Verification of Transactional Memories and Recursive Programs

by

Ariel Cohen

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

September 2008

Amir Pnueli and Lenore Zuck

To Sharon

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisers, Amir Pnueli and Lenore Zuck, for introducing me to the wonderful world of verification, and for their continuous guidance and support ever since.

Many thanks to Kedar Namjoshi, for giving me the opportunity to experience research at Bell-Labs and for many enjoyable and fruitful discussions. I would like to thank the other members of my thesis committee – Clark Barrett, Benjamin Goldberg and John O'Leary – for their constructive comments and valuable suggestions. Many thanks to Ittai Balaban, Ron van der Meyden and Mark Tuttle for interesting and exciting discussions while collaborating on various research topics.

Above all I would like to thank my family. To my parents for encouraging me to pursue a Ph.D and giving me their support along the way. To my son Yonatan for making me smile, especially during the days when the theorem prover did not seem to appreciate my proofs. To my wife Sharon – without your love and support none of this would have been possible.

ABSTRACT

Transactional memory is a programming abstraction intended to simplify the synchronization of conflicting concurrent memory accesses without the difficulties associated with locks. In the first part of this thesis we provide a framework and tools that allow to formally verify that a transactional memory implementation satisfies its specification. First we show how to specify transactional memory in terms of admissible interchanges of transaction operations, and give proof rules for showing that an implementation satisfies its specification. We illustrate how to verify correctness, first using a model checker for bounded instantiations, and subsequently by using a theorem prover, thus eliminating all bounds. We provide a mechanical proof of the soundness of the verification method, as well as mechanical proofs for several implementations from the literature, including one that supports non-transactional memory accesses.

Procedural programs with unbounded recursion present a challenge to symbolic model-checkers since they ostensibly require the checker to model an unbounded call stack. In the second part of this thesis we present a method for model-checking safety and liveness properties over procedural programs. Our method performs by first augmenting a concrete procedural program with a well founded ranking function, and then abstracting the augmented program by a finitary state abstraction. Using procedure summarization the procedural abstract program is then reduced to a finite-state system, which is model checked for the property.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

In this thesis we focus on two topics in system verification: verification of transactional memories, and verification of recursive programs.

Multicore is becoming the mainstream architecture for microprocessor chips and it requires developers to produce concurrent programs in order to gain full advantage of the multiple number of processors. Concurrent programs, however, are hard to write since they require careful coordination between threads that access the same memory locations. Conventional lock-based solutions are difficult to compose, and when applied incorrectly may introduce various faulty situations. *Transactional memory* [HM93b] simplifies parallel programming by transferring the burden of concurrency management from the programmers to the system designers, and enables safe composition of scalable applications.

A *transaction* is a sequence of memory access operations that appears to be *atomic*: the operations either all complete successfully (and the transaction commits), or none completes (and the transaction aborts). Transactions run in *isolation* – each transaction executes as if running alone on the system, without any interleaving with other transactions. A *conflict* occurs when two transactions access the same memory location and at least one writes to it. A conflict is resolved by aborting at least one of the conflicting transactions. Implementations of transactional memories are often parameterized by their properties. These may include the conflicts they are to avoid, when conflicts are de-

tected, how they are resolved, when the memory is updated, whether transactions can be nested, etc. Larus and Rajwar [LR07] survey nearly 40 implementations of transactional memory in their comprehensive book on the subject.

In the first part of this thesis, we define an abstract model for specifying transactional memory, represented by a fair state machine that is parameterized by a set of *admissible interchanges* — a set of rules specifying when a pair of consecutive operations in a sequence of transactional operations can be swapped without introducing or removing a conflict. We provide proof rules for verifying that an implementation satisfies a transactional memory specification. We demonstrate the method by first modeling a small instantiation of a well-known transactional memory implementation in TLA$^+$ [Lam02] and proving its correctness with the model checker TLC [Lam02]. Subsequently we construct a framework that allows for a mechanical formal verification using the PVS-based theorem prover TLPVS [PA03]. We illustrate how to apply the framework by presenting TLPVS proofs of three implementations from the literature. Finally, we extend the model and the framework to allow the verification of transactional memory implementations that support *non-transactional* memory accesses.

Procedural programs with unbounded recursion present a challenge to symbolic model-checkers since they ostensibly require the checker to model an unbounded call stack. In the second part of this thesis, we explore the integration of ranking abstraction [KP00b, BPZ05], finitary state abstraction, procedure summarization [SP81], and model-checking into a combined method for the automatic verification of linear temporal logic (LTL) properties of infinite-state recursive procedural programs. While most components of

the proposed method have been studied before, we reduce the verification problem to that of symbolic model-checking. Furthermore, the method allows for application of ranking and state abstractions while still relegating all summarization computation to the model-checker.

The rest of this thesis is organized as follows: Chapter 1 provides a general background, describing fair discrete systems (FDS), linear temporal logic and temporal testers. Chapter 2 begins with an overview of the conventional methods for concurrency control and of transactional memory. It then provides preliminary definitions related to transactions, and presents the concept of admissible interchanges and a specification model of a transactional memory. It subsequently gives proof rules for verifying that implementations satisfy their specifications, and illustrates how to apply these rules using a model checker and a theorem prover. The model is then extended to support non-transactional memory accesses. Finally, related work, conclusions and future work are discussed. Chapter 3 begins with an overview of predicate abstraction, ranking abstraction and programs, which are presented as transition graphs. It then illustrates a method for verifying termination of procedural programs by computing abstractions and summarization, constructing a procedure-free FDS, and finally, model-checking. It also provides a method for model-checking general LTL properties of recursive procedural programs. Finally, it discusses related work and conclusions.

1

BACKGROUND

This section begins by introducing the computational model that is used in this work, *fair discrete systems* (FDS), which enables the representation of fairness constraints arising from both the system and the property. Next we present the specification language for reactive systems *linear temporal logic* (LTL). Finally, to allow model checking LTL formulas, we explain a method to construct temporal testers that is compositional in the structure of the formula.

1.1 Fair Discrete Systems

The computation model, *fair discrete systems* (FDS) $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ [KP00a], consists of the following components:

- V – A set of *state variables* over possibly infinite domains. A *state* of $\mathcal{D}$ provides a type-consistent interpretation of the variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of all states over V.

- Θ – The *initial condition*: An assertion characterizing all the initial states of the FDS. A state is called *initial* if it satisfies Θ.

- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to

its $\mathcal{D}$-successor $s' \in \Sigma$ by referring to both unprimed and primed versions of the state variables. We assume that every state has a $\mathcal{D}$-successor.

- $\mathcal{J} = \{J_1, \ldots, J_k\}$: A set of assertions expressing the *justice* (*weak fairness*) requirements. The justice requirement $J \in \mathcal{J}$ stipulates that every computation contains infinitely many J-states (states satisfying J).

- $\mathcal{C} = \{\langle p_1, q_1\rangle, \ldots \langle p_n, q_n\rangle\}$: A set of assertions expressing the *compassion* (*strong fairness*) requirements. The compassion requirement $\langle p, q\rangle \in \mathcal{C}$ stipulates that every computation containing infinitely many p-states also contains infinitely many q-states.

For an assertion ψ, we say that a state $s \in \Sigma$ is a ψ-state if $s \models \psi$.

Definition 1.1. A *run* of an FDS $\mathcal{D}$ is a possibly infinite sequence of states $\sigma : s_0, s_1, \ldots$ satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.

- *Consecution* — For each $\ell = 0, 1, \ldots$, the state $s_{\ell+1}$ is a $\mathcal{D}$-successor of s_ℓ. That is, $\langle s_\ell, s_{\ell+1}\rangle \models \rho(V, V')$ where, for each $v \in V$, we interpret v as $s_\ell[v]$ and v' as $s_{\ell+1}[v]$.

Definition 1.2. A *computation* of $\mathcal{D}$ is an infinite run that satisfies:

- *Justice* — for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.

- *Compassion* – for every $\langle p, q\rangle \in \mathcal{C}$, either σ contains finitely many occurrences of p-states, or σ contains infinitely many occurrences of q-states.

A state s is said to be *reachable* if it participates in some run of $\mathcal{D}$. An FDS $\mathcal{D}$ is said to be *feasible* if it has at least one computation.

Definition 1.3. A *synchronous parallel composition* of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 ||| \mathcal{D}_2$, is specified by the FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$V = V_1 \cup V_2, \quad \Theta = \Theta_1 \wedge \Theta_2, \quad \rho = \rho_1 \wedge \rho_2, \quad \mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2, \quad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$$

Synchronous parallel composition is used for the construction of an observer system $\mathcal{D}_2$, which evaluates the behavior of another system $\mathcal{D}_1$. That is, running $\mathcal{D}_1 ||| \mathcal{D}_2$ allows $\mathcal{D}_1$ to behave as usual while $\mathcal{D}_2$ evaluates it.

Definition 1.4. An *asynchronous parallel composition* of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \parallel \mathcal{D}_2$, is specified by the FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$V = V_1 \cup V_2, \quad \Theta = \Theta_1 \wedge \Theta_2, \quad \rho = (\rho_1 \wedge pres(V_2 - V_1)) \vee (\rho_2 \wedge pres(V_1 - V_2)),$$
$$\mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2, \quad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$$

The predicate $pres(U)$ stands for the assertion $U' = U$, implying that all the variables in U are preserved by the transition. Asynchronous parallel composition represents the interleaving-based concurrency which is the assumed concurrency in shared-variables models.

1.2 Linear Temporal Logic

As the language for specifying properties of systems we use *linear-time temporal logic* (LTL) [MP92]. Assume an underlying (first-order) assertion language. A *temporal formula* is constructed out of state formulas (assertions) to which we apply the boolean operators $\neg$ and $\vee$, the basic temporal future operators $\bigcirc$ (Next) and $\mathcal{U}$ (Until), and the temporal past operators, $\ominus$ (Previous) and $\mathcal{S}$ (Since).

Denote true by T. Other temporal operators can be defined in terms of the basic ones as follows:

$\Diamond p$	$=$	$\mathrm{T}\ \mathcal{U}\ p$:	Eventually
$\Box p$	$=$	$\neg \Diamond \neg p$:	Henceforth
$p\,\mathcal{W}\,q$	$=$	$\Box p \vee (p \mathcal{U} q)$:	Waiting-for, Unless, Weak Until
$\blacklozenge p$	$=$	$\mathrm{T}\ \mathcal{S}\ p$:	Sometimes in the past
$\blacksquare p$	$=$	$\neg \blacklozenge \neg p$:	Always in the past
$p\,\mathcal{B}\,q$	$=$	$\blacksquare p \vee (p \mathcal{S} q)$:	Back-to, Weak Since

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, ...$ where each state s_j provides an interpretation for the variables of p. We define the notion of a temporal formula p holding at a position j, $j \geq 0$, in σ, denoted by $(\sigma, j) \models p$:

- For an assertion p,

  $(\sigma, j) \models p \quad \iff \quad s_j \models p$

  That is, we evaluate p locally on state s_j.

- $(\sigma, j) \models \neg p \quad \iff \quad (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \quad \iff \quad (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \quad \iff \quad (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \quad \iff \quad$ for some $k \geq j, (\sigma, k) \models q$,

  and for every i such that $j \leq i < k$, $(\sigma, i) \models p$
- $(\sigma, j) \models \ominus p \quad \iff \quad j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p \mathcal{S} q \quad \iff \quad$ for some $k \leq j, (\sigma, k) \models q$,

  and for every i such that $j \geq i > k, (\sigma, i) \models p$

This implies the following semantics for the derived operators:

- $(\sigma, j) \models \diamondsuit p \quad \iff \quad (\sigma, k) \models p$ for *some* $k \geq j$
- $(\sigma, j) \models \square p \quad \iff \quad (\sigma, k) \models p$ for *all* $k \geq j$
- $(\sigma, j) \models p \mathcal{W} q \quad \iff \quad (\sigma, k) \models p \mathcal{U} q$ or $(\sigma, k) \models \square p$
- $(\sigma, j) \models \diamondsuit\!\!\!-\, p \quad \iff \quad (\sigma, k) \models p$ for *some* k, $0 \leq k \leq j$
- $(\sigma, j) \models \boxminus p \quad \iff \quad (\sigma, k) \models p$ for *all* k, $0 \leq k \leq j$
- $(\sigma, j) \models p \mathcal{B} q \quad \iff \quad (\sigma, k) \models p \mathcal{S} q$ or $(\sigma, k) \models \boxminus p$

If $(\sigma, 0) \models p$ we say that p *holds over* σ and write $\sigma \models p$. Formula p is *satisfiable* if it holds over *some* model. We say that p is *valid* if it holds over *all* models.

Formulas p and q are *equivalent*, denoted by $p \sim q$, if $p \leftrightarrow q$ is a valid formula. Namely, p and q have the same truth value in the first position of every model. They are

called *congruent*, denoted by $p \approx q$, if $\square(p \leftrightarrow q)$ is a valid formula. Namely, p and q have the same truth value in all positions of every model. Note that $p \sim q$ does not imply $p \approx q$. For example, $\mathrm{T} \sim \neg \ominus \mathrm{T}$ since both are true in the first position of every model (since all positions satisfy T, $\neg \ominus \mathrm{T}$ means that there is no previous position); however, $\mathrm{T} \not\approx \neg \ominus \mathrm{T}$ since T holds at all positions of every model, while $\neg \ominus \mathrm{T}$ holds only in the first position.

1.3 Temporal Testers

Every LTL formula φ is associated with a *temporal tester* [KPR98], an FDS denoted by $T[\varphi]$. A tester contains a distinguished boolean variable x such that for every computation σ of $T[\varphi]$, for every position $j \geq 0$, $x[s_j] = 1 \Longleftrightarrow (\sigma, j) \models \varphi$.

For example, the temporal tester for the basic path formula $\square p$ is given by:

$$T[\square p]: \quad \begin{cases} V & : \quad V_p \cup \{x\} \\ \Theta & : \quad \mathrm{T} \\ \rho & : \quad x = (p \ \wedge \ x') \\ \mathcal{J} & : \quad \{x \ \vee \ \neg p\} \\ \mathcal{C} & : \quad \varnothing \end{cases}$$

The justice requirement $x \vee \neg p$ is intended to guarantee that we will not have a computation in which continuously $p = 1$, while $x = 0$.

The following construction is used for model-checking an FDS $\mathcal{D}$:

- Construct a temporal tester $T[\neg\varphi]$ which is initialized with $x = 1$, that is an FDS that comprises only φ-falsifying computations.

- Form the synchronous parallel composition $\mathcal{D} \,|||\, T[\neg\varphi]$, that is an FDS for which all computations are of $\mathcal{D}$ and violate φ.

- Check feasibility of $\mathcal{D} \,|||\, T[\neg\varphi]$. $\mathcal{D} \models \varphi$ if and only if $\mathcal{D} \,|||\, T[\neg\varphi]$ is infeasible.

Next, we present an incremental construction of a tester for a general LTL formula. We restrict our attention to LTL formulas whose principal operator is temporal (rather than boolean).

Let $f(\psi)$ be a principally temporal path formula containing one or more occurrences of the LTL formula ψ. We denote by $f(x)$ the formula obtained from f by replacing all occurrences of ψ by the boolean variable x. Then the construction principle is presented by the following recursive reduction formula:

$$T[f] \quad = \quad T[f(x_\psi)] \,|||\, T[\psi] \tag{1.1}$$

Namely, we conjoin the tester for ψ to the recursively constructed tester for the simpler formula $f(x_\psi)$.

We next illustrate the construction of the LTL formula $\Box \diamondsuit p$ for the case that p is a simple proposition (boolean variable). Application of the reduction formula leads to

$$T[\Box \diamondsuit p] \quad = \quad T[\Box x_{\diamondsuit}] \,|||\, T[\diamondsuit p]$$

Computing $T[\diamondsuit p]$ and $T[\square x_{\diamondsuit}]$ separately and forming their synchronous parallel composition yields the following tester whose output variable is $x_{\square}$.

$$T[\square \diamondsuit p]: \quad \begin{cases} V & : & \{p, x_{\diamondsuit}, x_{\square}\} \\ \Theta & : & \mathrm{T} \\ \rho & : & \left(x_{\diamondsuit} = p \vee x'_{\diamondsuit}\right) \wedge \left(x_{\square} = \left(x_{\diamondsuit} \wedge x'_{\square}\right)\right) \\ \mathcal{J} & : & \{\neg x_{\diamondsuit} \vee p, \quad x_{\square} \vee \neg x_{\diamondsuit}\} \\ \mathcal{C} & : & \varnothing \end{cases}$$

In general, for a principally temporal formula φ, $T[\varphi] = T_1 ||| \cdots ||| T_k$, where $T_1, \ldots, T_k$ are the temporal testers constructed for the principally temporal sub-formulas of φ. $T[\varphi]$ contains k auxiliary boolean variables, and the output variable of $T[\varphi]$ is the output variable of T_1 — the last constructed tester.

In general, the recursive reduction described by Equation (1.1) is carried until we obtain the tester T_1 which is a tester for a basic path formula. We can carry it one step further and obtain an assertion that contains no further temporal operators. We refer to this assertion as the *redux* of the original LTL formula φ, denoted by *redux*(φ). For the case that φ is principally temporal, *redux*(φ) is the single output variable x_{φ}. If we apply the recursive construction Equation (1.1) to an LTL formula which is not principally temporal, we may obtain a more complex assertion as the resulting redux.

Consider, for example, the LTL formula $\varphi : \square p \vee \diamondsuit q$. The corresponding tester is given by:

$$T[\varphi] \quad = \quad T[\square p] ||| T[\diamondsuit q]$$

while ***redux***$(\varphi) = x_{\Box} \ \vee \ x_{\Diamond}$, where $x_{\Box}$ and $x_{\Diamond}$ are the output variables of $T[\Box p]$ and $T[\Diamond q]$, respectively.

2

VERIFICATION OF TRANSACTIONAL MEMORIES

Multicore architectures have become common in the design of microprocessor chips, and they require developers to produce parallel programs in order to gain full advantage of the multiple number of processors. Parallel programming, however, is very challenging. It compels programmers to carefully coordinate and synchronize access to shared data in order to ensure that programs do not produce inconsistent, incorrect or nondeterministic results. Locks, semaphores, mutexes, and similar constructs are difficult to compose, and if applied incorrectly may introduce undesirable effects such as deadlocks, priority inversion, and convoying.

Transactional Memory avoids these pitfalls, and simplifies parallel programming by transferring the burden of concurrency management from the programmers to the system designers, thus enabling programmers to safely compose scalable applications. A transaction specifies program semantics in which a computation executes as if accessing the memory exclusively, thus releasing the programmer from reasoning directly about concurrency. Consequently, transactional memory is considered to be a promising alternative method for coordinating objects. In 1993 Herlihy and Moss [HM93a] introduced a hardware transactional memory intended to make lock-free synchronization efficient. Two years later, Shavit and Touitou [ST95] proposed the first software transactional

memory, to be applied to existing processors. Since then, numerous new implementations comprising hardware, software and their integration have been proposed (see [LR07] for an excellent survey).

Transactional memories are often parameterized by their properties. These include the conflicts they are to avoid, policies for resolving conflicts, support for nested transactions and non-transactional memory accesses, etc. Each set of parameters defines a unique set of sequences of events that can occur in the system so as to guarantee atomicity and serializability. We refer to the specification of those allowed sequences of events as the specification of the transactional memory. A particular implementation need not generate all allowed sequences, but it should not generate any sequence that is not permitted. In this work we provide a framework and tools that allow to formally verify that a transactional memory implementation satisfies its specification.

The rest of the chapter is organized as follows: Section 2.1 reviews the traditional methods for concurrency control and their drawbacks. It also examines different parameters of transactional memory implementations. Section 2.2 first defines transactions and the concept of admissible interchanges of transactional operations. It then presents a specification model and an implementation from the literature called TCC. Section 2.3 presents a proof rule for verifying implementations using an abstraction mapping, and applies it to prove TCC's correctness using a model checker. Section 2.4 generalizes the proof rule by assuming an abstraction relation and applies the new rule in a hand proof of TCC. Section 2.5 shows how to mechanically verify correctness of implementations using a theorem prover, provides a framework for doing so in TLPVS and shows how

the framework is applied in proving TCC and two additional implementations from the literature. Section 2.6 extends the model to support non-transactional memory accesses and provides a correctness proof of a variation of TCC that handles non-transactional accesses. Sections 2.7, 2.8 and 2.9 discuss related work, conclusions and future work, respectively.

2.1 Background

2.1.1 Conventional Solutions for Concurrency Control

Concurrent programming is notoriously difficult. *Race conditions*, in which the output of concurrent threads is faulty as a result of an unexpected order of memory accesses, are created easily. In this section we show why some of the well known methods for coordinating accesses to shared resources in multiprocessor applications are intricate and make it hard to design computer systems that are reliable and scalable.

Lock Based Mechanisms In lock based mechanisms each process acquires a lock before accessing the shared resource. The lock then prevents other processes from accessing the resource until it is released. *Coarse-grained locks* protect relatively large amounts of data, even parts that the process does not access, whereas *fine-grained locks* protect a single resource (or a small number of them) and are therefore held by the processes for as little time as possible.

There are many implementations of lock based mechanisms. *MUTual EXclusion*

(mutex) algorithms give the process that successfully locks them ownership of the resource until it unlocks the mutex. Any process that in the meantime attempts to lock the mutex must wait until the owner unlocks it. *Monitors* are abstract data types, i.e. combinations of data structures and operations, that allow only one operation to be executed at a time. The required protections are enforced by the compiler implementing the monitor. *Semaphores* (sometimes referred to as counting semaphores) are counters that are always greater than or equal to 0. Any process can decrement the counter to lock the semaphore, but attempting to decrement it below 0 causes the calling process to wait for another process to unlock it first. No ownership is associated with a semaphore, namely, a process that never locked the semaphore may unlock it, which could cause unpredictable application behavior.

Coarse-grained locks are a source of contention since they protect parts of the data that are not accessed, thus processes cause one another to block even when they do not really interfere. Fine-grained locks often require sophisticated lock ordering to prevent *deadlock*, a state in which the system halts forever and further progress is impossible because processes are waiting for locks that are never be released. Consider the two threads of Fig. 2.1. If thread 1 locks x first and immediately after thread 2 locks y

Thread 1:	*Thread 2:*
`lock(x)`	`lock(y)`
`lock(y)`	`lock(x)`
`...`	`...`
`unlock(y)`	`unlock(x)`
`unlock(x)`	`unlock(y)`

Figure 2.1: Two threads locking the same data.

(before thread 1 locks y), then thread 1 must wait until thread 2 unlocks y before it may lock it. But since thread 2 must lock x before releasing y, both would wait forever.

Most mutual exclusion algorithms have side-effects including *starvation*, in which a process never gets sufficient resources to complete its task, *priority inversion* in which a higher priority process waits for a lower-priority process that holds the necessary resources, and *high latency* in which response to interrupts is not prompt.

Locks have other disadvantages. They are vulnerable to failures and faults – if one process dies, stalls/blocks or goes into any sort of infinite loop while holding a lock, other processes waiting for the lock may wait forever. Bugs are often very subtle and may be almost impossible to reproduce. Locks are not compassable. For example, deleting an item from one table and inserting it into a different table cannot be combined as one single atomic operation using locks. They also suffer from *convoying* which occurs when several processes wait for a process holding a lock that is scheduled due to a time-slice interrupt or a page fault.

Modern programming languages that have more abstract concurrency control mechanisms are error prone as well. Java code enclosed within a synchronized block is guaranteed to be executed by a single process at any given time. Internally, the synchronized keyword is implemented by the run-time machine as a mutex. The process that succeeds acquiring the mutex, runs the code and releases the mutex only when exiting the synchronized block. Consider the example from [Hol98], provided in Fig. 2.2, of a thread that consists of two variables that are modified by executing the synchronized procedure modify().

```
class My_thread extends Thread{
  private int field_1 = 0;
  private int field_2 = 0;
  public void run(){
    setDaemon(true);
    while(true){
      System.out.println("field_1=" + field_1 +
                         "field_2=" + field_2);
      sleep(100);
    }
  }

  synchronized public void modify(int new_value){
    field_1 = new_value;
    field_2 = new_value;
  }
}
```

Figure 2.2: An example for a concurrency control mechanism in Java.

