#}$. □

We can then recast the verification technique from [LR09] in our setting by taking the above SMA along with the transformation of the control-flow from [LR09]. Lemma 3.3, and Theorems 3.1 and 3.2 show the correctness of the resulting verification method. Similarly, we can combine $\mathcal{M}_{sc}^{ti}$ with a control-flow part that at each transition nondeterministically selects the next thread to execute. The resulting system captures the verification technique from [LMP10], and correctness is again ensured by Lemma 3.3, and Theorems 3.1 and 3.2. We remark that actual implementations of both these techniques require parameterization over the number of threads and rounds, as in the original implementations.

3.3.1.2 Extension to weak memory models.

The discussed verification algorithms can be extended to handle programs under weak memory model semantics by giving the corresponding shared memory abstractions. This can be done for TSO and PSO by explicitly adding the store buffers to $\mathcal{M}_{sc}^{ti}$, or for TSO by augmenting $\mathcal{M}_{sc}^{ti}$ with guesses on the round when a write will be visible to all threads, as done in [ABP11].

3.4 Related Work

The need for a general and reusable framework to accommodate different weak memory models in the analysis of programs has been identified in earlier papers. In [AM14], the verification algorithm works on a generic relaxed memory model that can be refined into actual memory models by adding constraints. The BMC approach from [AKT13] allows to handle different memory models by adding a conjunct to the formula. Our work differs from these both in the scope and the techniques. In particular, we give a general approach that allows to combine different verification algorithms with different implementations of memory models, not just a specific algorithm. The development of the two parts can be done independently as long as Theorems 3.1 and 3.2 hold. All the interaction between the two parts is through the API of the shared memory abstraction.

Another important aspect of our approach is to identify a class of implementations of memory models that allows for a full rearrangement of the thread interleavings in the analysis. As already observed, this is a feature that has been already exploited in verifying concurrent programs [LR09] also with weak memory model semantics [ABP11].

3.5 Conclusions

In this chapter we have described a new verification approach for concurrent programs over different memory models. Our main design goal was to break the coupling between computation (i.e., individual threads) and the communication (i.e., shared memory) concerns of multi-threaded programs, without loosing the efficiency of existing approaches. To achieve this goal, we have introduced shared memory abstractions, which capture the standard concurrency operations in multi-threaded programs. We have then shown that reachability is preserved if we exchange a program by a thread-wise equivalent one (assuming the SMA is thread-asynchronous) or an SMA for its thread-asynchronous closure. This allows us to generalize existing concurrent verification approaches to different memory models simply by implementing the corresponding different SMAs.

In the next chapter, we extend our Lazy sequentialization [ITF+14a] given for SC, to handle TSO and PSO memory models by introducing efficient *thread-synchronous* SMA implementations that capture their semantics.

In Chapter 5, we give efficient *thread-asynchronous* SMA implementations, based on the concept of *memory unwinding*, that capture the semantics of the SC memory model.

In Chapter 6, we extend the concept of *memory unwinding* to *individual shared memory location unwinding* and give efficient *thread-asynchronous* SMA implementations that, additionally, capture the semantic of the TSO and PSO memory models.

Chapter 4

Lazy Sequentialization for TSO and PSO via SMA

Lazy sequentialization is one of the most effective approaches for the bounded verification of concurrent programs. Existing tools assume sequential consistency (SC), thus the feasibility of lazy sequentializations for weak memory models (WMMs) remains untested.

In this chapter, we give efficient *thread-synchronous* SMA implementations for TSO and PSO that are based on temporal circular doubly-linked lists, a new data structure that allows an efficient simulation of the store buffers. We combine them with our Lazy sequentialization for SC memory model from [ITF+14a] on top of bounded model checking; in such a way we obtain a novel lazy sequentialization approach for the total store order (TSO) and partial store order (PSO) memory models. We replace all shared memory accesses with operations on the *thread-synchronous*[1] (SMA) that encapsulates the semantics of the underlying WMM and implements it under the simpler SC model. We show experimentally, that this approach works well both on (relatively) simple concurrency benchmarks and a real world instance. This chapter is largely based on our paper [TNI+16a].

The chapter starts by giving a short introduction in Section 4.1. In Section 4.2, we introduce two SMA implementations that capture the TSO memory model, the first one is a reference implementation called TSO-SMA that represents the standard TSO semantics [SSO+10b] directly; the second one, called eTSO-SMA, introduces a new representation that is based on temporal circular doubly-linked lists. Theorem 4.1 shows that eTSO-SMA captures the semantics of the TSO memory model. In Section 4.4, we extend our TSO implementation to PSO. We conclude this chapter with Section 4.5, where we give an implementation of our approach by combining our SMA implementations with Lazy-CSeq in the prototype tool *LazySMA*; we show that it works well over a set of

[1] To preserve the correct semantics of the memory operations, these must be invoked in the same order as they appear along the run

benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite, and also on a real world instance.

4.1 Introduction

SAT/SMT-based *bounded model checking* (BMC) has been used successfully to discover subtle errors in sequential software, even at large scale [KT14, SKB+15]. Sequential BMC tools can be extended symbolically to the concurrent case by conjoining the formula representing the effect of each individual thread in isolation with a second formula representing the possible interferences caused by concurrent accesses to the shared memory [SW11, AKT13]. Since this second formula effectively includes an axiomatization of the underlying memory model, this approach can in principle work for both sequential consistency (SC) and different WMMs. However, embodying a memory model at the formula level requires extensive (and non-reusable) modifications of the underlying sequential BMC tool, and can affect scalability since the resulting expressions are large and complex.

An alternative approach is *sequentialization*, which translates concurrent programs into sequential programs with data non-determinism that (under certain assumptions) behave equivalently, so that the different interleavings do not need to be treated explicitly during the analysis. This allows the reuse of existing sequential BMC tools. Eager sequentializations [LR09] guess the different values of the shared memory before the verification, which means that they can explore infeasible computations that need to be pruned away afterwards. Lazy sequentializations [LMP09a] only explore feasible computations and can be used as basis of very effective verification tools, such as Lazy-CSeq [ITF+14a]. This is witnessed by Lazy-CSeq's top rankings in recent software verification competitions but also borne out in practice: for example, using Lazy-CSeq we discovered in 30 minutes a bug in the safestack benchmark [Vyu10], while all other approaches, including testing, failed [TDB14]. However, to the best of our knowledge, lazy sequentializations have been developed only for SC, and not for any of the WMMs that are prevalent in modern computer architectures.

In this chapter, we therefore develop the first lazy sequentialization for multi-threaded programs for the total store order (TSO) [SSO+10b] and partial store order (PSO) memory models. More specifically, we replace all accesses to shared memory items (i.e., reads from and writes to shared memory locations, and synchronization primitives like lock and unlock) by explicit calls to *API operations* over a *shared memory abstraction* (SMA, see Section 3.2.1). For example, if `x` and `y` are two shared scalar variables then the statement $\texttt{x} = \texttt{y} + \texttt{x} + 3$ is translated into $\texttt{write}(\texttt{x}, \texttt{read}(\texttt{y}) + \texttt{read}(\texttt{x}) + 3)$. The SMA can be seen as an abstract data type that encapsulates the semantics of the underlying WMM and implements it under the simpler SC model. This isolates the WMM from the

remaining concurrency aspects, and allows us to reuse existing (lazy) sequentialization techniques and tools for SC. Our approach bears some similarity to the axiomatic representation of memory models [SW11, AKT13] but the fundamental difference is that we work at the code level—in effect, we apply the very idea of sequentialization to WMMs themselves.

In this chapter, we introduce a new data structure, temporal circular doubly-linked list (T-CDLL), for an efficient encoding of the store-buffers. We then describe TSO and PSO implementations of the SMA, based on T-CDLL, that lead, in combination with a lazy sequentialization targeting BMC tools, to efficient SAT/SMT encodings. Sections 4.2 and 4.3 describe an efficient implementation of SMA for TSO, while Section 4.4 extends this to PSO. In Section 4.5, we give an implementation of our approach in a verification tool, called `LazySMA`, and give its experimental evaluation. In particular, we compare our prototype implementation to CBMC (which implements TSO and PSO at the formula level) [AKT13] and Niddhug [AAA+15] (which combines stateless model checking and dynamic partial order reduction). The experiments show that our approach delivers a comparable performance on simple benchmarks, but outperforms both CBMC and Nidhugg on the more complex safestack problem. It also shows that the number of timestamps (i.e., writes to the store buffers) required to expose TSO and PSO bugs is generally small.

SMA implementations. The semantics of concurrent programs, and in particular the concurrency aspects, can vary with the underlying memory model. For a program P, we can capture such a semantics by plugging a corresponding SMA implementation into P and thus interpreting the resulting program according to the standard interleaving semantics that assumes atomicity and sequential consistency of the memory operations. An SMA implementation consists of variables and data structures to capture the shared memory and functions that manipulate them, as listed in the API. Thus, we can model it as a transition system in the usual way.

4.2 Designing a TSO Shared Memory Abstraction

Here, we introduce two SMA implementations that capture the TSO memory model. We start with a reference implementation called TSO-SMA that represents the standard TSO semantics [SSO+10b] directly but leads to complex formulas when used in a BMC-based sequentialization tool chain. We therefore introduce a new representation where the *per thread* store buffers are replaced by *per variable* write lists. We show how this, together with an indexing scheme, allows us to perform the shared memory updates implicitly, and in fact even to entirely remove any explicit representation of the shared memory. This reduces the size of the formulas for two reasons. First, BMC tools perform *function inlining* (i.e., replace each function call with the actual function code), so removing the

updates, which can happen at any time and thus need to be inlined at every visible operation[2], reduces the size of the program and thus the formula. Second, because we remove the memory we do not need (propositional) variables to represent each memory write; we can instead reuse those used to represent the variable write lists. We describe an efficient shared memory abstraction eTSO-SMA that is based on this representation and is equivalent to TSO-SMA. In this section, we give both SMA implementations at an abstract level; in the next section, we give the details of eTSO-SMA including concrete data structures and code for the API operations, and argue the equivalence of eTSO-SMA and TSO-SMA.

4.2.1 TSO-SMA

The total store ordering (TSO) memory model is a *relaxed-consistency* shared memory model where the ordering of write and read operations of the same variable but by different threads can be exchanged. More details of the TSO memory model have been given in Section 2.3.1.

The reference implementation TSO-SMA directly simulates the behavior of the TSO memory model according to the architecture shown in Figure 2.1. We use an array-based queue of bounded size for each thread store buffer, and a copy of the shared variables of the initial program to store one configuration of the shared memory.

Here, the simulation of each write operation results in a small formula: we just need to encode a single element write into the queue. However, read operations lead to larger and more complex formulas. The main source of complexity is in the number of steps required to determine the most recently performed write operation still cached in the queue. All these steps need to be encoded in the formula. Similarly, a flush operation involves every element of a store buffer which again need to be encoded in the formula. The formula for simulating shared memory updates is even bigger. In fact, even though each single update requires only a constant number of steps (to dequeue the write and then modify the content of a memory location), memory updates can occur nondeterministically at each step and in the limit the writes from all store buffers can be passed into the shared memory. Therefore, memory updates require a formula of size proportional to the sum of the maximum number of elements that can be stored in each (bounded) queue. Since these updates need to be simulated at each shared memory access, this leads to a considerable blow-up of the formula size for the whole sequentialized program, rendering this direct implementation hopelessly inefficient.

[2] A statement is *visible* if its execution involves either a read or a write operation of a shared variable, and *invisible* otherwise.

4.2.2 Timestamping writes

The handling of memory updates can be improved by performing the dequeueing operations implicitly. For this, we keep track of the (discrete) time in the executions with a global variable `clock` and add a *timestamp* to each write that is enqueued in a store buffer. These timestamps represent the future time at which a cached write will be used to update the shared memory. Since timestamps refer to future events and writes from different buffers can occur in any order (because the memory updates are nondeterministic), timestamps must be guessed. To ensure consistency with the TSO semantics, we must enforce that the timestamps of successive writes in the same store buffer follow a non-decreasing order. Further, a timestamp must be assigned a value that is at least the current `clock` value.

By adopting this time-stamping schema we can keep the content of a store buffer up-to-date without actually dequeuing the writes when a memory update occurs. In fact, we can just increase the value of `clock`, and all writes that have an *expired* timestamp (i.e., a timestamp that is less or equal to the value of `clock`) can be treated as removed from their respective store buffers.

4.2.3 eTSO-SMA

We now describe an efficient implementation where each SMA operation can be encoded with a formula of constant size and where there is no need to explicitly encode memory updates. For ease of presentation, let us assume that the original program uses only shared scalar variables and that memory locations are never accessed using their addresses. The two main ingredients of this implementation are: (1) the combined use of timestamps and the variable `clock` as above and (2) a re-arrangement of the writes of the store buffers into lists per shared variables. We refer to each such list as a *variable write list* (vw-list, for short). For each shared variable x, we denote with Q_x its associated vw-list. vw-lists contain writes as pairs (val, ts) where val is the written value and ts is the associated timestamp. A write is *expired* if its timestamp is less than or equal to the current value of `clock`.

In addition to the writes of x that are currently cached in store buffers, Q_x also contains the last write of x that has been used in a shared memory update. This gives the current value of x in the shared memory, and is the only expired write in Q_x. Thus, as sketched above, we do not need additional variables to track the shared memory. Further, we keep Q_x ordered by non-decreasing timestamps and in case of writes with the same timestamp, in the order of insertion in the list. Hence, the current valuation in the shared memory of each variable is easy to retrieve from the front of the list.

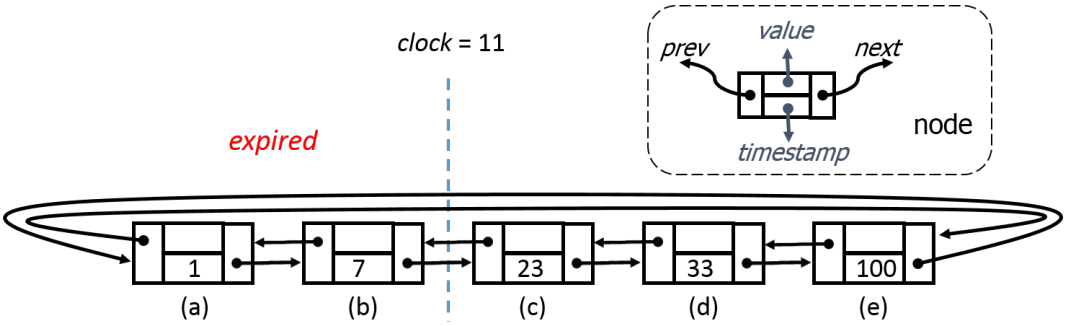

FIGURE 4.1: T-CDLL example.

4.2.4 Temporal Circular Doubly Linked List

To implement the vw-lists efficiently, we introduce *temporal circular doubly linked lists* (T-CDLL, for short), which are circular doubly linked lists where:

1. nodes are of the form shown in Figure 4.1; fields *prev* and *next* contain respectively the link to the predecessor and the successor in the list as usual, and the fields *value* and *timestamp* contain the write;
2. there is a unique *sentinel node*; it does not correspond to an actual write and its timestamp is maximal;
3. the sequence of nodes from the successor of the sentinel node through the sentinel node, via the *next* link, is ordered by non-decreasing timestamps;
4. the *head* of the list is defined as the only node that (i) contains an expired write and (ii) whose successor contains a non-expired write; it is uniquely determined by `clock`.

An example of T-CDLL is given in Figure 4.1. There the sentinel node is (e), the head node is (b) for `clock` value 11 and would change to (c) as soon as `clock` reaches 23.

Note that the portion of T-CDLL from the head through the sentinel node ensures the properties stated for the vw-lists, and thus we can easily use this portion to store each such list. Moreover, a T-CDLL also manages the list of available nodes in the remaining part: when the value of `clock` is increased so that the current head node expires, this becomes available and its successor becomes the new head. However, the node remains linked in the list; hence, all nodes between the sentinel and the head constitute a "free list" for further writes. Caching a write only requires checking that the successor of the sentinel is expired (but not the head of the list), and if so, overwriting the values and re-linking the node.

For example, in the T-CDLL of Figure 4.1, only node (a) is available. By updating `clock` to 30, the head becomes (c) and (b) becomes available but remains linked to the list of

available nodes that starts from (a). If we then try to cache a write with timestamp 40 in the vw-list, we can take (a), unlink it from the T-CDLL and then link it back between (d) and (e) with value and timestamp updated according to the write.

The T-CDLL for a vw-list is initialized with a fixed number of nodes that stay unchanged along the computation (i.e., the size of lists we can encode is bounded). The initial value for the timestamps is 0 except for the sentinel node, whose timestamp is set to the maximum allowed timestamp. In the next section we show how to implement T-CDLLs efficiently by using four parallel arrays, one for each node field.

4.3 Implementation of eTSO-SMA

The code of eTSO-SMA comprises a *module* whose *abstract view* is essentially the API given in Section 3.2.1 and whose *implementation view* (i.e., the complete data type declaration along with the actual code implementing the API operations) will be discussed in this section.

4.3.1 Memory bounds.

We assume that during the program execution all threads can access the memory, which consists of a finite sequence of locations of the same size. Each location has its own memory address which corresponds to its position in the memory. Shared variables are allocated to distinct memory locations. However, eTSO-SMA does not capture the entire memory explicitly but only tracks a bounded number of memory locations.

We use several parameters to bound our analysis and in particular the memory representation. `T` denotes the maximum thread identifier used in the program, `N` the number of nodes in each T-CDLL, `V` the maximum number of memory locations tracked along any execution (including the locations of the shared variables), and `K` the maximum timestamp.

We assume that each shared variable has an integer identifier in the range $[0, \texttt{V} - 1]$. Further, we can have an additional number of memory locations to track that can be accessed only through their memory addresses.

4.3.2 Auxiliary data structures

We use the following global auxiliary variables of type integer:

- `clock` keeps track of the number of writes performed by all threads; it is initialized to 0 and bounded by `K`;

```
1: static uint clock;
2: static uint address[V];
3: static int value[V][N];
4: static uint timestamp[V][N+1];
5: static uint prev[V][N+1];
6: static uint next[V][N+1];
7: static uint last_value[V][T];
8: static uint last_timestamp[V][T];
9: static uint max_last_timestamp[T];
```

FIGURE 4.2: eTSO-SMA variable declarations.

- `address[v]` contains the physical memory address of `v`, for any `v`$\in [0, \mathtt{V}-1]$; the `init` function initializes it with distinct nondeterministic values; the values do not change during the program simulation;
- `value[v][node]` and `tstamp[v][node]` store the value and timestamp of each write associated with each location;
- `max_tstamp[t]` stores for each thread the largest timestamp of any executed write;
- `prev[v][node]` and `next[v][node]` store the link to the previous and next node in the T-CDLL for each location;
- `last_value[v][t]` and `last_tstamp[v][t]` store the last value written and the timestamp of the last write performed by each thread for each location.

The nodes of the T-CDLL corresponding to variable `v` are kept in `prev[v][i]`, `value[v][i]`, `tstamp[v][i]` and `next[v][i]` for `i`$\in [0, \mathtt{N}-1]$.

4.3.3 Malloc and init

During its execution a thread can require a block of n consecutive locations by invoking `malloc`(n), which returns the address of the first location of a newly allocated heap block. Memory addresses can be used to access this shared memory. Let p be a shared pointer variable and `x` be a local variable. A location with address i is *pointed to* by p if the value of p is i. Then, `*p = x` copies the value of `x` into the location pointed to by p, and statement `x = *p` copies the value of the location pointed to by p into `x`. Note that we do not represent the heap memory explicitly as we only track some of its locations. Concerning to `malloc` we maintain a bounded sequence, say of fixed size m, where each element represents a memory block. In particular, for each block we store its base address, its size, and whether it has been allocated. This sequence is implemented using arrays. The `init` function initializes each of these blocks with a nondeterministic base address and a nondeterministic size, making sure that block do not overlap. These values do not change during program executions.

4.3.4 Clock update

`clock_update` is a service routine that updates the variable `clock` with a nondeterministic value picked in the range from its current value to its allowed maximum value `K` (see Figure 4.3, lines 11-15). As a consequence of such an update, some of the writes in the T-CDLLs may expire, thus modifying some of the head nodes and therefore, the valuation of variables in the shared memory and the configuration of the T-CDLLs.

```
11: void clock_update( ){
12:   int tmp;
13:   assume( tmp <= K && tmp >= clock );
14:   clock = tmp;
15: }
```

FIGURE 4.3: eTSO-SMA `clock_update` function.

We stress that this is a very convenient way to implement the shared memory updates since we achieve this without altering the underlying data structures. Moreover, it is correct w.r.t. the TSO semantics since the writes flow into the shared memory by increasing values of the timestamps, and by the ordering we ensure on the timestamps, this enforces a correct simulation of the TSO semantics on the memory updates.

4.3.5 Write operation

We recall that function `write` takes as input a variable identifier `v`, a value `val`, and a thread identifier `t`. `write` first updates the clock value by invoking `clock_update` and then simulates the write operation by changing the state of the memory representation by adding a new write in the T-CDLL associated with `v`. The code for this function is given in Figure 4.4 at lines 17-45.

Take an available node from the T-CDLL of the variable identifier (lines 20-23). Variable `node` is set to a position of the array encoding the T-CDLL for `v` that corresponds to the successor of its sentinel node. From the property of part 3 of the definition of T-CDLLs (that we maintain as an invariant in our implementation), this is the node with the smallest timestamp in the list. We are going to use this node to cache the write. The `assume` statement at line 21 ensures that the successor of `node` is also expired. Otherwise, `node` would be the head node and thus there are no available nodes, therefore the computation must be blocked. We then remove the node from the list by appropriately setting the fields `next` and `prev` of the successor and the predecessor of `node`, respectively (lines 22-23).

Select position for insertion (lines 25-27). We nondeterministically guess a node `succ` that is the candidate to be the successor of `node` in the list (after insertion). We make sure that `succ` encodes a non-expired write by checking that its timestamp is greater

```
17: void write(uint vid,int val,uint tid){
18:   clock_update();
19:   // remove expired node from list
20:   uint node = next[vid][N];
21:   assume(tstamp[vid][next[vid][node]]
                        <= clock);
22:   next[vid][N] = next[vid][node];
23:   prev[vid][next[vid][N]] = N;
24:   // select position in
      //the list for insertion
25:   uint succ = nondet();
26:   assume( succ<=N &&
              tstamp[vid][succ]>clock );
27:   uint pred = prev[vid][succ];
28:   // guess suitable timestamp
29:   uint ts=nondet();
30:   assume(ts >= clock &&
31:   ts >= max_last_tstamp[tid] &&
32:   ts >= tstamp[vid][pred] &&
33:   ts < tstamp[vid][succ]);
34:   // insert node at selected position
35:   value[vid][node] = val;
36:   tstamp[vid][tid] = ts;
37:   next[vid][node] = succ;
38:   prev[vid][node] = pred;
39:   next[vid][pred] = node;
40:   prev[vid][succ] = node;
41:   // update auxiliary data
42:   max_last_tstamp[tid] = ts;
43:   last_tstamp[vid][tid] = ts;
44:   last_value[vid][tid] = val;
45: }
46:
47: void ind_write(uint addr,
                    int val,uint tid){
48:   // select identifier of the
      //memory address
49:   int vid = nondet();
50:   assume( vid < V );
51:   assume( address[vid] == addr );
52:   write( vid, val, tid );
53: }
```

FIGURE 4.4: eTSO-SMA `write` and `write_to_address` functions.

than the current value of `clock`. Note that, a node with this feature always exists in since the timestamp of the sentinel node is `K`. We then set the local variable `pred` to the predecessor of `succ`. Node `pred` will be the predecessor of `node` after its reinsertion in the list.

Guess a suitable timestamp for the new write (lines 29-33). First, we guess a value (line 29), then we check that it is not smaller than the current `clock` value (line 30), and the last timestamp used for the same thread (line 31). This last check guarantees that the writes from the same thread update the shared memory in the FIFO order. The last two constraints at lines 32-33 guarantee the T-CDLL invariant that nodes must be in a non-decreasing order (part 3 of T-CDLL definition). Note that we allow the timestamp of the new write to be the same as that of the previous write in the list (which corresponds to a situation where they are passed to the shared memory in the same update). However, the inequality is strict w.r.t. the next write in the list (line 33). In this way we guarantee that a write cannot overtake another write from the same thread that is already cached in the store buffer with the same timestamp (which would violate the TSO semantics).

Node insertion at selected position (lines 35-40). We can now insert the new write into the T-DCLL. We first set its value, then its selected timestamp, and finally we insert `node` between `prec` and `succ` (lines 37-40). Clearly, the resulting list is still a T-DCLL.

Update auxiliary data (lines 42-44). We update the auxiliary variables to ensure that their invariants are maintained.

4.3.6 Ind-write operation

The first step of function `ind_write` *(lines 47-53)* is to select the identifier corresponding to address, if any, and then call `write`.

4.3.7 Read operations

The function `read` takes as input a variable identifier `v` and a thread identifier `t`, and returns the value of `v` retrieved from the current state of the memory system. The implementation of `read` is shown in Figure 4.5. It first updates the clock. Then, it checks

```
55: int read( uint vid, uint tid ){
56:   clock_update();
57:   // retrieve the value from thread store-buffer
58:   if ( last_tstamp[vid][tid] > clock )
59:      return last_value[vid][tid];
60:   // retrieve the value from shared memory
61:   int node = nondet();
62:   assume( node < N &&
63:   tstamp[vid][node] <= clock &&
64:   tstamp[vid][next[vid][node]]>clock);
65:   return value[vid][node];
66: }
67:
68: int ind_read( uint addr, uint tid ) {
69: // selecting the id for the memory address
70: int vid = nondet();
71: assume( vid < V );
72: assume( address[vid] == addr );
73: return( read( vid, tid ) );
74: }
```

FIGURE 4.5: eTSO-SMA `read` and `read_from_address` functions.

whether the last performed write by `t` on `v` has not expired yet. This condition ensures that if there is still a pending write in `t`'s store buffer, then the value of the latest such write is returned, as per the TSO semantics. Otherwise, it returns the value of the latest expired write in the T-CDLL of `v` (lines 61-65), which is always guaranteed to exist by the invariant property of T-CDLLs, and as previously argued, it corresponds to the valuation of `v` in the shared memory. When a read is performed using a memory address (lines 68-74), we first retrieve the location identifier, say `v`, corresponding to the memory address `a` and then return the value returned by calling `read` on `v`.

4.3.8 Fence operation

To flush the store-buffer of a thread it is sufficient to mark all its writes as expired. Thus, function `fence` shown in Figure 4.6 at lines 76-81 first updates the clock to the current time and then sets again the clock to the value of the maximum ticket issued by thread `t` in case it results greater than the current clock.

```
76: void fence( uint tid ){
77:   clock_update();
78:   // make all thread's write expired
79:   if ( clock < max_tstamp[tid] )
80:      clock = max_tstamp[tid];
81: }
```

FIGURE 4.6: eTSO-SMA `fence` function.

4.3.9 Correctness

We have already observed the main properties concerning the correctness of our implementation. The most complicated case of the proof is for the `write` function. From the observations in the write-operations section above, we have that after the execution of `write` the corresponding T-CDLL is correctly updated. Also the current write is added with a timestamp that is not smaller than the timestamp of the last previously cached write from the same thread (line 31), and in case the two timestamps are equal and the two writes are on the same variable, line 33 guarantees that we keep the same order as in the store buffer. This implies that if we start from equivalent configurations of T and $T_{\texttt{TSO}}$, then the configurations of T and $T_{\texttt{TSO}}$ resulting after the invocation of `write` are also equivalent. Therefore, we get:

Theorem 4.1. *eTSO-SMA and TSO-SMA are equivalent.*

A formal proof of this can be given by showing that the transition system T that captures eTSO-SMA is equivalent to the transition system $T_{\texttt{TSO}}$ that captures TSO-SMA (and thus the semantics of the TSO memory model) in the sense that they can simulate each other behaviors going through equivalent configurations.

In this section, we show that the implementation eTSO-SMA based on the notion of vw-lists correctly captures the semantics of the TSO memory model. For this, we prove that the implementation schemes eTSO-SMA and TSO-SMA are equivalent in the sense that they can simulate each other step-by-step going through equivalent configurations as formalized below.

We start by describing the transition systems T and $T_{\texttt{TSO}}$ that capture the semantics of eTSO-SMA and TSO-SMA respectively.

A store buffer for a thread t is a sequence of zero or more tuples of the form (v, val, t) where v and val are respectively the location and the value of the write. According to the architecture from Figure 2.1, a *TSO memory configuration* is a tuple of the form $(\nu, \langle b_t \rangle_{t \in Th}, Ter)$ where ν is a valuation of the shared memory, $\langle b_t \rangle_{t \in Th}$ denotes a tuple of valuations of store buffers b_t for each thread $t \in Th$, and $Ter \subseteq Th$ is the set of terminated threads.

The transition system $T_{\texttt{TSO}}$ that captures the semantics of TSO-SMA is defined as follows. The set of states $S_{\texttt{TSO}}$ is the set of the TSO memory configurations augmented with a new

state that is taken as the initial state. Transitions are labeled with the corresponding action, i.e., the call of a function from the API or a thread creation/join/termination actions. From the initial state only transition that correspond to the init function can be taken, and the effect is to reach a TSO memory configuration that gives the initial valuation to the shared locations and has only one store buffer that is empty (the store buffer of the main thread, that is the only one created in the beginning). From each TSO memory configuration there is a transition that adds an empty store buffer for a thread creation and a transition that adds a thread from Th to Ter for thread termination. Transitions over a join are allowed only from configurations where the waited thread is terminated. When a thread is terminated no more actions from this thread are allowed. The other transitions are defined for the remaining API functions and according to the TSO semantics given in Section 4.2. For example, a write of v with value 3 by thread t_1 from a state $(\nu, \langle b_t \rangle_{t \in Th})$ is captured by a transition to a state $(\nu, \langle b'_t \rangle_{t \in Th})$ where $b'_t = b_t$ for $t \neq t_1$ and $b'_{t_1} = b_{t_1}.(v, 3, t_1)$. Similarly, a read of v by t_1 that returns 5 is captured by a self-loop onto all the states of the form $(\nu, \langle b_t \rangle_{t \in Th})$ such that either the last write of v cached in b_{t_1} gives value 5 or, in case there is no such write in b_{t_1}, ν evaluates v to 5. Dynamic memory allocation extends the valuation ν with new locations.

Analogously, one can define the semantics of eTSO-SMA by a transition system T according to the description given in Section 4.2. In particular, a valuation of a vw-list for a location v is a sequence of tuples of the form (v, val, ts, t) where val is the value assigned to v, ts is the associated timestamp and t is the thread that has performed the write (note that for the ease of presentation, differently from the description given in Section 4.2, we annotate into each tuple also the thread and the location name). Then, the set of states S of T is the set of all the tuples of the form $(d, Th, Ter, \langle q_v \rangle_{v \in Var})$ where d is the current timestamp, Th is the set of created threads, $Ter \subseteq Th$ is the set of terminated threads and q_v is a valuation of the vw-list of location $v \in Var$. We observe that for dynamic memory allocation, we add a new vw-list for each newly created location and thread creation just adds the newly created thread to Th.

We recall that a run of a transition system is a sequence of transitions $s_0 \overset{a_1}{\rightarrow} s_1 \overset{a_2}{\rightarrow} \dots s_h$ where s_0 is an initial state and for each $i = 0, \dots, h-1$, $s_i \overset{a_{i+1}}{\rightarrow} s_{i+1}$ is a transition from s_i to s_{i+1}.

Informally, two SMAs are equivalent if they can simulate each other step-by-step going through equivalent configurations. Precisely, given two SMAs, denote with T_i the respective transitions system, and let S_i be the set of states of T_i, for $i = 1, 2$. We say that T_1 is *equivalent* to T_2 if there is a function λ that maps states from S_1 to states from S_2 and for each sequence of actions $a_1 \dots a_h$:

- for each run $s_0 \overset{a_1}{\rightarrow} s_1 \overset{a_2}{\rightarrow} \dots s_h$ of T_1, there is a run $s'_0 \overset{a_1}{\rightarrow} s'_1 \overset{a_2}{\rightarrow} \dots s'_h$ of T_2 such that $\lambda(s_i) = s'_i$ for $i = 1, \dots, h$;

- for each run $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots s_h$ of T_2, there is a run $s'_0 \xrightarrow{a_1} s'_1 \xrightarrow{a_2} \dots s'_h$ of T_1 such that $\lambda(s'_i) = s_i$ for $i = 1, \dots, h$.

Note that the above notion of equivalence ensures that the two SMAs update the memory consistently on every sequence of concurrent statements that can be issued in a multithreaded program. Therefore, they can be used interchangeably without altering the semantics of the programs.

Since TSO-SMA is the SMA reference implementation for the TSO memory model, we get that any implementation that is equivalent to it correctly captures the TSO memory model.

We can now show Theorem 4.1.

Proof. We start by defining a function λ.

For each state $s = (d, Th, Ter, \langle q_v \rangle_{v \in Var}) \in S$, we define the corresponding TSO memory configuration $\lambda(s)$ as the state $(\nu, \langle b_t \rangle_{t \in Th}, Ter) \in S_{\texttt{TSO}}$ such that $|\nu| = |Var|$ and:

- ν is the valuation that assigns to each location v the value given by the head of the corresponding list q_v, i.e., for each shared location v, the head of the vw-list is tuple of the form $(\nu[v], ts, t)$ where $\nu[v]$ denotes the valuation of v by ν;
- each store buffer b_t is the sequence of writes by thread t taken from the tails of the lists q_v for any $v \in Var$ and ordered according to the timestamps; more precisely, let α be the minimal sequence such that each q'_v, $v \in Var$, is a subsequence of α where $q_v = (v, val_v, ts_v, t_v).q'_v$ (i.e., q'_v is the tail of q_v); for each thread $t \in Th$, b_t is the maximal subsequence of α containing writes by t (modulo projecting out t from the tuples).

We also set $\lambda(s_{in})$, where s_{in} is the initial state of T, to the initial state of $T_{\texttt{TSO}}$.

The "if" direction directly follows from the fact that the relation $\{(s, \lambda(s)) \mid s \in S\}$ is a simulation. This can be shown by case inspection and follows the description given in Section 4.2.

For the "only if" direction, we fix a run ρ of $T_{\texttt{TSO}}$. From this run, we assign increasing timestamps to the write operations according to the order in which they occur in the run. Then, we simulate step-by-step the transitions of ρ in T by choosing at the write operations the timestamps computed above. Note that in T the timestamps are guessed nondeterministically and are bounded from below by the current timestamp, thus the computed timestamps can be selected. By case inspection and according to the description given in Section 4.2, we can show that this way from any run $s_0 s_1 \dots s_h$ of $T_{\texttt{TSO}}$ we can compute a run $s'_0 s'_1 \dots s'_h$ of T such that $\lambda(s'_i) = s_i$ for $i = 1, \dots, h$. □

4.4 Extension to the PSO memory model

We recall that the semantics of PSO is the same as for TSO except that each thread is endowed with a store buffer for each shared memory location. To handle PSO we just need to modify the implementation of eTSO-SMA with the following changes in the write function from Figure 4.4. In the write of a location `v` by a thread `t` we do the following:

- the guessed timestamp `ts` must be not lower than the timestamp of the last write of `t` on `v` (according to the PSO semantics, a write by a thread t of a variable following a previous write by t can overtake it, but cannot overtake a previous write of the same variable);
  line 31 of Figure 4.4 must be replaced with
  ```
  && ts >= last_tstamp[v][t]
  ```
- `ts` must be the last timestamp of `t` if it is greater than the current one; line 42 of Figure 4.4 must be replaced with
  ```
  max_last_tstamp[t] =
      (ts<=max_last_tstamp[t])?
           max_last_tstamp[t]:ts;
  ```

Denoting with ePSO-SMA our prototype tool obtained from eTSO-SMA by the above changes and with PSO-SMA the reference implementation for PSO (obtained similarly to TSO-SMA for TSO), we get:

Theorem 4.2. *ePSO-SMA and PSO-SMA are equivalent.*

4.5 LazySMA Implementation and Evaluation

Prototype implementation. We implemented our approach for C programs with POSIX threads in a prototype tool called LazySMA. It is based on the open-source CSeq framework [Inv15, INF+15] (see Section 2.6.3 for an overview) which allows the development of sequentializations following a modular approach. In this framework, tools are built as pipelines of source-to-source transformations where the result of the last transformation is fed into a sequential analysis backend. For our prototype, we implemented a new transformation that replaces each memory access with the corresponding operation from the API, as described in this chapter. In order to combine this new transformation with Lazy-CSeq [ITF+14a], we needed to "inject" the new memory management layer into a few locations where memory and concurrency handling overlap. We used CBMC v5.2 as sequential verification backend for our experiments.

The complete configuration (see Figure 4.7) is defined by a sequence of 17 modules (16 `Translators` and 1 `Wrapper` module), which can be conceptually grouped according to the following categories:

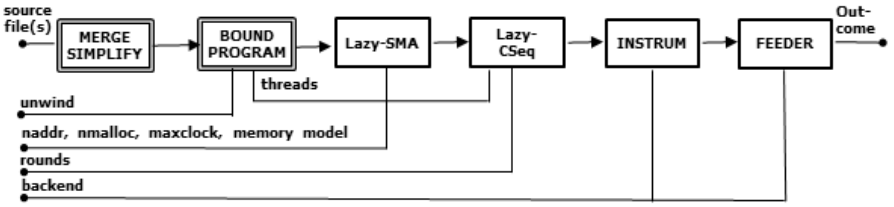

FIGURE 4.7: Architecture of the LazySMA tool. Double framed boxes denote modules composed by multiple submodules.

1. the source merging module that preprocesses the source files merging them into a single file (built-in CSeq Framework), followed by eight simple translators that rewrite the input program into a simplified syntax (built-in CSeq Framework);
2. four `Translators` for program flattening to produce a bounded multi-threaded program (built-in CSeq Framework);
3. a module that replaces each memory access with the corresponding operation from the API, as described in this chapter (a new module, with about 3k lines of code);
4. a module implementing the lazy sequentialization for SC memory model [ITF+14a] that yields a backend-independent sequentialized file (built-in CSeq Framework);
5. a module that instruments the sequentialized file for a specific backend (built-in CSeq Framework);
6. a `Wrapper` for backend invocation and user report generation (built-in CSeq Framework).

Our tool can be downloaded from this link[3]. More information about its usage is available in the README file in the installation package at the URL above.