The user can initialize a thread and modify the variables as follows:

```
My_thread test = new My_thread;
test.start();
...
test.modify(1);
```

The only functions executed by the new thread are run() itself and println() (which run() calls). The method modify() never runs on the same thread as println(); instead, it runs on the thread that was running when the call to run() was made made (in this case,

it runs on whatever thread main() is running on). Depending on scheduling, the earlier fragment outputs: `field_1=0, field_2=0` or `field_1=0, field_2=1` or `field_1=1, field_2=1`. In the first and last cases, the thread is outside the println() statement when modify() is called. In the second case, the thread is halfway through evaluating the arguments to println(), having fetched `field_1` but not `field_2`. It thus prints the unmodified `field_1` and the modified `field_2`.

There is no simple solution to this race condition. The method modify() is indeed synchronized in the earlier example, but run() cannot be. If it could, the thread would have started, then entered the monitor and locked the object. Thereafter, any other thread that would have called any synchronized method on the object (such as modify()) would have block until the monitor was released. Since run() does not return, the release could have never happened, and any other thread that would have called any of its synchronized methods, would have been blocked forever. In the current example, the main thread would have been suspended, and the program would hang. Thus just using the synchronized keyword in a naive way can get inconsistent results. [Hol98] gives other examples that lead to faulty results.

Non-Blocking Mechanisms Non-blocking algorithms are an alternative for lock-based mechanisms as they avoid some of the problems associated with locks; in particular they ensure one of the following types of progress. *Wait-free* guarantees that every process completes its task within a bounded number of steps. *Look-free* ensures that at least one process makes progress within a bounded number of steps. And last, *obstruction-*

free [HLM03], ensures that if, at any point, a single process executes in isolation for a bounded number of steps, then it will complete its task. All wait-free algorithms are lock-free and all lock-free algorithms are obstruction-free. Non-blocking algorithms use atomic primitives provided by the hardware, e.g., *compare and swap* (an atomic instruction that compares the contents of a memory location to a given value and, if the same, writes a new given value to that location), however, they are extremely difficult to implement.

2.1.2 Transactional Memory

Transactional Memory avoids most pitfalls of traditional solutions, and simplifies parallel programming by transferring the burden of concurrency management from the programmers to the system designers, thus enabling programmers to safely compose scalable applications.

A *transaction* is a sequence of operations that are executed *atomically* – either all complete successfully (and the transaction commits), or none completes (and the transaction aborts). Moreover, transactions run in *isolation* – each transaction appears as if running alone on the system (without interleaving of other transactions). Transactional memory allows transactions to run concurrently as long as the atomicity and isolation of each transaction are preserved.

Although transactional memory simplifies parallel programming and avoids many of the drawbacks that other solutions for concurrency control have, it is still not ideal and

```
bool flagA=0, flagB=0;

 Thread 1:                    Thread 2:
 atomic {                     atomic {
   while(!flagA);               flagA = true;
   flagB = true;                while(!flagB);
 }                            }
```

Figure 2.3: Example: non-terminating transactions.

could be used incorrectly. Consider the example from [BLM05], provided in Fig. 2.3. The while loops in both transactions never terminate and thus the transactions never complete.

2.1.2.1 Comparison to Transactions in Database Systems

Transactions have been used for a long time in database systems and have been proven to be a good mechanism for constructing parallel computations. Although there are many similarities, the objectives and the implementation techniques when accessing memory differ greatly. Database systems store data on discs rather than in memory which requires a much longer time to access, therefore computation time is negligible relative to access time. Transactional memory implementations do not have to worry about durability since as opposed to database systems; the data does not last after the program terminates. This makes the transactional memory implementations much simpler in that sense. Transactional memory usually has to coexist and support legacy code that access the memory without using transactions, while database systems are self contained.

Database transactions have four attributes, known as ACID: atomicity, consistency,

isolation and durability. Atomicity requires that either all the actions in a transaction complete successfully or none complete. Consistency requires that the changes made by a transaction leave the data consistent. Isolation requires that the result of each transaction is correct regardless of the other transactions executing in parallel. Durability requires that after a transaction commits, its modifications to the data are permanent and available to all subsequent transactions. The first three attributes, namely ACI, also apply to transactional memory – durability is less important since the data does not last in memory after the program terminates.

2.1.2.2 Implementation Approaches

Transactional memory can be implemented in software (STM), hardware (HTM) or a hybrid of the two (HyTM).

Software Transactional Memory STM implementations are entirely in software, i.e., they are implemented through languages, compilers and libraries. They implement algorithms for updating the memory and detecting conflicts in software using read and write barriers and software data structures. Therefore they permit a wide variety of algorithms and can be easily modified and enhanced, however there is usually a cost of a high runtime overhead. STM can be integrated with existing software systems and programming languages, and there is usually no inherent limitation on the size of the data structures or transactions.

STM systems differ in the type of storage unit (sometimes called *transaction granularity*) they handle. Systems implemented in object oriented languages, such as Java,

usually have an *object granularity* (e.g., see [HLMW03, GHP05, MSS05]). Other systems have a *word granularity* [HF03, Fra03] or a *block granularity* ([DS06] supports all three types of granularity), depending on whether they reference single words or blocks of words. Since metatdata is often associated with the referenced unit, in object granularity systems the objects are simply extended with a field that records the metadata and which allows quick access to it. Systems with word or block granularity maintain a separate table that records the metatdata, thus access time is slower. One of the disadvantages with object granularity is that conflicts (explained shortly) are detected even if concurrent transactions access different fields of the same objects. In this work we assume word granularity.

Hardware Transactional Memory HTM implementation are hardware systems that have a fairly small software and a large hardware components. These implementations eliminate most overheads introduced by STM systems and therefore yield a better performance. They use memory consistency models to reason about the ordering of reads and writes from multiple processes. To enable easy detection of conflicts between concurrent transactions and to store data that is frequently referenced local caches are used. Thus, there is often more than a single copy for each memory location, and cache coherence protocols are used to keep all of the copies up-to-date. These protocols are able to locate all copies of a given memory location and ensure that these copies hold the latest values, and that processes observe updates to the same location in the right order. The hardware support makes the implementations complex and expensive to modify. Trans-

action size is limited by hardware resources and overflow of operations is considered a major challenge of HTM.

Hybrid Transactional Memory HyTM implementations combine HTM with STM so to maintain the flexibility of STM (e.g. unbounded transactions) while using a hardware mechanism. Some implementations [DFL+06, Lie04] first try to execute transactions in hardware and only switch to software when the hardware resources reach their limits or to detect conflicts. In contrast [SSH+07] proposes to use hardware for optimizing the performance of transactions that are controlled by software. Other approaches such as [KCH+06, SMD+06] do not directly integrate STM with HTM and instead provide an interface for the STM to control specific HTM operations.

2.1.2.3 Correctness Criteria

In this section we give an informal intuition about some of the correctness criteria which have been considered for transactional memory.

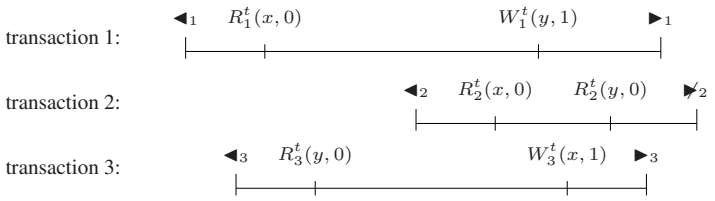
Serializability A concurrent execution is *serializable* [EGLT76] with respect to specification $\mathcal{S}$, if there exists a sequential execution (transactions appear one after the other) of its committed transactions that satisfies $\mathcal{S}$.

Strict Serializability *Strict serializability* [Pap79] is a restricted variation of serializability that requires real time ordering of the committed transactions. Namely, a con-

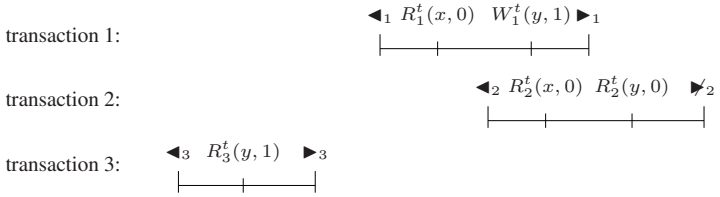
current execution is strictly serializable with respect to specification $\mathcal{S}$ if there exists a sequential execution of its committed transactions that satisfies $\mathcal{S}$ and has the same order between every two non-overlapping transactions.

Abort Consistent [GK08] studies different correctness criterions and introduces *Opacity* as a new alternative. Opacity requires that: (i) operations of every committed transactions appear as if they happened at some single point during the transaction duration, (ii) operations of aborted transactions can not be observed by other transactions and (iii) transactions (committed and aborted) always observe a consistent state of the memory. *Abort consistency*, a product of Opacity's first and third requirements, requires that in addition to strict serializability aborted transactions do not observe an inconsistent state of the memory.

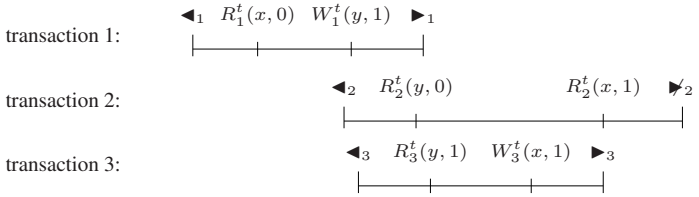
Figure 2.4 illustrates various scenarios that satisfy different correctness criterions, when read operations are required to return the most recent value written to the referenced location by a committed transaction. $\blacktriangleleft_p$, $R_p^t(x, v)$, $W_p^t(y, v)$, $\blacktriangleright_p$ and $\not\blacktriangleright_p$ denote the transactional operations of client p for opening a transaction, reading v from address x, writing v to address y, committing a transaction and aborting a transaction, respectively. Detailed semantics are provided later in Subsection 2.2.1. In all scenarios we assume that x and y are initialized to 0. The first scenario described in Fig. 2.4 is not



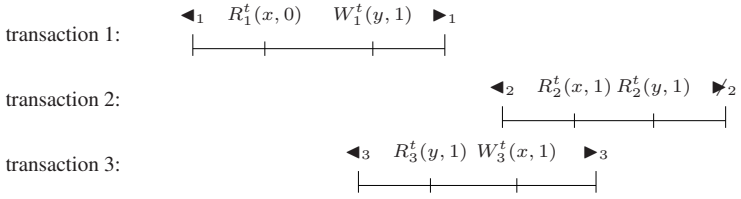
First scenario: not serializable.



Second scenario: serializable.



Third scenario: serializable and strictly serializable.



Fourth scenario: serializable, strictly serializable and abort consistent.

Figure 2.4: Various scenarios that satisfy different correctness criterions.

serializable since neither the first nor the last transaction can appear before the other in a sequential execution. The second scenario is not strictly serializable since the first and third transactions do not overlap, thus requiring the third transaction to appear earlier in any sequential execution. Abort consistency is not satisfied by the third scenario since there is no sequential execution during which $y = 0$ and $x = 1$. All correctness conditions are satisfied by the fourth scenario of Fig. 2.4.

In Subsection 2.2.1 we formally define serializability with respect to a set of admissible interchanges of transactional operations.

2.1.2.4 Design Alternatives

Transactional memory implementations are classified using a number of parameters that may define the programming model and performance of the system. In this section we discuss alternatives for some of the parameters.

Version Management An implementation employs *eager version management* (sometimes called direct update) if it updates the memory immediately when a transactional write occurs. It uses *lazy version management* (also called deferred update) if it differs the update until the transaction commits. There is a tradeoff between the amount of work and additional data structures that are required when committing and aborting. Under eager version management aborts may require rolling-back the memory to hold its old values, which means that transactions have to maintain a log with the original values of all locations that were modified. Under lazy version management, transactions keep a private record of all the write events that take place. On commit, the memory is updated

based on these records. Aborts require no further ado.

Conflict Detection A *conflict* occurs when two overlapping transactions (each begins before the other ends) access the same location and at least one writes to it. An implementation uses *eager conflict detection* if it detects conflicts as soon as they occur, and it uses *lazy conflict detection* if it delays the detection until a transaction requests to commit. When a conflict occurs, eager conflict detection helps to avoid worthless work by a transaction that is eventually aborted. Yet, deciding to keep one transaction (and to cause the other to abort) does not guarantee that the "surviving" transaction can commit since it may conflict with a third transaction. On the other hand, lazy conflict detection may allow doomed transactions to perform worthless work. Note that lazy conflict detection may not be combined with eager version management.

Arbitration A conflict between two transactions is resolved by aborting at least one of the transactions. Transactional memory systems often have an arbitration mechanism (sometimes called contention management) that determines which of the two transactions should be aborted. [SS04, SS05a, SS05b, GHP05, GHKP05, GC08] propose several arbitration policies and examine their effect on the system's performance and on the progress of each process. The arbitration mechanism may implement more than one policy from which it chooses automatically or based on a predefinition made by the user.

2.1.2.5 Non-Transactional Memory Access

Accesses to the memory are not necessarily all transactional and for various reasons (e.g., legacy code) they can be also *non-transactional*, i.e., executed not as part of a transaction. Unlike transactional accesses, non-transactional accesses cannot be aborted. The atomicity and serializability requirements remain intact, whereas two types of isolation are considered, weak and strong. Transactional memory systems that guarantee weak isolation ensure that isolation is kept only between transactions, namely they do not guarantee consistency and correctness when non-transactional accesses are executed. With strong isolation transactional memory systems assures that isolation is kept between transactions but also between transactions and non-transactional accesses.

In our work we first consider a model that does not support non-transactional accesses and later in Section 2.6 extend it to support them under the assumptions of strong isolation where non-transaction operations are cast as a successfully committed, single operation, transaction.

Note that implementations that have eager version management, in which aborts may require rolling-back the memory to hold its old values, can not handle non-transactional operations. A non-transactional read may obtain a value that was written by a transaction that is later aborted, thus the value is not valid since the transaction is never committed.

2.1.2.6 Nested Transactional Memory

There are many reasons for supporting nested transactions, for example, a programmer's transactional code may use library procedures that contain transactions as well. Another

reason is to avoid discarding all the operations in very large aborted transactions when some portions may be committed safely.

There are many issues to consider when allowing nested transactions, for example, what values are read when referring to locations that have already been updated, when to update the memory, how conflicts should be defined between sibling transactions, etc. We shall refer to the transactions' structure as a tree with the root being the top-level transaction and the leaves transactions that have no inner nested transactions.

Flat nesting is the easiest way to deal with nested transactions; the inner transaction is simply merged entirely with the outer one. When a transaction reads, the returned value is the most recent one written to the same location by any of the transactions in the same tree, if such exists. There are no conflicts between transactions in the same tree. Only when the top-level transaction commits the writes of all the inner transactions become permanent. Aborting a transaction implies that all other transactions in the same tree are aborted as well.

Two types of more sophisticated nesting have been considered in the literature so far: *closed* and *opened* nesting. We follow the definitions of [MH06]. In *closed nesting*, only transactions with no pending inner transactions may access data. The memory is updated only when the top-level transaction commits. When a transaction attempts to read, it sees the most recent value it wrote to that location, if such exists; otherwise it reads the value seen by its parent. A top-level transaction sees the latest committed value. Two Transactions conflict if they access the same memory location, at least one writes to it, and neither one is an ancestor of the other. When a nested transaction

commits, its reads and writes are unioned with those of its parent. When a top-level transaction commits, all writes become permanent. When a transaction aborts, its reads and writes (and tentatively written values by its inner committed transactions) are simply discarded.

Open nesting allows for more concurrency, by releasing resources earlier and applying conflict detection at a higher level of abstraction (at the cost of both software and hardware complexity). When a transaction commits the memory is updated immediately and the changes are visible to all other transactions in the system. When a transaction aborts, the updates made by inner nested transactions remain committed. Conflict definitions are identical to those of closed nesting.

Most implementations that support nested transactions support flat [HLMW03, HF03, AR05, MSH+06] or closed nesting [Fra03, GHP05, SATH+06, ATLM+06]. A few implementations support open nesting, such as [MBM+06b] that extends the original LogTM [MBM+06a], TCC [Car06, MCC+06] and the Java-based STM of [NMA+07], which is the first to support open nesting in a software implementation. In our work we assume that transactions may not be nested.

2.2 Specification and Implementation

2.2.1 Transactional Sequences

Assume n *clients* that direct requests to a *memory system*, denoted by *memory*. Let the notation of an invocation begins with an ι. For every client p, let the set of *transactional invocations by client* p consists of:

- $\iota \blacktriangleleft_p$ – An open transaction request.
- $\iota R_p^t(x)$ – A transactional read request from address $x \in \mathbb{N}$.
- $\iota W_p^t(y, v)$ – A transactional request to write the value $v \in \mathbb{N}$ to address $y \in \mathbb{N}$.
- $\iota \blacktriangleright_p$ – A commit transaction request.
- $\iota \not\blacktriangleright_p$ – An abort transaction request.

The memory provides a response for each invocation. Erroneous invocations (e.g., a $\iota \blacktriangleleft_p$ while client p has a pending transaction) are responded by the memory returning an error flag err. Non-erroneous invocations, except for $\iota \not\blacktriangleright_p$, are responded by $abort$ if the transaction should abort. Otherwise, non-erroneous invocations, except for ιR^t (e.g., $\iota \blacktriangleleft_p$ when there is no pending p transaction), are responded by the memory returning an acknowledgment ack, whereas for $\iota R_p^t(x)$ the memory returns the (natural) value of the memory at location x. Since we observe the responses, we assume that invocations

and responses occur atomically and consecutively, i.e., there are no other operations that interleave an invocation and its response.

Let E_p^t: $\{ \blacktriangleleft_p, R_p^t(x,u), W_p^t(x,v), \blacktriangleright_p, \not\blacktriangleright_p \}$ be the *set of transactional observable events associated with client* p. We consider as observable events only requests that are accepted, and abbreviate invocation and its response by omitting the ι-prefix of the invocation. Thus, $\blacktriangleleft_p$, $W_p^t(x,v)$, $\blacktriangleright_p$ and $\not\blacktriangleright_p$ abbreviate $\iota \blacktriangleleft_p \ ack_p$, $\iota W_p^t(x,v) ack_p$, $\iota \blacktriangleright_p \ ack_p$ and $\iota \not\blacktriangleright_p \ ack_p$, respectively. For read actions, we include the value read, that is, $R_p^t(x,u)$ abbreviates $\iota R^t(x)$ and the response u. When the value written/read has no relevance, we write the above as $W_p^t(x)$ and $R_p^t(x)$. When both values and addresses are of no importance, we omit the addresses, thus obtaining W_p^t and R_p^t. The output of each action is its relevant observable event when the invocation is accepted, and undefined otherwise. Let E^t be the set of all transactional observable events over all clients, i.e., $E^t = \bigcup_{p=1}^{n} E_p^t$. We denote by E the set of all observable events (note, however, that until Section 2.6, in which the model is extended with non-transactional operations, $E = E^t$. We use E^t when the content refers only to transactional events and E when it refers to any observable events.

Definition 2.1. Let σ: $e_0, e_1, \ldots, e_k$ be a finite sequence of observable E^t-events. The sequence σ is called a *well-formed transactional sequence* (TS for short) if the following all hold:

1. For every client p, let $\sigma|_p$ be the sequence obtained by projecting σ onto E_p^t. Then $\sigma|_p$ satisfies the regular expression T_p^*, where T_p is the regular expression

$\blacktriangleleft_p\ (R_p^t + W_p^t)^*(\blacktriangleright_p + \not\blacktriangleright_p)$. For each occurrence of T_p in $\sigma|_p$, we refer to its first and last elements as *matching*. The notion of matching is lifted to σ itself, where $\blacktriangleleft_p$ and $\blacktriangleright_p$ (or $\not\blacktriangleright_p$) are matching if they are matching in $\sigma|_p$;

2. The sequence σ is *locally read-write consistent*: for any subsequence of σ of the form $W_p^t(x,v)\,\eta\,R_p^t(x,u)$, where η contains no event of the form $\blacktriangleright_p$, $\not\blacktriangleright_p$, or $W_p^t(x,u)$, we have $u = v$.

We denote by $\mathcal{T}$ the set of all well-formed transactional sequences, and by *pref*$(\mathcal{T})$ the set of $\mathcal{T}$'s prefixes. Note that the requirement of local read-write consistency can be enforced by each client locally.

Definition 2.2. A TS σ is called *atomic* if:

1. σ satisfies the regular expression $(T_1 + \cdots + T_n)^*$. That is, there is no overlap between any two transactions;

2. σ is *globally read-write consistent*: for any subsequence $W_p^t(x,v)\eta R_q^t(x,u)$ in σ, where η contains $\blacktriangleright_p$, which is not preceded by $\not\blacktriangleright_p$, and contains no $W_k^t(x)$ followed by an $\blacktriangleright_k$, it is the case that $u = v$.

2.2.2 Interchanging Events

The notion of a correct implementation is that every TS can be transformed into an atomic TS by a sequence of interchanges that swap two consecutive events. This definition is parameterized by the set $\mathcal{A}$ of *admissible interchanges* which may be used in

the process of serialization. Rather than attempt to characterize $\mathcal{A}$, we choose to characterize its complement $\mathcal{F}$, the set of *forbidden interchanges*. To simplify the verification process, we restrict to swaps whose soundness depends only on the history leading to them.

In all our discussions, we assume *commit preserving serializability* that implies that while serializing a TS, the order of committed transactions has to be preserved.

Consider a temporal logic over E^t using the past operators $\ominus$ (previously), $\diamondsuit$ (sometimes in the past), and $\mathcal{S}$ (since). Let σ be a prefix of a well-formed TS over E. We define a satisfiability relation $\models_{end}$ between σ and a temporal logic formula φ so that $\sigma \models_{end} \varphi$ if at the end of σ, φ holds. (The more standard notation is $(\sigma, |\sigma| - 1) \models_{end} \varphi$, but since we always interpret formulae at the end of sequences we chose the simplified notation.)

$\mathcal{F}$ always includes two structural restrictions: The formula $\blacktriangleright_p \wedge \ominus \blacktriangleright_q$, $p \neq q$ forbids the interchange of closures of transactions belonging to different clients. This guarantees that commit order is preserved by the serializability process. The restriction $e_p \wedge \ominus \tilde{e}_p$, where $e_p, \tilde{e}_p \in E^t_p$, forbids interchanging two events belonging to the same client.

Let $\mathcal{F}$ be a set of forbidden formulae characterizing all the forbidden interchanges, and let $\mathcal{A}$ denote the set of interchanges that do not satisfy any of the formulas in $\mathcal{F}$. Assume that $\sigma = a_0, \ldots, a_k$. Let σ' be obtained from σ by interchanging two elements, say a_{i-1} and a_i. We then say that σ' is *1-derivable from σ with respect to $\mathcal{A}$* if $(a_0, \ldots, a_i) \not\models_{end} \bigvee \mathcal{F}$. Similarly, we say that σ' is *derivable from σ with respect to $\mathcal{A}$*

if there exist $\sigma = \sigma_0, \ldots, \sigma_\ell = \sigma'$ such that for every $i < \ell$, σ_{i+1} is 1-derivable from σ_i with respect to $\mathcal{A}$.

Definition 2.3. A TS is *serializable with respect to* $\mathcal{A}$ if there exists an atomic TS that is derivable from it with respect to $\mathcal{A}$, or, correspondingly, a TS is *not serializable with respect to* $\mathcal{A}$ if an atomic TS cannot be derived from it with respect to $\mathcal{A}$.