Experimental set-up. We have compared our prototype implementation against two tools with built-in support for analysis under weak memory models: CBMC [HK15], a mature bounded model checker tool for C/C++ programs, and Nidhugg [AAA+15], a bug-finding tool that combines stateless model checking with dynamic partial order reduction on relaxed memory executions.

We ran the experiments on a dedicated machine with a Xeon W3520 2.6GHz processor and 12GB of physical memory running 64-bit linux 3.0.6. We set a 10GB memory limit and a 600s timeout for the simple benchmarks and timeout of 14,400s for the safestack example. For each tool and benchmark, we set the parameters to the minimum value needed to expose the error.

[3] http://users.ecs.soton.ac.uk/gp4/cseq/fmcad16.zip

TABLE 4.1: Analysis runtimes under TSO and PSO

		parameters							TSO runtime (s)					PSO runtime (s)				
	l.o.c	unwind	qsize (N)	naddr	nmalloc	bitwidth	rounds	maxclock (K)	bug?	files	LazySMA	CBMC	NIDHUGG	bug?	files	LazySMA	CBMC	NIDHUGG
dekker	52	1	2	0	0	4	2	2	•	1	0.77	0.29	**0.04**	•	1	0.75	0.25	**0.05**
lamport	78	1	2	0	0	4	2	2	•	1	0.88	0.31	**0.05**	•	1	0.88	0.29	**0.05**
peterson	40	1	3	0	0	4	2	2	•	1	0.66	0.26	**0.04**	•	1	0.65	0.25	**0.04**
szymanski	57	1	3	0	0	4	2	3	•	1	0.81	0.34	**0.07**	•	1	0.80	0.32	**0.04**
fib_longer_unsafe	30	6	2	0	0	10	6	2	•	1	**6.47**	8.19	94.84	•	1	6.51	**1.69**	135.45
fib_longer_safe	30	6	2	0	0	10	6	2		1	**9.78**	22.5	t.o.		1	**8.82**	31.8	t.o.
parker	110	1	2	0	0	4	2	3	•	1	1.68	0.31	**0.05**	•	1	2.19	0.28	**0.05**
stack_unsafe	110	2	2	1	2	5	2	2	•	1	1.50	0.41	**0.05**	•	1	1.49	0.35	**0.05**
litmus_safe (avg)	297k	5	2	0	0	10	2	20		5526	1.26	**0.17**	2.35		4835	1.22	**0.15**	6.65
litmus_unsafe (avg)		5	2	0	0	10	2	20	•	277	1.27	**0.16**	3.86	•	968	1.26	**0.12**	1.58

Simple benchmarks. We first evaluated our approach over a set of (relatively simple) benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite. The results are summarized in Table 4.1. The `unwind` parameter was used by all the three tools considered in the comparison. `naddr` and `nmalloc` bound the number of locations that can be accessed only through their memory addresses and the number of malloc operations along any computation, respectively. The parameter `bitwidth` gives the size of integers (in bits) used in the sequential analysis and the parameter rounds is the number of rounds used by Lazy-CSeq.

The first block contains results for the classical mutual exclusions algorithms (`dekker`, `lamport`, `peterson`, `szymanski`). These implementations are correct under SC but not under TSO or PSO. All tools find the errors, but because of their small size[4], Nidhugg outperforms both our prototype and CBMC on these programs.

The second block of the table contains variations of the fibonacci-benchmark, in which two worker threads concurrently increase two shared counters, and a main thread checks whether any the two counters can reach a defined value. A full exploration of the thread interleavings is required to identify the error (or show its absence) in this program. Techniques such as partial-order reduction do not apply, and several tools struggle to analyze it. We have included both the safe and unsafe versions. Here, Nidhugg is generally slower than both CBMC and our prototype tool, and fails to terminate on the safe version. Our prototype beats CBMC on the unsafe cases and on the safe case under TSO, but is slower on the safe one under PSO.

The next two benchmarks originate from industrial code: `parker` models a semaphore-like synchronization class that breaks under TSO [AAA+15] (and thus also under PSO), and `stack` which was taken from SV-COMP [Bey15]. Here, Nidhugg outperforms both our prototype and CBMC.

[4]On small programs the sequentialization overhead, which is for all intents and purposes a constant effort for our prototype, is high compared to the verification time. Hence, the other tools are faster on the small programs.

The last two lines show the average values for 5803 litmus tests for WMMs; note that we ran these under TSO and under PSO. For TSO, both our prototype and CBMC successfully identified the 277 test cases containing a reachable error, while Nidhugg failed to find one of them. For PSO, CBMC claims that there are 971 unsafe instances while Nidhugg and IMU-CSeq both find only 968 unsafe ones. We have manually inspected the counter-examples produced by CBMC (v5.2). We confirm that CBMC produces spurious counter-examples on those benchmarks. The performance gap between CBMC and our tool could be reduced with a more efficient implementation, as our prototype transforms each file nearly 20 times, each time requiring parsing and unparsing.

Safestack. We have conducted further experiments on a real world benchmark, Safestack [CJZ+13], which is a lock-free stack implementation designed for weak-memory models. It contains a rare bug that is hard to find with automatic bug-finding techniques already under SC (including random testing, Nidhugg, CIVL [ZRL+15], and other approaches based on BMC) [TDB14]. The only tool we are aware of that can automatically find a genuine counter-example is Lazy-CSeq. It requires a minimum of 3 loop unwindings and 4 rounds of computation to expose a bug. This actually shows that the error is quite deep which explains why other approaches based on explicit handling of interleaving fail.

Safestack is written in C++. We manually translated it into C, providing simulation functions for the C11 atomic functions used in the test. We experimented with this C version, with different bounds for the queue size of each memory address, and the maximal timestamp along any bounded computation. Table 4.2 summarises these experiments. Note that we only used three bits to represent integers during the analysis. We then checked whether counterexamples found also hold for full 32-bits integers, by running Lazy-CSeq over the exact schedule extracted from the counterexample. A “Yes” entry in the CEX? column means that the counterxample holds, thus there are no spurious counterexamples (due to overflow).

Because SC and TSO coincide if maxclock is set to 1, the first four lines indirectly show the overhead paid for our TSO encoding. Since the SC analysis using Lazy-CSeq (not shown in the table) requires approximately 3 minutes, the TSO encoding itself thus introduces an approximately 3x-4x overhead. The last two lines show that we can still find the error under “proper” TSO. It also shows that the weaker memory model reduces to 3 (from 4) the number of rounds required to expose this error; however, the analysis time grows noticeably, by almost an order of magnitude. Finally, increasing maxclock (for fixed values of qsize and rounds) shows that the analysis explores more reorderings of reads over writes (witnessed by the increased memory consumption).

TABLE 4.2: Analysis runtimes for SafeStack under TSO and PSO

parameters			TSO analysis (3 bits)			CEX check (32 bits)		PSO analysis (3 bits)		CEX check (32 bits)	
m	q	r	Time	Memory	Reach?	CEX?	Time	Time	Reach?	CEX?	Time
1	2	4	10m18s	0.8GB	Yes	Yes	23s	11m42s	Yes	Yes	4.82s
1	2	3	12m2s	0.6GB	No	-	-	11m16s	No	-	-
1	3	4	13m45s	1.2GB	Yes	Yes	30s	21m6s	Yes	Yes	6.40s
1	3	3	12m50s	0.9GB	No	-	-	12m20s	No	-	-
3	2	4	26m55s	1.4GB	Yes	Yes	24s	20m47s	Yes	Yes	4.33s
3	2	3	24m34s	1.0GB	No	-	-	27m15s	No	-	-
3	3	4	74m22s	3.4GB	Yes	Yes	31s	31m16s	Yes	Yes	5.47s
3	3	3	62m22s	1.0GB	Yes	Yes	30s	20m7s	Yes	Yes	2.84s
3	3	2	12m14s	0.6GB	No	-	-	11m14s	No	-	-
7	2	4	47m17s	2.4GB	Yes	Yes	27s	104m35s	Yes	Yes	6.05s
7	2	3	35m7s	1.3GB	No	-	-	36m14s	No	-	-

4.6 Related Work

The transformation of concurrent programs under TSO into equivalent programs under SC is intrinsic in the architecture from Figure 2.1. However, the explicit modeling of the store buffers in the resulting program introduces a substantial overhead for standard SC verification tools.

In [ABP11], the authors replace the store buffers with $O(k)$ local variables per thread, where k is the number of context-switches for each thread that is allowed in the analysis. The main intuition there is: when a write operation occurs, a future context number is guessed with the meaning that the write will be visible to the other threads at that context. This is similar to our guess of a future timestamp in the sense that it implicitly gives a total ordering of the shared memory updates, but in our setting this is completely unrelated to the thread context at which the memory update will occur.

In [WT15], the authors replace the store buffers by embedding each of them symbolically in the thread locations. Their translation goes through the construction of the corresponding transitions systems that seems appropriate for a backend as SPIN but in our experiments works poorly with BMC since it introduces a lot of redundancy in the constructed formulas.

Another main difference of our approach with the above-mentioned two is that we rearrange the contents of store buffers *per variable* instead of *per thread* and entirely maintain them in T-CDLLs of bounded size. This, along with the strong invariant properties of T-CDLLs, results in smaller formula encodings computed by the BMC backend tools (see Section 4.2).

Other recent work that have dealt with the verification of concurrent programs under weak memory model semantics are [AAA+15, AM14, AKNT13, AKT13, BCDM15, BAM07, ZKW15]. The most related to ours is [AKNT13] where the authors give a general reduction technique to SC by augmenting the programs with arrays to simulate the

caching and buffering due to the weak memory models and use it in combination with CBMC. In [AKT13], CBMC is enhanced with a reduction based on partial orders.

We have designed our translation to target BMC backends and used the tool Lazy-CSeq [ITF+14b, ITF+14a] for our experiments. Lazy-CSeq implements an efficient lazy sequentialization of concurrent programs that works exceptionally well with BMC backends and has won the SV-COMP twice [Bey15]. It performs a bounded context-switching analysis [QR05]. The idea of sequentialization was originally proposed in [QW04] but became popular with the first scheme for an arbitrary but bounded number of context switches [LR09].

4.7 Conclusions

In this chapter, we have given efficient implementations of *thread-synchronous* SMAs that work well in combination with Lazy-CSeq [ITF+14a]. We have demonstrated the effectiveness of this approach for finding bugs under TSO and PSO: our prototype tool is competitive with existing tools on standard benchmarks used in the literature; it also works for more complex benchmarks that are, to the best of our knowledge, out of reach for existing bug-finding approaches. We have also observed that the designed translation puts a reasonable overhead on the backend tool. Rare bugs require analysis approaches that handle interleavings symbolically. We have empirically demonstrated, using the safestack benchmark, that we can indeed find deep bugs, in contrast with other mature bug-finding techniques. We are currently looking at harder benchmarks, and in particular, in the domain of concurrent data structures. SafeStack is part of this effort.

We have developed our approach for TSO memory model, but a simple extension lets us also handle the PSO memory model. The main change was to organize the cached writes per variable and thread, and not just per variable. We believe that our approach can also be extended to further weak memory models, and leave this for future work.

Our approach has been optimized for bounded model-checking, but, in principle, sequentializations can be used with any backend analysis tool. To improve scalability, we are currently investigating a new approach that combines abstract interpretation and bug-finding, which is built on the top of the code-to-code translation presented in this chapter.

Chapter 5

Bug finding in Concurrent Programs by Memory Unwinding

In the previous chapter, we have described implementations of shared memory abstractions for TSO and for PSO that work well in combination with lazy sequentialization. Although this approach has been successful in combination with Lazy-CSeq, extending verification methods to other memory models by simply replacing the SMA implementation might not result in scalable bug finding tools.

In this chapter and in the next one, we follow the approach illustrated in Chapter 3 and describe efficient implementations of *thread-asynchronous* SMAs for SC memory model, which main novelty is the idea of *memory unwinding* (MU for short) [TIF+15b], i.e., an explicit representation of the sequence of write operations into the shared memory. This approach is complementary to other approaches [FIP13a, LR09, LMP10] and explores an orthogonal dimension, i.e., the number of write operations. We show experimentally, over the SV-COMP concurrency benchmarks that this approach works well in combination with eager sequentialization on top of bounded model checking.

In the next chapter, we introduce the concept of *individual shared memory location unwinding* that refines the notion of *memory unwinding* and give efficient BMC-implementations for bug finding also under TSO and PSO program semantics.

The chapter starts with a short introduction in Section 5.1. In Section 5.2, we recall the concept of *memory unwinding* and we define a MU-based SMA transition system that capture the semantics of the SC memory model. In Sections 5.3 and 5.4, we describe concrete C-implementations of SMAs and conclude this chapter with Section 5.5, where we give an implementation of our approach by combining our MU-based SMA implementations with the sequentialization from [TIF+15b] in a prototype tool that we call *MU-CSeq*; we show that our tool compares well with existing tools over the set of SV-COMP concurrency benchmarks.

5.1 Introduction

In this chapter, we describe an efficient implementation of *thread-asynchronous* shared memomory abstraction for SC memory model, that is based on the idea of *memory unwinding* (MU for short) [TIF+15b], i.e., an explicit representation of the sequence of write operations into the shared memory.

For the analysis, we nondeterministically guess this unwinding and then explore the behavior of the program according to any scheduling that respects this guess. This approach is complementary to other approaches and explores an orthogonal dimension, i.e., the write operations of the shared memory. It also simplifies the implementation of several important optimizations, in particular the targeted exposure of individual writes.

We provide several implementations of SMAs whose semantics is captured by the transition system $\mathcal{M}_{sc}^{mu}$ defined in Section 5.2.1, that targets bounded model checking (BMC) as underlying analysis technique. Here, we assume the SC memory model, while in the next chapter we give an extension of our approach to TSO and PSO memory models.

We first recall that for BMC technique, given a program, a property, and a bound k, BMC translates the program into a verification condition (VC) that is satisfiable if and only if the property has a counterexample of depth k or less.

In the case of BMC for programs, each statement of the bounded programs contributes to a constant portion of the formula. Therefore, to minimize the formula size of the program encoding, one needs to reduce the number of statements of the bounded program as much as possible.

Here, we discuss different implementation strategies of the memory unwinding approach that are characterized by orthogonal choices. The first choice we make is either to store in the MU all the writes of the shared variables in a run (*fine-grained* MU) or to expose only some of them (*coarse-grained* MU).

In a *fine-grained* MU, we still have different implementations: depending on how we store the values of the variables that are not written at a position of the memory unwinding, we have two implementation alternatives that we call *read-explicit* (where all the shared variables are duplicated to each position, not only those that are changed in the writes) and *read-implicit* (where only the modified variables are duplicated at each position).

In a coarse-grained MU we store at each position a partial mapping from the shared variables to values, with the meaning that the variables in the domain of the mapping are modified from the previous position and the value given by the mapping is their value at this position. A variable that is modified at position $i+1$ could also be modified between positions i and $i+1$ by other writes that are not exposed in the memory unwinding.

In Section 5.5, we give an implementation of our memory unwinding-based approach in the prototype tool *Mu-CSeq*. We evaluate our tool over the SV-COMP15 [Bey15] concurrency benchmarks; we show that it finds all the errors achieving performance on par with those of the current best BMC tools with built-in concurrency handling as well as other sequentializations.

5.2 Memory unwinding.

A $\mathtt{W}$-*memory unwinding* (MU) [TIF+15b] $\mathcal{M}$ of a concurrent program P is a sequence of writes $w_0 \dots w_{\mathtt{W}-1}$ of P's shared variables; each w_i is a triple (t_i, var_i, val_i) where t_i is the identifier of the thread that has performed the write operation, var_i is the name of the written variable and val_i is the new value of var_i. A *position* in a $\mathtt{W}$-memory unwinding $\mathcal{M}$ is an index in the interval $[0, \mathtt{W}-1]$. An execution of P *conforms to* a memory unwinding $\mathcal{M}$ if the sequence of its writes in the shared memory exactly matches $\mathcal{M}$.

5.2.1 MU transition system.

A corresponding transition system guesses an MU on the init()-transition and then executes the operations consistently with this guess. For SC, the corresponding transition system $\mathcal{M}_{sc}^{mu}$ will keep for each thread the current position in the MU and for any input sequence α, it ensures that:

- on write(v,val,t) (resp. ind_write(a,val,t)), the next write in the MU for thread t matches the value val and variable identifier v (resp. address a);
- on read(val,v,t) (resp. ind_read(val,a,t)), there must be in the MU a write at a position i from the current position of t through the next write of t, that assigns value val to the location identified by v (resp. a); the current position of t is updated to i in the next state;
- for each thread, the writes are matched exactly in the same order as in the MU.

In order to accept α, create(t,f,pt) must occur in α for each thread t with writes guessed in the MU and the writes in the MU should be mapped 1-to-1 to the writes in α.

The transition system $\mathcal{M}_{sc}^{mu}$ is thread-wise equivalent to $\mathcal{M}_{sc}$, and additionally, it can execute all the computations of $\mathcal{M}_{sc}$ by advancing each involved thread in any order. Moreover, due to the fact that all writes are guessed in advance, the ordering in which we interleave the threads is irrelevant. Thus, the following lemma holds.

Lemma 5.1. $L(\mathcal{M}_{sc}^{mu}) = (L(\mathcal{M}_{sc}))^{\#}$.

Proof. We start showing that $L(\mathcal{M}_{sc}^{mu}) \supseteq (L(\mathcal{M}_{sc}))^{\#}$. For $\alpha \in L(\mathcal{M}_{sc})$, denote with μ the MU that corresponds to the sequence of writes in α and with ρ an accepting run of $\mathcal{M}_{sc}$ over α. We recall that $\mathcal{M}_{sc}^{mu}$ on the `init` transition can guess any MU and is built on the top of $\mathcal{M}_{sc}$. Thus, $\mathcal{M}_{sc}^{mu}$ on the initial transition can enter a state storing the initial configuration γ as in ρ and μ. Now, since μ and the initial configuration γ fully capture the configurations of the shared memory along ρ (memory locations that are not assigned can be neglected), $\mathcal{M}_{sc}^{mu}$ can simulate the execution ρ by arbitrarily advancing the execution of each involved thread in any order. Thus, $\mathcal{M}_{sc}^{mu}$ accepts all words in $\{\alpha\}^{\#}$ and therefore, $L(\mathcal{M}_{sc}^{mu}) \supseteq (L(\mathcal{M}_{sc}))^{\#}$.

For the other direction, i.e., $L(\mathcal{M}_{sc}^{mu}) \subseteq (L(\mathcal{M}_{sc}))^{\#}$, let $\alpha \in L(\mathcal{M}_{sc}^{mu})$ and denote with μ the MU that is guessed on an accepting run over α. Note that for each word in $\{\alpha\}^{\#}$ there is an accepting run of $\mathcal{M}_{sc}^{mu}$ such that μ is the guessed MU. Now, let $\alpha' \in \{\alpha\}^{\#}$ be a word where the write operations are ordered as in μ and the read operations are ordered such that for each pair of matching read and write: 1) the read follows the write, and 2) there are no other writes involving the same location between them. Clearly, $\alpha' \in L(\mathcal{M}_{sc})$ and therefore $\alpha \in (L(\mathcal{M}_{sc}))^{\#}$. □

In the following section, we give several implementations of thread-asynchronous fine-grained MU-based SMAs whose semantics is captured by $L(\mathcal{M}_{sc}^{mu})$. In Section 5.5 we combine them with the sequentialization from [TIF+15b] which transforms a multi-threaded program P_1 into a thread-wise equivalent sequential program P_2. Lemma 5.1, and Theorems 3.1 and 3.2 show the correctness of the resulting verification method. Our implementations would require parameterization on the number of writes and threads.