The sequence $\breve{\sigma}$ is called the *purified version* of TS σ if $\breve{\sigma}$ is obtained by removing from σ all aborted transactions, i.e., removing the opening and closing events for such a transaction and all the read-write events by the same client that occurred between the opening and closing events. When we specify the correctness of a transactional memory implementation, only the purified versions of the implementation's transaction sequences will have to be serializable.

Definition 2.4. Let TS σ be $\sigma = a_0, a_1, \ldots, a_n$. Event a_i *precedes* event a_j in σ, denoted by $a_i \prec_\sigma a_j$, if $i < j$.

When the TS is clear from the context, we use $\prec$ instead of $\prec_\sigma$. If the order between events a_i and a_j is insignificant we simply write a_i, a_j. Thus, $a_i, a_j \prec a_k$ means that both a_i and a_j precede a_k but the order between a_i and a_j is insignificant.

2.2.2.1 Forbidding Conflicts

Forbidding certain interchanges guarantees the absence of specific conflicts. For example, following [Sco06], there is a *lazy invalidation* conflict (see Fig. 2.5) when committing one transaction may invalidate a read of another. More formally, there is a lazy invalidation conflict if for some p, q and a memory address x, $R^t_p(x), W^t_q(x) \prec \blacktriangleright_q \prec \blacktriangleright_p$.

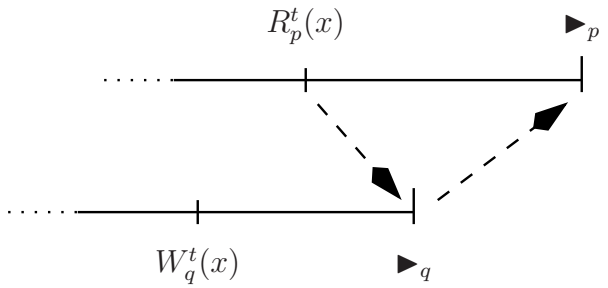


Figure 2.5: Conflict lazy invalidation.

We associate conflicts with TL formulae that determine, for any prefix of a TS whether the last two events in the prefix can be safely interchanged without removing the conflict. For a conflict c, the formula that, jointly with the structural restrictions of $\mathcal{F}$, forbid interchanges that may remove instances of this conflict is called *the maintaining formula for* c and is denoted by m_c. For lazy invalidation conflict we define m_{li} to be

$$\blacktriangleright_q \ \wedge \ominus (R_p^t(x) \ \wedge \ (\neg \blacktriangleright_q) \mathcal{S} \ W_q^t(x)) \tag{2.1}$$

Let $\mathcal{F}_{li}$ be the forbidden set that only includes the basic structural restrictions and m_{li}, and let $\mathcal{A}_{li}$ be its complement.

Theorem 2.5. *A TS σ is serializable with respect to $\mathcal{A}_{li}$ iff there is no lazy invalidation conflict in σ.*

Proof. In one direction, let p and q be clients, and x be a memory location, such that $R_p^t(x), W_q^t(x) \prec_\sigma \blacktriangleright_q \prec_\sigma \blacktriangleright_p$. Assume, by way of contradiction, that σ is serializable with respect to $\mathcal{A}_{li}$. Definition 2.3 then implies that there exists TS σ', which is derivable from σ with respect to $\mathcal{A}_{li}$ such that all events of one of T_p and T_q appear before all events

of the other. Since we assume interchanges are commit preserving and thus require $\mathcal{F}_{li}$ to forbid interchanges of $\blacktriangleright$ events, and since $\blacktriangleright_q \prec_\sigma \blacktriangleright_p$, it follows that $\blacktriangleright_q \prec_{\sigma'} \blacktriangleright_p$, and, consequently, that in σ' all of T_q's events appear before all of T_p's events. This, however, implies that the series of interchanges from σ to σ' must include an interchange between the pair $(R_p^t(x), \blacktriangleright_q)$ which is forbidden by $\mathcal{F}_{li}$. We therefore conclude that σ is not serializable with respect to $\mathcal{A}_{li}$, contradicting our assumption.

In the other direction, assume that TS σ is not serializable with respect to $\mathcal{A}_{li}$. Therefore, no atomic TS can be derived from σ with respect to $\mathcal{A}_{li}$. Hence, there are two transactions, belonging to different clients, T_p and T_q, that (i) are overlapping in σ, (ii) $\blacktriangleright_q \prec_\sigma \blacktriangleright_p$, and (iii) moving the events of one transaction to precede those of the other involves violation of $\mathcal{A}_{li}$. From the definition of $\mathcal{A}_{li}$ it now follows that for some x, $R_p^t(x), W_q^t(x) \prec_\sigma \blacktriangleright_q \prec_\sigma \blacktriangleright_p$, and, consequently, σ has a lazy invalidation conflict. □

We next provide formulae for each of the other conflicts defined by [Sco06] except for mixed invalidation which requires future operators.

1. An *overlap* conflict (see Fig. 2.6) occurs if for some transactions T_p and T_q, we have $\blacktriangleleft_p \prec \blacktriangleright_q$ and $\blacktriangleleft_q \prec \blacktriangleright_p$. We define the maintaining formula m_o to be $\blacktriangleright_q \wedge \ominus \blacktriangleleft_p$, for every $p \neq q$. Let $\mathcal{F}_o$ be the forbidden set that only includes the basic structural restrictions and m_o, and let $\mathcal{A}_o$ be its complement.

   Theorem 2.6. *A TS σ is serializable with respect to $\mathcal{A}_o$ iff there is no overlap conflict in σ.*

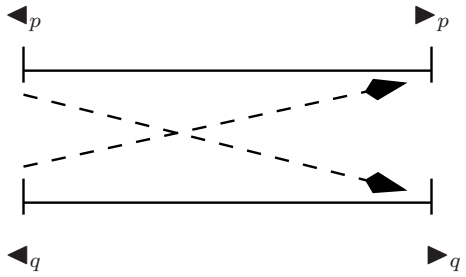


Figure 2.6: Conflict overlap.

Proof. In one direction, let p and q be clients, and x be a memory location, such that $\blacktriangleleft_p \prec_\sigma \blacktriangleright_q$ and $\blacktriangleleft_q \prec_\sigma \blacktriangleright_p$. Assume, by way of contradiction, that σ is serializable with respect to $\mathcal{A}_o$. Definition 2.3 then implies that there exists TS σ', which is derivable from σ with respect to $\mathcal{A}_o$ such that all events of one of T_p and T_q appear before all events of the other. The series of interchanges from σ to σ' must include an interchange between either the pair $(\blacktriangleleft_p, \blacktriangleright_q)$ or the pair $(\blacktriangleleft_q, \blacktriangleright_p)$, which are both forbidden by $\mathcal{F}_o$. We therefore conclude that σ is not serializable with respect to $\mathcal{A}_o$, contradicting our assumption.

In the other direction, assume that TS σ is not serializable with respect to $\mathcal{A}_o$. Therefore, no atomic TS can be derived from σ with respect to $\mathcal{A}_o$. Hence, there are two transactions, belonging to different clients, T_p and T_q, that (i) are overlapping in σ, (ii) moving the events of one to precede those of the other involves violation of $\mathcal{A}_o$. From the definition of $\mathcal{A}_o$ it now follows that for some x, $\blacktriangleleft_p \prec \blacktriangleright_q$ and $\blacktriangleleft_q \prec \blacktriangleright_p$, and, consequently, σ has an overlap conflict. □

2. A *writer overlap* conflict (see Fig. 2.7) occurs if two transactions overlap and one

writes to any address before the other terminates, i.e., for some T_p and T_q, we have $\blacktriangleleft_p \prec W_q^t \prec \blacktriangleright_p$ or $W_q^t \prec \blacktriangleleft_p \prec \blacktriangleright_q$. We define the maintaining formula m_{wo} to be $(\blacktriangleright_p \wedge \ominus W_q^t) \vee (\blacktriangleright_q \wedge \ominus(\blacktriangleleft_p \wedge (\neg \blacktriangleright_q) \mathcal{S} W_q^t))$. Let $\mathcal{F}_{wo}$ be the forbidden set that only includes the basic structural restrictions and m_{wo}, and let $\mathcal{A}_{wo}$ be its complement.

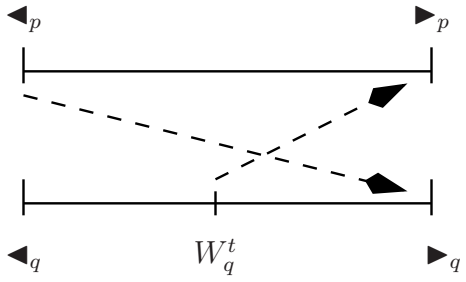


Figure 2.7: Conflict writer overlap.

Theorem 2.7. *A TS σ is serializable with respect to $\mathcal{A}_{wo}$ iff there is no writer overlap conflict in σ.*

Proof. In one direction, let p and q be clients, and x be a memory location, such that $\blacktriangleleft_p \prec_\sigma W_q^t \prec_\sigma \blacktriangleright_p$ or $W_q^t \prec_\sigma \blacktriangleleft_p \prec_\sigma \blacktriangleright_q$. Assume, by way of contradiction, that σ is serializable with respect to $\mathcal{A}_{wo}$. Definition 2.3 then implies that there exists TS σ', which is derivable from σ with respect to $\mathcal{A}_{wo}$ such that all events of one of T_p and T_q appear before all events of the other. If all events of T_p appear before all events of T_q, it implies that the series of interchanges from σ to σ' must include an interchange between the pair $(W_q^t, \blacktriangleright_p)$ which is forbidden by $\mathcal{F}_{wo}$. If all events of T_q appear before those of T_p then the series of interchanges must

include an interchange between the pair $(\blacktriangleleft_p, \blacktriangleright_q)$ which is forbidden by $\mathcal{F}_{wo}$ since $W_q^t \prec_\sigma \blacktriangleright_p$. We therefore conclude that σ is not serializable with respect to $\mathcal{A}_{wo}$, contradicting our assumption.

In the other direction, assume that TS σ is not serializable with respect to $\mathcal{A}_{wo}$. Therefore, no atomic TS can be derived from σ with respect to $\mathcal{A}_{wo}$. Hence, there are two transactions, belonging to different clients, T_p and T_q, that (i) are overlapping in σ, and (ii) moving the events of one to precede those of the other involves violation of $\mathcal{A}_{wo}$. From the definition of $\mathcal{A}_{wo}$ it now follows that for some x, $\blacktriangleleft_p \prec_\sigma W_q^t \prec_\sigma \blacktriangleright_p$ or $W_q^t \prec_\sigma \blacktriangleleft_p \prec_\sigma \blacktriangleright_q$, and, consequently, σ has a writer overlap conflict. □

3. An *eager W-R* conflict (see Fig. 2.8) occurs if for some transactions T_p and T_q a lazy invalidation conflict occurs, or if for some memory address x, we have $W_p^t(x) \prec R_q^t(x) \prec \blacktriangleright_p$. We define the maintaining formula m_{ewr} to be $m_{li} \vee (R_q^t(x) \wedge \ominus W_p^t(x))$. Let $\mathcal{F}_{ewr}$ be the forbidden set that only includes the basic structural restrictions and m_{ewr}, and let $\mathcal{A}_{ewr}$ be its complement.

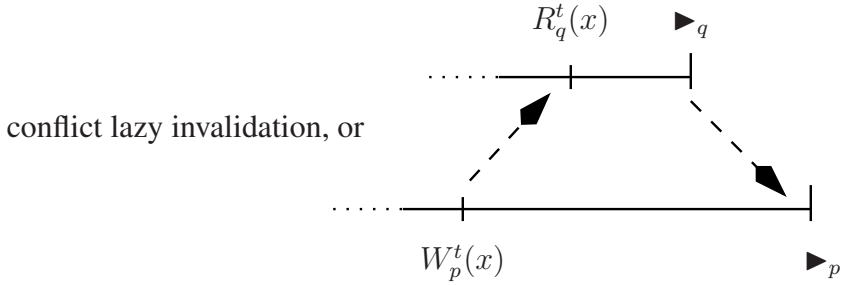


Figure 2.8: Conflict eager W-R.

Theorem 2.8. *A TS σ is serializable with respect to $\mathcal{A}_{ewr}$ iff there is no eager W-R conflict in σ.*

Proof. In one direction, let p and q be clients, and x be a memory location, such that T_p and T_q have a lazy invalidation conflict or $W_p^t(x) \prec R_q^t(x) \prec \blacktriangleright_p$. Assume, by way of contradiction, that σ is serializable with respect to $\mathcal{A}_{ewr}$. Definition 2.3 then implies that there exists TS σ', which is derivable from σ with respect to $\mathcal{A}_{ewr}$ such that all events of one of T_p and T_q appear before all events of the other. If T_p and T_q have a lazy invalidation conflict, by Theorem 2.5 σ is not serializable with respect to $\mathcal{A}_{li}$, and thus not serializable with respect to $\mathcal{A}_{ewr}$, contradicting our assumption. Otherwise $W_p^t(x) \prec R_q^t(x) \prec \blacktriangleright_p$, and note that $\blacktriangleright_q \prec_\sigma \blacktriangleright_p$ must hold. Since we assume interchanges are commit preserving and thus require $\mathcal{F}_{ewr}$ to forbid interchanges of $\blacktriangleright$ events, it follows that $\blacktriangleright_q \prec_{\sigma'} \blacktriangleright_p$, and, consequently, that in σ' all of T_q's events appear before all of T_p's events. This, however, implies that the series of interchanges from σ to σ' must include an interchange between the pair $(W_p^t(x), R_q^t(x))$ which is forbidden by $\mathcal{F}_{ewr}$. We therefore conclude that σ is not serializable with respect to $\mathcal{A}_{ewr}$, contradicting our assumption.

In the other direction, assume that TS σ is not serializable with respect to $\mathcal{A}_{ewr}$. Therefore, no atomic TS can be derived from σ with respect to $\mathcal{A}_{ewr}$. Hence, there are two transactions, belonging to different clients, T_p and T_q, that (i) are overlapping in σ, and (ii) moving the events of one to precede those of the other involves violation of $\mathcal{A}_{ewr}$. From the definition of $\mathcal{A}_{ewr}$ it now follows that for

some x, either T_p and T_q have a lazy invalidation conflict or $W_p^t(x) \prec R_q^t(x) \prec \blacktriangleright_p$, and, consequently, σ has an eager W-R conflict. □

4. An *eager invalidation* conflict (see Fig. 2.9) occurs if for some transactions T_p and T_q an eager W-R conflict occurs, or if for some memory address x, we have $R_p^t(x) \prec W_q^t(x) \prec \blacktriangleright_p$. We define the maintaining formula m_{ei} to be $m_{ewr} \vee (\blacktriangleright_p \wedge \ominus (W_q^t(x) \wedge (\neg \blacktriangleright_p) \mathcal{S} R_p^t(x)))$. Let $\mathcal{F}_{ei}$ be the forbidden set that only includes the basic structural restrictions and m_{ei}, and let $\mathcal{A}_{ei}$ be its complement.

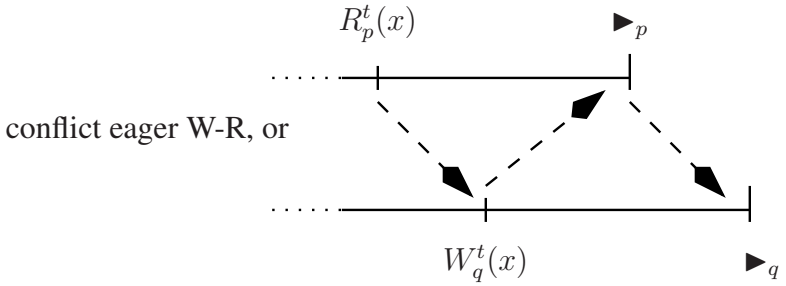


Figure 2.9: Conflict eager invalidation.

Theorem 2.9. *A TS σ is serializable with respect to $\mathcal{A}_{ei}$ iff there is no eager invalidation conflict in σ.*

Proof. In one direction, let p and q be clients, and x be a memory location, such that T_p and T_q have an eager W-R conflict or $R_p^t(x) \prec W_q^t(x) \prec \blacktriangleright_p$. Assume, by way of contradiction, that σ is serializable with respect to $\mathcal{A}_{ei}$. Definition 2.3 then implies that there exists TS σ', which is derivable from σ with respect to $\mathcal{A}_{ei}$ such that all events of one of T_p and T_q appear before all events of the other.

If T_p and T_q have an eager W-R conflict, by Theorem 2.8 σ is not serializable with respect to $\mathcal{A}_{ewr}$, and thus not serializable with respect to $\mathcal{A}_{ei}$, contradicting our assumption. Otherwise $R_p^t(x) \prec W_q^t(x) \prec \blacktriangleright_p$, and note that $\blacktriangleright_p \prec_\sigma \blacktriangleright_q$ must hold. Since we assume interchanges are commit preserving and thus require $\mathcal{F}_{ei}$ to forbid interchanges of $\blacktriangleright$ events, it follows that $\blacktriangleright_p \prec_{\sigma'} \blacktriangleright_q$, and, consequently, that in σ' all of T_p's events appear before all of T_q's events. This, however, implies that the series of interchanges from σ to σ' must include an interchange between the pair $(W_q^t(x), \blacktriangleright_p\ (x))$ which is forbidden by $\mathcal{F}_{ei}$ since $R_p^t(x) \prec W_q^t(x)$. We therefore conclude that σ is not serializable with respect to $\mathcal{A}_{ei}$, contradicting our assumption.

In the other direction, assume that TS σ is not serializable with respect to $\mathcal{A}_{ei}$. Therefore, no atomic TS can be derived from σ with respect to $\mathcal{A}_{ei}$. Hence, there are two transactions, belonging to different clients, T_p and T_q, that (i) are overlapping in σ, and (ii) moving the events of one to precede those of the other involves violation of $\mathcal{A}_{ei}$. From the definition of $\mathcal{A}_{ei}$ it now follows that for some x, either T_p and T_q have an eager W-R conflict or $W_p^t(x) \prec R_q^t(x) \prec \blacktriangleright_p$, and, consequently, σ has an eager invalidation conflict. □

2.2.3 Specification

Let $\mathcal{A}$ be a set of admissible interchanges which we fix for the remainder of this section. We next describe *Spec*$_{\mathcal{A}}$ – a specification of transactional memory that generates all

sequences whose corresponding TSs are serializable with respect to $\mathcal{A}$.

Spec$_{\mathcal{A}}$ is a process that is described as an FDS. In every step, it outputs an element in $E_\perp = E \cup \{\perp\}$ where $\perp$ indicates no observable event. The sequence of outputs it generates, once the $\perp$ elements are projected away, is the set of TSs that are admissible with respect to $\mathcal{A}$. *Spec*$_{\mathcal{A}}$ maintains a queue-like structure $\mathcal{Q}$ to which elements are appended, interchanged, deleted, and removed. The sequence of elements removed from this queue-like structure defines an atomic TS that can be obtained by serialization of *Spec*$_{\mathcal{A}}$'s output with respect to $\mathcal{A}$. *Spec*$_{\mathcal{A}}$ uses the following data structures:

- *spec_mem*: $\mathbb{N} \to \mathbb{N}$ — Persistent memory. Initially for all $i \in \mathbb{N}$, *spec_mem*$[i] = 0$;
- $\mathcal{Q}$: **list over** $E^t \cup \{mark_p\}$ initially empty ($mark_p$ explained later);
- *spec_out*: **scalar in** $E_\perp = E \cup \{\perp\}$ — An output variable, initially $\perp$;
- *spec_doomed*: **array** $[1..n]$ **of booleans** — An array recording which pending transactions are doomed to be aborted. Initially *spec_doomed*$[p] =$ F for every p.

Table 2.1 summarized the steps taken by *Spec*$_{\mathcal{A}}$. The first column describes the value of *spec_out* with each step – it is assumed that every step produces an output. The second column describes the effects of the step on the other variables. The third column describes the conditions under which the step can be taken. We use the following abbreviations in Table 2.1:

- A client p is *pending* if *spec_doomed*$[p] =$ T or if $\mathcal{Q}|_p$ is not empty and does not terminate with $\blacktriangleright_p$;

- a client p is *marked* if $\mathcal{Q}|_p$ terminates with $mark_p$; otherwise, it is *unmarked*;
- a p action a is *locally consistent with* $\mathcal{Q}$ if $\mathcal{Q}|_p, a$ is a prefix of some locally consistent p-transaction;
- a transaction is *consistent with* spec_mem if every $R^t(x, v)$ in it is either preceded by some $W^t(x, v)$, or $v =$ *spec_mem*$[x]$;
- the *update of* spec_mem *by a (consistent) transaction* is *spec_mem*$'$ where for every location x for which the transaction has no $W^t(x, v)$ actions, *spec_mem*$'[x] =$ *spec_mem*$[x]$, and for every memory location x such that the transaction has some $W^t(x, v)$ actions, *spec_mem*$'[x]$ is the value written in the last such action in the transaction;
- an $\mathcal{A}$*-valid transformation to* $\mathcal{Q}$ is a sequence of interchanges of consecutive $\mathcal{Q}$ entries such that each is consistent with $\mathcal{A}$. To apply the transformations, each $mark_p$ is treated as if it is $\blacktriangleright_p$.

The role of *spec_doomed* is to allow *Spec*$_{\mathcal{A}}$ to be implemented with various arbitration policies. It can, at will, schedule a pending transaction to be aborted by setting *spec_doomed*$[p]$to T. The variable *spec_doomed*$[p]$ is reset once the transaction is actually aborted (when *Spec*$_{\mathcal{A}}$ outputs $\not\blacktriangleright_p$). Note that actions of doomed transactions are not recorded on $\mathcal{Q}$.

$mark_p$ is added to $\mathcal{Q}$ just before client p requests to commit its pending transaction. *Spec*$_{\mathcal{A}}$ considers $mark_p$ as a $\blacktriangleright_p$ when trying to shift p's events to the front of $\mathcal{Q}$ using

	spec_out	other updates	conditions
a_1	$\blacktriangleleft_p$	append $\blacktriangleleft_p$ to $\mathcal{Q}$	p is not pending
a_2	$R_p^t(x,v)$	append $R_p^t(x,v)$ to $\mathcal{Q}$	p is pending, unmarked and *spec_doomed*$[p] =$ F; $R(x,v)$ is locally consistent with $\mathcal{Q}$
a_3	$R_p^t(x,v)$	none	p is pending, unmarked and *spec_doomed*$[p] =$ T
a_4	$W_p^t(x,v)$	append $W_p^t(x,v)$ to $\mathcal{Q}$	p is pending, unmarked and *spec_doomed*$[p] =$ F
a_5	$W_p^t(x,v)$	none	p is pending, unmarked and *spec_doomed*$[p] =$ T
a_6	$\not\blacktriangleright_p$	delete p's pending transaction from $\mathcal{Q}$; set *spec_doomed*$[p]$ to F	p is pending
a_7	$\blacktriangleright_p$	replace $mark_p$ with $\blacktriangleright_p$	p has a consistent transaction at the front of $\mathcal{Q}$ that ends with $mark_p$ (p is pending and marked)
a_8	$\perp$	update *spec_mem* according to p's committed transaction; remove p's transaction from $\mathcal{Q}$	p has a consistent committed transaction at the front of $\mathcal{Q}$ (ends with $\blacktriangleright_p$)
a_9	$\perp$	set *spec_doomed*$[p]$ to T; delete all pending p-events from $\mathcal{Q}$	p is pending and *spec_doomed*$[p] =$ F
a_{10}	$\perp$	apply a $\mathcal{A}$-valid transformation to $\mathcal{Q}$	none
a_{11}	$\perp$	append $mark_p$ to $\mathcal{Q}$	p is pending and unmarked
a_{12}	$\perp$	none	none

Table 2.1: Steps of *Spec*$_{\mathcal{A}}$.

interchanges in $\mathcal{A}$. If it succeeds $mark_p$ is replaced by $\mathcal{A}$ (a_7), otherwise p's transaction is aborted.