5.3 Fine-grained MU-based SMA

In our implementations of the fine-grained MU-based SMA we primarily target to minimize the formula size. We first discuss of a straightforward implementation that does not lead to compact formulas. This allows us to motivate our approaches incrementally.

A simple implementation would represent the memory unwinding as a sequence of triples each of which represents a write operation into the shared memory. A triple has the form (var, val, t) where var is the name of the written variable, val is the written value, and t is the identifier of the thread that has performed that write operation. The memory also keeps track of the memory position of each thread at each point in the computation. A *write operation* performed by thread t can be realized by moving forward the position of t along the memory sequence to the next write operation performed by t, and then check that the variable and the value in the triple at the new position are consistent with those passed to the `write` function as parameters. Similarly, a *read operation* first non-deterministically jumps forward in the sequence at a position between the current

thread position and the position of the next write operation performed by t (if any). Then, going backward on the memory sequence we retrieve the first triple that corresponds to a write of the read variable. Thus, we return the value from that triple. Although, this does not provide a full detailed implementation of the shared memory class, it is easy to see that each write and read operation requires a number of steps that is proportional to the length of the memory sequence, that is `W`. Thus, for a bounded program P this leads to a formula whose size is proportional to $n + (r + w) \cdot \texttt{W}$, where n is the size of P, and r and w are the number of read and write operations of P, respectively.

Our first implementation is better suited for concurrent programs with a small number of shared variables, as opposed to the second implementation that is particularly adequate for programs with a large number of shared variables (e.g., large arrays). We also give a third implementation which is an hybrid of the first two.

5.3.1 Fine-grained MU-based SMA implementations

In this section, we provide a detailed implementation of the shared memory that is more suitable when we use our analysis in combination with BMC tools. Low level implementation details are important for scalable solutions. We provide three implementations which are the same except the way in which we implement the *read* function. In this section, we first describe the common part of these implementations, and then we specialize them only describing the differences.

We assume that shared variables and threads are each identified by an unsigned integer.

The implementations are parameterized over several constants.

- `T` denotes the maximal number of threads that can be spawned during any execution of the input program,
- `V` denotes the number of tracked memory locations,
- `N` denotes the number of shared scalar variables (*locations with names*),
- `U` denotes the maximum number of locations that can be accessed only through their memory addresses (*locations without names*),
- `M` denotes the maximum number of dynamic memory allocations, and
- `W` denotes the maximum number of write operations of the shared memory.

Note that `V=N+U` holds.

5.3.1.1 Data structures and invariants.

We represent the memory sequence using 3 arrays:

- `uint thread`[`W`];
- `uint var`[`W`];
- `int value`[`W`].

For every memory position $i \in [0, \mathtt{W}-1]$, `thread`$[i]$ is the identifier of the thread that performs the i-th write operation into the memory sequence, `var`$[i]$ is the written variable, and `value`$[i]$ is the value written into variable `var`$[i]$. When the memory is instantiated, all elements of these arrays are assigned with non-deterministic values, and remain unchanged along the computation of the concurrent program.

Except for `value`, the elements of the arrays defined above are constrained as follows. For each memory position $i \in [0, \mathtt{W}-1]$,

- `var`$[i] \in [0, \mathtt{V}-1]$;
- `thread`$[i] \in [0, \mathtt{max\_th}-1]$;

where,

```
uint max_th
```

is an additional variable, that is non deterministically assigned with a value in the interval $[1, \mathtt{T}]$; that value represents the exact number of threads that will be created in the current execution. This variable is set when the memory is instantiated and will never be modified.

We also employ a variable named

```
uint last_write
```

to guess the exact position of the last write operation in the memory sequence. This variable is non deterministically assigned with a value in the range $[0, \mathtt{W}-1]$ and is never modified.

We use a global array which contains the physical memory address of v, for any $v \in [0, \mathtt{V}-1]$; this array is nondeterministically initialized with distinct increasing values (i.e.,

`address`$[i] <$ `address`$[i+1]$ for $i \in [0, V-2]$) and its values stay unchanged during the program execution

```
uint address[V]
```

Concerning to `malloc` we maintain a bounded sequence, say of fixed size `M`, of memory blocks that can be allocated dynamically. In particular, for each block we store its base address and whether it has been allocated. This sequence is implemented using the following arrays

- `uint mallocP[M];`
- `uint mallocPallocated[M]`.

The array `mallocP` is nondeterministically initialized with distinct increasing values; we additionally impose that the address guessed for the last named location is less than the one assigned to the base location of the first memory allocation (i.e., `address[N-1]<mallocP[0]`). These values will not change during the program execution.

Next thread operation. When a thread, say t, invokes a write operation, as described at the end of Section 5.3, we need to logically jump forward in the memory sequence at the position of the next write operation performed by t. To avoid a scan of the memory sequence, we define a new array that allows jumping directly to that position. More precisely, we define

```
uint th_next_write[W][T]
```

where for every memory position $i \in$ [0,`W`-1] and thread identifier $t \in [0, \texttt{T}-1]$, `th_next_write`$[i][t]$ is the first position in the memory sequence after i corresponding to a write operation labelled by t, if any; otherwise, it is set to `W`. More formally, let `th_next_write`$[i][t] = j$. Then, either (1) $i < j <$ `W`, `thread`$[j] = t$, and `thread`$[z] \neq t$ for every $z \in]i, j[$, or (2) $j =$ `W` and `thread`$[z] \neq t$, for every $z \in]i, \texttt{W}-1]$.

5.3.1.2 Simulation variables and auxiliary functions.

During the execution, we keep track of the memory position of each thread, using the array

```
uint th_pos[T]
```

Position t in `th_pos[T]` is initialized when a thread with identifier t is created with the current memory position of the parent thread just before the creation of t. The idea is that t is created when the parent was at that position, and therefore t cannot use any previous memory position. We also use a variable

```
uint th_count
```

```
bool is_th_terminated(uint t){
   if(ret[t]||nondet()) {ret[t]=1; return 1;}
   return 0;
}
uint Jump(uint t){
   uint jump=*;

   assume(   (jump <= last_write)
          && (jump <  th_next_write[t])
          && (jump >= th_pos[t]));
   return jump;
}
```

FIGURE 5.1: Fine-grained MU: Auxiliary functions for `MU` implementation.

to keep track of the number of created threads along the computation. This variable is initialized to 0. At the end of the execution we verify that its value coincides with the initially guessed value for `max_th`.

Also, for each thread `t`$\in$[0,`T-1`] we use the following variables:

- `last_t_write`[`t`] is the position of the last write operation in the MU by thread `t`; (this value is guessed nondeterministically in the initialization and is never changed; it should match `th_pos`[`t`] in the end of the current computation);
- `ret`[`t`] is set to 1 to mean that `t` has been interrupted before reaching the end of its execution;
- `terminated`[`t`] is set to 1 to mean that we expect that thread `t` will be stopped before the execution of its last statement (this value is guessed nondeterministically in the initialization and is never changed).

We also define the two auxiliary procedures illustrated in Figure 5.1. The function `is_th_terminated` returns 1, if `ret[t]` is already set to 1, and nondeterministically chooses either to set `ret[t]` to 1 and then return 1, or to return 0. Function `Jump` modifies the memory position of thread `t` to a new non-deterministic position that is within the current position of the thread and the position of the next write operation of that thread, if any. The first condition of the assume statement makes sure that we will not overcome the index of the last write operation of the sequence, i.e. it will never jump in a not valid position. This procedure is often used in the memory procedures to move along the memory sequence.

All introduced variables and arrays are declared as global, and their constraints are imposed using the procedure in Figure 5.2.

Function `init_address` ensures that array `address` is nondeterministically initialized with increasing values (i.e., `address`$[i]$ < `address`$[i+1]$ for $i \in [0, V-2]$). Function `init_malloc` ensures the same for array `mallocP` and additionally imposes that the address guessed for the last named location is less than the one assigned to the base

```
int common_constraints(uint V, uint W,
                       uint T, uint M){

  init_address(V);
  init_malloc(W);

  th_count = 0;

  last_write = *;
  assume( last_write < W );

  max_th = *;
  assume( max_th <= T );

  int i=0;
  while (i<W){
    thread[i] = *;
    assume( thread[i] < max_th);
    var[i] = *;
    assume( var[i] <= V);
    value[i] = *;
    i=i+1;
  }
```

```
  int t=0;
  while (t<T) {
    terminated[t] = *;
    last_t_write[t] = *;
    assume (last_t_write[t] < W);
    t=t+1;
  }

  t=0;
  while (t<=T) {
    th_next_write[W-1][t] = W;
    t=t+1;
  }

  i=W-2;
  while (i>=0) {
    t=1;
    while (t<T) {
      if (thread[i+1] == t)
         th_next_write[i][t]=i+1
      else
         th_next_write[i][t]=
                 th_next_write[t][i+1];
      t=t+1;
    }
    i=i+1;
  }
}
```

FIGURE 5.2: Fine-grained MU: Constraints on memory structures.

location of the first memory allocation (i.e., `address[N-1]<mallocP[0]`). Functions `init_malloc` and `init_address` are illustrated in Figure 5.3.

```
void init_address(uint V){
  int i=0;
  while (i<V){
    address[i] = *;
    if(i>0)
      assume( address[i] > address[i-1]);
    i=i+1;
  }
}
```

```
void init_malloc(uint M){
  int i=0;
  while (i<M){
    mallocPallocated[i]=0;
    mallocP[i] = *;
    if(i>0)
     assume( mallocP[i] > mallocP[i-1]);
    i=i+1;
  }
  assume( mallocP[0] > address[N-1]);
}
```

FIGURE 5.3: Fine-grained MU: init_address and init_malloc functions for `MU` implementation.

Variable `th_count` is initialized to 0. Then, we assign non deterministic legal values to variables `last_write` and `max_th`. In the first while-block of Figure 5.2, arrays `var`, `value` and `thread` are nondeterministically assigned to legal values. Proper values of `terminated` and `last_t_write` are nondeterministically guessed in the second while-block, for each thread t. The rest of `common_memory_constraints` initializes `th_next_write` such that for each thread t, all the writes from t in the MU are linked in the proper order (value `W` is used as a sentinel to denote the end of the MU).

5.3.1.3 Schema with Explicit Read Operations

The first schema that we propose inherits all structures defined in the section above and, in addition defines a new matrix `mem` that allows us to simulate read operations using a constant number of steps:

```
int mem[W][V];
```

Matrix `mem` is logically characterized as follows: for every memory position $i \in [0, \mathtt{W}-1]$ and variable (index) $\mathtt{v} \in [0, \mathtt{V}-1]$, $\mathtt{mem}[i][\mathtt{v}]$ is the valuation of variable `v` after the i-th write operation (assuming the values of arrays `thread`, `var`, and `value`). At memory position zero, all variables $v \in [0, \mathtt{V}-1]$ have value 0 which corresponds to the initial value for shared variables. For all the other memory positions, `mem` has the same valuation for all variables as in the previous position except for that one written at that position (see Figure 2.4). The initialization of `mem` is implemented by the procedure shown in Figure 5.4.

```
int memory_constraints_explicit_read(uint V, uint W){
  int v=0;
  while (v<V) {
    mem[0][v] = 0;
    v=v+1;
  }
  i=1;
  while (i<W){
    v=1;
    while (v<V){
      if (var[i] == v)
        mem[i][v]=value[i];
      else
        mem[i][v]=mem[i-1][v];

      v=v+1;
    }
    i=i+1;
  }
}
```

FIGURE 5.4: Fine-grained MU: Computation of array `mem` (explicit read-schema).

Shared memory procedures We are now ready to define the procedures for the shared memory. All structures defined above for the representation of the shared memory are declared as global. We assume that only the memory functions can access them.

Memory initialization. The memory initialization procedure is illustrated in Figure 5.5. It consists of the initialization of the general structures and the matrix `mem`.

```
void init(uint V, uint W, uint T, uint M){
  common_constraints(V,W,T,M);
  memory_constraints_explicit_read(V,W);
}
```

FIGURE 5.5: Fine-grained MU: procedure `init` (explicit read schema).

Thread creation, termination, join, and finish. The procedure for memory thread creation is shown in Figure 5.6. If the maximal number of allowed threads is reached, the procedure immediately returns -1 meaning that this thread will never be scheduled. We then increment the number of created thread and initialize the thread

```
uint create(void *f, uint pt){
  if (th_count >= max_th) return -1;
  if (pt == 0){
     th_pos[th_count]=0;
  }else{
     th_pos[th_count]=th_pos[pt];
     assume(th_next_write[0][th_count]>=th_pos[pt]);
  }
  uint th_id=th_count++;
  return th_id;
}
```

FIGURE 5.6: Fine-grained MU: Function `create`.

position of the newly created threads to the position of the thread invoking the thread creation in the original concurrent program. If the index of the parent thread is 0, then the created thread corresponds to the `main` function of the original program and the position of this new thread is set to 0, and the thread identifier is returned. Note that, in our implementation we assign identifiers to thread incrementally starting from 0.

The procedure for memory thread termination is shown in Figure 5.7. We remark that,

```
void terminate(uint t){
  uint pos;
  pos=th_pos[t];
  assume(th_next_write[pos][t] > last_write);
  assume( ret[t]==terminated[t] &&
     last_t_write[t]==pos );

}
```

FIGURE 5.7: Fine-grained MU: Procedure `terminate`.

for a correct usage of the memory at the end of the execution of each thread, we must invoke procedure thread `terminate` with the identifier of that thread. The assume statement makes sure that all write operations guessed for thread `t` have been performed. Furthermore, the concluding assume checks that the values guessed by function `init` for `terminated[t]` and `last_t_write[t]` are consistent with the explored computation. We recall that `ret[t]` is initialized to 0 and can be nondeterministically set to 1 by the auxiliary function `is_th_terminated` (this has the effect of stopping the execution of the current thread).

This is exploited by the function `join` (see Figure 5.8). This function returns immediately if the execution of thread `t1` is stopped. Otherwise, the calling thread (`t1`) first jumps forward in the memory sequence by invoking the function `Jump`, and then abort the computation in case the other thread (`t2`) either will not terminate (i.e., `terminated[t2]=1`) or has not terminated yet at the current memory position of the calling thread (but it is supposed to terminate).

```
void join(uint t1, uint t2){
   if is_th_terminated(t1) return;

   uint jump = Jump(t1);
   assume(   (!terminated[t2])
          && (jump >= last_t_write[t2]));
   th_pos[t1]=jump;
}
```

```
void finish(){
  assume (max_th==th_count);
 }
```

FIGURE 5.8: Fine-grained MU: Procedures join and `finish`

At the end of the computation, we must be sure that the guessed number of threads that were supposed to be created have been indeed created. This check has done by the procedure `finish` illustrated in Figure 5.8. It checks that the value guessed for *max_th* is equal to the number of spawned threads *th_count*.

Read and write operations. The implementation of the read and write procedures are straightforward (see Figures 5.9). For a read operation, the thread first jumps forward into the memory sequence by invoking the function `Jump` described above and then returns the valuation of the variable at the new position, recovering the value from matrix `mem`. Similarly, in a write operation, the thread first jumps to the next write operation of thread `t` and checks that the variable and the value coincides with those in the memory sequence at the new position. Notice that both operations require only a constant number of steps using the auxiliary data structures for the memory representation. This reduces the size of the bounded program quite considerably when several read and write operations are performed by the program.

Ind_read and ind_write operations. When a read or write operation is performed using a memory address, i.e. `*p = 3` for a pointer variable `p`, we invoke `ind_read` and `ind_write` methods. The implementation of these procedures are straightforward (see Figures 5.9). We first search for the location corresponding to the one whose address corresponds to the given parameter by invoking the function `address` described below, and then invoke the read/write operation at that location.

```
int read(uint v, uint t){
   if is_th_terminated(t) return;

   uint jump=Jump(t);
   th_pos[t]=jump;
   return (mem[jump][v]);
}

void write(uint v,int val,uint t){
  if is_th_terminated(t) return;

  uint jump=th_next_write[th_pos[t]][t];
  assume (    (jump<=last_write)
           && (var[jump]==v)
           && (value[jump]==val) );
  th_pos[t]=jump;
}
```

```
int ind_read(uint addr,uint t){
  if(is_th_terminated(t)) return 0;
  uint pos;
  assume(pos<V);
  assume(address(pos,t)==addr);
  return read(pos,t);
}


void ind_write(uint addr,int val,uint t){
  if(is_th_terminated(t)) return;
  uint pos;
  assume(pos<V);
  assume(address(pos,t)==addr);
  write(pos, val,t);
}
```

FIGURE 5.9: Fine-grained MU: Read and write functions (explicit read-schema).

Address operation. Method `address` is used to recover the address of a given position $v \in [0, V-1]$. The implementation for this method is given in figure 5.10. It returns the value from $address[v]$.

```
int address (uint v, uint t){

   if(is_th_terminated(t)) return 0;
   return address[v];
}
```

```
int malloc(uint n, uint t){
   uint pos;

   if(is_th_terminated(t)) return 0;
   assume(pos<M);
   assume(!mallocPallocated[pos]);
   assume(mallocP[pos]+n < mallocP[pos+1]);
   mallocPallocated[pos]=1;
   return mallocP[pos];
}
```

FIGURE 5.10: Functions `address` and `malloc`.

Malloc operation. During its execution a thread can require a block of n consecutive unallocated locations by invoking `malloc`(n). When `malloc` is invoked, say with argument n, a block is chosen non deterministically, and it is allocated if its size is at least n by returning its base address. The `malloc` procedure is implemented as shown in fig 5.10. We first find a position *pos* that corresponds to a not sill allocated block, by checking the value of *mallocPallocated* at that position. We recall that addresses stored in *mallocP* are ordered in ascending order; then in order to know if there is enough space we simply check that $mallocP[\text{pos}] + size < mallocP[\text{pos+1}]$. Then we set $mallocPallocated[pos]$ to true to indicate that the address at position *pos* has been allocated. Finally, we return the base address corresponding to the position *pos*.

Lock and unlock mutex variables. A thread can take or release a lock on a shared mutex variable by calling the procedure `lock` and `unlock`, respectively; their implementations are provided by Figure 5.11. For a mutex variable, we assign value

T when the lock is not acquired by any thread, and we assign value `t` if the mutex is held by thread `t`. We assume that all mutex are initially unlocked; then each of them is initialized to T.

```
void lock(uint mut, uint t){
  write(mut, t, t);
  assume(ret[t] ||
      mem[th_pos[t]-1][mut]==T);
}
```

```
void unlock(uint mut, uint t){
  write(mut, T, t);
  assume(ret[t] ||
      mem[th_pos[t]-1][mut]==t);
}
```

FIGURE 5.11: Mutex `lock` and `unlock` operations.

For efficient implementation, we modify the value of variable `mut` using a write operation. For a `lock` operation we first write the value of `t` in `mut`; however, it may be the case that the mutex was already held by some thread. Thus, we check that the previous value of `mut` was T. The implementation of the method `unlock` procedure is similar, the only difference is that we write T in to the `mut` variable.

Considerations. The memory implementation provided in this Section allows a straightforward implementation of the read and write operations that involves, for both of them, a constant number of steps. This comes at a cost of using an extra matrix of size $\texttt{W} \cdot \texttt{V}$. This amount of extra space is negligible for concurrent programs with a small number of variables. In contrast, a considerable number of shared variables (for example, involving arrays of big size), will penalise this approach. The reason is the formula encoding an execution of the initialization procedure for this matrix (see Figure 5.4) will involve a considerable number of statements. For example, a program with a shared array of $1k$ elements and 20 memory writes will lead to formula encoding of size proportional to $20k$, though there will be no more than 20 elements of them that will be written in the given bound. One solution we have implemented is to consider arrays as pointers, but with the difference that we initially allocate the space in the `init` method by invoking the `malloc` function. An alternative solution is given in the next section.

5.3.2 Schema with Implicit Read Operations

In this section we propose an alternative solution that avoids expensive data structures, but will make the read operation more costly in terms of formula size as it requires searching into arrays.

We assume all data structures defined in Section 5.3.1. Additionally, we use another array called

```
uint var_next_write[W];
```

This array is initialized as follows. For every memory position $i \in$ `[0,W-1]`, `var_next_write`$[i]$ is the smallest memory position j greater than i with $\text{var}[j] = \text{var}[i]$. This array is initialized using the procedure of Figure 5.12, and it is never modified along the program

computation. Furthermore, we use an additional auxiliary array

```
uint var_first_write[V];
```

to keep track of the position of each variable's first write in the memory sequence.

```
int memory_constraints_implicit_read(uint V, uint W){
  uint i=W-1;

  while (i>=0) {
    if (var_first_write[ var[i] ]==0)
      var_next_write[i] =  W;
    else
      var_next_write[i] = var_first_write[ var[i] ]
    var_first_write[ var[i] ] = i;
    i=i-1;
  }
}
```

FIGURE 5.12: Fine-grained MU: Computation of array `var_next_write` (implicit read-schema).

Memory initialization. The memory initialization procedure is illustrated in Figure 5.13. It initializes both the general structures and arrays `var_next_write` and `var_first_write`.

```
void init(uint V, uint W, uint T, uint M){
  common_constraints(V,W,T,M);
  memory_constraints_implicit_read(V,W);
}
```

FIGURE 5.13: Fine-grained MU: Procedure `init` (implicit read schema).

All memory procedures are the same as in the explicit-read schema, except for the read operation (that cannot use matrix `mem` anymore) and implicitly the `lock` and `unlock` procedures.

Read operation. The read procedure for the implicit-read schema is illustrated in Figure 5.14. The procedure first checks if the variable has never written in the memory sequence; in such case, it returns its initial value (which is 0). Otherwise, it guesses a non-deterministic position *jump* that is less than the position of the next write operation of the currently active thread, if any. Thus, if *jump* is less than the thread's current position, we check that *jump* coincides with the last write operation for v before the current position; otherwise, if the first write of v is next to *jump* it returns 0, otherwise it checks that the variable which is written at position *jump* coincides with the input variable v, then updates the thread's current position to *jump*.

```
int read(uint t, uint v){
  uint pos=th_pos[t];
  uint jump;

  if is_th_terminated() return;

  if  (var_first_write[v]==0) return 0;

  assume(   (jump <= last_write )
         && (jump < th_next_write[th_pos[t]][t]) );

  if (jump<pos){
    assume( ( var[jump] == v ) && (var_next_write[jump] > pos) ) ;
  }else{
    if (( jump < var_first_write[v] ) ) return 0;
    assume( var[jump] == v );
    th_pos[t]=jump;
  }
  return (value[jump]);
}
```

FIGURE 5.14: Fine-grained MU: Procedure `read` (implicit read schema).

5.3.3 Hybrid Schema with Implicit and Explicit Read Operations

This third schema is an hybrid of the first two. It uses an explicit representation for scalar variables and an implicit representation for arrays.

5.4 Coarse-grained MU-based SMA encodings

The main idea of this approach is to expose only some of the writes of a memory unwinding. This has two main consequences in terms of explored runs of the original multithreaded program. On the one side, we restrict the number of possible runs that can match a memory unwinding. In fact, the unexposed writes cannot be read externally, and thus some possible interleavings of the threads are ruled out. On the other side, we can handle larger number of writes by nondeterministically deeming few of them as interesting for the other threads.

We store a sequence of *clusters of writes* $c_0, \ldots, c_{W-1}$ where, each cluster is a snapshot of the shared memory at a given time during the execution of the program P (i.e. it is a mapping from the shared variables to values) and `W` is the bound on the number of clusters we consider in our analysis. The intended meaning is as follows: each cluster is formed by an unbounded number of writes from multiple threads, assuming a round robin schedule. The cluster c_0 is obtained starting from the initial valuation of the shared memory; clusters c_i for each $i \in [1, \texttt{W-1}]$ is formed starting from the valuation of the cluster c_{i-1}. This sequence is non deterministically guessed in the `init` function and will never change during the execution.

In our implementation, we use an additional copy of the shared variables c'_i for each position $i \in [0, \texttt{W-1}]$ in the sequence of clusters where, for each $v \in [0, V-1]$, $c'_0[v]$ is

initialized with the initial valuation of the shared memory and $c'_i[v]$ is initialized with the value of $c_{i-1}[v]$ for each $i \in [1, \texttt{W-1}]$. During the program computation, each thread executes read and write operations from c'. At the end of the execution, we check that all clusters have been matched, i.e. the value of $c'_i[v]$ coincides with the value of $c_i[v]$ for each $i \in [0, \texttt{W-1}]$ and for each variable $v \in [0, \texttt{V-1}]$.

For a write operation of a variable v, we first jump forward in the memory to any position k that has not been matched yet, between the current position and the last valid memory position, then we update the value of $c'_k[v]$; finally we non deterministically decide whether to match the cluster. In this case, we check that the valuation of the shared variables in c'_k coincides with the guessed valuation in c_k; no other writes will be allowed at this position.

A read of a variable v from thread t at a position i is performed by jumping to any valid position $k \geq i$; then we return the value of $c'_k[v]$.

5.4.1 Auxiliary Data Structures

Let `T` be a constant denoting the maximal number of threads that can be spawned during any execution of the program, `V` be the maximum number of tracked memory locations, `M` be the maximum number of dynamic memory allocations, `S` be the number of shared scalar variables, and `W` be the maximum number of clusters.

During an execution of P, the sequentialized program maintains the following data structures:

- `uint memory[V][W]`: for every location $v \in$ `[0, V-1]` and every memory position $i \in$ `[0, W-1]`, `memory`$[v][i]$ is the guessed value for the variable with identifier v, just after the cluster i has been matched, i.e. after the last write for cluster i has been executed;
- `uint last_cluster_pos` is the position of the last valid cluster in the memory sequence; its value is guessed nondeterministically in the initialization and is never changed during the execution;
- `uint work_memory[V][W]`: represents the additional copy described above. At memory position zero, all variables $v \in [0, \texttt{V} - 1]$ are initialized to 0 which corresponds to the initial value for shared variables. For all the other memory positions $i \in$ `[1, W-1]`, `work_memory`$[v][i]$ is initialized to `memory`$[v][i-1]$ for each variable $v \in [0, \texttt{V} - 1]$. At the end of the computation, we check that `work_memory[v][i]` $=$ `memory[v][i]` for each memory position $i \in [0, last\_cluster\_pos]$ and for each variable $v \in [0, \texttt{V} - 1]$;
- `bool is_matched[W]`: for every position $i \in$ `[W]`, `is_matched`$[i]$ is true if the cluster at position i has been matched;

– `uint last_t_cluster[T]`: for every thread $t \in [T]$, `last_t_cluster[t]` is the position of the last memory operation in the MU by thread `t`; (this value is guessed nondeterministically in the initialization and is never changed; it should match `th_pos[t]` in the end of the current computation);

– `uint th_pos[T]`, `uint max_th`, `uint th_count`, `uint address[V]`, `uint mallocP[M]`, `uint mallocPallocated[M]`, `bool terminated[T]` and `bool ret[T]`, have the same meaning described in Section 5.3.1.

5.4.2 Shared memory procedures (Coarse-grained)

Memory initialization. All introduced variables and arrays are declared as global in the sequentialised program. The implementation is very basic and it is illustrated in Figure 5.15.

```
int init(void){

  init_address(V);
  init_malloc(V);

  th_count = 0;

  max_th = *;
  assume( max_th <= T );

  last_cluster_pos = *;
  assume( last_cluster_pos < W );


  int i=0, v=0;
  while (i<W){
     while (v<V){
       memory[v][i] = *;
       if (i>0)
          work_memory[v][i] = memory[v][i-1]
       v=v+1;
     }
     v=0;
     i=i+1;
   }
   int t=0;
   while (t<T) {
     terminated[t] = *;
     last_t_cluster[t] = *;
     assume (last_t_write[t] < W);
     t=t+1;
   }
}
```

FIGURE 5.15: Coarse-grained MU: Constraints on memory structures.

As in the previous implementations, the function `init` invokes functions `init_address` and `init_malloc`. Variable `th_count` is initialized to 0. Then we assign non deterministic legal values to variables `last_cluster` and `max_th`. In the first while-block of Figure 5.15, the array `memory` is nondeterministically assigned to legal values and `work_memory` is built from `memory` as described above. Legal values of `terminated` and `last_t_cluster` are nondeterministically guessed for each thread in the second while-block.

Auxiliary functions As in the previous implementations we define the following two auxiliary procedures illustrated in Figure 5.16.

Procedure `is_th_terminated` is exactly the same of the procedure used by the other previous implementations.

Procedure `Jump` is a bit simpler, it jumps forward in the memory sequence at any position i between the current and the last valid position in the memory. We recall that, for this

```
bool is_th_terminated(uint t){
   if (ret[t]||nondet())
   {ret[t]=1; return 1;}
   return 0;
}
uint Jump(uint t){
   uint pos=th_pos[t];
   uint jump=*;

   assume(   (jump <= last_cluster_pos)
          && (jump >= pos));
}
```

```
void cluster_match(){
  uint pos=th_pos[t];
  uint v=0;
  while (v < V){
      assume(memory[pos][v] ==
                  work_memory[pos][v]);
      v=v+1;
  }
  is_matched[pos]=1;
}
```

FIGURE 5.16: Coarse-grained MU: Auxiliary functions.

implementation write operations are performed by multiple threads before to expose a snapshot of the shared memory to the other threads.

For this implementation, we introduce the `cluster_match` method, whose code is illustrated in Figure 5.16. It is called by the last write of each cluster and checks that the valuations of the arrays `memory` and `work_memory` coincide at the current memory position *pos*. Furthermore, we set *is_matched*[*pos*] to 1. We recall that, at the end of the execution, all clusters must be matched. This will be exploited by the function `finish` described below.

Thread creation, termination, join, and finish. The procedure for memory thread creation is shown in Figure 5.17. If the maximal number of allowed threads is reached, the procedure immediately returns -1 meaning that this thread will never be scheduled.

```
int create(void *f, uint pt){
  if (th_count >= max_th) return -1;
  if (pt == 0)
     th_pos[th_count]=0;
  else
     th_pos[th_count]=th_pos[pt];
  uint th_id=th_count++;
  return th_id;
}
```

```
void terminate(uint t){
  assume( ret[t]==terminated[t] &&
     last_t_cluster[t]==last_cluster[t] );
}
```

FIGURE 5.17: Coarse-grained MU: Procedures `create` and `terminate` .

Differently to the other implementations, here we do not have any check to do at the end of the computation of each thread, except that the values guessed by function `init` for `terminated[t]` and `last_t_cluster[t]` must be consistent with the explored computation. The implementation of the function `terminate` is shown in Figure 5.17.

Function `join` is illustrated in Figure 5.18. This function returns immediately if the execution of thread `t1` is stopped. Otherwise, the calling thread (`t1`) first jumps forward in the memory sequence by invoking the function `Jump`, and then aborts the computation

in case the other thread (`t2`) either will not terminate (i.e., `terminated[t2]=1`) or has not terminated yet at the current memory position of the calling thread (but it is supposed to terminate). This last condition holds either when the thread $t1$ jumps to a position less than the last position of $t2$ or when $t1$ jumps at the last position of $t2$ but it has been created after $t2$[1].

```
void join(uint t1, uint t2){
   if is_th_terminated(t1) return;

   uint jump = Jump(t1);
   assume(   (!terminated[t2])
          && ((jump > last_t_cluster[t2])
                || ( (jump == last_t_cluster[t2])
                      && (t2 > t1) )
              );
   th_pos[t1]=jump;
}
```

FIGURE 5.18: Coarse-grained MU: Procedure `join`.

At the end of the computation we invoke the procedure `finish` illustrated in Figure 5.19. It checks that the value guessed for *max_th* is equal to the number of spawned threads, *th_count*. Furthermore, we check that all valid clusters have been matched during the computation.

```
bool finish(){
  uint i=0;
  assume (max_th==th_count);
  while (i < W){
      assume(is_matched[i] || i>last_cluster_pos);
      i=i+1;
  }
  return 1;
}
```

FIGURE 5.19: Coarse-grained MU: Procedure `finish`.

Read and write operations. The implementation of the read and write procedures are shown in Figures 5.20.

For a read operation, the thread t first jumps forward into the memory invoking the function `Jump`, then updates its thread memory position and finally returns the valuation of the variable at the new position from matrix *work_memory*.

For a write operation of a variable with identifier v, we first jump forward in the memory invoking function `Jump` to any position that has not been matched yet, then it updates the value of v at the new position into the matrix *work_memory* with the value given as input; finally, we non deterministically choose whether it is the last write for the current cluster; in such case we invoke the method `cluster_match` to check the correspondence

[1] We are assuming a round robin schedule

```
int read(uint t, uint vid){
   uint jump;

   jump=Jump(t);
   if is_th_terminated(t) return;
   th_pos[t]=jump;

   return (work_memory[jump][vid]);
}
```

```
void write(uint t,uint vid,int val){
   uint jump;

   jump=Jump(t);
   if is_th_terminated(t) return;
   assume(! is_matched[jump]);
   th_pos[t]=jump;
   work_memory[vid][jump]=val;

   if (nondet()){
     is_matched[jump] = 1;
     cluster_match();
   }
}
```

FIGURE 5.20: Coarse-grained MU: Procedures `read` and `write`.

of the auxiliary copy of shared variables and what we have guessed for the memory sequence at that position.

Lock and unlock mutex variables. Figures 5.21 provides an implementation for the lock and unlock procedures, respectively.

```
void lock(uint t, uint mut){
   uint jump;

   jump=Jump(t);
   if is_th_terminated(t) return;
   assume(! is_matched[jump]);
   th_pos[t]=jump;
   assume (work_memory[mut][jump]==T);
   work_memory[mut][jump]=t;

   if (nondet()){
     is_matched[jump] = 1;
     cluster_match();
   }
}
```

```
void unlock(uint t, uint mut){
   uint jump;

   jump=Jump(t);
   if is_th_terminated(t) return;
   assume(! is_matched[jump]);
   th_pos[t]=jump;
   assume (work_memory[mut][jump]==t);
   work_memory[mut][jump]=T;

   if (nondet()){
     is_matched[jump] = 1;
     cluster_match();
   }
}
```

FIGURE 5.21: Coarse-grained MU: Procedures `lock` and `unlock`.

For a `lock` operation, we first jump forward into the memory sequence; the first assume makes sure that we can write at that position, i.e. the cluster has not been matched yet. Then, we check that the mutex `mut` is unlocked; finally we update the value in the *work_memory* corresponding to the variable `mut` and non deterministically decide whether to match the current cluster. The implementation of the method `unlock` procedure is similar, the only difference is that we write T in to the `mut` variable.

5.5 Mu-CSeq Implementation and Evaluation

Prototype implementation. We have implemented in MU-CSeq (v0.3) the different variants of the memory-unwinding based SMAs discussed in this chapter and the sequentialization from [TIF+15b]. Our transformation replaces the memory accesses with the corresponding operations from the APIs.

Our tool is built on one of the first versions of CSeq framework [INF+15] which allows the development of sequentialization following a modular approach. The output of MU-CSeq can, in principle, be processed by any analysis tool for sequential programs, but we primarily target BMC tools, in particular, CBMC [AKT13]. However, since the schema is very generic and the instrumentation for the different backends only differs in a few lines, backend integration is straightforward and not fundamentally limited to any underlying technology.

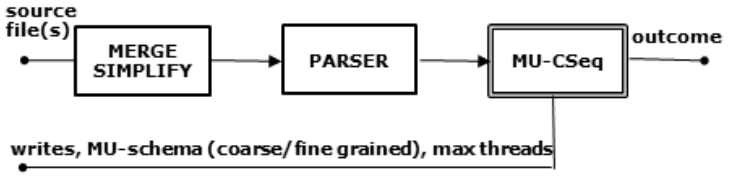

FIGURE 5.22: Architecture of the MU-CSeq tool. Double framed boxes denote modules composed by multiple submodules.

The complete configuration (see Figure 5.22) is defined by a sequence of 4 modules:

1. the source merging module that preprocesses the source files merging them into a single file (30 lines of code);
2. a parser module that builds the abstract syntax tree (AST), the symbol table, and other data structures (650 lines of code);
3. two modules that implement the sequentialization from [TIF+15b] and the different variants of the fine-and-coarse-grained MU based SMAs (4k lines of code); the sequentialized file is instrumented for the CBMC back-end which is invoked at the end of the transformation.

Our tool can be downloaded from the CSeq Web page[2]. More information about its usage is available in the README file in the installation package at the URL above.

Experimental set-up. We have evaluated MU-CSeq with CBMC (v4.9) as a backend on the benchmark set from the Concurrency category of the TACAS Software Verification Competition (SV-COMP15) [Bey15]. These are widespread benchmarks, and many state-of-the-art analysis tools have been trained on them; in addition, they offer a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms.

The whole benchmark set consists of 993 files, of which 786 have a reachable error location. The total number of lines of code of these 786 is 240k. Since we use a BMC tool as a backend, we only evaluate our approach on these unsafe files. We remark that in general

[2] http://users.ecs.soton.ac.uk/gp4/cseq/

TABLE 5.1: Performance comparison among different tools on the unsafe instances of the SV-COMP15 *Concurrency category*.

			CBMC 4.9			LR-CSeq 0.5a			Lazy-CSeq 0.5			MU-CSeq 0.3		
sub-category	files	l.o.c.	pass	fail	time	pass	fail	time	pass	fail	time	pass	fail	time
pthread	14	4381	12	2	33.4	6	8	76.5	11	3	134.2	14	0	19.9
pthread-atomic	2	202	2	0	0.3	2	0	3.0	2	0	6.1	2	0	2.0
pthread-ext	8	763	6	2	153.3	1	7	119.6	8	0	16.6	8	0	2.4
pthread-lit	3	117	2	1	166.8	2	1	166.9	3	0	7.0	3	0	2.7
pthread-wmm-mix	466	146448	466	0	57.2	466	0	8.9	466	0	6.3	466	0	21.8
pthread-wmm-podwr	16	4240	16	0	10.6	16	0	4.2	16	0	6.0	16	0	12.3
pthread-wmm-rfi	76	20981	76	0	9.3	76	0	4.1	76	0	6.2	76	0	12.7
pthread-wmm-safe	184	57391	184	0	27.3	184	0	6.6	184	0	6.2	184	0	18.6
pthread-wmm-thin	12	4008	12	0	9.6	12	0	6.2	12	0	6.1	12	0	29.3

BMC cannot prove correctness, but can only certify that an error is not reachable within the given bounds. MU-CSeq (v0.3) does not fully support dynamic memory allocation of shared memory[3], so five of the test cases are excluded here.