We assume a fairness requirement, namely, that for every client $p = 1, \ldots, n$, there are infinitely many states of *Spec*$_{\mathcal{A}}$ where $\mathcal{Q}|_p$ is empty and *spec_doomed*$[p] =$ F. This implies that every transaction eventually terminates (commits or aborts). It also guarantees that the sequence of outputs is indeed serializable. Note that progress can always be guaranteed by aborting transactions.

Definition 2.10. A sequence σ over E is *compatible with* Spec$_{\mathcal{A}}$ if it can be obtained by the sequence of *spec_out* that *Spec*$_{\mathcal{A}}$ outputs once all the $\perp$'s are removed.

2.2.4 Implementation

An *implementation TM*: $(\mathit{read}, \mathit{commit})$ of a transactional memory consists of a pair of functions

$$\begin{aligned} \mathit{read} &: \mathit{pref}(TS) \times [1..n] \times \mathbb{N} \to \mathbb{N} \qquad \text{and} \\ \mathit{commit} &: \mathit{pref}(TS) \times [1..n] \to \{\mathit{ack}, \mathit{err}\} \end{aligned}$$

For a prefix σ of a TS, $\mathit{read}(\sigma, p, x)$ is the response (value) of the memory to an accepted $\iota R_p^t(x)$ request immediately following σ, and $\mathit{commit}(\sigma, p)$ is the response (*ack* or *err*) of the memory to a $\iota \blacktriangleright_p$ request immediately following σ.

Definition 2.11. A TS σ is said to be *compatible with* TM if:

1. For every prefix $\eta R_p^t(x, u)$ of σ, $\mathit{read}(\eta, p, x) = u$.

2. For every prefix $\eta \blacktriangleright_p$ of σ, $\mathit{commit}(\eta, p) = ack$.

Definition 2.12. An implementation *TM*: $(\mathit{read}, \mathit{commit})$ is a *correct implementation of a transactional memory with respect to* $\mathcal{A}$ if every TS compatible with *TM* is also compatible with $\mathit{Spec}_{\mathcal{A}}$.

2.2.5 Example: TCC

We now provide an example of transactional memory, which is essentially TCC [HWC+04], and in the following sections verify it using different proof rules and techniques. Its specification is given by $Spec_{\mathcal{A}_{li}}$ where $\mathcal{A}_{li}$ is the admissible set of events corresponding to the lazy invalidation conflict described in Subsection 2.2.2.1. TCC has lazy conflict detection and lazy version control, namely, transactions execute speculatively in the clients' caches and when they commit the memory is updated accordingly. At commit, all pending transactions that contain some read events from addresses written to by the committed transaction are "doomed." A doomed transaction may execute new read and write events in its cache, but it must eventually abort.

Here we present the implementation, TM_1, which uses the following data structures:

- *imp_mem*: $\mathbb{N} \to \mathbb{N}$ — Persistent memory. Initially, for all $i \in \mathbb{N}$, *imp_mem*$[i] = 0$;
- *caches*: **array**$[1..n]$ **of list of** E^t — Caches of clients. For each client $p \in [1..n]$, *caches*$[p]$, initially empty, is a sequence over E^t_p that records the actions of p's pending transaction;
- *imp_out*: **scalar in** $E^t_\perp = E^t \cup \{\perp\}$ — An output variable recording responses to clients, initially $\perp$;
- *imp_doomed*: **array** $[1..n]$ **of booleans** — An array recording which transactions are doomed to be aborted. Initially, *imp_doomed*$[p] =$ F for every $p \in [1..n]$.

TM_1 receives requests from clients, and for each request it updates its state, including updating the output variable *imp_out*, and issues a response to the requesting client. The responses are either an err for syntactic errors (e.g., $\iota\blacktriangleright$ request when there is no pending transaction), an $abort$ for a $\iota\blacktriangleright$ request when the transaction is doomed for abortion, or an acknowledgment whereas for ιR^t requests it is a value in $\mathbb{N}$ and ack for all other cases. Table 2.2 describes the acknowledged actions (events) of TM_1, where for each request we describe the new output value, the other updates to TM_1's state, the conditions under which the updates occur, and the response to the client that issues the request. For now, ignore the comments in the square brackets under the "conditions" column. The last line represents the idle step where no actions occurs and the output is $\perp$.

Comment: Since the requirement of local read-write consistency can be enforced by each client locally, for simplicity, we assume that clients only issue read requests for locations they had not written to in the pending transaction.

	Request	*imp_out*	Other Updates	Conditions	Response
t_1	$\iota\blacktriangleleft_p$	$\blacktriangleleft_p$	append $\blacktriangleleft_p$ to *caches*[p]	[*caches*[p] is empty]	ack
t_2	$\iota R_p^t(x)$	$R_p^t(x,v)$	append $R_p(x,v)$ to *caches*[p]	$v =$ *imp_mem*[x]; [*caches*[p] is not empty] (see comment)	*imp_mem*[x]
t_3	$\iota W_p^t(x,v)$	$W_p^t(x,v)$	append $W_p(x,v)$ to *caches*[p]	[*caches*[p] is not empty]	ack
t_4	$\iota\not\blacktriangleright_p$	$\not\blacktriangleright_p$	set *caches*[p] to empty; set *imp_doomed*[p] to F;	[*caches*[p] is not empty]	ack
t_5	$\iota\blacktriangleright_p$	$\blacktriangleright_p$	set *caches*[p] to empty; for every x and $q \neq p$ such that $W_p^t(x) \in$ *caches*[p] and $R_p^t(x) \in$ *caches*[q] set *imp_doomed*[q] to T; update *imp_mem* by *caches*[p]	*imp_doomed*[p] $=$ F; [*caches*[p] is not empty]; *caches*[p] is consistent with *imp_mem*	ack
t_6	none	$\perp$	none	none	none

Table 2.2: The actions of TM_1.

The specification of Subsection 2.2.3 specifies not only the behavior of the Transactional Memory but also the combined behavior of the memory when coupled with a typical clients module. A generic clients module, *Clients*(n), may, at any step, invoke the next request for client p, $p \in [1..n]$, provided the sequence of E_p^t-events issued so far (including the current one) forms a prefix of a well-formed sequence. The justice requirement of *Clients*(n) is that eventually, every pending transaction issues an ack-ed $\iota\blacktriangleright$ or $\iota\not\blacktriangleright_p$.

Combining modules TM_1 and *Clients*(n) we obtain the complete implementation, defined by:

$$Imp_1 : \quad TM_1 \;|||\; Clients(n)$$

We interpret this composition in a way that combines several of the actions of each of the modules into one. Those actions can be described similarly to those given by Table 2.2, where the first (Request) and last (Response) column are ignored, and the conditions in the brackets are added. The justice requirements of *Clients*(n), together with the observation that both $\iota\not\blacktriangleright$ and an ack-ed $\iota\blacktriangleright$ cause the cache of the issuing client to be emptied, imply that Imp_1's justice requirement is that for every $p = 1, \ldots, n$, *caches*$[p]$ is empty infinitely many times.

2.3 Verification Based on Abstraction Mapping

In this section, we present a proof rule for verifying that an implementation satisfies a transactional memory specification. The premisses of our proof rule can be checked with a model checker, and we demonstrate the method by modeling TCC in TLA$^+$ [Lam02] and proving its correctness with the model checker TLC [Lam02].

2.3.1 A Proof Rule Based on Abstraction Mapping

We start by presenting a proof rule for verifying that an implementation satisfies the specification *Spec*. The approach is an adapted version of the rule presented in [KPSZ02].

To apply the underlying theory, we extend the model of FDS with a new component, $\mathcal{O}$, that defines the *observable variables*. $\mathcal{O} \subseteq V$ and can be externally observed. Also, we omit dealing with compassion (strong fairness) here since we believe compassion is not used in the TM framework. We call the extended model *observable fair discrete system* (OFDS) and assume that both the implementation and the specifications has the form $\mathcal{D} : \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J} \rangle$.

In the current application, we prefer to adopt an *event-based* view of reactive systems, by which the observed behavior of a system is a (potentially infinite) set of events. Technically, this implies that the set of observable variables consists of a single variable O, to which we refer as the *output variable*. It is also required that the domain of O always includes the value $\perp$, implying no observable event. In our case, the domain of

the output variable is $E_\perp = E \cup \{\perp\}$.

Let $\eta : e_0, e_1, \ldots$ be an infinite sequence of $E_\perp$-values. The $E_\perp$-sequence $\widetilde{\eta}$ is called a *stuttering variant* of the sequence η if it can be obtained by removing or inserting finite strings of the form $\perp, \ldots, \perp$ at (potentially infinitely many) different positions within η.

Let $\sigma : s_0, s_1, \ldots$ be a computation of OFDS $\mathcal{D}$. The *observation* corresponding to σ is the $E_\perp$ sequence $s_0[O], s_1[O], \ldots$ obtained by listing the values of the output variable O in each of the states. We denote by $\mathit{Obs}(\mathcal{D})$ the set of all observations of system $\mathcal{D}$.

Let $\mathcal{D}_C$ and $\mathcal{D}_A$ be two systems, to which we refer as the *concrete* and *abstract* systems, respectively. We say that system $\mathcal{D}_A$ *abstracts* system $\mathcal{D}_C$ (equivalently $\mathcal{D}_C$ *refines* $\mathcal{D}_A$), denoted $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$ if, for every observation $\eta \in \mathit{Obs}(\mathcal{D}_C)$, there exists $\widetilde{\eta} \in \mathit{Obs}(\mathcal{D}_A)$, such that $\widetilde{\eta}$ is a stuttering variant of η. In other words, modulo stuttering, $\mathit{Obs}(\mathcal{D}_C)$ is a subset of $\mathit{Obs}(\mathcal{D}_A)$.

Based on the *abstraction mapping* of [AL91], we present in Fig. 2.10 a proof rule that reduces the abstraction problem into a verification problem. There, we assume two comparable OFDS's, a *concrete* $\mathcal{D}_C : \langle V_C, \mathcal{O}_C, \Theta_C, \rho_C, \mathcal{J}_C \rangle$ and an *abstract* $\mathcal{D}_A : \langle V_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A \rangle$, and we wish to establish that $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$. Without loss of generality, we assume that $V_C \cap V_A = \varnothing$, and that there exists a 1-1 correspondence between the concrete observables $\mathcal{O}_C$ and the abstract observables $\mathcal{O}_A$.

The method assumes the identification of an *abstraction mapping* $\alpha : (V_A = \mathcal{E}^\alpha(V_C))$ which assigns to each abstract variable $X \in V_A$ an expression $\mathcal{E}^\alpha_X$ over the concrete variables V_C. For an abstract assertion φ, we denote by $\varphi[\alpha]$ the assertion obtained by replacing each abstract variable $X \in V_A$ by its concrete expression $\mathcal{E}^\alpha_X$. We say that the

abstract state S is an α-image of the concrete state s, written $S = \alpha(s)$, if the values of $\mathcal{E}^{\alpha}$ in s equal the values of the variables V_{A} in S.

$$
\begin{array}{lllll}
\mathbf{A1}. & \Theta_{C} & \rightarrow & \Theta_{A}[\alpha] & \\
\mathbf{A2}. & \mathcal{D}_{C} & \models & \Box(\rho_{C} \rightarrow \rho_{A}[\alpha][\alpha']) & \\
\mathbf{A3}. & \mathcal{D}_{C} & \models & \Box(\mathcal{O}_{C} = \mathcal{O}_{A}[\alpha]) & \\
\mathbf{A4}. & \mathcal{D}_{C} & \models & \Box \Diamond J[\alpha], & \text{for every } J \in \mathcal{J}_{A} \\
\hline
 & & & \mathcal{D}_{C} \sqsubseteq \mathcal{D}_{A} &
\end{array}
$$

Figure 2.10: Rule ABS-MAP.

Premise A1 of the rule states that if s is a concrete initial state, then $S = \alpha(s)$ is an initial abstract state. Premise A2 states that if concrete state s_2 is a ρ_{C}-successor of concrete state s_1, then the abstract state $S_2 = \alpha(s_2)$ is a ρ_{A}-successor of $S_1 = \alpha(s_1)$. Together, A1 and A2 guarantee that, for every run $s_0, s_1, \ldots$ of $\mathcal{D}_{C}$ there exists a run $S_0, S_1, \ldots$ of $\mathcal{D}_{A}$, such that $S_j = \alpha(s_j)$ for every $j \geq 0$. Premise A3 states that the observables of the concrete state s and its α-image $S = \alpha(s)$ are equal. Premise A4 ensures that the abstract fairness requirements (justice) hold in any abstract state sequence which is a (point-wise) α-image of a concrete computation. It follows that every α-image of a concrete computation σ obtained by applications of premises A1 and A2 is an abstract computation whose observables match the observables of σ. This leads to the following claim:

Claim 2.13. If the premises of rule ABS-MAP are valid for some choice of α, then $\mathcal{D}_{A}$ is an abstraction of $\mathcal{D}_{C}$.

2.3.2 Model Checking Using TLC

We verified the correctness of TCC using the explicit-state model checker TLC, the input of which are TLA$^+$ programs. Based on the similarity between TLC and the FDS model, we verified that TCC indeed implements its specification $Spec_{\mathcal{A}_{li}}$.

2.3.2.1 TLA and TLA+

Both TLA [Lam94] and TLA$^+$ [Lam02] have been introduced by Leslie Lamport for the description of reactive and distributed, especially asynchronous, systems. TLA is a logical language for expressing specifications and their properties. It has two levels: a low level containing formulas that describe states and state transitions, and a high level consisting of temporal formulas that are evaluated over infinite sequences of states.

Specifications are written by defining their initial conditions and next-state relations, augmented by liveness and fairness conditions. Abstractness in the sense of information hiding is ensured by quantification over state variables. The refinement problem is solved by allowing stuttering steps that do not modify the values of the state variables of interest; an implementation is allowed to refine such high-level stuttering into lower-level state changes.

An action is an expression containing primed (value in the next state) and un-primed variables. For an action A and a state function t (an expression over constants and unprimed variables), $[A]_t$ denotes $A \vee t' = t$ and $\langle A \rangle_t$ denotes $A \wedge \neg(t' = t)$. Namely, $[A]_t$ requires A to hold only if t changes value during a transition, whereas $\langle A \rangle_t$ requires

that t changes value while A holds true. Formulas $[A]_t$ allow for stuttering: besides state transitions that satisfy A, they also admit any transitions that do not change the state function t. In particular, duplications of states can not be observed by formulas of this form.

TLA$^+$ is a language intended for writing high-level specification of reactive, distributed, and asynchronous systems. It combines TLA and a set theory with choice. It also provides facilities for structuring a specification as a hierarchy of modules, for declaring parameters, and for defining operators. See [Lam02] for a thorough discussion of TLA$^+$.

2.3.2.2 The Model Checker TLC

TLC [Lam02] is an explicit-state model checker that accepts as an input TLA$^+$ specifications of the form:

$$Init \wedge \Box[Next]_{vars} \wedge L$$

where $Init$ is a predicate that defines the initial condition, $Next$ is the next-state action, $vars$ is a tuple of all variables, and L is a temporal formula that usually specifies a liveness condition. In addition to the TLA$^+$ description, TLC is supplied with a configuration file that provides the names of the specification and properties to be checked.

To verify that an implementation correctly implements its specification, one has to provide TLA$^+$ modules for both the specification and the implementation, and a mapping associating each of the specification's variables with an expression over the implementation's variables. TLC then (automatically) verifies that the proposed mapping

is a refinement mapping. Success means that, for the bounded instantiation taken, the implementation module implements the specification, i.e., that every run of the implementation implements some run of the specification, and that every fair computation of the implementation maps into a fair computation of the specification. In the first case, failure is indicated by a finite execution path leading from an initial state into a state in which the mapping is falsified. In the second case, failure is indicated by a finite execution path leading from an initial state to a loop in which the implementation meets all fairness requirements, and the associated specification does not.

Since TLC can handle only finite-state systems, all parameters must be bounded.

2.3.2.3 Model Checking TCC

We verified a bounded instantiation of TCC using TLC. In this subsection we provide two TLA$^+$ modules: The first describes $Spec_{\mathcal{A}_{li}}$ whereas the second describes TCC. The refinement mapping that associates each of the specification's variables with an expression over the implementation's variables is given as part of the implementation module.

Specification Module We constructed the specification module from two submodules, *Spec* and *Spec_Imp*. *Spec* is the heart of the specification and is uniform for all transactional memory specifications. It basically describes the specification of Subsection 2.2.3. *Spec_Imp* mostly defines features that are unique for each transactional memory by means of forbidding interchanges.

To reduce the size of the model, we added a new assumption to $Spec_{\mathcal{A}}$: The action for committing a transaction combines some of $Spec_{\mathcal{A}}$'s actions into one instead of pro-

MODULE *spec* (Declaration of Data Structures)

VARIABLES	$spec\_mem$,	A persistent memory, represented as an array of naturals;
	Q,	A queue of pending events;
	$spec\_out$,	The observable effect of the actions;
	$spec\_doomed$	For each client p, $spec\_doomed[p]$ can be T (doomed), or F;

Figure 2.11: Specification: declaration of data structures.

viding each separately, thus simplifying the proof but preserving the soundness (since the specification is restricted). At commit the module checks whether legal interchanges could bring the transaction to the front of $\mathcal{Q}$. If so, the transaction is removed form $\mathcal{Q}$, conflicting transactions are doomed for abortion, and the memory is updated. This restricts the set of $Spec_{\mathcal{A}}$'s runs but retains soundness. Formally, $TM_1 \sqsubseteq \widetilde{Spec_{\mathcal{A}}}$ implies that $TM_1 \sqsubseteq Spec_{\mathcal{A}}$, where $\widetilde{Spec_{\mathcal{A}}}$ is the restricted specification.

We next present the specification's sections and provide a detailed analysis of each one.

In the first part of *Spec*, Fig. 2.11, the data structures are declared. The second part, provided in Fig. 2.12, defines the initial condition, $Init$. Note that there are several

MODULE *spec* (Initial Condition)

$$
\begin{array}{l}
Init \triangleq \\
\quad \wedge\ spec\_mem \quad = [x \in Adr \mapsto DEF\_VAL] \\
\quad \wedge\ Q \qquad\quad = \langle\rangle \\
\quad \wedge\ spec\_out \quad = \langle \text{“NIL”} \rangle \\
\quad \wedge\ spec\_doomed = [p \in Clients \mapsto \text{FALSE}]
\end{array}
$$

Figure 2.12: Specification: initial condition.

MODULE *spec* (Actions - first part)

$IssueOpen(p) \triangleq$ Open transaction for client p
- $\land \neg ClientHasPendingTranscation(Q,\ p)$
- $\land\ spec\_out' = \langle p,\ \text{“Open”}\rangle$
- $\land\ Q' = Append(Q,\ \langle p,\ \text{“Open”}\rangle)$
- $\land$ UNCHANGED $\langle spec\_mem,\ spec\_doomed\rangle$

$IssueRead(p,\ x,\ v) \triangleq$ Client p reads v from address x, (assuming simplifying assumption)
- $\land\ ClientHasPendingTranscation(Q,\ p)$
- $\land \neg QueueIsMarked(Q)$
- $\land\ spec\_doomed[p] =$ FALSE
- $\land\ spec\_out' = \langle p,\ \text{“Read”},\ x,\ v\rangle$
- $\land\ Q' = Append(Q,\ \langle p,\ \text{“Read”},\ x,\ v\rangle)$
- $\land\ LocalConsistency(Q',\ p)$
- $\land$ UNCHANGED $\langle spec\_mem,\ spec\_doomed\rangle$

$AttemptRead(p,\ x,\ v) \triangleq$ Client p reads v from address x (not added to Q)
- $\land\ ClientHasPendingTranscation(Q,\ p)$
- $\land \neg QueueIsMarked(Q)$
- $\land\ spec\_doomed[p] =$ TRUE
- $\land\ spec\_out' = \langle p,\ \text{“Read”},\ x,\ v\rangle$
- $\land$ UNCHANGED $\langle spec\_mem,\ spec\_doomed,\ Q\rangle$

Figure 2.13: Specification: actions - first part.

constants that were not declared in the first part: Adr, $Clients$, DEF_VAL and NIL. These, and others, are declared in a separate configuration file and are used to define the bounds of the data structures. The initial condition sets each location (address) in $spec\_mem$ to a default value DEF_VAL. It also sets Q to be empty, $spec\_out$ to NIL, and for each client, its index in array $spec\_doomed$ is set to FALSE.

The third, and last part of *Spec*, is provided in Figures 2.13 and 2.14. It defines the actions, beside those required for committing a transaction. $IssueOpen$ accepts a client ID, p, and issues a new open event if p does not have a pending transaction in Q. It sets

MODULE *spec* (Actions - second part)

$IssueWrite(p,\ x,\ v) \triangleq$ Client p writes v to address x
$\quad \land\ ClientHasPendingTranscation(Q,\ p)$
$\quad \land\ \neg QueueIsMarked(Q)$
$\quad \land\ spec\_doomed[p] = \text{FALSE}$
$\quad \land\ spec\_out' = \langle p,\ \text{“Write”},\ x,\ v\rangle$
$\quad \land\ Q' = Append(Q,\ \langle p,\ \text{“Write”},\ x,\ v\rangle)$
$\quad \land\ \text{UNCHANGED}\ \langle spec\_mem,\ spec\_doomed\rangle$

$AttemptWrite(p,\ x,\ v) \triangleq$ Client p writes v to address x (not added to Q)
$\quad \land\ ClientHasPendingTranscation(Q,\ p)$
$\quad \land\ \neg QueueIsMarked(Q)$
$\quad \land\ spec\_doomed[p] = \text{TRUE}$
$\quad \land\ spec\_out' = \langle p,\ \text{“Write”},\ x,\ v\rangle$
$\quad \land\ \text{UNCHANGED}\ \langle spec\_mem,\ spec\_doomed,\ Q\rangle$

$IssueAbort(p) \triangleq$ Abort transaction of client p
$\quad \land\ ClientHasPendingTranscation(Q,\ p)$
$\quad \land\ spec\_out' = \langle p,\ \text{“Abort”}\rangle$
$\quad \land\ Q' = RemoveClientEvents(Q,\ p)$
$\quad \land\ spec\_doomed' = [spec\_doomed\ \text{EXCEPT}\ ![p] = \text{FALSE}]$
$\quad \land\ \text{UNCHANGED}\ \langle spec\_mem\rangle$

$Idle \triangleq$
$\quad \land\ spec\_out' = \langle \text{“NIL”}\rangle$
$\quad \land\ \text{UNCHANGED}\ \langle spec\_mem,\ Q,\ spec\_doomed\rangle$

Figure 2.14: Specification: actions - second part.

$spec\_out$ to $\langle p, \text{“}Open\text{”}\rangle$ ($\blacktriangleleft_p$) and also appends it to Q. $IssueRead$ accepts a client ID, a memory address, x, and a value, v. It issues a new read event if the client already has a pending transaction in Q that is not doomed, Q is not marked (does not contain $mark_p$) and the new event preserves local consistency. It also sets $spec\_out$ to $\langle p, \text{“}Read\text{”}, x, v\rangle$ ($R^t_p(x, v)$) and appends it to the end of Q. $AttemptRead$ issues a new read event if the client's transaction is doomed for abortion. It sets $spec\_out$ to $\langle p, \text{“}Read\text{”}, x, v\rangle$ but

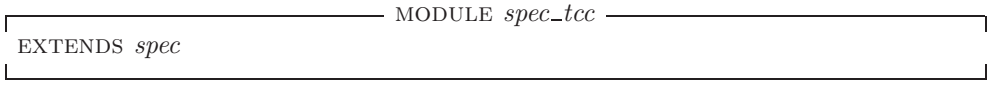


Figure 2.15: Specification: TCC specifics.

does not append a new event to Q. $IssueWrite$ issues a new write event if p already has a pending transaction in Q that is not doomed, and Q is not marked. It also sets $spec\_out$ to $\langle p, \text{“}Write\text{”}, x, v\rangle$ ($W_p^t(x, v)$) and append it to Q. $AttemptWrite$ issues a new write event if the client has a doomed pending transaction. It sets $spec\_out$ to $\langle p, \text{“}Write\text{”}, x, v\rangle$ but does not append it to Q. $IssueAbort$ issues a new abort event if the client has a pending transaction (doomed or not). It sets $spec\_out$ to $\langle p, \text{“}Abort\text{”}\rangle$ ($\not\blacktriangleright_p$), removes all of p's events from Q and sets the index of p in $spec\_doomed$ to FALSE. The last action is $Idle$. It sets $spec\_out$ to NIL and preserves the values of all the other variables.

Fig. 2.15 provides the first part of *Spec_Imp* which defines the module to be an extension of *Spec*.

The second part of *Spec_Imp*, Fig. 2.16, defines the action required for committing a transaction, assuming lazy invalidation conflicts are forbidden. $IssueCommit$ issues a new commit event if the client already has a pending transaction in Q that is not doomed, Q is not marked and the read events are consistent with the memory. Note that if these conditions hold, a transaction may always commit if transactions conflicting with it are doomed. $IssueCommit$ also sets $spec\_out$ to $\langle p, \text{“}Commit\text{”}\rangle$ ($\blacktriangleright_p$), update the memory

```
MODULE spec_tcc (Actions)
IssueCommit(p) ≜  Commit transaction of client p
  ∧ ClientHasPendingTranscation(Q, p)
  ∧ ¬QueueIsMarked(Q)
  ∧ spec_doomed[p] = FALSE
  ∧ LET readEvents ≜ ReadEventsOfClient(Q, p)
    IN  IsConsistent(readEvents, spec_mem)
  ∧ spec_out' = ⟨p, "Commit"⟩
  ∧ LET writeEvents ≜ WriteEventsOfClient(Q, p)
    IN  spec_mem' = (IF (Len(writeEvents) > 0)
                     THEN UpdateSpecMem(spec_mem, writeEvents)
                     ELSE spec_mem)
  ∧ LET MarkedQ ≜ Append(Q, ⟨p, "Commit"⟩)
    IN  ∧ Q' = RemoveClientEventsAndOfConflicting_LIC(MarkedQ, p)
        ∧ spec_doomed' = DoomConflictingTransactions_LIC(Clients, MarkedQ,
                                                         spec_doomed, p)
```

Figure 2.16: Specification: the commit action.

($UpdateSpecMem$) according to the write events ($WriteEvents$), dooms conflicting transactions ($DoomConflictingTransactions\_LIC$ where LIC stands for Lazy Invalidation Conflict), and removes p's events from Q as well as events of transactions conflicting with it ($RemoveClientEventsAndOfConflicting\_LIC$). Note that the admissible interchange set is not used directly to interchange p's events to the front of Q and instead the module uses it to reason which other transactions should be doomed. To capture lazy invalidation conflicts it uses $MarkedQ$, which abbreviates $\langle p, \text{“}Commit\text{”}\rangle$ appended to Q.

The third part of *Spec_Imp*, Fig. 2.17, sets the module's next state transition to one of the possible actions and defines the fairness condition. $Close(p)$ requires client ID as an input since the fairness requirement, “eventually every transaction is closed”, may be

MODULE *spec_tcc* (Transitions)

$a1 \triangleq \lor \exists\, p \in Clients : IssueOpen(p)$
$a2 \triangleq \lor \exists\, p \in Clients : \exists\, x \in Adr : \exists\, v \in Val \cup \{NoVal\} : IssueRead(p,\, x,\, v)$
$a3 \triangleq \lor \exists\, p \in Clients : \exists\, x \in Adr : \exists\, v \in Val \cup \{NoVal\} : AttemptRead(p,\, x,\, v)$
$a4 \triangleq \lor \exists\, p \in Clients : \exists\, x \in Adr : \exists\, v \in Val : IssueWrite(p,\, x,\, v)$
$a5 \triangleq \lor \exists\, p \in Clients : \exists\, x \in Adr : \exists\, v \in Val : AttemptWrite(p,\, x,\, v)$
$a6(p) \triangleq \lor \exists\, p \in Clients : IssueAbort(p)$
$a7\_a8\_a9\_a10\_a11(p) \triangleq \lor \exists\, p \in Clients : IssueCommit(p)$
$a12 \triangleq Idle$

$$Close(p) \triangleq \begin{array}{l} \lor\ a7\_a8\_a9\_a10\_a11(p) \\ \lor\ a6(p) \end{array}$$

$NextClose \triangleq \lor \exists\, p \in Clients : Close(p)$

$$Next \triangleq \begin{array}{l} \lor\ a1 \\ \lor\ a2 \\ \lor\ a3 \\ \lor\ a4 \\ \lor\ a5 \\ \lor\ NextClose \\ \lor\ a12 \end{array}$$

$w \triangleq \langle spec\_mem,\, Q,\, spec\_out,\, spec\_doomed \rangle$

$Fairness \triangleq$ eventually every transaction is closed
$\quad \land \forall\, p \in Clients : \mathrm{WF}_w(Close(p))$

Figure 2.17: Specification: next state transition.

MODULE *spec_tcc* (Specification)

$TCC\_Init \triangleq Init$
$TCC\_Next \triangleq Next$

$TCC\_Spec \triangleq TCC\_Init \land \Box[TCC\_Next]_w \land Fairness$

Figure 2.18: Specification: the complete specification assertion.

MODULE imp_tcc (Declaration of Data Structures)

VARIABLES	imp_mem,	A persistent memory which maintains the values of all commited transactions.
	$caches$,	Array of lists. For each client p, $caches[p]$ keeps the partial actions associated with the most recent pending transaction of p, if one exists.
	imp_out,	An output variable recording responses to clients.
	imp_doomed,	An array recording which transactions are doomed to be aborted.
	$history\_Q$,	Used for the refinement mapping - events that are written to any of the $caches[p]$ are also written to it. When a transaction is closed, its events are also removed from $history\_Q$.
	$counters$	$counters[p]$ the number of events in p's pending transaction

Figure 2.19: Implementation of TCC: declaration of data structures.

satisfied by either an abort or a commit, which are implemented separately.

Fig. 2.18 defines the last part of the specification, including the complete specification assertion, TCC_Spec.

Implementation Module The implementation module *Imp* synchronously composes the memory and the clients, such that every request by a client is immediately responded by the memory.

The abstraction mapping requires that every variable of *Spec* is expressed over *Imp*'s variables. No such expression, however, exists for Q and therefore we add a new auxiliary variable, called $history\_Q$, which is a queue over $E^t_\perp$ and consists of the undoomed pending transactions' events. When new events are issued they are appended to $history\_Q$, and removed when the corresponding transaction is doomed, committed or aborted. We next provide a detailed analysis of each of the implementation's parts.

In the first part of ***Imp***, provided in Fig. 2.19, the data structures are declared. Note that there is an additional array called $counters$. For each client, $counters[p]$ holds the number of events in p's pending transaction and is used for bounding the length of each transaction.

The second part part of ***Imp***, presented in Fig. 2.20, defines the initial condition, $Init$. It sets each location in imp_mem to the default value DEF_VAL, sets the cache of each client to empty, sets imp_out to NIL, sets the relevant index of each client in array imp_doomed to FALSE, initializes $history\_Q$ to empty, sets imp_out to NIL, and initializes the counter of the number of events in each client's transaction to 0.

The third section of ***Imp***, provided in Figures 2.21 and 2.22, defines the possible actions of the memory. $AttemptOpen(p)$ opens a new transaction for client p if the client's cache is empty. It sets imp_out and $caches[p]$ to $\langle p, \text{“}Open\text{”}\rangle$ and appends it to $history\_Q$ as well. $AttemptRead(p, x)$ issues a new read event only if p already has a pending transaction (i.e. $caches[p]$ is not empty). The value read, locally stored in u, is equal to $imp\_mem[x]$ (see comment in Section 2.2.5). $AttemptRead(p, x)$ also sets imp_out to $\langle p, \text{“}Read\text{”}, x, u\rangle$ and appends it to $caches[p]$. If p's transaction is not doomed, i.e. $imp\_doomed[p] = \text{FALSE}$, $\langle p, \text{“}Read\text{”}, x, u\rangle$ is appended to $history\_Q$ as well. $AttemptWrite(p, x, v)$ issues a new write event if the client's cache is empty. It sets imp_out to $\langle p, \text{“}Write\text{”}, x, v\rangle$ and also appends it to $caches[p]$. If p's transaction is not doomed, it appends $\langle p, \text{“}Write\text{”}, x, v\rangle$ to $history\_Q$. $AttemptCommit(p)$ issues a new commit event if the client already has a pending transaction, which is not doomed and consistent with the memory. It updates the memory according to the

MODULE *imp_tcc* (Initial Condition)

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ imp\_mem \quad = [x \in Adr \mapsto DEF\_VAL] \\
\quad \land\ caches \quad = [p \in Clients \mapsto \langle\rangle] \\
\quad \land\ imp\_out \quad = \langle \text{“NIL”} \rangle \\
\quad \land\ imp\_doomed \quad = [p \in Clients \mapsto \text{FALSE}] \\
\quad \land\ history\_Q \quad = \langle\rangle \\
\quad \land\ counters \quad = [p \in Clients \mapsto 0]
\end{array}
$$

Figure 2.20: Implementation of TCC: initial condition.

MODULE *imp_tcc* (Memory Actions - first part)

$$
\begin{array}{l}
AttemptOpen(p) \triangleq \quad \text{Open cachesaction for client } p \\
\quad \land\ caches[p] = \langle\rangle \\
\quad \land\ caches' = [caches \text{ EXCEPT } ![p] = Append(caches[p], \langle p, \text{“Open”} \rangle)] \\
\quad \land\ imp\_out' = \langle p, \text{“Open”} \rangle \\
\quad \land\ history\_Q' = Append(history\_Q, \langle p, \text{“Open”} \rangle) \\
\\
AttemptRead(p, x) \triangleq \quad \text{Client } p \text{ requests to read from address } x \\
\quad \land\ caches[p] \neq \langle\rangle \\
\quad \land \text{ LET } u \triangleq imp\_mem[x] \\
\qquad \text{IN} \quad \land\ caches' = [caches \text{ EXCEPT } ![p] = Append(caches[p], \langle p, \text{“Read”}, x, u \rangle)] \\
\qquad\quad \land\ imp\_out' = \langle p, \text{“Read”}, x, u \rangle \\
\qquad\quad \land \text{ IF } (imp\_doomed[p]) \\
\qquad\qquad \text{THEN } history\_Q' = history\_Q \\
\qquad\qquad \text{ELSE } history\_Q' = Append(history\_Q, \langle p, \text{“Read”}, x, u \rangle) \\
\\
AttemptWrite(p, x, v) \triangleq \quad \text{Client } p \text{ requests to write } v \text{ in address } x \\
\quad \land\ caches[p] \neq \langle\rangle \\
\quad \land\ caches' = [caches \text{ EXCEPT } ![p] = Append(caches[p], \langle p, \text{“Write”}, x, v \rangle)] \\
\quad \land\ imp\_out' = \langle p, \text{“Write”}, x, v \rangle \\
\quad \land \text{ IF } (imp\_doomed[p]) \\
\qquad \text{THEN } history\_Q' = history\_Q \\
\qquad \text{ELSE } history\_Q' = Append(history\_Q, \langle p, \text{“Write”}, x, v \rangle)
\end{array}
$$

Figure 2.21: Implementation of TCC: memory actions - first part.

```
MODULE imp_tcc (Memory Actions - second part)
AttemptCommit(p) ≜  Client p requests to Commit
  ∧ caches[p] ≠ ⟨⟩
  ∧ imp_doomed[p] = FALSE
  ∧ IsConsistentWithMem(caches[p], imp_mem)
  ∧ LET WriteEvents ≜ GetWriteEvents(caches[p])
    IN  imp_mem' = IF (Len(WriteEvents) > 0)
                   THEN UpdateMem(imp_mem, WriteEvents)
                   ELSE imp_mem
  ∧ imp_out' = ⟨p, "Commit"⟩
  ∧ imp_doomed' = DoomConflictingTransactions_LIC(caches, imp_doomed, p)
  ∧ caches' = [caches EXCEPT ![p] = ⟨⟩]
  ∧ history_Q' = RemoveClientEventsAndOfConflicting_LIC(caches, history_Q, p)

AttemptAbort(p) ≜  Abort the transaction of client p
  ∧ caches[p] ≠ ⟨⟩
  ∧ imp_out' = ⟨p, "Abort"⟩
  ∧ imp_doomed' = [imp_doomed EXCEPT ![p] = FALSE]
  ∧ caches' = [caches EXCEPT ![p] = ⟨⟩]
  ∧ history_Q' = RemoveClientEvents(history_Q, p)

Idle ≜  An idle step
  ∧ imp_out' = ⟨"NIL"⟩
```

Figure 2.22: Implementation of TCC: memory actions - second part.

write events, sets imp_out to $\langle p, \text{“}Commit\text{”}\rangle$, initializes the cache of client p to empty, dooms all other pending transactions that conflicts with that of p, and removes from $history\_Q$ all p's events as well as events of clients which their transactions are doomed. $AttemptAbort(p)$ issues a new abort event only if the client has a pending transaction. It sets imp_out to $\langle p, \text{“}Abort\text{”}\rangle$, sets its own doomed bit to FALSE, initializes its cache to empty and removes all its events from $history\_Q$. The last action is $Idle$. It sets imp_out to NIL and preserves the values of the other variables.

MODULE *imp_tcc* Transitions - first part

$t1(p) \triangleq$ Open transaction
$\land\ counters[p] = 0$
$\land\ counters' = [counters \text{ EXCEPT } ![p] = 1]$
$\land\ AttemptOpen(p)$
$\land$ UNCHANGED $\langle imp\_mem,\ imp\_doomed \rangle$

$t2(p,\ x) \triangleq$ Read
$\land\ counters[p] > 0$
$\land\ counters[p] < TMsize$
$\land\ NoPreviousWriteEvents(history\_Q,\ p)$
$\land\ AttemptRead(p,\ x)$
$\land\ counters' = [counters \text{ EXCEPT } ![p] = counters[p] + 1]$
$\land$ UNCHANGED $\langle imp\_mem,\ imp\_doomed \rangle$

$t3(p,\ x,\ v) \triangleq$ Write
$\land\ counters[p] > 0$
$\land\ counters[p] < TMsize$
$\land\ AttemptWrite(p,\ x,\ v)$
$\land\ counters' = [counters \text{ EXCEPT } ![p] = counters[p] + 1]$
$\land$ UNCHANGED $\langle imp\_mem,\ imp\_doomed \rangle$

Figure 2.23: Implementation of TCC: transitions - first part.

The fourth section of *Imp*, provided in Figures 2.23 and 2.24, defines the transitions of the implementation. Each one of these transitions requires a matching enabled action of the memory thus when combined together with a client's request, forms a single event. $t1(p)$ opens a new transaction for client p if the client does not have a pending transaction, i.e., the counter of the number of events in the client's current pending transaction is 0. It calls $AttemptOpen(p)$, the matching action of the memory for opening a new transaction. $t2(p)$ issues a new read transition for client p if it already has a pending transaction ($counters[p]$ is greater than 0) with less than $TMsize$ events. This restric-

```
MODULE imp_tcc  Transitions - second part
t4(p) ≜  Abort transaction
  ∧ counters[p] > 0
  ∧ AttemptAbort(p)
  ∧ counters' = [counters EXCEPT ![p] = 0]
  ∧ UNCHANGED ⟨imp_mem⟩

t5(p) ≜  Commit transaction
  ∧ counters[p] > 0
  ∧ AttemptCommit(p)
  ∧ counters' = [counters EXCEPT ![p] = 0]

t6 ≜
  ∧ Idle
  ∧ UNCHANGED ⟨imp_mem, caches, history_Q, counters, imp_doomed⟩
```

Figure 2.24: Implementation of TCC: Transitions - second part.

tion bounds the length of the transactions and allows model checking. $t2(p)$ follows the simplified assumption of Section 2.2.5, requiring that no write events have been issued before. It finally increases the counter by 1. The analysis of the other actions is very similar.

The last part, provided in Fig. 2.25, sets the next state in terms of the possible transitions and defines the fairness condition. It also issues an instance of the specification and constructs a abstraction mapping from expressions over *Imp*'s variables to the variables of *Spec*. The mapping used for verifying TCC is the most trivial one – all variables are mapped one-to-one. The last line of the implementation tells TLC to check that the specification of *Spec*'s instance holds when the *Spec*'s variables are expressed using the abstraction mapping.

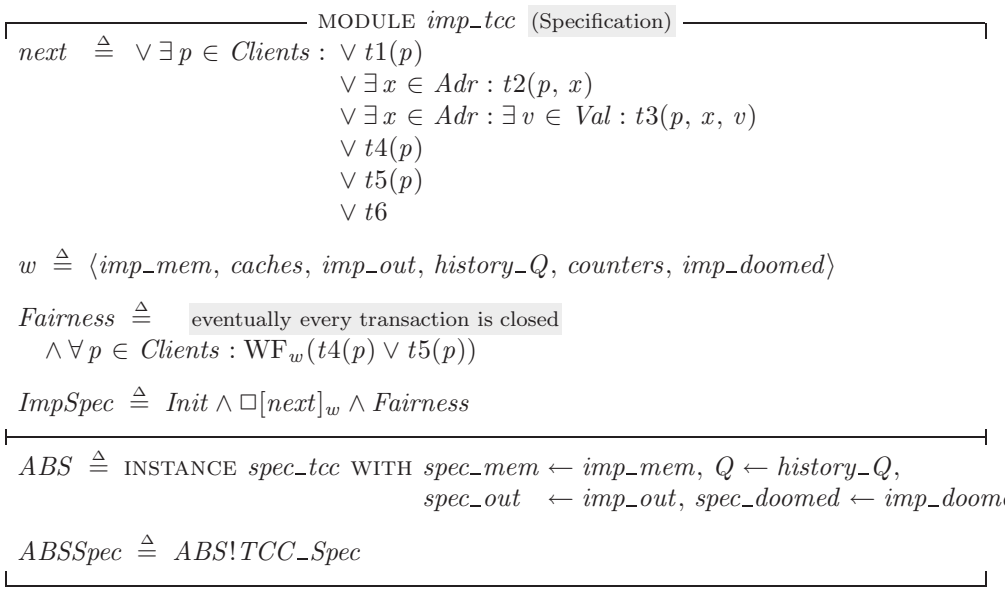


Figure 2.25: Implementation of TCC: the complete specification assertion.

TLC verified that mapping is a refinement mapping and verified that for the bounded instantiation taken, the implementation module implements the specification.

2.4 Verification Based on Abstraction Relation

It is not always possible to relate abstract to concrete states by a functional correspondence which maps each concrete state to a unique abstract state. In many cases, we cannot find an abstraction mapping, but can identify an *abstraction relation* $R(V_C, V_A)$ (which induces a relation $R(s, S)$). In this section we present a new proof rule that only assumes an abstraction relation and shows how it can be used to verify the correctness

of TCC.

2.4.1 A Proof Rule Based on an Abstraction Relation

In Fig. 2.26, we present proof rule ABS-REL which only assumes an *abstraction relation* $R(V_C, V_A)$ between the concrete and abstract states.

$$\begin{array}{lllll}
\textbf{R1}. & \Theta_C & \rightarrow & \exists V_A : R \wedge \Theta_A & \\
\textbf{R2}. & \mathcal{D}_C & \models & \Box(R \wedge \rho_C \rightarrow \exists V'_A : R' \wedge \rho_A^+) & \\
\textbf{R3}. & \mathcal{D}_C & \models & \Box(R \rightarrow \mathcal{O}_C = \mathcal{O}_A) & \\
\textbf{R4}. & \mathcal{D}_C & \models & \Box \Diamond (\forall V_A : R \rightarrow J), & \text{for every } J \in \mathcal{J}_A \\
\hline
 & & & \mathcal{D}_C \sqsubseteq \mathcal{D}_A &
\end{array}$$

Figure 2.26: Rule ABS-REL.

Premise R2 of the rule allows a single concrete transition to be emulated by a sequence of abstract transitions. This is done via the transitive closure ρ_A^+ which is defined as follows. Let $S_0, S_1, \ldots, S_k$, $k > 0$, be a sequence of abstract states, such that $\langle S_i, S_{i+1} \rangle \models \rho_A$ for every $i \in [0..k-1]$, and for some $\ell \in [1..k]$, for every $i \in [1..k]$, if $i \neq \ell$ then $S_i[\mathcal{O}] = \perp$. Then $\langle S_0, \widetilde{S}_k \rangle \models \rho_A^+$, where $\widetilde{S}_k = S_k[\mathcal{O} := S_\ell[\mathcal{O}]]$ is obtained from S_k by assigning the variable $\mathcal{O}$ (the single output variable) the value that it has in state S_ℓ. This definition allows to perform first some "setting up" transitions that have no externally observable events, followed by a transition that produces a non-trivial observable value, followed by a finite number of "clean-up" transitions that again have no

externally observable events. The observable effect of the composite transition is taken to be the observable output of the only observable transition in the sequence.

Premise R1 of the rule states that for every initial concrete state s, it is possible to find an initial abstract state $S \models \Theta_{A}$, such that $\langle s, S \rangle \models R$.

Premise R2 states that for every pair of concrete states, s_1 and s_2, such that s_2 is a ρ_{C}-successor of s_1, and an abstract state S_1 which is a R-related to s_1, it is possible to find an abstract state S_2 such that S_2 is R-related to s_2 and is also a ρ_{A}^{+}-successor of S_1. Together, R1 and R2 guarantee that, for every run $s_0, s_1, \ldots$ of $\mathcal{D}_{C}$ there exists a run $S_0, \ldots, S_{i_1}, \ldots, S_{i_2}, \ldots$, of $\mathcal{D}_{A}$, such that for every $j \geq 0$, S_{i_j} is R-related to s_j and all abstract states S_k, for $i_j < k < i_{j+1}$, have no observable variables. Premise R3 states that if abstract state S is R-related to the concrete state s, then the two states agree on the values of their observables. Premise R4 ensures that the abstract fairness requirements (justice) hold in any abstract state sequence which is a (point-wise) R-related to a concrete computation. It follows that every sequence of abstract states which is R-related to a concrete computation σ and is obtained by applications of premises R1 and R2 is an abstract computation whose observables match the observables of σ. This leads to the following claim:

Claim 2.14. If the premises of rule ABS-REL are valid for some choice of R, then $\mathcal{D}_{A}$ is an abstraction of $\mathcal{D}_{C}$.

2.4.2 Verifying TCC

The model checker TLC does not enable to apply a relation between the abstract and concrete systems and therefore the proof rule of Subsection 2.4.1 cannot be applied. As an alternative we now sketch a proof, using rule ABS-REL, showing that $\mathit{Imp}_1 \sqsubseteq \mathit{Spec}_{\mathcal{A}_{li}}$. Later, in Subsection 2.5.3, we explain how an adjusted version of this rule can be applied when using a theorem prover.

The application of rule ABS-REL requires the identification of a relation R which holds between concrete and abstract states. We use the relation R defined by:

$$\mathit{spec\_out} = \mathit{imp\_out} \ \wedge \ \mathit{spec\_mem} = \mathit{imp\_mem} \ \wedge \ \mathit{spec\_doomed} = \mathit{imp\_doomed}$$
$$\wedge \ \textstyle\bigwedge_{p=1}^{n} \neg \mathit{imp\_doomed}[\mathrm{p}] \longrightarrow (\mathcal{Q}|_p = \mathit{caches}[p])$$

The relation R stipulates equality between *spec_out* and *imp_out*, between *spec_mem* and *imp_mem* and between *spec_doomed* and *imp_doomed*, and that, for each $p \in [1..n]$ that its transaction is not doomed, the projection of $\mathcal{Q}$ on the set of events pertinent to client p equals $\mathit{caches}[p]$.