We have performed the experiments on an otherwise idle machine with a Xeon W3520 2.6GHz processor and 12GB of memory, running a Linux operating system with 64-bit kernel 3.0.6. We set a 10GB memory limit and a 500s timeout for the analysis of each test case.

The experiments are summarized in Table 6.1. Each row corresponds to a sub-category of the SV-COMP15 benchmarks, where we report the number of files and the total number of lines of code. The table reports the evaluation of CBMC [AKT13], LR-CSeq [FIP13a], Lazy-CSeq [ITF+14b, ITF+14a], and MU-CSeq on these benchmarks. For each tool and sub-category we consider the best parameters (i.e., minimal loop unrolling, number of rounds, etc.). Furthermore, we indicate with *pass* the number of correctly found bugs, with *miss* the number of unsuccessful analyses including tool crashes, backend crashes, memory limit hits, and timeouts, and with *time* the average time in seconds to find the bug.

For MU-CSeq the *fine-grained MU-based* SMA implementation has better performances on the first 4 sub-categories, except for one file. On those benchmarks a 24-memory unwinding allows to expose the bugs. In contrast, the *coarse-grained MU-based* SMA implementation is more efficient in the remaining sub-categories, where a memory unwinding with at least 90 writes is required for finding the bugs.

The table shows that MU-CSeq is competitive with other tools based on BMC, and in particular is the only approach that is able to find all bugs.

The MU-CSeq source code, static Linux binaries and benchmarks are available at http://users.ecs.soton.ac.uk/gp4/cseq/files/CSeq-MU-TACAS.tar.gz.

[3]Dynamic memory allocation of shared memory has been introduced in our prototype tool IMU-CSeq, which is described in the next Chapter.

5.6 Related work

Our approach has some similitudes with Lal and Reps [LR09]. The basic idea of the LR schemas is to simulate in the sequential program all round-robin schedules of the threads in the concurrent program, in such a way that (*i*) each thread is run to completion, and (*ii*) each simulated round works on its own copy of the shared global memory. The initial values of all memory copies are nondeterministically guessed in the beginning (*eager* exploration), while the context switch points are guessed during the simulation of each thread. At the end a checker prunes away all infeasible runs where the initial values guessed for one round do not match the values computed at the end of the previous round. This requires a second set of memory copies.

Similarly to LR, sequentialization by memory unwinding runs each thread only once and simulates it to completion; however, there are several differences. First, for our fine-grained MU implementation, the threads are not scheduled in a fixed ordering and in rounds. Instead, any scheduling that matches the memory unwinding is taken into account, including schedules with unboundedly many context switches (although one can show that a subset of them using a bounded number of context-switches suffices to expose the same bugs). Second, the consistency check to prune away unfeasible computations is interleaved with the simulation, thus many unfeasible runs can be found earlier and not only at the end of the simulation. This can improve the performance, in particular for BMC backends. Third, it is possible to show that the assertion violations that can be exposed by our sequentialization is equivalent to those that can be exposed with LR, with different parameter values though. For example, for our coarse-grained MU implementation, the exectutions that can be captured up to a size $\mathtt{W}$ in the memory unwinding can also be captured by LR with at most $2\,\mathtt{W} - 1$ rounds, and vice-versa all the computations of LR up to k context-switches (note that $k = r\,n - 1$ where n is the number of threads and r is the number of rounds) can be captured with at most k clusters.

MU can also be seen as a hybrid eager/lazy technique. It guesses the thread interactions at the beginning of the simulation, like the eager techniques in the Lal/Reps mould. However, it prunes away unfeasible computations incrementally, like Lazy-CSeq [ITF+14b, ITF+14a], but it calls the thread simulation function only once and runs it to completion, rather then repeatedly traversing it. Unlike the original lazy techniques [LMP09a], it also does not need to recompute the values of the local variables.

5.7 Conclusions

In this chapter, we have given implementations of different SMAs for the SC memory model. Our implementations have been optimized for an efficient analysis using BMC

tools.

We have presented a new approach to finding bugs in concurrent programs based on the concept of *memory unwinding*, i.e. a sequence of write operations of the shared memory that are exposed in the interaction between threads.

The reachability analysis used in our algorithm is bounded on the number of writes of the shared memory. This follows up on the observation of Lu et al. [LPSZ08], that only a few memory accesses are relevant to find concurrency bugs. It is an orthogonal bounding parameter with respect to the well-known bounded context-switching [QR05].

We have designed different strategies for the SMA implementations that vary on which write operations are exposed to the other threads, and have combined them with our eager sequentialization from [TIF+15b], obtaining a tool, called Mu-CSeq.

Mu-CSeq supports the full C-language, but the handling of dynamic memory allocation is still limited. We have evaluated our different strategies over the SV-COMP15 [Bey15] concurrency benchmarks, finding all the errors and achieving performance on par with those of the current best BMC tools with built-in concurrency handling as well as other sequentializations.

We have found that in general our fine-grained MU implementations work well for most problem categories. However, for the problems in the weak memory model category the size of the fine-grained unwindings becomes too big; here, coarse-grained MU works better.

The discussed verification algorithms can be extended to handle programs under weak memory model semantics by giving the corresponding shared memory abstractions. This can be done for TSO and PSO by explicitly adding the store buffers to $\mathcal{M}_{sc}^{mu}$. In the next chapter, we introduce a new implementation that refines the notion of MU and that works especially well for bounded model checking (BMC), and thus gives efficient BMC-implementations for bug finding under TSO and PSO program semantics.

Chapter 6

Individual Memory Location Unwindings

In this chapter, we describe *thread-asynchronous* SMA implementations for the SC, TSO, and PSO memory models that are based on the idea of *individual shared memory location unwinding*, which extends the concept of *memory unwinding* by splitting the whole sequence of write operations into different sequences, one for each individual shared memory location. We instantiate our approach by developing a new, efficient BMC-based bug finding tool for multi-threaded C programs under SC, TSO, or PSO based on these SMAs, and show experimentally that it is competitive with existing tools. The material used in this chapter is largely based on our paper [TNF+17].

In Section 6.1, we give a brief introduction and we describe the main differences between *memory unwinding* and *individual shared memory location unwinding*. In Section 6.2, we give IMU-based SMA transition systems corresponding to the SC, TSO and PSO memory models and show that they capture the semantics of the corresponding memory models. In Section 6.3, we describe concrete C-implementations of SMAs whose semantics is captured by the introduced transition systems. We conclude this chapter with Section 6.4, where we give an implementation of our approach by combining our IMU-based SMA implementations with MU-CSeq [TIF+15b] in a prototype tool *IMU-CSeq*; we show that our tool compares well with existing tools.

6.1 Introduction

In this chapter, we discuss an efficient implementation of thread-asynchronous shared memory abstractions for SC, TSO and PSO memory models. It builds on the idea of memory unwinding introduced in the previous chapter. The two main innovations are:

- the splitting of the memory unwinding into different sequences, one for each individual shared memory location (*location unwinding*, LU for short), and
- the introduction for each write of a *timestamp*, i.e., a natural number that denotes the time of occurrence of a write according to a discrete-time global clock.

An *individual memory-location unwinding* (IMU) is then a set containing exactly one LU for each memory location and such that the timestamps determine a total order among all the writes of all the LUs.

Splitting the memory unwinding into smaller sequences works well when used in combination with BMC bug finding tools: the read and write operations result in much smaller verification conditions; for each memory access, only the corresponding individual sequence needs to be duplicated and not the whole sequence of writes. Further, the shared memory abstraction capturing SC based on IMU can be easily extended to accommodate TSO and PSO. In fact, this can be done at the cost of adding for each write a second timestamp denoting the time at which the write is moved to the shared memory (and thus becomes visible to all threads).

6.2 IMU-based SMAs.

In this section, we give the IMU-based SMA transition systems corresponding to the SC, TSO and PSO memory models, denoted with $\mathcal{M}_{sc}^{imu}$, $\mathcal{M}_{tso}^{imu}$ and $\mathcal{M}_{pso}^{imu}$ respectively. We also show that these transition systems capture the semantics of the corresponding memory models.

IMU-based SMA for SC. An LU for a memory location v, denoted by v-LU, is a sequence of triples (t, val, d) where t and val denote the thread identifier and the value of the write and d is a positive integer denoting the time at which val is written into v according to a discrete global clock (*timestamp*). If Var is the set of location names and μ_v a v-LU for each $v \in Var$, an IMU is a set $\{\mu_v \mid v \in Var\}$ such that a) the tuples in each LU are ordered by increasing timestamps, and b) for each pair of different location names $v_1, v_2 \in Var$ and for each (t_i, val_i, d_i) in μ_{v_i} with $i = 1, 2$, then also $d_1 \neq d_2$. Note that timestamps define a total order among all the writes in the IMU.

The IMU-based shared memory abstraction for SC can be constructed similarly to $\mathcal{M}_{sc}^{mu}$. We only remark here the main differences: on the $\mathsf{init}()$-transition an IMU is guessed instead of an MU; the *current* timestamp (i.e., the timestamp of the last executed SMA operation) is maintained for each thread; in a read by a thread t, the position of the matching write is guessed such that the corresponding timestamp d is between the current timestamp of t and the timestamp of the next write by t (the current timestamp of t is

updated to d after the transition); the current timestamp of a thread t is also updated to the timestamp of a write when this is executed.

The total ordering of timestamps across all the IMU ensures the equivalence with a corresponding MU where the writes are written by increasing timestamps, and vice-versa (in an MU the timestamps are given implicitly by the order of the writes in the sequence). We thus get the following lemma for the resulting transition system $\mathcal{M}_{sc}^{imu}$.

Lemma 6.1. $L(\mathcal{M}_{sc}^{imu}) = L(\mathcal{M}_{sc}^{mu})$.

IMU-based SMA for TSO and PSO. To capture the TSO and PSO semantics, we introduce into the IMU a second timestamp for each write. In particular, we now make a distinction between the time a write occurs (*occurrence timestamp*) and the time the shared memory is updated with an occurred write (*update timestamp*). For correctness, we impose on the IMU that for each write the occurrence timestamp should not be greater than the update timestamp.

For TSO, in order to ensure the FIFO policy for the store buffers along any program execution, we also require that for each thread the writes must be following the same order, if ordered by non-decreasing timestamps according to either one of the sequences of timestamps (i.e., either the occurrence or the update timestamps). For PSO, instead this requirement is replaced with a weaker one that ensures a FIFO policy only for the writes of a same location performed by the same thread.

$\mathcal{M}_{tso}^{imu}$ can be obtained from $\mathcal{M}_{sc}^{imu}$ with a few changes: on the $\mathsf{init}()$-transition we now guess the IMU with occurrence and update timestamps as observed above; in a read of location v by a thread t the position of the matching write is the last occurred write still in the store buffer of t (i.e., current timestamp of t is between the occurrence timestamp and the update timestamp of the last write of v by t), if any, and the last updated write of v, otherwise (this case works as the read in $\mathcal{M}_{sc}^{imu}$); the current timestamp of a thread t is also updated to the occurrence timestamp of a write when this is executed; a $\mathsf{fence(t)}$-transition updates the current timestamp to the largest update timestamp of the already occurred writes performed by t. Obtaining $\mathcal{M}_{pso}^{imu}$ from $\mathcal{M}_{tso}^{imu}$ is very simple and the only difference is hidden in the properties that are required on the guessed IMU as observed above.

By the above observations we can derive that the described transition systems capture the semantics of the corresponding memory models. Moreover, as for the MU case, since all the writes are guessed in advance, the ordering in which we interleave the threads is again irrelevant. Thus, we get the following lemma:

Lemma 6.2. *For* $m \in \{tso, pso\}$, $L(\mathcal{M}_{m}^{imu}) = (L(\mathcal{M}_{m}))^{\#}$.

Verification by IMU. By composing the transformation of the control-flow from [TIF+15b] along with the SMA implementations $\mathcal{M}_{sc}^{imu}$, $\mathcal{M}_{tso}^{imu}$ and $\mathcal{M}_{pso}^{imu}$ we get new methods for the verification of multi-threaded programs under SC, TSO and PSO semantics, respectively. The correctness of such methods is a consequence of the lemmas given above in this section, and Theorems 3.1 and 3.2. We will give concrete implementations of these methods in the next section and evaluate them in Section 6.4.

6.3 IMU-based SMA implementations

In this section, we discuss concrete C-implementations of SMAs whose semantics is captured by $\mathcal{M}_{sc}^{imu}$, $\mathcal{M}_{tso}^{imu}$ and $\mathcal{M}_{pso}^{imu}$, respectively. Each of them implements the SMA API defined in Section 3.2. In the remainder of this chapter we will give details of the implemented code. Our code is optimized for an efficient analysis using BMC tools but implementations for other backends are possible.

6.3.1 IMU implementation for SC

The implementation is parameterized over several constants; note that `V=N+U` holds.

- `T` denotes the maximal number of threads that can be spawned during any execution of the input program,
- `V` denotes the number of tracked memory locations,
- `N` denotes the number of shared scalar variables (*locations with names*),
- `U` denotes the maximum number of locations that can be accessed only through their memory addresses (*locations without names*),
- `M` denotes the maximum number of dynamic memory allocations, and
- `W` denotes the maximum number of write operations for each location.

Data structures and invariants. We use several scalar variables and arrays to maintain the LUs and support the implementation of the SMA operations. All are declared global such that they are visible and can be modified in all the functions. For simplicity, we assume that all data is represented by unsigned integers.

The triples (t, val, d) of the LUs are maintained by three different arrays `thread`, `value` and `tstamp`. For every location `v` $\in [0,$ `V-1`$]$ and `i` $\in [0,$ `W-1`$]$, the triple at position `i` in the `v`-LU is stored in `thread`[`v`][`i`], `value`[`v`][`i`] and `tstamp`[`v`][`i`]. We link the writes of a same thread in each LU by an additional array `th_next_write`. All these arrays are

nondeterministically assigned in the function `init` and never changed in the program execution. Function `init` also ensures that:

- timestamps are assigned in increasing order for each LU;
- no two writes in the IMU are assigned the same timestamp;
- for every location `v` $\in [0, \texttt{V-1}]$, position `i` $\in [0, \texttt{W-1}]$ and thread identifier `t` $\in [0, \texttt{T-1}]$, `th_next_write`[v][i][t] is the first position in the v-LU after `i` that corresponds to a write by `t`, if any; otherwise, it is set to `W`, denoting that no further writes of `v` by `t` are expected;

To keep track of the execution of each thread in the IMU, we use the arrays `th_pos`, `last_write` and `cur_tstamp`, and maintain the following invariants for every location `v` $\in [0, \texttt{V-1}]$ and thread identifier `t` $\in [0, \texttt{T-1}]$:

- `th_pos`[v][t] stores the current position of thread `t` in the v-LU;
- `last_write`[v] stores the position `i` $\in [0, \texttt{W-1}]$ of the last executed write operation of location `v` in the v-LU;
- `cur_tstamp`[t] stores the current timestamp of thread `t` during its execution.

  Concerning to the management of threads, we keep some additional information. Variables `max_th` and `th_count` contain respectively the total number of threads that we assume should be created in the current program execution (a different value is guessed for each simulated execution) and the counting of the threads that have been actually created (this should match the guessed total number of threads in the end of computation). Also, for each thread `t`$\in$[0,`T-1`]:

  - `last_tstamp`[t] is the timestamp corresponding to the last write in the entire IMU by thread `t`; (this value is guessed nondeterministically in the initialization and is never changed; it should match `cur_tstamp`[t] in the end of a computation);
  - `ret`[t] is set to 1 to mean that `t` has been interrupted before reaching the end of its execution;
  - `terminated`[t] is set to 1 to mean that we expect that thread `t` will be stopped before the execution of its last statement (this value is nondeterministically guessed in the initialization and is never changed).

  To handle dynamic memory allocation and pointer arithmetics, for each location `v`$\in$[0,`V-1`] and for each `i`$\in$[0,`M-1`] we use:

  - `address`[v] to store the physical memory address of `v`;

– `mallocP[i]` to store the base address for each memory block that can be allocated dynamically;

– `mallocPallocated[i]` to track the dynamically allocated memory blocks.

IMU initialization. All the variables and arrays introduced above are declared global. On initializing the IMU we impose several constraints on them (see function `init()` in Figure 6.1).

```
void init(){
  bool ts_used[V*W] = [0];
  int v=0,w=0,t=0;

  th_count = 0;

  max_th = *;
  assume( max_th <= T );

  init_address(V);
  init_malloc(M);

  while (v<V) {
    last_write[v] = *;
    assume( last_write[v] < W );
    w=0;
    while (w<W){
      tstamp[v][w] = *;
      assume( (tstamp[v][w] < V*W) &&
        (!ts_used[tstamp[v][w]]) );
      ts_used[tstamp[v][w]]=1;
      if(w>0)
        assume(tstamp[v][w]>tstamp[v][w-1]);
      thread[v][w] = *;
      assume(thread[v][w] < max_th);
      w=w+1;
    }
    v=v+1;
  }
  while (t<T) {
    terminated[t] = *;
    last_tstamp[t] = *;
    assume (last_tstamp[t] < V*W);
    t=t+1;
  }
  v=0;
  while (v<V){
    t=0;
    while (t<T) {
      th_next_write[v][W-1][t] = W;
      t=t+1;
    }
    v=v+1;
  }

  v=0;
  while (v<V) {
    w=W-2;
    while (w>=0) {
      t=1;
      while (t<T) {
        if(thread[v][w+1] == t)
         th_next_write[v][w][t]=w+1;
        else
         th_next_write[v][w][t]=
             th_next_write[v][w+1][t];
        t=t+1;
      }
      w=w-1;
    }
    v=v+1;
  }
}
```

FIGURE 6.1: IMU initialization.

Function `init_address` ensures that array `address` is nondeterministically initialized with increasing values (i.e., `address`$[i]$ < `address`$[i+1]$ for $i \in [0, V-2]$). Function `init_malloc` ensures the same for array `mallocP` and additionally imposes that the address guessed for the last named location is less than the one assigned to the base location of the first memory allocation (i.e., `address[N-1]<mallocP[0]`). Functions `init_malloc()` and `init_address()` are illustrated in Figure 6.2.

In the first while-block of Figure 6.1, arrays `last_write`, `tstamp` and `thread` are nondeterministically assigned to legal values. Additionally, for each LU, timestamps are nondeterministically assigned in increasing order. The local array `ts_used` is used to ensure that different timestamps are assigned to each write in the IMU.

Legal values of `terminated` and `last_tstamp` are nondeterministically guessed in the second while-block. The rest of `init` initializes `th_next_write` such that for

each thread t and each location v, all the writes from t in the v-LU are linked in the proper order (value W is used as a sentinel to denote the end of each LU).

```
void init_address(){
  int i=0;
  while (i<V){
    address[i] = *;
    if(i>0)
      assume( address[i] > address[i-1]);
    i=i+1;
  }
}
```

```
void init_malloc(){
  int i=0;
  while (i<M){
    mallocPallocated[i]=0;
    mallocP[i] = *;
    if(i>0)
     assume( mallocP[i] > mallocP[i-1]);
    i=i+1;
  }
  assume( mallocP[0] > address[N-1]);
}
```

FIGURE 6.2: IMU: init_address and init_malloc functions for IMU implementation.