To simplify the proof, we assume (see comment in Section 2.2.5) that all transactions have the form

$$\blacktriangleleft_p \ R_p^t(x_1, u_1) \cdots R_p^t(x_r, u_r) W_p^t(y_1, v_1) \cdots W_p^t(y_w, v_w) \{\blacktriangleright_p, \not\blacktriangleright_p\}$$

It is not difficult to see that premise R1 of rule ABS-REL holds, since the two initial conditions are given by

$$\begin{array}{lllll}
\Theta_C: & \bigwedge_{p=1}^{n}(\mathit{imp_doomed}[p] = \text{F}) & \wedge & \bigwedge_{p=1}^{n}(\mathit{caches}[p] = \Lambda) & \wedge \\
& \mathit{imp_mem} = \lambda i.0 & \wedge & \mathit{imp_out} = \bot & \\
\Theta_A: & \bigwedge_{p=1}^{n}(\mathit{spec_doomed}[p] = \text{F}) & \wedge & \mathcal{Q} = \Lambda & \wedge \\
& \mathit{spec_mem} = \lambda i.0 & \wedge & \mathit{spec_out} = \bot &
\end{array}$$

and the relation R guarantees equality between the relevant variables.

We will now examine the validity of premise R2. This can be done by considering each of the concrete transitions $t_1, \ldots, t_6$.

t_1. Transition t_1 appends the event $\blacktriangleleft_p$ to an empty *caches*$[p]$ and outputs it to *imp_out*. Note that *caches*$[p]$ is initially empty or set to empty only by t_4 and t_5, whereas a precondition for t_4 is that *imp_doomed*$[p] =$ F and t_5 by itself sets *imp_doomed*$[p]$ to F. Therefore an empty *caches*$[p]$ implies *imp_doomed*$[p] =$ F. This can be emulated by an instance of abstract transition a_1 which outputs $\blacktriangleleft_p$ to *spec_out* and places this event at the end of $\mathcal{Q}$. The precondition for a_1 is that client p is not pending, i.e. *spec_doomed*$[p] =$ F and $\mathcal{Q}|_p$ is empty, which is guaranteed by R. It can be checked that this joint action preserves the relation R, in particular, $\bigwedge_{p=1}^{n} \neg$*imp_doomed*[p] $\longrightarrow (\mathcal{Q}|_p =$ *caches*$[p])$.

t_2. Transition t_2 appends to *caches*$[p]$ (and outputs) the event $R_p^t(x, v)$. The preconditions are that *caches*$[p]$ is not empty and that $v =$ *imp_mem*$[x]$. If *imp_doomed*$[p] =$

F, this can be matched by an instance of abstract transition a_2, if p is unmarked. The preconditions for abstract transition a_2 are that client p is pending, p is unmarked, *spec_doomed*$[p] =$ F and that $R_p^t(x, v)$ is locally consistent with $\mathcal{Q}$. Since we assume (in the implementation) that clients only issue read requests for locations they had not written to in the pending transaction, the requirement for local read-write consistency is always (vacuously) satisfied. All other preconditions are guaranteed by R. If *imp_doomed*$[p] =$ T, this can be matched by an instance of abstract transition a_3. The preconditions for a_3 are that client p is pending, p is unmarked and *spec_doomed*$[p] =$ T, which are guaranteed by R. The joint action preserves the relation R for both cases.

t_3. Transition t_3 appends to *caches*$[p]$ (and outputs) the event $W_p^t(y, v)$. It is matched by an instance of either a_4 or a_5. The rationalization is very similar to the previous case.

t_4. Transition t_4 aborts the transaction pending in *caches*$[p]$ while outputting the event $\not\blacktriangleright_p$. The precondition is that *caches*$[p]$ is not empty. The transition clears *caches*$[p]$ and sets *imp_doomed*$[p] =$ F. It is matched with the abstract transition a_6 which outputs the event $\not\blacktriangleright_p$, removes all of p's events from $\mathcal{Q}$ and sets *spec_doomed*$[p] =$ F. The precondition for a_6 is that client p is pending, i.e. *spec_doomed*$[p] =$ T or $\mathcal{Q}|_p$ is not empty. Since *caches*$[p]$ is not empty, if *spec_doomed*$[p] =$ F then $\mathcal{Q}|_p$ is not empty. If *spec_doomed*$[p] =$ T, the precondition trivially holds. The joint action preserves the relation R.

t_5. Transition t_5 commits the current transaction contained in *caches*$[p]$ while outputting the event $\blacktriangleright_p$. This is possible only if *imp_doomed*$[p] = \text{F}$ and p has a pending transaction, which is consistent with *imp_mem*. Otherwise, if one of the read events does not match the value of the relevant memory address, there must have been an update by a transaction that committed after the read, which would have also set *imp_doomed*$[p] = \text{T}$. Transition t_5 also updates *imp_mem* according to *caches*$[p]$, clears *caches*$[p]$ and sets *imp_doomed*$[q] = \text{T}$ if there exist W_p^t and R_q^t that access the same location.

The emulation of this transition begins by an instance of a_{11} which appends $mark_p$ to $\mathcal{Q}$, followed by a sequence of applications of abstract transition a_{10} which attempt to move all the events of p to the front of $\mathcal{Q}$, where $\mathcal{A}$ corresponds to the lazy invalidation conflict, i.e. forbid interchanging a $\blacktriangleright_p$ on the right with $R_q^t(x)$ if $\mathcal{Q}$ also contains $W_p^t(x)$. To resolve conflicts abstract transition a_9 is applied, setting *spec_doomed*$[q] = \text{T}$ for all q such that $\mathcal{Q}$ contains R_q^t and W_p^t that access the same location. When p has a consecutive transaction at the front of $\mathcal{Q}$, we apply abstract transition a_7 which confirms that $\mathcal{Q}|_p$ is consistent with *spec_mem* (must be true due to the R-conjunct *spec_mem* $=$ *imp_mem*), outputs $\blacktriangleright_p$, and replaces $mark_p$ with $\blacktriangleright_p$. Finally we apply a_8 which updates *spec_mem* according to $\mathcal{Q}|_p$ (thus making it again equal to *imp_mem*), and removes all elements of p from $\mathcal{Q}$, thus reestablishing the R-conjunct $\bigwedge_{p=1}^{n} \neg$*imp_doomed*[p] $\longrightarrow$ $(\mathcal{Q}|_p =$ *caches*$[p])$.

t_6. The idling concrete transition t_6 may be emulated by the idling abstract transition

a_{11}.

The R-conjunct *spec_out* $=$ *imp_out* guarantees the validity of premise R3. It remains to verify premise R4. This premise requires showing that any concrete computation that visits infinitely many times states satisfying $\forall V_{_A} : R \rightarrow J_{_A}$, where $J_p : \mathcal{Q}|_p = \Lambda \wedge$ *spec_doomed*$[p] =$ F, characterizes the set of abstract states in which the queue contains no E_p^t events and p does not have a pending transaction which is doomed to be aborted. Since R requires that $\mathcal{Q}|_p =$ *caches*$[p]$, we obtain that Premise R4 is valid.

Note that ABS-MAP does not suffice to construct step t_5, where the power of ABS-REL is demonstrated. We obtained a similar proof for a bounded instantiation using TLC, however, there *Spec* is defined as performing "meta-steps," without which TLC, that uses an ABS-MAP-like rule, cannot construct the relations ABS-REL does.

2.5 Mechanical Verification

In Section 2.3 we described the verification of TCC with the (explicit-state) model checker TLC. It has several drawbacks: TLC does not allow to specify abstraction relations between states, but rather mappings between variables thus a proof rule that is weaker than ABS-REL is used. For example, the relation $\mathcal{Q}|_p =$ *caches*$[p]$, that equates the projection of the specification's queue on a certain client to the client's cache in the implementation, cannot be expressed in TLA$^+$. We consequently use an auxiliary queue to record the order in which events are invoked in the implementation and map it to Q. Also, TLC can only verify finite instances of the system, with rather small numbers chosen for

memory size, number of clients, number of pending transactions, etc. Thus far, we have no "small number" properties of such systems, and the positive results can only serve as a sanity check. Hence, to obtain a full mechanical verification we have to use a tool that is based on deductive techniques.

Our tool of choice is TLPVS [PA03], which was developed to reduce the substantial manual effort required to complete deductive temporal proofs of reactive systems. It embeds temporal logic and its deductive framework within the high-order theorem prover, PVS [OSRSC99]. It includes a set of theories defining linear temporal logic (LTL), proof rules for proving soundness and response properties, and strategies that aid in conducting the proofs. In particular, it has a special framework for verifying unbounded systems and theories. See [OSRSC99] and [PA03] for thorough discussions for proving with PVS and TLPVS, respectively.

Since, however, TLPVS only supports the model of *parameterized fair systems* (PFS) [PA03], we formulate a new computational model in the PVS specification language. We then define a new version of the rule ABS-REL that is cast in terms of the formal framework used in TLPVS. We provide a mechanical verification of the adapted proof rule and of three implementations from the literature, including TCC.

2.5.1 Observable Parameterized Fair Systems

For working with TLPVS we use *observable parameterized fair systems* (OPFS) as the computational model. It extends the model PFS with an *observation domain*. PFS is

a variation of FDS and the differences are due to the need to embed LTL and its proof theory within PVS. This representation allows the verification of parameterized systems where the size of the system is kept symbolic. The change from a PFS to the OPFS model is to allow verifying refinement relations between concrete and abstract systems and comparing their observables. Under OPFS, a system $\mathcal{D} : \langle \Sigma, d_{\mathcal{O}}, \mathcal{O}, \Theta, \rho, \mathcal{F}, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components:

- Σ — A non-empty set of *system states*. Typically, a state is structured as a record whose fields are typed *system variables*, V.
- $d_{\mathcal{O}}$ — A non-empty *observation domain* which is used for observing the external behavior of the states.
- $\mathcal{O}$ — An *observation function*. A mapping from Σ to $d_{\mathcal{O}}$.
- Θ — An *initial condition*: A predicate characterizing the initial states.
- $\rho(s, s')$ — A *transition relation*: A *bi-predicate*, relating the state s to state s' – a $\mathcal{D}$-successor of s. We assume that every state has a ρ-successor.
- $\mathcal{F}$ — A non-empty *fairness domain* which is used for parameterizing the justice and compassion fairness requirements.
- $\mathcal{J}$ — A mapping from $\mathcal{F}$ to predicates. For each $f \in \mathcal{F}$, $\mathcal{J}(f)$ is a *justice* (*weak fairness*) requirement (predicate). For each $f \in \mathcal{F}$, a computation must include infinitely many states satisfying $\mathcal{J}(f)$.

- $\mathcal{C}$ — A mapping from $\mathcal{F}$ to pairs of predicates, called a *compassion (strong fairness)* requirement. For each $f \in \mathcal{F}$, let $\mathcal{C}(f) = \langle p, q \rangle$ be the compassion requirement corresponding to f. It is required that if a computation contains infinitely many p-states, then it should also contain infinitely many q-states. We present compassion so to have a complete model but we omit dealing with it later since we believe it is not used in the TM framework.

The definitions of a run and computation are the same as those in Section 1.1.

We formulate OPFS in PVS specification language as follows:

```
OPFS: TYPE =
   [# initial:   PREDICATE,
      rho:       BI_PREDICATE,
      obs:       OBSERVABLE,
      justice:   JUSTICE_TYPE #]
```

where

```
PREDICATE:     TYPE = [STATE → bool]
BI_PREDICATE:  TYPE = [STATE, STATE → bool]
OBSERVABLE:    TYPE = [STATE → OBSERVABLE_DOMAIN]
JUSTICE_TYPE:  TYPE = [FAIRNESS_DOMAIN → PREDICATE]
```

When importing theory OPFS, the user must instantiate it with STATE, FAIRNESS_DOMAIN and OBSERVABLE_DOMAIN. These are uninterpreted types that must be defined by the user for each system. The other components are identical to those of PFS, and we refer the reader to [PA03] for further details.

2.5.2 An Adjusted Proof Rule

Working with TLPVS required us to cast the proof rule in terms of a different formal framework, and to capture a general stuttering equivalence as opposed of assuming stuttering only on the part of the specification.

To apply the underlying theory, we assume that both the implementation and the specifications are represented as OPFSs. The observation function is then defined by $\mathcal{O}(s) = s[O]$.

$$
\begin{array}{lllll}
\textbf{R1}. & \Theta_{_C}(s) & \rightarrow & \exists S : R(s,S) \ \wedge \ \Theta_{_A}(S) & \\
\textbf{R2}. & \mathcal{D}_{_C} & \models & \Box\,(R(s,S) \ \wedge \ \rho_{_C}(s,s') \quad \rightarrow \quad \exists S' : R(s',S') \ \wedge \ \rho_{_A}(S,S')) & \\
\textbf{R3}. & \mathcal{D}_{_C} & \models & \Box\,(R(s,S) \ \rightarrow \ \mathcal{O}_{_C}(s) = \mathcal{O}_{_A}(S)) & \\
\textbf{R4}. & \mathcal{D}_{_C} & \models & \Box \Diamond\, (\forall S : R(s,S) \rightarrow \mathcal{J}(f)(S)), & \text{for every } f \in \mathcal{F}_{_A} \\
\hline
 & & & \mathcal{D}_{_C} \sqsubseteq \mathcal{D}_{_A} &
\end{array}
$$

Figure 2.27: (Adjusted) Rule ABS-REL.

The method advocated by the rule of Fig. 2.27 assumes the identification of an *abstraction relation* $R(s,S) \subseteq \Sigma_{_C} \times \Sigma_{_A}$. If the relation $R(s,S)$ holds, we say that the abstract state S is an R-image of the concrete state s.

Premise R1 of the rule states that for every initial concrete state s, it is possible to find an initial abstract state $S \models \Theta_{_A}$, such that $R(s,S) = \text{T}$. Premise R2 states that for every pair of concrete states, s and s', such that s' is a $\rho_{_C}$-successor of s, and an abstract state S which is a R-related to s, there exists an abstract state S' such that S' is R-related

to s' and is also a ρ_A-successor of S. Together, R1 and R2 guarantee that, for every run $s_0, s_1, \ldots$ of $\mathcal{D}_C$ there exists a run $S_0, S_1, \ldots,$ of $\mathcal{D}_A$, such that for every $j \geq 0$, S_j is R-related to s_j. Premise R3 states that if abstract state S is R-related to the concrete state s, then the two states agree on the values of their observables. Together with the previous premises, we conclude that for every σ, a run of $\mathcal{D}_C$, there exists a corresponding run of $\mathcal{D}_A$, which has the same observation as σ. Premise R4 ensures that the abstract justice requirements hold in any abstract state sequence which is a (point-wise) R-related to a concrete computation. It follows that every sequence of abstract states which is R-related to a concrete computation σ and is obtained by applications of premises R1 and R2 is an abstract computation whose observables match the observables of σ. This leads to the following claim which was proved using TLPVS:

Claim 2.15. If the premises of rule ABS-REL are valid for some choice of R, then $\mathcal{D}_A$ is an abstraction of $\mathcal{D}_C$.

A new theory that constructs the proof rule was defined. It assumes two OPFSs, one for the abstract system (`abs`) and another for the concrete system (`con`). We next illustrate a couple of the theory's functions that are used to define abstraction:

```
observable_equivalent(seq_c,obs_c,seq_a,obs_a): bool=
    (FORALL t:
       obs_c(seq_c(t)) = obs_a(seq_a(t)))
```

Function `observable_equivalent` accepts a sequence and an observable for each of the systems (concrete and abstract). It then checks if the observables are equal for every

corresponding (by position) states of the concrete and abstract sequences.

Function `abstract` accepts two OPFSs, `abs` and `con`. It checks whether for each computation of `con` there exists a computation of `abs` such that the observables of every corresponding states are equal (using `observable_equivalent`).

```
abstract(con,abs): bool=
    (FORALL seq_c: computation(seq_c,con) IMPLIES
       (EXISTS seq_a: computation(seq_a,abs) AND
          observable_equivalent(seq_c,con`obs,seq_a,abs`obs)))
```

Note that this version of abstraction is simpler since it assumes no stuttering.

TLPVS has all temporal operators implemented, including $\Diamond$ (Eventually) and $\Box$ (Always), which are denoted by `F` and `G`, respectively. It also has a function called `is_P_valid` that accepts a temporal operator and checks whether it holds in the first state of every computation. Let `f` be a `FAIRNESS_DOMAIN`, `s_c` a concrete state, `s_a` an abstract state and `rel` a relation between an abstract and a concrete states. The rule of Fig. 2.27 is defined in TLPVS as follows:

```
ABS_REL: LEMMA
  ((FORALL st_c:
       con`initial(st_c) IMPLIES
          EXISTS st_a:
              rel(st_c, st_a) AND
              abs`initial(st_a)))

    AND (is_P_valid(G(LAMBDA seq_c, t:
                        (FORALL st_a, st_cp:
                           ((rel(seq_c(t), st_a) AND
                             con`rho(seq_c(t), st_cp))) IMPLIES
                              (EXISTS st_ap:
                                  (rel(st_cp, st_ap) AND
                                   abs`rho(st_a, st_ap))))), con))

    AND (is_P_valid(G(LAMBDA seq_c, t:
                        (FORALL st_a:
                           ((rel(seq_c(t), st_a) IMPLIES
                                con`obs(seq_c(t)) = abs`obs(st_a))))), con)))
    AND (is_P_valid(G(F(LAMBDA seq_c, t:
                        (FORALL st_a, f:
                           rel(seq_c(t), st_a) IMPLIES
                              abs`justice(f)(st_a)))))), con)))
   IMPLIES abstract(con, abs)
```

The soundness of the rule was proved in TLPVS and is provided in [CPZ08].

2.5.3 Mechanical Verification Using TLPVS

In this section we explain how to use TLPVS for verifying the correctness of transactional memory implementations. We provide a framework and illustrate how it is applied for

proving the correctness of TCC. The framework was used to prove the correctness of additional implementations (Subsections 2.5.3, 2.5.3 and 2.6.2) and may also be used to verify other implementations from the literature.

Framework for Verifying Transactional Memories

Several theories were defined so to enable the verification of transactional memory implementations in TLPVS. The most basic one is `Event` which defines a new type for capturing events and a set of utility functions for getting the event's ID, type, value etc. Another theory defines lists and allows to capture the queue-like structures used in both specification and implementation. This theory requires, in addition to the regular list operations, the definition of projection (|), which, in turn, requires the proofs of several auxiliary lemmas. The data structure list is defined as follows:

```
list: TYPE = [# size: nat, entry: [posnat -> EVENT_TYPE]#]
```

where `nat` is the non-negative integers, `posnat` is the positive integers and `EVENT_TYPE` is a type for transactional events defined in `Event`. Projection accept a condition and a list and preserves only those elements which satisfy the condition. It is defined recursively as follows:

```
projection(condition: [EVENT_TYPE -> bool],l: list): RECURSIVE list =
    IF empty(l)
       THEN l
       ELSE IF condition(car(l))
                 THEN cons(car(l),projection(condition,cdr(l)))
                 ELSE projection(condition,cdr(l))
            ENDIF
    ENDIF
    MEASURE l`size
```

where car, cdr, empty and size are the regular list operations. **MEASURE** `l`size` indicates the maximal number of times that `projection` may be called recursively. Other operations are provided in [CPZ08].

Verifying TCC

Both a specification that forbids lazy invalidation conflicts and the implementation of TCC were defined under the same theory.

Some of the specifications steps are combined to simplify the proofs in TLPVS. For example, when $Spec_{\mathcal{A}}$ commits a transaction, we combine the steps of interchanging events, removing them from $\mathcal{Q}$, and setting *spec_out* to $\blacktriangleright$. This restricts the set of $Spec_{\mathcal{A}}$'s runs but retains soundness. Formally, $TM_1 \sqsubseteq \widetilde{Spec}_{\phi_{li}}$ implies that $TM_1 \sqsubseteq Spec_{\phi_{li}}$, where $\widetilde{Spec}_{\phi_{li}}$ is the restricted specification. We next present the relevant part of the specification's transition relation ρ:

```
% a7,a8,a9 and a10 were combined into one action - Commit:
    (EXISTS (id: CLIENT_ID):
       (is_pending(st_a`Q,id) AND
        st_a`spec_doomed(id) = FALSE AND
        read_events_consistent_with_mem(projection(of_id(id))(st_a`Q),st_a`spec_mem) AND
        stp_a = st_a WITH
                  [spec_out := (#id := id, act := COMMIT, adr := 0, val := 0#),
                   spec_mem := update_mem(st_a`spec_mem, st_a`Q, id),
                   Q := fix_Q_at_commit(is_conflict,id,st_a`Q),
                   spec_doomed := doom_others(is_conflict,id,projection(of_id(id))(st_a`Q),
                                              st_a`spec_doomed)])
```

where `is_conflict` consists of the forbidden interchanges corresponding to conflict lazy invalidation, `fix_Q_at_commit(is_conflict,id,st_a`Q)` removes from `st_a`Q` the events of `id`'s committed transaction and events of other transactions that conflict with it (i.e. for which `is_conflict` is satisfied), while `doom_others` dooms all the transactions that conflicts with that of `id` and sets `spec_doomed[id]` to F. Thus, when a client tries to commit an undoomed transaction several actions are combined such that its transaction is committed, other transactions that conflict with it are doomed for abortion, the transaction's events are removed from $\mathcal{Q}$ and the memory is updated according to the write events.

The implementation's key component, ρ_c, is illustrated next:

```
rho_c: BI_PREDICATE_C =
  (LAMBDA (st_c, stp_c):
     % t1 - Open:
       (EXISTS (id: CLIENT_ID):
         empty(st_c`caches(id)) AND
         stp_c = st_c WITH
                   [imp_out := (#id := id, act := OPEN, adr := 0, val := 0#),
                    caches := st_c`caches WITH
                                [(id) := append_open(id,st_c`caches(id))]])
     % t2 - Read:
     OR (EXISTS (id: CLIENT_ID, adr: MEMORY_ADDRESS):
          NOT empty(st_c`caches(id)) AND
          NOT some(is_write_of_id(id),st_c`caches(id)) AND
          LET read_val = get_val(adr,st_c`imp_mem) IN
             stp_c = st_c WITH
                       [imp_out := (#id := id, act := READ,
                                    adr := adr, val := read_val#),
                        caches := st_c`caches WITH
                                    [(id) := append_read(id,adr,read_val,
                                                         st_c`caches(id))]])
     % t3 - Write:
     OR (EXISTS (id: CLIENT_ID, adr: MEMORY_ADDRESS, val: VALUE):
          NOT empty(st_c`caches(id)) AND
          stp_c = st_c WITH
                   [imp_out := (#id := id, act := WRITE,
                                adr := adr, val := val#),
                    caches := st_c`caches WITH
                                [(id) := append_write(id,adr,val,
                                                      st_c`caches(id))]])
```

```
% t4 - abort:
OR (EXISTS (id: CLIENT_ID):
     NOT empty(st_c`caches(id)) AND
     stp_c = st_c WITH
               [imp_out := (#id := id, act := ABORT, adr := 0, val := 0#),
                caches := st_c`caches WITH [(id) := null],
                imp_doomed := st_c`imp_doomed WITH [(id) := FALSE]])
% t5 - Commit:
OR (EXISTS (id: CLIENT_ID):
     NOT empty(st_c`caches(id)) AND
     st_c`imp_doomed(id) = FALSE  AND
     read_events_consistent_with_mem(st_c`caches(id),st_c`imp_mem) AND
     stp_c = st_c WITH
               [imp_out := (#id := id, act := COMMIT, adr := 0, val := 0#),
                imp_mem := update_mem(st_c`imp_mem, st_c`caches(id)),
                caches := st_c`caches WITH [(id) := null],
                imp_doomed := doom_others(is_conflict, id,
                                          st_c`caches, st_c`imp_doomed)])
% t6 - idle:
OR stp_c = st_c WITH [imp_out := (#id := 0, act := NULL, adr := 0, val := 0#)])
```

where `stp_c` is the primed state, `some(condition,caches(id))` checks if one of the events in `caches(id)` satisfies `condition`, `get_val(adr,mem)` gets the value of `mem` at `adr`, `read_events_consistent_with_mem(caches(id),mem)` checks if all the read events in `caches(id)` are consistent with `mem`, `update_mem(mem, caches(id))` updates `mem` according to the write events in `caches(id)` and `doom_others(is_conflict, id, caches, doomed)` dooms all the transactions that conflict with `id`'s transaction. It is easy to see that the description matches Table 2.2.