Auxiliary functions. We make use of two auxiliary functions illustrated in Figure 6.3.

```
bool is_th_terminated(uint t){
   if(ret[t]||nondet()) {ret[t]=1; return 1;}
   return 0;
}
uint Jump(uint t, uint v){
   uint jump=*;

   ts_jump = tstamp[v][jump];
   assume(   (jump <= last_write[v])
          && (jump <  th_next_write[v][t][th_pos[v]])
          && (tstamp[v][jump+1] > cur_tstamp[t]));
   cur_tstamp[t] = (ts_jump > cur_tstamp[t]) ? ts_jump : cur_tstamp[t];
   return jump;
}
```

FIGURE 6.3: IMU: Auxiliary functions for IMU implementation.

Function is_th_terminated returns 1, if ret[t] is already set to 1, and nondeterministically chooses either to set ret[t] to 1 and then return 1, or to return 0. The purpose of function Jump is to determine the position jump in the v-LU of the write that determines the current value contained in v. If the timestamp of the selected write is past the current thread timestamp, the last is updated to this value by acknowledging the fact that the corresponding write into the shared memory has occurred. The value of jump is selected nondeterministically within a range of proper values. Namely, jump should not pass the last legal write position for v and must be strictly less than the position of the next write of v by the same thread t (that has not occurred yet). Further, we require that the timestamp at position jump+1 is greater than the current timestamp of t (we wish to point to a write of v that is not superseded by an already occurred write).

Thread creation, termination, and join. The implementations of functions create, terminate and join are shown in Figure 6.4.

In function create, if the maximal number of allowed threads is reached, the procedure immediately returns -1 meaning that this thread will never be scheduled. Otherwise, the count of the created threads is incremented and the current

```
int create(void *f, uint pt){
  if(th_count >= max_th) return -1;
  uint v=0;
  if(pt == 0){
    while (v < V){
      th_pos[v][th_count]=0;
      v=v+1;
    }
    cur_tstamp[th_count]=0;
  }else{
    cur_tstamp[th_count]=cur_tstamp[pt];
    while (v < V){
      th_pos[v][th_count]=th_pos[v][pt];
      assume(th_next_write[v][0][th_count]
                    >=th_pos[v][th_count]);
      v=v+1;
    }
  }
  uint th_id=th_count++;
  return th_id;
}
```

```
void terminate(uint t){
  uint i, v=0;
  while (v < V){
    i=th_pos[v][t];
    assume( th_next_write[v][i][t]
                  > last_write[v] );
    v=v+1;
  }
  assume( ret[t]==terminated[t] &&
     last_tstamp[t]==cur_tstamp[t] );
}

void join(uint t1, uint t2){
  if(is_th_terminated(t1)) then return;
  uint v;
  assume( v < V );
  Jump(t1,v);
  assume( (terminated[t2] == 0) &&
   (cur_tstamp[t1]>=last_tstamp[t2]));
}
```

FIGURE 6.4: IMU: Functions `create`, `terminate` and `join`.

timestamp and LU positions of the new created thread are initialized such that: they coincide with those of the parent thread.

The assume statement ensures that no write operations are entitled to the new created thread before its creation. Since we update the positions of each thread in the LUs forward only, this will ensure also that each thread will not use any LU position corresponding to a write operation that is supposed to occur before its creation.

Function `terminate` checks that all write operations guessed for thread `t` have been done (while-loop). Furthermore, the concluding assume checks that the values guessed by function `init` for `terminated[t]` and `last_tstamp[t]` are consistent with the explored computation. We recall that `ret[t]` is initialized to 0 and can be nondeterministically set to 1 by the auxiliary function `is_th_terminated` (this has the effect of stopping the execution of the current thread).

Function `join` returns immediately if the execution of thread `t1` is stopped. Otherwise, the timestamp of `t1` is updated invoking `Jump` on a nondeterministically guessed variable (this ensures a choice of the new timestamp among all the LUs of `t1`). The computation is aborted whenever the other thread (`t2`) either will not terminate (i.e., `terminated[t2]==1`) or has not terminated yet at the current timestamp of `t1` (but it is supposed to terminate).

Read and write operations. The implementation of functions `read` and `write` are illustrated in Figure 6.5. For a read operation, the thread under simulation t first jumps forward into the `v`-LU corresponding to the variable given as parameter by invoking the auxiliary procedure `Jump` described above and then returns the valuation of the variable at the new position from matrix `value`.

```
int read(uint v,uint t){
  if(is_th_terminated(t)) return 0;
  uint jump = Jump(t,v);
  return (value[v][jump]);
}

void write(uint v,int val,uint t){
  if(is_th_terminated(t)) return;
  uint i, jump;
  i = th_pos[v][t];
  jump=th_next_write[v][i][t];
  assume( (jump<=last_write[v])
    && (value[v][jump] == val)
    && (tstamp[v][jump] > cur_tstamp[t])
  );
  th_pos[v][t]=jump;
  cur_tstamp[t]=tstamp[v][jump];
}
```

```
int ind_read(uint addr,uint t){
  if(is_th_terminated(t)) return 0;
  uint pos;
  assume(pos<V);
  assume(address(pos,t)==addr);
  return read(pos,t);
}


void ind_write(uint addr,int val,uint t){
  if(is_th_terminated(t)) return;
  uint pos;
  assume(pos<V);
  assume(address(pos,t)==addr);
  write(pos, val,t);
}
```

FIGURE 6.5: IMU: Read and write functions.

In a write operation, the thread first jumps to its next write operation for that variable and makes sure that the value coincides with that in the memory sequence at the new position. Furthermore, we also check that the timestamp associated to the new position is greater than the actual timestamp of t; this to prevent to simulate already simulated write operations. Then we update the current position of thread `t` in the v-LU and the current timestamp.

Address and malloc operations. Method `address` is used to recover the address of a given location $v \in [\mathtt{V} - 1]$. The implementation for this method is given in Figure 6.6. It returns the value from `address`[v].

```
int address (uint v, uint t){

   if(is_th_terminated(t)) return 0;
   return address[v];
}
```

```
int malloc(uint n, uint t){
   uint pos;

   if(is_th_terminated(t)) return 0;
   assume(pos<M);
   assume(!mallocPallocated[pos]);
   assume(mallocP[pos]+n < mallocP[pos+1]);
   mallocPallocated[pos]=1;
   return mallocP[pos];
}
```

FIGURE 6.6: IMU: Functions `address` and `malloc`.

The `malloc` procedure is implemented as shown in Figure 6.6. We first find a position *pos* that corresponds to a not sill allocated block, by checking the value of `mallocPallocated` at that position. In order to know if there is enough space we simply check that `mallocP`[`pos`]$+n <$ `mallocP`[`pos`+1]. Then we set `mallocP`[`pos`] to true to indicate that the address at position `pos` has been allocated. Finally, we return the base address corresponding to the position `pos`.

Ind_read and ind_write operations. When a read or write operation is performed using a memory address, i.e. `*p = 3` for a pointer variable `p`, we invoke. The implementation of `ind_read` and `ind_write` methods are illustrated in Figure 6.5.

We first search for the location corresponding to that whose address corresponds to the given parameter and then simulate the read/write operation at that location.

Lock and unlock mutex variables. A thread can take or release a lock on a shared mutex variable by calling the procedure `lock` and `unlock`, respectively; their implementations are provided by Figure 6.7. For a mutex variable, we assign value T when the lock is not acquired by any thread, and we assign value `t` if the mutex is held by thread `t`.

```
void lock(uint mut, uint t){
  write(mut, t, t);
  assume(ret[t] ||
    value[mut][th_pos[mut][t]-1]==T);
}
```

```
void unlock(uint mut, uint t){
  write(mut, T, t);
  assume(ret[t] ||
    value[mut][th_pos[mut][t]-1]==t);
}
```

FIGURE 6.7: IMU: Mutex `lock` and `unlock` operations.

For efficient implementation, we modify the value of variable `mut` using a write operation. For a `lock` operation we first write the value of `t` in `mut`; however, it may be the case that the mutex was already held by some thread. Thus, we check that the previous value of `mut` was T. The implementation of the method `unlock` procedure is similar, the only difference is that we write T in to the `mut` variable. Note that, two consecutive write operations of `mut` are performed by the same thread (`lock` and `unlock`). Furthermore, the value written at the even positions of the `mut`-LU are always T. These constrains can be added in the `init` function to reduce the number of runs to consider.

6.3.2 IMU implementation for TSO

We give this implementation incrementally on that given for SC.

To be consistent with the notation used in the implementation for SC, we use `tstamp`[`v`][`i`] to store the update timestamp concerning the $(\texttt{i}+1)^{th}$ write of location `v`, and `cur_tstamp` [`t`] to keep track of the current timestamp in the execution of thread `t` (i.e., the occurrence timestamp of the last occurred read or write). Additionally, we use two new arrays `btstamp` (buffer timestamps) and `ts_lastW` such that:

- `btstamp`[`v`][`i`] is the occurrence timestamp of the $(\texttt{i}+1)^{th}$ write of `v` (that is also the time at which it is stored in the local buffer of the thread that performs the write operation);
- `ts_lastW`[`t`] is the update timestamp of the last occurred write by thread `t`.

To implement the SMA API, we only need to give an implementation of `fence` and modify those given for SC of `init`, `read`, `write`, `lock` and `unlock`. The rest of the implementation is the same as for SC.

For `init`, we add to the implementation given for SC the following. We nondeterministically guess initial values for `btstamp`[`v`][`i`] and then impose that `btstamp`[`v`][`i`] $\leq$ `tstamp`[`v`][`i`] must hold (i.e., the update of the shared memory according to an occurred write may be delayed w.r.t. its occurrence time).

Note that here we slightly diverge from the transition system $\mathcal{M}_{tso}^{imu}$ described in Section 6.2. In fact, since we do not require any other condition on the guessed update timestamps, we can carry over an IMU with timestamps that may violate the FIFO policy on the store buffers. This is fixed by checking the proper ordering on matching the writes (we return on this when discussing the write implementation).

Function `lock` from Figure 6.7 is modified such that the write is done by a routine `Write_SC` that is exactly the write given for SC instead of the `write` for TSO. This ensures that lock acquisition is immediately visible to all the other threads. For function `unlock`, we do the same and further before returning we call `fence`. This way, we make immediately visible to all the other threads all the writes that occurred in the critical section.

The code of functions `fence`, `read` and `write` are illustrated in Figure 6.8.

```
int read(uint v,uint t){
  if(is_th_terminated(t)) return 0;

  uint ts_jump, i;
  i = th_pos[v][t];
  uint nxt_write=th_next_write[v][i][t];
  uint fst_write=th_next_write[v][0][t];
  assume (
    (ts_jump >= cur_tstamp[t]) &&
    (ts_jump < btstamp[v][nxt_write])
  );
  cur_tstamp[t]=ts_jump;
   if( (fst_write <= i)  &&
     (tstamp[v][i] > cur_tstamp[t])
    )
       return value[v][i];
    return Read_SC(v,t);
}
```

```
void fence(uint t){
  if(ts_lastW[t]>cur_tstamp[t])
    cur_tstamp[t] = ts_lastW[t];
}


void write(uint v,int val,uint t){
  if(is_th_terminated(t)) return;
  i = th_pos[v][t];
  jump=th_next_write[v][i][t];
  th_pos[v][t]=jump;
  assume(
    (btstamp[v][jump] > cur_tstamp[t])
    && (value[v][jump] == val) &&
    (tstamp[v][jump] > ts_lastW[t])
   ) ;
   ts_lastW[t] = tstamp[v][jump];
   cur_tstamp[t] = btstamp[v][jump];
}
```

FIGURE 6.8: IMU: Functions `read`, `fence` and `write` for TSO.

A memory *fence* flushes the store buffer of the thread executing it and thus we need to synchronize the current thread timestamp with its last update timestamp. Namely, if `ts_lastW`[`t`] is larger than the timestamp of the last occurred write by `t`, we assign `ts_lastW`[`t`] to `cur_tstamp`[`t`]. Note that if this is not the case then the local store buffer of `t` is certainly empty (recall `btstamp`[`v`][`i`] $\leq$ `tstamp`[`v`][`i`]).

Function `read` first updates nondeterministically the current timestamp of thread `t` such that it is not smaller than the current timestamp of `t` and is smaller than the update timestamp of the next write of `t`. Now, if at least a write of location `v` by `t` has occurred and the last write of `v` by `t` is still in the thread buffer, then we return the value of this write. Otherwise, a read from the shared memory is

performed by invoking the auxiliary function `Read_SC` that is exactly the function `read` from Figure 6.5.

Observe that the update of the current thread timestamp by `read` can cause this value to be larger than the update timestamp of the last write and this may be correct. To avoid that we wrongly move the time back, in `fence` we make the assignment only when this is not the case.

Function `write` first updates the current position in the v-LU of thread `t` to the next write provided that the time of occurrence of this write is larger than the current thread timestamp, the value of the write matches the guessed value for it and the update timestamp of the next write is larger than that of the last occurred write (the last one ensures that the thread store buffers are emptied according to a FIFO policy). Note that, in the case of a wrong guess of the update timestamps in `init`, this condition would not hold and thus the execution would abort. Before returning, the update timestamp of the last write and the current timestamp of thread `t` are modified consistently.

6.3.3 IMU implementation for PSO

We can give an implementation of SMA for PSO by slightly modifying the implementation given for TSO as follows.

We use a new array `max_tsW` in substitution of `ts_lastW` and change a few lines in the implementation of function `write`. Array `max_tsW` maintains for each thread `t` the maximum update timestamp among all the occurred writes of `t`.

In function `write` (Figure 6.8), we do not require any more that the update timestamp of the current write is larger than the update timestamp of the previous write by `t`. Recall that this was required in the TSO implementation in order to ensure that for each thread `t`, the guessed occurrence and update timestamps for the sequence of writes by `t` (that may be contained in different LU's) are indeed consistent with the FIFO policy of a store buffer; in PSO we only need to ensure that the FIFO policy holds for each of the maximal subsequences containing all the writes of a same location which is ensured by the remaining constraints and function `init`. Moreover, the update of `ts_lastW[t]` is replaced with the update of `max_tsW[t]` as follows:

```
max_tsW[t] =
  (tstamp[v][jump] > max_tsW[t]) ?  tstamp[v][jump] : max_tsW[t];
```

6.4 IMU-CSeq Implementation and Evaluation

Prototype implementation. We have implemented our approach, described in this Chapter, in a prototype tool called IMU-CSeq, where we first use the

sequentialization from [TIF+15b] to transform the original multi-threaded program into a sequential one (*sequentialization*), then link this against an IMU-based SMA implementation, and finally verify the resulting program with a BMC tool for sequential programs, in particular CBMC (v5.2).

IMU-CSeq is based on the open-source CSeq framework [Inv15, INF+15] (see Section 2.6.3 for an overview) which allows the development of sequentializations following a modular approach. Depending on the chosen SMA implementations we then obtain a tool for finding bugs in multi-threaded programs under SC, TSO, and PSO, respectively.

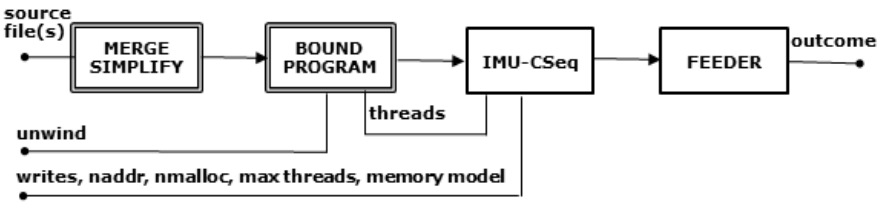

FIGURE 6.9: Architecture of the IMU tool. Double framed boxes denote modules composed by multiple submodules.

The complete configuration(see Figure 6.9) is defined by a sequence of 15 modules (14 `Translators` and 1 `Wrapper` module), which can be conceptually grouped according to the following categories:

1. the source merging module that preprocesses the source files merging them into a single file (built-in CSeq Framework), followed by eight simple translators, that rewrite the input program into a simplified syntax (built-in CSeq Framework);
2. four `Translator`s for program flattening to produce a bounded multi-threaded program (built-in CSeq Framework);
3. a module implementing the IMU sequentialization for SC, TSO and PSO memory models, as described in this chapter; the sequentialized file is instrumented for the CBMC back-end (a new module, with about 3k lines of code);
4. a `Wrapper` for backend invocation and user report generation (built-in CSeq Framework).

Our tool can be downloaded from this link[1]. More information about its usage is available in the README file in the installation package at the URL above.

Experimental set-up. *SC benchmarks.* We have evaluated this prototype on benchmarks from the Concurrency-category of the TACAS Software Verification Competition (SV-COMP16) [Bey16]. These are widespread benchmarks, and many

[1] http://users.ecs.soton.ac.uk/gp4/cseq/files/IMU-2017.zip

state-of-the-art analysis tools have been trained on them; in addition, they offer a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms.

The whole benchmark set consists of 1015 files, of which 791 have a reachable error location. Since we use a BMC tool as a backend, we cannot prove correctness, but can only show that an error is not reachable within the given bounds. We therefore only evaluate our prototype on such unsafe files. In particular, we used the files from the sub-categories shown in Table 6.1.

TABLE 6.1: Performance comparison among different tools for SC semantics on unsafe instances from the SV-COMP16 *Concurrency category*.

			CBMC svc16			CIVL svc16			Lazy-CSeq svc16			MU-CSeq svc15			IMU-CSeq		
sub-category	files	l.o.c.	pass	fail	time	pass	fail	time	pass	fail	time	pass	fail	time	pass	fail	time
pthread	15	2301	14	1	84.23	15	0	33.31	15	0	48.58	15	0	5.42	15	0	**4.88**
pthread-atomic	2	156	2	0	0.59	2	0	17.5	2	0	**1.39**	2	0	1.4	2	0	3.15
pthread-ext	8	616	7	1	154	8	0	13.12	8	0	11.23	8	0	5.45	8	0	**4.88**
pthread-lit	2	73	2	0	0.3	2	0	10.33	2	0	**0.56**	2	0	2.55	2	0	0.88
ldv-races	8	616	3	5	66.96	3	0	14.5	8	0	1.73	-	-	-	8	0	**1.61**

The experiments were run on a dedicated machine with a Xeon E5-2650 v2 with 2.60 GHz and 132 of RAM, running a Linux 4.2.0-22-generic operating system. The verifiers were given a a 15GB memory limit and a 900s timeout. The files are analyzed under SC semantics. The experiments are summarized in Table 6.1. Each row corresponds to a sub-category of the SV-COMP16 benchmarks, where we report the number of files and the total number of lines of code. Note that the different *pthread-wmm-** sub-categories are missing. Our current prototype cannot currently handle these benchmarks, which have a large number of shared variables and write operations. However, the original MU-approach had similar problems which we could overcome by exposing only a subset of the write operations (*coarse-grained unwinding*, see [TNI+16b]), and we are currently exploring similar ideas for the IMU-approach. The table shows the results for CBMC [AKT13], CIVL [ZRL+15], Lazy-CSeq [ITF+14b, ITF+14a], MU-CSeq [TIF+15b],[2] and IMU-CSeq on these benchmarks. Furthermore, we indicate with *pass* the number of correctly found bugs, with *fail* the number of unsuccessful analyses including tool crashes, memory limit hits, and timeouts, and with *time* the average time in seconds to find the bug. The results clearly show that our approach is competitive with existing tools; in particular, the IMU-based SMA-implementation improves over the MU-based MU-CSeq.

We have implemented a hybrid prototype tool combining IMU-CSeq and MU-CSeq and we won the gold medal at SV-COMP16 in the Concurrency-category [Bey16].

WMM benchmarks. We also compared our prototype against two tools with built-in support for analysis under weak memory models, CBMC [HK15], and

[2]Note that this refers to the SV-COMP15 version and results, which did not include the *ldv-races* sub-category. For SV-COMP16 we submitted (under the label MU-CSeq) a hybrid tool that uses IMU for the shown sub-categories, and the unchanged MU-CSeq for the *pthread-wmm-** sub-categories.

TABLE 6.2: Analysis runtime under TSO/PSO

		parameters					TSO runtime (s)					PSO runtime (s)				
	l.o.c.	unwind	writes (W)	naddr (U)	nmalloc (M)	bitwidth	bug?	files	IMU-CSeq	CBMC	NIDHUGG	bug?	files	IMU-CSeq	CBMC	NIDHUGG
dekker	52	1	3	0	0	5	•	1	0.76	0.29	**0.04**	•	1	0.76	0.25	**0.05**
lamport	78	1	3	0	0	5	•	1	0.97	0.31	**0.05**	•	1	0.97	0.29	**0.05**
peterson	40	1	3	0	0	5	•	1	0.67	0.26	**0.04**	•	1	0.68	0.53	**0.04**
szymanski	57	1	4	0	0	5	•	1	0.84	0.34	**0.07**	•	1	0.84	0.32	**0.04**
fib_longer_unsafe	30	6	7	0	0	10	•	1	**2.10**	8.19	94.84	•	1	2.50	**1.69**	135.45
fib_longer_safe	30	6	7	0	0	10		1	**4.75**	22.5	t.o.		1	**3.90**	31.8	t.o.
parker	110	1	5	0	0	5	•	1	1.22	0.31	**0.05**	•	1	1.21	0.28	**0.05**
stack_unsafe	110	2	4	1	2	5	•	1	1.46	0.41	**0.05**	•	1	1.44	0.35	**0.05**
litmus_safe (avg)	297K	1	6	1	0	10		5526	1.20	**0.17**	2.35		4835	1.06	**0.15**	6.65
litmus_unsafe (avg)		1	6	1	0	10	•	277	1.67	**0.16**	3.86	•	968	1.28	**0.12**	1.58

Nidhugg [AAA+15], a bug-finding tool that combines stateless model checking with dynamic partial order reduction on relaxed memory executions. These experiments were run on a dedicated machine with a Xeon W3520 2.6GHz processor and 12GB of physical memory running 64-bit linux 3.0.6. We set a 10GB memory limit and a 600s timeout for the analysis of each of the simple benchmarks and timeout of 14,400s for safestack. For each tool and benchmark, we set the parameters to the minimum value needed to expose the error.

Simple benchmarks. We first used a set of (relatively simple) benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite. The results are summarized in Table 6.2. The unwind parameter was used by all the three tools considered in the comparison. The parameters `W`, `U`, and `M` are used by IMU-CSeq, with the meaning as given in Section 6.3.1. The parameter bitwidth gives the size of integers (in bits) used in the sequential analysis.

The first block contains results for some classical mutual exclusions algorithms (`dekker`, `lamport`, `peterson`, `szymanski`). The implementations are correct under SC but not under TSO and PSO. All tools find the errors, but because of their small size[3], Nidhugg outperforms both our prototype and CBMC on these programs.

The second block contains the safe and unsafe versions of one of the fibonacci-benchmarks, in which two worker threads concurrently increase two shared counters, and a main thread checks whether any of the two counters can reach a defined value. A full exploration of the thread interleavings is required to identify the error (or show its absence) in this program. Techniques such as partial-order reduction do not apply, and several tools struggle to analyze it. Here, Nidhugg is generally slower than both CBMC and our prototype tool, and fails to terminate on the safe version.

The next block contains two benchmarks that originate from industrial code: `parker` models a semaphore-like synchronization class that breaks under TSO [AAA+15],

[3]On small programs the sequentialization overhead, which is for all intents and purposes a constant effort for our prototype, is high compared to the verification time. Hence, the other tools are faster on the small programs.

and `stack` which was taken from SV-COMP [Bey16]. Here, all tools report the expected results; the performance differences between Nidhugg and CBMC are small, while the performance of our tool could be improved with a better implementation, as it currently transforms each file nearly 20 times, each time requiring parsing and unparsing.

The last two lines show the average values over 5803 litmus tests for WMMs. For TSO, both our tool and CBMC successfully identified the 277 test cases containing a reachable error, while Nidhugg failed to find one of them. For PSO, CBMC claims that there are 971 unsafe instances while Nidhugg and IMU-CSeq both find only 968 unsafe ones. We have manually inspected the counter-examples produced by CBMC (v5.2). We confirm that CBMC produces spurious counter-examples on those benchmarks.

Safestack. We have conducted further experiments on a real world benchmark, Safestack [CJZ+13], which is a lock-free stack implementation designed for weak-memory models. Safestack is written in C++ but we manually translated it into C, providing simulation functions for the C11 atomic functions used in the test, and have conducted the experiments with this version. It contains a rare bug that is hard to find with automatic bug-finding techniques already under SC (including random testing, Nidhugg, CIVL [ZRL+15], and other approaches based on BMC) [TDB14]. The only tool we are aware of that can automatically find a genuine counter-example is Lazy-CSeq [ITF+14b, ITF+14a]. It requires a minimum of 3 loop unwindings and 4 rounds of computation to expose a bug. This actually shows that the error is quite deep which explains why other approaches based on explicit handling of interleaving fail. Both Nidhugg and CBMC failed to find the error with the given timeout but IMU-CSeq was able to find it also for a TSO- and PSO-semantics, respectively. IMU-CSeq required approx. 3.5 minutes and 1.5GB of memory to find the error under TSO, and approx. 17 minutes and 1.8GB of memory to find the error under PSO.

6.5 Conclusions

In this chapter, we have described efficient implementations SMAs for SC, TSO, and PSO that are based on the idea of individual memory-location unwindings. These implementations have allowed us to instantiate our approach into an efficient BMC-based bug-finding tool, and we have shown experimentally that it compares well with existing tools. We have also implemented a hybrid prototype tool combining IMU-CSeq and MU-CSeq that has won the gold medal at SV-COMP16 in the Concurrency-category [Bey16].

Chapter 7

Conclusions

7.1 Summary of work

The main goal of this thesis was to extend existing successful techniques that have been implemented for SC to more general WMMs. We have described a general approach that allows to combine different verification techniques with different memory models in the style of a plug-and-play architecture.

The main idea of our approach is to introduce an abstraction that allows us to separate the computation and the communication concerns of concurrent programs, without loosing the efficiency of existing approaches. For this, we have introduced an abstract data type, *called shared memory abstraction* (SMA), which captures the standard concurrency operations in multi-threaded programs, such as shared memory reads, writes, and allocations, thread creation and termination, and synchronization operations such as thread join and mutex locking and unlocking. All concurrent operations are executed by invoking the corresponding SMA operations. Such SMA has provided a separation of concerns between the shared memory and the control-flow related aspects of concurrent programs, such that the verifier design can be focused on each of these aspects in isolation.

We have then proposed novel SMA implementations for TSO and PSO that are carefully optimized for lazy sequentialization on top of bounded model checking. We have implemented this approach in our *Lazy-CSeq* tool, that implements our Lazy Sequentialization for SC [ITF+14a]. We have evaluated our tool on a set of benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite, and also on a real world instance. Results are in line with those of the current best tools for concurrency handling under weak memory models on simple benchmarks, but outperforms them on more complex problems.

Although this approach has been successful in combination with Lazy-CSeq, it may be the case that by extending verification methods to other memory models by simply replacing the SMA implementation might not result in scalable verification

tools. Some approaches from the literature [LMP10, LR09, MC81, Jon83] explore the program executions by rearranging the order in which the memory operations of the different threads are executed.

We have introduced the concepts of *thread-asynchronous transition systems* and *thread-wise equivalent* programs. We have then shown that, when the underlying SMA implementation is *thread-asynchronous*, intra-thread analysis can be optimized by rearranging the operations of the threads. This allows us to freely transform the threads as long as we stay within the class of *thread-wise equivalent* programs. We have also discussed how previous successful approaches implemented for SC memory model from the literature fit into our setting, yielding correctness proofs for free, and then extended them to more realistic memory models.

We have also developed *thread-asynchronous* shared memory abstraction implementations for SC, TSO and PSO memory models, based on the idea of *memory unwinding*, i.e. an explicit representation of the sequence of write operations into the shared memory, and *individual shared memory location unwinding*, that extends the concept of MU. For the verification, we nondeterministically guess this unwinding and then explore all the executions of the program that respect this guess; each thread is simulated in isolation in according with the initially guessed memory unwinding. This approach follows up on the observation of Lu et al. [LPSZ08], that only a few memory accesses are relevant to find concurrency bugs. It is complementary to other approaches [FIP13a, LR09, LMP10] and explores an orthogonal dimension, i.e., the number of write operations of the shared memory.

We have first implemented different SMAs for SC in our prototype tool *MU-CSeq* and combined them with our eager sequentialization [TIF+15b]. *MU-CSeq* [TIF+14, ITF+14a] has won two silver medals at the Software Verification Competition SV-COMP in the concurrency category.

Finally, we have extended the idea of *memory unwinding* to *individual shared memory location unwinding* and implemented *thread-asynchronous* SMA implementations for the SC, TSO and PSO memory models that are based on this new concept.

We have combined our IMU-based *thread-asynchronous* SMAs with our sequentialization [TIF+15b] in the protoype tool *IMU-CSeq* and empirically shown, that this approach is very effective in finding bugs also under TSO and PSO on the same set of benchmarks we have used to evaluate *LazySMA*.

We have also implemented a tool that combines IMU-CSeq and MU-CSeq [TNI+16b] that has won the gold medal in the Software Verification Competition SV-COMP2016.

7.2 Future work

Rare bugs require analysis approaches that handle interleavings symbolically. We have empirically demonstrated, using the SafeStack benchmark, that we can indeed find deep bugs, in contrast with other mature bug-finding techniques. We are currently looking at harder benchmarks, and in particular, in the domain of concurrent data structures. SafeStack is part of this effort.

In Chapter 4 and 6, we have described two different SMAs for TSO memory model, but we have also shown that a simple extension lets us handle the PSO memory model. We believe that our approach can also be extended to further weak memory models, and leave this for future work.

Although, our approach has been targeted for bounded model-checking, sequentializations can be used with any backend analysis tool. To improve scalability, we are currently investigating a new approach that combines abstract interpretation and bug-finding, similar to [NFLP17, NIF+17], which is built on the top of the code-to-code translation presented in this thesis.

The combination of IMU-CSeq with a sequential symbolic execution backend such as KLEE [CDE08] immediately gives us a concolic testing tool for concurrent programs.

Recently, a source-to-source translation has been used as a method to parallelize Lazy-CSeq [NSF+17], i.e., a parametrizable translation that generates a set of simpler program instances, each capturing a reduced set of the original programs interleavings. These instances can then be checked independently in parallel. This can reduce the wall-clock run times for difficult verification problems by orders of magnitude. We currently investigate how to embed this technique into the approaches proposed in this thesis.

We are also investigating other concurrency models like message passing, which should fit into our framework as well; here, the message buffers play the same role as the storage buffers in TSO.

Bibliography

[AAA+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In Baier and Tinelli [BT15], pages 353–367.

[ABP11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting rid of store-buffers in TSO analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2011.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[ÁH14] Erika Ábrahám and Klaus Havelund, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*. Springer, 2014.

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 512–532. Springer, 2013.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*

- *25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[AM14] Tatsuya Abe and Toshiyuki Maeda. A general model checking framework for various memory consistency models. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 332–341. IEEE Computer Society, 2014.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In Ferrante and McKinley [FM07], pages 12–21.

[BCDM15] Ahmed Bouajjani, Georgel Calin, Egor Derevenetc, and Roland Meyer. Lazy TSO reachability. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9033 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2015.

[BE12] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. In Cormac Flanagan and Barbara Knig, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2012.

[Bey15] Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In Baier and Tinelli [BT15], pages 401–416.

[Bey16] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In Chechik and Raskin [CR16], pages 887–904.

[Bie09] Armin Biere. Bounded model checking. In *Handbook of Satisfiability*, pages 457–481. 2009.

[BM09] Ahmed Bouajjani and Oded Maler, editors. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *LNCS*. Springer, 2009.

[Boe05] Hans-Juergen Boehm. Threads cannot be implemented as a library. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and*

Implementation, Chicago, IL, USA, June 12-15, 2005, pages 261–268. ACM, 2005.

[BR] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. *SIGPLAN Not*, page 2002.

[BT15] Christel Baier and Cesare Tinelli, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*. Springer, 2015.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[CF11] Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *ICSE*, pages 331–340, 2011.

[CGBD12] David R. Cok, Alberto Griggio, Roberto Bruttomesso, and Morgan Deters. The 2012 smt competition. In *SMT@IJCAR*, pages 131–142, 2012.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CGS11] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Time-bounded Analysis of Real-time Systems. In *FMCAD*, pages 72–80, 2011.

[CGW15] Myra B. Cohen, Lars Grunske, and Michael Whalen, editors. *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE, 2015.

[CJZ+13] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Trans. Dependable Sec. Comput.*, 10(6):368–379, 2013.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of*

Computing, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CR16] Marsha Chechik and Jean-François Raskin, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*. Springer, 2016.

[FIP13a] Bernd Fischer, Omar Inverso, and Gennaro Parlato. CSeq: A Concurrency Pre-processor for Sequential C Verification Tools. In *ASE*, pages 710–713. IEEE, 2013.

[FIP13b] Bernd Fischer, Omar Inverso, and Gennaro Parlato. CSeq: A sequentialization tool for c - (competition contribution). In *TACAS*, pages 616–618, 2013.

[FM07] Jeanne Ferrante and Kathryn S. McKinley, editors. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 2007.

[GHR10] Naghmeh Ghafari, Alan J. Hu, and Zvonimir Rakamaric. Context-Bounded Translations for Concurrent Software: An Empirical Evaluation. In Jaco van de Pol and Michael Weber, editors, *SPIN*, volume 6349 of *LNCS*, pages 227–244. Springer, 2010.

[HK15] Alex Horn and Daniel Kroening. On partial order semantics for sat/smt-based symbolic encodings of weak memory concurrency. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.

[INF+15] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded c-programs. In Cohen et al. [CGW15], pages 807–812.

[Inv15] Omar Inverso. Bounded model checking of multi-threaded programs via sequentialization. November 2015.

[ISO09] ISO/IEC. *Information technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, ISO/IEC/IEEE 9945:2009.* 2009.

[ISO11] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C.* International Organization for Standardization, Geneva, Switzerland, December 2011.

[ITF+14a] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2014.

[ITF+14b] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-CSeq: A lazy sequentialization tool for C - (competition contribution). In Ábrahám and Havelund [ÁH14], pages 398–401.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1975.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[KSMS11] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing*, SAT'11, pages 343–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[KT14] Daniel Kroening and Michael Tautschnig. Automating software analysis at large scale. In Petr Hlinený, Zdenek Dvorak, Jirí Jaros, Jan Kofron, Jan Korenek, Petr Matula, and Karel Pala, editors, *Mathematical and Engineering Methods in Computer Science - 9th International Doctoral Workshop, MEMICS 2014, Telč, Czech Republic, October 17-19, 2014, Revised Selected Papers*, volume 8934 of *Lecture Notes in Computer Science*, pages 30–39. Springer, 2014.

[Lam79] Leslie Lamport. On the proof of correctness of a calendar program. *Commun. ACM*, 22(10):554–556, 1979.

[LMP09a] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In Bouajjani and Maler [BM09], pages 477–492.

[LMP09b] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 211–222. ACM, 2009.

[LMP10] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

[LMP12] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Sequentializing Parameterized Programs. In Sebastian S. Bauer and Jean-Baptiste Raclet, editors, *FIT*, volume 87 of *EPTCS*, pages 34–47, 2012.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, March 2008.

[LQR09] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In Bouajjani and Maler [BM09], pages 509–524.

[LR09] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[LS90] M. E. Lesk and E. Schmidt. Unix vol. ii. chapter Lex&Mdash;a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[MA15] Adam Morrison and Yehuda Afek. Temporally bounding TSO for fence-free asymmetric synchronization. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 45–58. ACM, 2015.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Ferrante and McKinley [FM07], pages 446–455.

[MQB07] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software, 2007.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[Mül06] Markus Müller-Olm. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*, volume 3800 of *Lecture Notes in Computer Science*. Springer, 2006.

[NFLP15] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Unbounded Lazy-CSeq: A lazy sequentialization tool for C programs with unbounded context switches - (competition contribution). In Baier and Tinelli [BT15], pages 461–463.

[NFLP16] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 174–191, 2016.

[NFLP17] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Concurrent program verification with lazy sequentialization and interval analysis. In Amr El Abbadi and Benoît Garbinato, editors, *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, volume 10299 of *Lecture Notes in Computer Science*, pages 255–271, 2017.

[NIF+17] Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq 2.0: Combining lazy sequentialization with abstract interpretation - (competition contribution). In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April*

22-29, 2017, Proceedings, Part II, volume 10206 of *Lecture Notes in Computer Science*, pages 375–379, 2017.

[NSF+17] Truc L. Nguyen, Peter Schrammel, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Parallel bug-finding in concurrent programs via reduced interleaving instances. In Massimiliano Di Penta and Tien N. Nguyen, editors, *32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana-Champaign, IL, USA, October 30 - November 3, 2017*, pages 753–764. IEEE Computer Society, 2017.

[Qad11] Shaz Qadeer. Poirot - a concurrency sleuth. In *ICFEM*, page 15, 2011.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[QW04] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, March 2000.

[SKB+15] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Successful use of incremental BMC in the automotive industry. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, volume 9128 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2015.

[SSO+10a] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[SSO+10b] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, pages 108–125, 2000.

[SW11] Nishant Sinha and Chao Wang. On interference abstractions. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 423–434. ACM, 2011.

[TDB14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: an empirical study. In José Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 15–28. ACM, 2014.

[TIF+14] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. MU-CSeq: Sequentialization of C Programs by Shared Memory Unwindings - (Competition Contribution). In Ábrahám and Havelund [ÁH14], pages 402–404.

[TIF+15a] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. MU-CSeq 0.3: Sequentialization by Read-Implicit and Coarse-Grained Memory Unwindings - (Competition Contribution). In Baier and Tinelli [BT15], pages 436–438.

[TIF+15b] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Verifying concurrent programs by memory unwinding. In Baier and Tinelli [BT15], pages 551–565.

[TNF+17] Ermenegildo Tomasco, Truc Lam Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Using shared memory abstractions to design eager sequentializations for weak memory models. In Alessandro Cimatti and Marjan Sirjani, editors, *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*, volume 10469 of *Lecture Notes in Computer Science*, pages 185–202. Springer, 2017.

[TNI+16a] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for TSO and PSO via shared memory abstractions. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 193–200. IEEE, 2016.

[TNI+16b] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. MU-CSeq 0.4: Individual Memory Location Unwindings - (Competition Contribution). In Chechik and Raskin [CR16], pages 938–941.

[Vyu10] Dmitry Vyukov. Bug with a context switch bound 5, 2010.

[WT15] Heike Wehrheim and Oleg Travkin. TSO to SC via symbolic execution. In Nir Piterman, editor, *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, volume 9434 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2015.

[ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 250–259. ACM, 2015.

[ZRL+15] Manchun Zheng, Michael S. Rogers, Ziqing Luo, Matthew B. Dwyer, and Stephen F. Siegel. CIVL: formal verification of parallel programs. In Cohen et al. [CGW15], pages 830–835.