The definitions of the other components, of both the implementation and specifica-

tion, can be found in [CPZ08].

The abstraction relation R between concrete and abstract states is described in TLPVS as follows:

```
rel: RELATION =
    (LAMBDA s_c, s_a:
       s_c`imp_out = s_a`spec_out              AND
       s_c`imp_mem = s_a`spec_mem              AND
       s_c`imp_doomed = s_a`spec_doomed        AND
       FORALL (id: CLIENT_ID):
         (NOT s_c`imp_doomed(id)) IMPLIES
          project(id,s_a`Q) = s_c`caches(id)
```

Given a concrete and abstract states, `rel` equates the values of the output variable, the memory and the list of doomed transactions. It also checks that if the transaction of a client is not doomed, then its projection on the abstract queue equals the client's cache in the concrete system

In order to prove that $TM_1 \sqsubseteq Spec_{\mathcal{A}}$, we checked that `abstract(con,abs)` holds by instantiating `ABS_REL` and proving all of its premises. The proof can be found in [CPZ08].

Verifying Eager Conflict Detection, Lazy Version Management

In this section we prove the correctness of an implementation that supports eager conflict detection and lazy version management. A representative of this class of implementations is LTM of [AAK+05]. As in TCC, transactions execute speculatively in the clients' caches and when they commit the memory is updated accordingly. The definition of

conflict is slightly stronger than eager invalidation by disallowing writes to the same memory location. Namely, the memory detects a conflict if it receives $R_i^t(x)$ such that $W_j^t(x)$ is in some open transaction, or $W_i^t(x,v)$ such that $W_j^t(x)$ or $R_j^t(x)$ is in some open transaction. In case of a conflict, the transaction that requests the second "offensive" memory access is aborted.

The specification is given by $\mathit{Spec}_{\mathcal{A}}$ where $\mathcal{A}$ is the admissible set of events corresponding to the eager invalidation conflict described in Subsection 2.2.2.1 excluding also the exchange of write events to the same location. Namely, the corresponding maintaining formula is $m_{ei} \ \vee \ (\blacktriangleright_n \wedge \ominus W_m^t(x) \wedge (\neg \blacktriangleright_n) \mathcal{S} \, W_n^t(x))$.

We next present the implementation, TM_2, which uses the following data structures:

- *imp_mem*: $\mathbb{N} \to \mathbb{N}$ — Persistent memory. Initially, for all $i \in \mathbb{N}$, *imp_mem*$[i] = 0$;
- *caches*: **array**$[1..n]$ **of list of** E^t — Client's caches. For each $p \in [1..n]$, *caches*$[p]$, initially empty, is a sequence over E_p^t that records the actions of p's pending transaction;
- *imp_out*: **scalar in** $E_\perp^t = E^t \cup \{\perp\}$ — An output variable recording responses to clients, initially $\perp$;

The responses to clients requests are either an error err (for illegal syntactical requests, e.g., a request to commit while the client has no pending transaction), an $abort$ if the request causes the client's transaction to conflict with another pending transaction, or an acknowledgment whereas for ιR^t requests and non-conflicting ιR^t requests it is

a value in $\mathbb{N}$ and ack for all other cases. Note that legal commit requests are always acknowledged.

We assume a generic clients module, *Clients*(n), that may, at any step, invoke the next request for client p, $p \in [1..n]$, provided the sequence of E_p^t-events issued so far (including the current one) forms a prefix of a well-formed sequence. The justice requirement of *Clients*(n) is that eventually, every pending transaction issues an ack-ed $\iota\blacktriangleright$ or $\iota\not\blacktriangleright_p$. Combining modules *TM*$_2$ and *Clients*(n) we obtain the complete implementation, defined by:

$$\mathit{Imp}_2 : \quad \mathit{TM}_2 \;|||\; \mathit{Clients}(n)$$

Table 2.3 describes the acknowledged actions (events) of *Imp*$_2$. The justice requirements of *Clients*(n), together with the observation that ack-ed $\iota\not\blacktriangleright$ and ack-ed $\iota\blacktriangleright$ cause the cache of the issuing client to be emptied, imply that *Imp*$_2$'s justice requirement is that for every $p = 1, \ldots, n$, *caches*$[p]$ is empty infinitely many times.

For simplicity, we again assume that clients only issue reads for locations they had not written to in the pending transaction.

To prove that *Imp*$_2$ satisfies its specifications, we used the following abstraction relation R in TLPVS:

	imp_out	Other Updates	Conditions
t_1	$\blacktriangleleft_p$	append $\blacktriangleleft_p$ to *caches*$[p]$	*caches*$[p]$ is empty
t_2	$R_p^t(x, v)$	append $R_p(x, v)$ to *caches*$[p]$	$v =$ *imp_mem*$[x]$; *caches*$[p]$ is not empty; $\forall q \neq p, W_q^t(x) \notin$ *caches*$[q]$
t_4	$W_p^t(x, v)$	append $W_p(x, v)$ to *caches*$[p]$	*caches*$[p]$ is not empty $\forall q \neq p, W_q^t(x), R_q^t(x) \notin$ *caches*$[q]$
t_6	$\not\blacktriangleright_p$	set *caches*$[p]$ to empty	*caches*$[p]$ is not empty
t_7	$\blacktriangleright_p$	set *caches*$[p]$ to empty; update *imp_mem* by *caches*$[p]$	*caches*$[p]$ is not empty
t_8	$\perp$	none	none

Table 2.3: The actions of Imp_2.

```
rel: RELATION =
    (LAMBDA s_c, s_a:
       s_c`imp_out = s_a`spec_out                AND
       s_c`imp_mem = s_a`spec_mem                AND
       s_c`imp_doomed = s_a`spec_doomed          AND
       FORALL (id: CLIENT_ID):
          project(id,s_a`Q) = s_c`caches(id)
```

Verifying Eager Conflict, Eager Version Control

In this section we prove the correctness of an implementation that supports eager conflict detection and eager version management. A representative of this class of implementations is LogTM of [MBM+06a]. The definition of conflict and its resolution are identical to those of the implementation in Subsection 2.5.3, i.e., the memory detects a conflict if it receives $R_i^t(x)$ such that $W_j^t(x)$ is in some open transaction, or $W_i^t(x, v)$ such that $W_j^t(x)$ or $R_j^t(x)$ is in some open transaction. In case of a conflict, the transaction that requests the second "offensive" memory access is aborted. Since eager version control

is employed, however, the implementation updates the memory upon a write. If later it is necessary to abort the transaction, then the memory is rolled back to its previous value. Since the protocol does not allow for more than one overlapping write, there is no need to record any information but the previous value of the locations that are updated. The implementation thus includes an array $log[1..n]$ of tuples from type $\langle \mathbb{N}, \mathbb{N} \rangle$. For client p, $log[p]$ stores the previous value of every location that has been updated by the client in its pending transaction.

The specification is given by $Spec_{\mathcal{A}}$ where the corresponding maintaining formula is $m_{ei} \ \vee \ (\blacktriangleright_n \wedge \ominus W_m^t(x) \wedge (\neg \blacktriangleright_n) \mathcal{S} \, W_n^t(x))$.

The implementation, TM_3, uses the following data structures:

- *imp_mem*: $\mathbb{N} \rightarrow \mathbb{N}$ — Persistent memory. Initially, for all $i \in \mathbb{N}$, *imp_mem*$[i] = 0$;
- *caches*: **array**$[1..n]$ **of list of** E^t — Client's caches. For each $p \in [1..n]$, *caches*$[p]$, initially empty, is a sequence over E_p^t that records the actions of p's pending transaction;
- log: **array**$[1..n]$ **of**$\langle \mathbb{N}, \mathbb{N} \rangle$ — Log that stores the previous value of every location that has been updated by the client in its pending transaction. For each $p \in [1..n]$, $\log[p]$, initially empty, is a list of tuples of the type $\langle \mathbb{N}, \mathbb{N} \rangle$, where the first element is a memory location and the second is the location's previous value.
- *imp_out*: **scalar in** $E_\perp^t = E^t \cup \{\perp\}$ — An output variable recording responses to clients, initially $\perp$;

The responses to clients requests are either an error err (for illegal syntactical requests, e.g., a request to commit while the client has no pending transaction), an $abort$ if the request causes the client's transaction to conflict with another pending transaction, or an acknowledgment whereas for ιR^t requests and non-conflicting ιR^t requests it is a value in $\mathbb{N}$ and ack for all other cases. If a transaction is aborted the memory is rolled back to its previous value based on the client's $\log$.

Combining module TM_3 and the generic clients module $Clients(n)$ we obtain the complete implementation, defined by:

$$Imp_3 : \quad TM_3 \;|||\; Clients(n)$$

The justice of Imp_3's requires that for every $p = 1, \ldots, n$, $caches[p]$ is empty infinitely many times.

Table 2.4 describes the actions of Imp_3. For simplicity we assume that clients only issue reads for locations they had not written to in the pending transaction.

To prove that Imp_3 satisfies its specifications, we cannot use the same R which is used to verify Imp_2; rather, we look at the "rolled back" version of memory values, which can be determined by log. Formally, for each memory address $x \in \mathbb{N}$, we define

$$rolled\_back[x] \;=\; \begin{cases} v & \text{for some } j,\ \langle x, v\rangle \in log[j] \\ imp\_mem[x] & \text{otherwise} \end{cases}$$

	imp_out	Other Updates	Conditions
t_1	$\blacktriangleleft_p$	append $\blacktriangleleft_p$ to *caches*$[p]$	*caches*$[p]$ is empty
t_2	$R_p^t(x,v)$	append $R_p(x,v)$ to *caches*$[p]$	$v =$ *imp_mem*$[x]$; *caches*$[p]$ is not empty; $\forall q \neq p, W_q^t(x) \notin$ *caches*$[q]$
t_4	$W_p^t(x,v)$	append $W_p(x,v)$ to *caches*$[p]$; append $\langle x,$ *imp_mem*$[x]\rangle$ to *log*$[p]$; update *imp_mem*$[x]$ to v	*caches*$[p]$ is not empty $\forall q \neq p, W_q^t(x), R_q^t(x) \notin$ *caches*$[q]$
t_6	$\blacktriangleright\!\!\!\!\!\!\checkmark_p$	set *caches*$[p]$ to empty; $\forall\langle x,u\rangle \in$ *log*$[p]$ set *imp_mem*$[x]$ to u; set *log*$[p]$ to empty	*caches*$[p]$ is not empty
t_7	$\blacktriangleright_p$	set *caches*$[p]$ to empty; set *log*$[p]$ to empty	*caches*$[p]$ is not empty
t_8	$\perp$	none	none

Table 2.4: The actions of Imp_3.

The abstraction relation R is then defined in TLPVS as:

```
rel: RELATION =
    (LAMBDA s_c, s_a:
       s_c`imp_out = s_a`spec_out                AND
       s_a`spec_mem = rolled_back(s_c`imp_mem)   AND
       FORALL (id: CLIENT_ID):
          s_a`spec_doomed(id) = FALSE            AND
       FORALL (id: CLIENT_ID):
          project(id,s_a`Q) = s_c`caches(id)
```

2.6 Supporting Non-Transactional Accesses

As pointed out in Subsection 2.1.2.5 accesses to the memory are not necessarily all transactional and for various reasons they can be also non-transactional. In this section

we show how to extend our model to handle non-transactional accesses. We apply our proof rule using TLPVS and produce a machine checkable, deductive proof for the correctness of an implementation that handles non-transactional memory accesses.

Non-transactional accesses cannot be aborted and may not be issued by a client that has a pending transaction. The atomicity and serializability requirements remain intact, where non-transaction operations are cast in the specification as a successfully committed, single operation, transaction. A conflict occurs when two overlapping transactions, or a non-transactional operation and an overlapping transaction, access the same location and at least one writes to it. In case of the latter, it is always the transaction that is aborted. Our method for handling the non-transactional operations allows using the same proof rule and the same admissible interchange sets for each one of [Sco06]'s conflicts, as defined in Subsection 2.2.2.1.

It is important to point out that we make a strong assumption, under which the transactional memory is aware of non-transactional accesses, as soon as they occur. While the transactional memory implementations cannot abort such accesses, it may use them in order to abort transactions that are under its control. It is only with such or similar assumption that total consistency or coherence can be maintained.

2.6.1 Extended Specification

In this subsection we extend the model and the specification to handle non-transactional memory accesses. We present only the extensions and modifications and do not repeat

the unchanged definitions. For every client p, let the set of *non-transactional invocations by client* p consists of:

- $\iota R_p^{nt}(x)$ – A non-transactional request to read from address $x \in \mathbb{N}$.
- $\iota W_p^{nt}(y, v)$ – A non-transactional request to write value $v \in \mathbb{N}$ to address $y \in \mathbb{N}$.

The memory provides a response for each non-transactional invocation where ιW^{nt} is responded by an acknowledgment ack and $\iota R_p^{nt}(x)$ by the value of the memory at location x. The memory returns an err if the requesting client has a pending transaction. Non-transactional invocations are never responded with an $abort$. We define E_p^{nt}: $\{R_p^{nt}(x, u), W_p^{nt}(x, v)\}$ to be the set of *non-transactional observable events*, and keep the same definition for the set of transactional observable events, namely, E_p^t: $\{\blacktriangleleft_p$, $R_p^t(x, u), W_p^t(x, v), \blacktriangleright_p, \not\blacktriangleright_p\}$. We define E^{nt} to be the set of all non-transactional events over all clients, i.e., $E^{nt} = \cup_{p=1}^{n} E_p^{nt}$, and redefine E, the set of all observable events, to include non-transactional events, namely $E = E^{nt} \cup E^t$. As with transactional events, we also abbreviate the pair invocation, non-err response by omitting the ι-prefix of the invocation for transactional requests. Thus, $W_p^{nt}(x, v)$ abbreviates $\iota W_p^{nt}(x, v), ack_p$. For read actions, we include the value read, that is, $R_p^{nt}(x, u)$ abbreviates $\iota R^{nt}(x), \rho R(u)$. Let σ: $e_0, e_1, \ldots, e_k$ be a finite sequence of observable E-events. We say that the sequence $\hat{\sigma}$ over E^t is *σ's transactional sequence*, where $\hat{\sigma}$ is obtained from σ by replacing each R_p^{nt} and W_p^{nt} by $\blacktriangleleft_p$ R_p^t $\blacktriangleright_p$ and $\blacktriangleleft_p$ W_p^t $\blacktriangleright_p$, respectively. That is, each non-transactional event of σ is transformed into a singleton committed transaction in

$\hat{\sigma}$. The definitions for well-formed transactional sequence, local read-write consistency, atomicity and global read-write consistency, remain the same.

Table 2.5 summarizes the new non-transactional steps, a_{13} and a_{14}, for $Spec_{\mathcal{A}}$.

	Request	*imp_out*	Other Updates	Conditions	Response
a_{13}	$R_p^{nt}(x)$	$R_p^{nt}(x,v)$	append $\blacktriangleleft_p, R_p^t(x,v), \blacktriangleright_p$ to $\mathcal{Q}$	p is not pending	*spec_mem*$[x]$
a_{14}	$W_p^{nt}(x,v)$	$W_p^{nt}(x,v)$	append $\blacktriangleleft_p, W_p^t(x,v), \blacktriangleright_p$ to $\mathcal{Q}$	p is not pending	ack

Table 2.5: Non-transactional steps of $Spec_{\mathcal{A}}$.

A non-transactional memory-access a_p^{nt} (that can only be accepted when p-has no pending transaction) is treated by appending $\blacktriangleleft_p, a_p^t, \blacktriangleright_p$ to the queue. (Note that the non-transactional event is replaced by its transactional counterpart.) These transactions, corresponding to the non-transactional operations, *cannot* be "doomed to abort" since they are not pending by definition. The extended $Spec_{\mathcal{A}}$ does not support "eager version management" that eagerly updates the memory with every W^t-action (that doesn't conflict any pending transaction) which is reasonable since eager version management and the requirement to commit each non-transactional access are contradictory.

2.6.2 Verifying TCC Augmented with Non-Transactional Accesses

We augment the implementation of Section 2.2.5 to support non-transactional accesses and verify its correctness. As before, transactions execute speculatively in the clients' caches. When a transaction commits, all pending transactions that contain some read events from addresses written to by the committed transaction are "doomed." Similarly,

non-transactional writes cause pending transactions that read from the same location to be "doomed." A doomed transaction may execute new read and write events in its cache, but it must eventually abort.

The specification is given by $Spec_{\mathcal{A}}$ where $\mathcal{A}$ is the admissible set of events corresponding to the lazy invalidation conflict described in Subsection 2.2.2.1.

We refer to the implementation as TM_4. It uses the following data structures:

- *imp_mem*: $\mathbb{N} \to \mathbb{N}$ — Persistent memory. Initially, for all $i \in \mathbb{N}$, *imp_mem*$[i] = 0$;
- *caches*: **array**$[1..n]$ **of list of** E^t — Client's caches. For each $p \in [1..n]$, *caches*$[p]$, initially empty, is a sequence over E^t_p that records the actions of p's pending transaction;
- *imp_out*: **scalar in** $E_\perp = E \cup \{\perp\}$ — An output variable recording responses to clients, initially $\perp$;
- *imp_doomed*: **array** $[1..n]$ **of booleans** — An array recording which transactions are doomed to be aborted. Initially *imp_doomed*$[p] =$ F for every p.

Note that *caches* is over E^t, i.e., only over transactional events, whereas *imp_out* is over $E_\perp$, which consists also of non-transactional events.

The responses are either a value in $\mathbb{N}$ for ιR^t requests, an err for syntactic errors (e.g., $\iota\blacktriangleright$ request when there is no pending transaction), an $abort$ for a $\iota\blacktriangleright$ request when the transaction is doomed for abortion, or an acknowledgment ack for all other cases.

TM$_4$ receives requests from clients, to each it updates its state, including updating the output variable *imp_out*, and issues a response to the requesting client. The responses are either a value in $\mathbb{N}$ for ιR^t or ιR^{nt} requests, an error err for syntactic errors, an $abort$ for a $\iota \blacktriangleright$ request when the transaction is doomed for abortion, or an acknowledgment ack for all other cases. Table 2.6 describes the actions of *TM*$_4$ (for now, ignore the comments in the square brackets under the "conditions" column).

	Request	*imp_out*	Other Updates	Conditions	Response
t_1	$\iota \blacktriangleleft_p$	$\blacktriangleleft_p$	append $\blacktriangleleft_p$ to *caches*[p]	[*caches*[p] is empty]	ack
t_2	$\iota R_p^t(x)$	$R_p^t(x,v)$	append $R_p(x,v)$ to *caches*[p]	v = *imp_mem*[x]; [*caches*[p] is not empty]	*imp_mem*[x]
t_3	$\iota W_p^t(x,v)$	$W_p^t(x,v)$	append $W_p(x,v)$ to *caches*[p]	[*caches*[p] is not empty]	ack
t_4	$\iota \not\blacktriangleright_p$	$\not\blacktriangleright_p$	set *caches*[p] to empty set *imp_doomed*[q] to F;	[*caches*[p] is not empty]	ack
t_5	$\iota \blacktriangleright_p$	$\blacktriangleright_p$	set *caches*[p] to empty; for every x and $q \neq p$ such that $W_p^t(x) \in$ *caches*[p] and $R_p^t(x) \in$ *caches*[q] set *spec_doomed*[q] to T; update *imp_mem* by *caches*[p]	*imp_doomed*[p] = F; [*caches*[p] is not empty]	ack
t_6	none	$\perp$	none	none	none
t_7	$\iota R_p^{nt}(x)$	$R_p^{nt}(x,v)$	none	v = *imp_mem*[x]; [*caches*[p] is empty]	*imp_mem*[x]
t_8	$\iota W_p^{nt}(x,v)$	$W_p^{nt}(x,v)$	set *imp_mem*[x] to v; for every q such that $R^t(x) \in$ *caches*[q] set *imp_doomed*[q] to T	[*caches*[p] is empty]	ack

Table 2.6: The actions of *TM*$_4$.

We assume a generic clients module, *Clients*(n), that may, at any step, invoke the next request for client p, $p \in [1..n]$. Combining modules *TM*$_4$ and *Clients*(n) we obtain the complete implementation, defined by:

$$\mathit{Imp}_4 : \quad \mathit{TM}_4 \;|||\; \mathit{Clients}(n)$$

The actions of Imp_4 can be described similarly to the one given by Table 2.6, where the first and last column are ignored, the conditions in the brackets are added. The justice of Imp_4 requires that for every $p = 1, \ldots, n$, $\mathit{caches}[p]$ is empty infinitely many times.

We now sketch a proof, using rule ABS-REL, showing that $\mathit{Imp}_4 \sqsubseteq \mathit{Spec}_{\mathcal{A}}$ (a mechanical proof using TLPVS is provided in [CPZ08]). The application of the rule requires the identification of a relation R which holds between concrete and abstract states. We use the same relation R as of Subsection 2.4.2, defined by:

$$\mathit{spec\_out} = \mathit{imp\_out} \;\wedge\; \mathit{spec\_mem} = \mathit{imp\_mem} \;\wedge\; \mathit{spec\_doomed} = \mathit{imp\_doomed}$$
$$\wedge\; \textstyle\bigwedge_{p=1}^{n} \neg \mathit{imp\_doomed}[\mathrm{p}] \longrightarrow (\mathcal{Q}|_p = \mathit{caches}[p])$$

Premises R1,R3 and R4 of rule ABS-REL hold for the same reason given in Subsection 2.4.2. In fact, the justifications for the concrete transitions $t_1, \ldots, t_4$ and t_6 that were given in Subsection 2.4.2 to verify R2 may as well be used here. It thus remains only to consider the other transitions for validating premise R2.

t_5. Transition t_5 commits the current transaction contained in $\mathit{caches}[p]$ while outputting the event $\blacktriangleright_p$. This is possible only if p has a pending transaction and $\mathit{imp\_doomed}[p] = \mathrm{F}$. The pending transaction must be consistent with *imp_mem*;

otherwise, one of the read events does not match the value of the referred memory address, which means that there must have been an update (according to a write in a committed transaction or to a non-transactional write) that occurred after the read and implied *imp_doomed*$[p] = \text{T}$. The transition also updates *imp_mem* according to *caches*$[p]$, clears *caches*$[p]$ and sets *imp_doomed*$[q] = \text{T}$ if there exist $W_p^t(x)$ and $R_q^t(x)$.

The emulation of this transition begins by the instance of a_{11} which appends $mark_p$ to $\mathcal{Q}$, followed by a sequence of applications of abstract transition a_{10} which attempt to move all the events of p to the front of $\mathcal{Q}$, where $\mathcal{A}$ corresponds to the lazy invalidation conflict, i.e. forbids interchanging $\blacktriangleright_p$ on the right with $R_q^t(x)$ if there also exists a $W_p^t(x)$. If there are conflicts we apply abstract transitions a_9 and set *spec_doomed*$[q] = \text{T}$ for all q such that $\mathcal{Q}$ contains $R_q^t(x)$ and $W_p^t(x)$. As we shall see next, during the emulation for transition t_7 all read events that are added to $\mathcal{Q}$ as a result of a non-transactional read are removed from it before any other client tries to commit its pending transaction. Thus all read events in $\mathcal{Q}$, when p tries to commit, are originally transactional and belong to transactions that may be aborted. After aborting conflicting transaction, and when p has a consecutive transaction at the front of $\mathcal{Q}$, we apply abstract transition a_7 which confirms that $\mathcal{Q}|_p$ is consistent with *spec_mem* (must be true due to the R-conjunct *spec_mem* $=$ *imp_mem*), outputs $\blacktriangleright_p$, and replaces $mark_p$ with $\blacktriangleright_p$. Finally we apply a_8 which updates *spec_mem* according to $\mathcal{Q}|_p$ (thus making it again equal to *imp_mem*), and removes all elements of p from $\mathcal{Q}$, thus reestablishing the

R-conjunct $\bigwedge_{p=1}^{n} \neg$*imp_doomed*[p] $\longrightarrow$ $(\mathcal{Q}|_p = \mathit{caches}[p])$.

t_7. Transition t_7 outputs the non-transactional event $R_p^{nt}(x, u)$. The preconditions are that *caches*$[p]$ is empty and that $u =$ *imp_mem*$[x]$. The emulation of this transition begins by an instance of a_{13} which appends $\blacktriangleleft_p, R_p^t(x, v), \blacktriangleright_p$ to $\mathcal{Q}$ and outputs $R_p^{nt}(x, v)$, where v equals *spec_mem*$[x]$. This is followed by a sequence applications of abstract transition a_{10} which attempt to shift all the events of p to the front of $\mathcal{Q}$. Since $\mathcal{A}$ corresponds to the lazy invalidation conflict, and there are no $W_p^t(x)$, this is feasible. Next we apply abstract transition a_8 which confirms that $\mathcal{Q}|_p$ is consistent with *spec_mem* (must be true since $v =$ *spec_mem*$[x]$), and removes all events of p from $\mathcal{Q}$, thus reestablishing the R.

t_8. Transition t_7 outputs the non-transactional event $W_p^{nt}(x, v)$. The only precondition is that *caches*$[p]$ is empty. The emulation of this transition begins by an instance of a_{14} which appends $\blacktriangleleft_p, W_p^t(x, v), \blacktriangleright_p$ to $\mathcal{Q}$ and outputs $W_p^{nt}(x, v)$. This is followed by a sequence applications of abstract transition a_{10} which attempt to shift all the events of p to the front of $\mathcal{Q}$, without interchanging $\blacktriangleright_p$ on the right with $R_q^t(x)$. Since they are forbidden, to avoid such interchanges we apply abstract transitions a_9 and set *spec_doomed*$[q] =$ T for all q such that $\mathcal{Q}$ contains $R_q^t(x)$. Next we apply abstract transition a_8 which confirms that $\mathcal{Q}|_p$ is consistent with *spec_mem* (obvious since the transaction consists only of $W_p^t(x)$), updates *spec_mem*$[x]$ to v (thus making it again equal to *imp_mem*), and removes the events of p from $\mathcal{Q}$, thus reestablishing R.

2.7 Related Work

To the best of our knowledge, the work presented here is the first to verify the correctness of transactional memories. A very recent work [GHJS08] presents a model checking technique for verifying safety and liveness properties of software transactional memories. It considers strict serializability and abort consistency as safety requirements, and obstruction freedom and livelock freedom as liveness requirements. By exploiting symmetries in STM implementations they reduce the verification problem of unbounded state to a problem that involves only two threads and two shared variables. For verifying safety, they first define a finite state transition system that generates exactly the permitted executions of programs with two threads and two shared variables. Next, a transition system is defined also for the implementation, and a check for the existence of a simulation between the two transition systems is performed. The liveness properties are checked over the implementation's transition system, by trying to detect a loop which violates the relevant property.

The work of [AMP00] studies the model checking of serializability, linearizablity [HW90] and sequential consistency [Lam79]. Most relevant to our work is the verification of serializability, which they show to be PSPACE-complete. They generate a nondeterministic finite state automaton for the implementation and another automaton that accepts only strings that are not serializable with respect to the specification. They then check whether the intersection of the automata is empty. Their method is intended for verifying serializability in its most general form and does not capture any special as-

pects of transactional memory. It also may be used only to verify bounded instantiations. They do not report whether their method was implemented or not.

2.8 Conclusions

In this work we developed an abstract model for specifying transactional memory and a methodology for verifying the correctness of transactional memory implementations against their specifications. The first essential contribution of this work is the notion of interchangeable events which allows to specify correctness criteria and to model different conflicts. We constructed a proof rule based on abstraction mapping for verifying that an implementation satisfies its specification and applied it using model checker TLC for verifying a bounded instantiation of TCC. Since it is not always possible to relate abstract to concrete states by a functional correspondence, we constructed a new proof rule which only assumes an abstraction relation. We provided a mechanical verification of the new proof rule and applied it by using TLPVS to mechanically verify three transactional memory implementations drawn from the literature.

Since practical transactional memory implementations are expected to deal with memory accesses that occur outside of transactions, we extended our initial specification to support non-transactional memory accesses, based on the assumption that the implementation can detect them. We then proved the correctness of an implementation that accommodate non-transactional memory accesses.

We provided a framework that enables to verify the correctness of other transactional

memory implementations, to be used in the theorem prover TLPVS.

2.9 Future Work

While our model captures the important algorithmic aspects of the implementations, it is still quite a bit more abstract than implementations written in C++, Java etc. One obvious next step is to formally analyze more detailed models of the implementations.

The extension for supporting non-transactional accesses is based on the assumption that an implementation can detect them. We would like to weaken this assumption and extend our framework to enable the verification of implementations that cannot distinguish between transactional and non-transactional accesses. It would also be quiet interesting to extend our model to support nesting transactions, since transactional code may use library procedures that also contain transactions.

We would like to harness the power of new verification technologies like satisfiability modulo theories (SMT) that have already shown so much potential for software verification. Interesting questions are whether SMT and other software verification technologies provide us additional leverage for efficient reasoning about transactional memory, and whether there are theories and decision procedures specific to transactional memory that could be added to the SMT arsenal.

We are planning as well to study liveness properties of transactional memory, while considering different arbitration policies.

3

RANKING ABSTRACTION OF RECURSIVE PROGRAMS

Procedural programs with unbounded recursion present a challenge to symbolic model-checkers since they ostensibly require the checker to model an unbounded call stack. In this chapter we propose the integration of ranking abstraction [KP00b, BPZ05], finitary state abstraction, procedure summarization [SP81], and model-checking into a combined method for the automatic verification of LTL properties of infinite-state recursive procedural programs. The inputs to this method are a sequential procedural program together with state and ranking abstractions. The output is either "success", or a counterexample in the form of an abstract error trace. The method is sound, as well as complete, in the sense that for any valid property, a sufficiently accurate *joint* (state and ranking) abstraction exists that establishes its validity.

The method centers around a fixpoint computation of procedure summaries of a finite-state program, followed by a subsequent construction of a behaviorally equivalent nondeterministic procedure-free program. Since we begin with an infinite-state program that cannot be summarized automatically, a number of steps involved in abstraction and LTL model-checking need to be performed over the procedural (unsummarized) program. These include augmentation with non-constraining observers and fairness con-

straints required for LTL verification and ranking abstraction, as well as computation of state abstraction. Augmentation with global observers and fairness is modeled in such a way as to make the associated properties observable once procedures are summarized. The abstraction of a procedure call is handled by abstracting "everything but" the call itself, i.e., local assignments and binding of actual parameters to formals and of return values to variables.

The method relies on machinery for computing abstractions of first order formulas, but is orthogonal as to how an abstraction is actually computed. We have implemented a prototype based on the TLV symbolic model-checker [Sha00] by extending it with a model of procedural programs. Specifically, given a symbolic finite-state model of a program, summaries are computed using BDD techniques in order to derive an FDS free of procedures to which model-checking is applied. The tool is provided, as input, with a concrete program and with predicates and ranking components. It computes a *predicate abstraction* [GS97] automatically using the method proposed in [BPZ05]. We have used this implementation to verify a number of canonical examples, such as Ackerman's function, the Factorial function and a procedural formulation of the 91 function.

While most components of the proposed method have been studied before, our approach is novel in that it reduces the verification problem to that of symbolic model-checking. Furthermore, it allows for the application of ranking and state abstractions while still relegating all summarization computation to the model-checker. Another advantage is that fairness is supported directly by the model and related algorithms, rather than it being specified in a property.

The rest of the chapter is organized as follows: Section 3.1 provides an overview of predicate abstraction and ranking abstraction, and formalizes procedural programs as flow-graphs. In Section 3.2 we present a method for verifying the termination of procedural programs using ranking abstraction, state abstraction, summarization, construction of a procedure-free FDS, and finally, model-checking. In Section 3.3 we present a method for LTL model-checking of recursive procedural programs. Finally, Section 3.5 concludes and discusses future work.

3.1 Background

In this subsection we provide background necessary for constructing our technique, and which was not already given in Chapter 1. It includes predicate abstraction to deal with the infinite domains, ranking abstraction to preserve program termination which might disappear once predicate abstraction is applied and an overview of programs with and without procedures presented as transition graphs.

3.1.1 Predicate Abstraction

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS and let $V_A = \{u_1, \ldots, u_n\}$ be a set of *abstract variables* that range over finite domains. An *abstract state* is an interpretation that assigns to each variable u_i a value in its domain. We denote by Σ_A the set of all abstract states. An

abstraction mapping is defined

$$\alpha_\epsilon\colon\ u_1 = \mathcal{E}_1(V),\ \ldots,\ u_n = \mathcal{E}_n(V)$$

where each $\mathcal{E}_i$ is an expression over V ranging over the domain of u_i.

When the abstract variables are boolean, and $\mathcal{E}_i$ are predicates over V, the abstraction is often referred to as *predicate abstraction*. The abstraction mapping α_ϵ can be expressed by:

$$V_A = \mathcal{E}(V)$$

From here on, we shall refer to α_ϵ as α. For assertion $p(V)$, we define its α-abstraction by:

$$\alpha(p)\colon\ \exists V.(V_A = \mathcal{E}(V) \wedge p(V))$$

The semantics of $\alpha(p)$ is $\| \alpha(p) \| : \{\alpha(s) \mid s \in \| p \|\}$. Note that an abstract state S is in $\| \alpha(p) \|$ iff there exists a concrete p-state that is abstracted into S. A bi-assertion $\beta(V, V')$ is abstracted by:

$$\alpha^2(\beta)\colon\ \exists V, V'.(V_A = \mathcal{E}(V) \wedge V'_A = \mathcal{E}(V') \wedge \beta(V, V'))$$

The *dual contracting abstraction* $\underline{\alpha}$ is defined by:

$$\underline{\alpha}(p)\colon\ \alpha(1) \wedge \neg\alpha(\neg p)$$

where $\alpha(1)$ restricts the range of $\underline{\alpha}$ to contain only abstract states that have at least one concrete state mapped by α into S.

For a temporal formula ψ in positive normal form, ψ^α is the formula obtained by abstracting every maximal state sub-formula p in ψ into $\underline{\alpha}(p)$.

The abstraction of $\mathcal{D}$ by α is the FDS

$$\mathcal{D}^\alpha \;=\; \langle V_A, \alpha(\Theta), \alpha^2(\rho), \{\alpha(J) \mid J \in \mathcal{J}\}, \{\langle \underline{\alpha}(p), \alpha(q)\rangle \mid (p,q) \in \mathcal{C}\}\rangle$$

Theorem 3.1. *For a system $\mathcal{D}$, abstraction α, and temporal formula ϕ:*

$$\mathcal{D}^\alpha \models \phi^\alpha \implies \mathcal{D} \models \phi$$

3.1.2 Ranking Abstraction

A *well-founded domain* is a pair ($\mathcal{W}$,$\succ$) such that $\mathcal{W}$ is a set, and $\succ$ is a partial order over $\mathcal{W}$ that does not admit infinite $\succ$-decreasing chains. A *ranking function* maps program states into some well-founded domain.

Ranking abstraction is a method of augmenting the concrete program by a non-constraining progress monitor, which measures the progress of the program relative to a given ranking function. Once a program is augmented, a conventional state abstraction can be used to preserve the ability to monitor progress in the abstract system. This method was introduced in [KP00b] and further clarified in [KPV01].

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ be an FDS, ($\mathcal{W}$,$\succ$) be a well-founded domain and δ be a

ranking function over the domain. Let dec be a fresh variable. The *augmentation* of $\mathcal{D}$ by δ, written $\mathcal{D}{+}\delta$, is the system

$$\mathcal{D}{+}\delta : \langle V \cup \{dec\}, \Theta, \rho \wedge \rho_\delta, \mathcal{J}, \mathcal{C} \cup \{(dec > 0, dec < 0)\}\rangle$$

where ρ_δ is defined by:

$$\rho_\delta \colon dec' = \begin{cases} 1 & \delta > \delta' \\ 0 & \delta = \delta' \\ -1 & otherwise \end{cases}$$

$\mathcal{D}{+}\delta$ acts like $\mathcal{D}$ but also keeps track of the changes in δ's value. The new compassion requirement restricts δ from decreasing infinitely often without increasing infinitely often (well-foundedness of $\mathcal{W}$). The behavior of $\mathcal{D}{+}\delta$ is exactly like $\mathcal{D}$'s, and therefore any property is valid over $\mathcal{D}$ iff it is valid over $\mathcal{D}{+}\delta$. In order to a verify liveness of $\mathcal{D}$, predicate abstraction is applied to $\mathcal{D}{+}\delta$ and the satisfiability of the abstracted property is checked.

3.1.3 Programs

Our programming language will be based on *transition graphs*. Some of this material is taken from [Pnu05] and is provided here for convenience.

3.1.3.1 Procedureless Programs

We assume a set of typed program variables V.

A *transition graph* is a labeled directed graph such that:

- All nodes are labeled by *locations* ℓ_i.
- There is one *initial node*, usually labeled by ℓ_0, and having no incoming edges.
- There is one *terminal node*, labeled ℓ_t with no outgoing edges.
- Nodes are connected by directed edges labeled by an instruction of the form

$$c \rightarrow [\vec{y} := \vec{e}]$$

  where c is a boolean expression over V, $\vec{y} \subseteq V$ is a list of variables, and $\vec{e}$ is a list of expressions over V. In cases the assignment part is empty, we can abbreviate the label to a pure condition c.

- Every node is on a path from ℓ_0 to ℓ_t.

Example 3.1 (Integer Square Root Program).

Program INT-SQUARE presented in Fig. 3.1 computes in y_1 the integer square root of the input variable $x \geq 0$.

States and Computations: For simplicity, we assume that all program variables range over the same domain D. For example, for program INT-SQUARE, D is the domain of

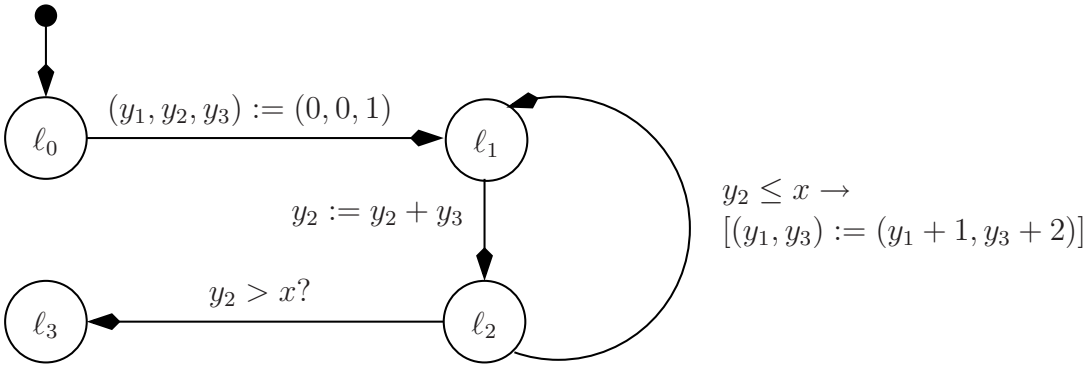


Figure 3.1: Integer square root program.

integers. We denote by $d = (d_1, \ldots, d_n)$ a sequence of D-values, which represent an *interpretation* (i.e., an assignment of values) of the program variables V.

A *state* of program P is a pair $\langle \ell, d \rangle$ consisting of a label ℓ and a data-interpretation d.

A *computation* of program P is a maximal sequence

$$\sigma : \quad \langle \ell^0, d^0 \rangle, \langle \ell^1, d^1 \rangle, \ldots, \langle \ell_k, d^k \rangle \ \ldots,$$

such that

- $\ell^0 = \ell_0$.
- For each $i = 0, 1, \ldots,$ there exists an edge connecting ℓ^i to ℓ^{i+1} and labeled by the instruction $c \rightarrow [\vec{y} := \vec{e}]$, such that $d^i \models c$ and $d^{i+1} = d^i$ with $\vec{y} := \vec{e}(d^i)$.

We denote by $\mathcal{C}omp(P, d)$ the set of computations of program P starting at data-state d.

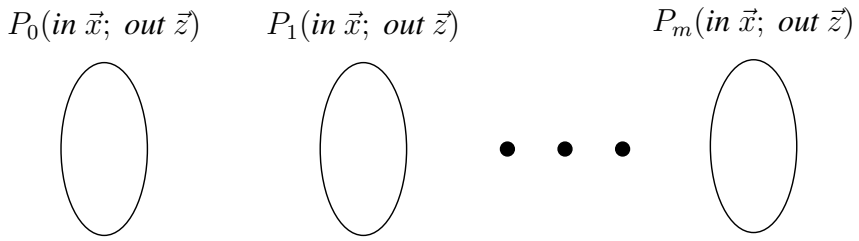
Example 3.2 (A Computation). Reconsider program INT-SQUARE. The computation

generated for $x = 5$ is:

$$\langle l_0; (-,-,-)\rangle,$$
$$\langle l_1; (0,0,1)\rangle, \quad \langle l_2; (0,1,1)\rangle, \quad \langle l_1; (1,1,3)\rangle, \quad \langle l_2; (1,4,3)\rangle,$$
$$\langle l_1; (2,4,5)\rangle, \quad \langle l_2; (2,9,5)\rangle, \quad \langle l_3; (2,9,5)\rangle$$

3.1.3.2 Recursive Programs

A program P consists of $m+1$ modules: $P_0, P_1, \ldots, P_m$, where P_0 is the main module, and $P_1, \ldots, P_m$ are procedures that may be called from P_0 or from other procedures.

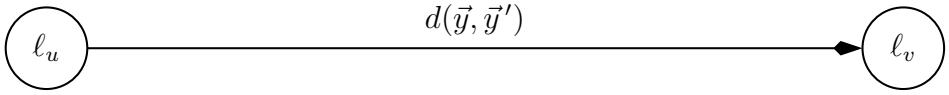


Each module P_i is presented as a flow-graph with its own set of locations $\mathcal{L}_i = \{\ell_0^i, \ell_1^i, \ldots, \ell_t^i\}$. It must have ℓ_0^i as its only entry point, ℓ_t^i as its only exit, and every other location must be on a path from ℓ_0^i to ℓ_t^i. It is required that the entry node has no incoming edges and that the terminal node has no outgoing edges.

The variables of each module P_i are partitioned into $\vec{y} = (\vec{x}; \vec{u}; \vec{z})$. We refer to $\vec{x}, \vec{u}$, and $\vec{z}$ as the input, working (local), and output variables, respectively. A module cannot modify its own input variables.

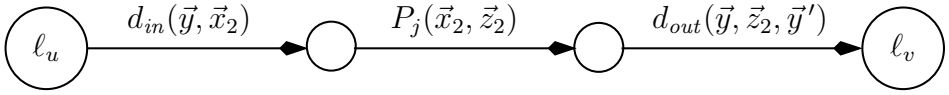
General Format: Edges in the graph are labeled by an instruction which must have one of the following forms:

- A *local change* $d(\vec{y}, \vec{y}')$, where d is an assertion over two copies of the module variables.



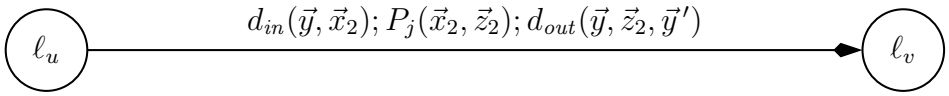
  It is required that $d(\vec{y}, \vec{y}')$ implies *pres*$(\vec{x})$. That is, all $\vec{x}$ variables are preserved by this transition.

- A *procedure call* $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$, where $\vec{x}_2$ and $\vec{z}_2$ are fresh copies of the input and output parameters $\vec{x}$ and $\vec{z}$, respectively.



  This instruction represents a procedure call to procedure P_j. Assertion $d_{in}(\vec{y}, \vec{x}_2)$ determines the actual arguments that are fed in the variables of $\vec{x}_2$. It may also contain an enabling condition under which this transition is possible. The assertion $d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$ updates the module variables $\vec{y}$ based on the values returned by the procedure via the output parameters $\vec{z}_2$. It is required that $d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$ implies *pres*$(\vec{x})$. Unless otherwise stated, we shall use the following description

as abbreviation for a procedure call.



Example 3.3 (The 91 Function). Consider the functional program given by

$$F(x) \quad = \quad \textbf{if } x > 0 \textbf{ then } x - 10 \textbf{ else } F(F(x + 11)) \tag{3.1}$$

We refer to this function as F_{91}. Fig. 3.3 shows the procedural version of F_{91} given in general format.

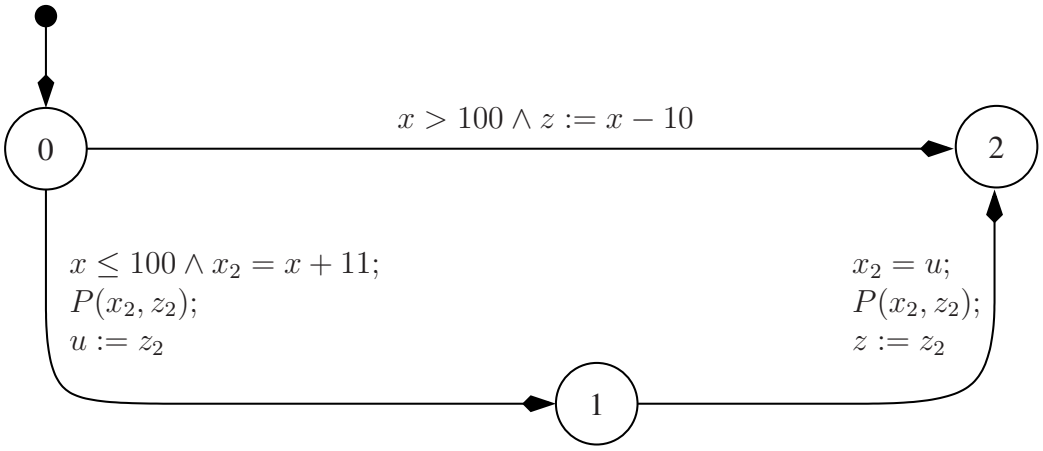
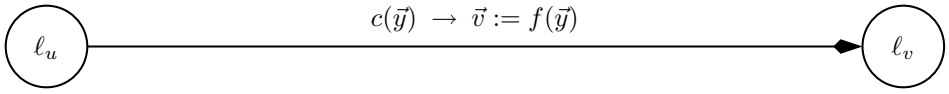


Figure 3.2: Procedural program F_{91}. Note that $z := x - 10$ is an abbreviation of $x' = x \wedge y' = y \wedge z' = x - 10$.

Deterministic Format: Conventional procedural programs are usually deterministic. The more general non-deterministic form arises only due to the process of abstraction.

We therefore introduce also the deterministic notation and show how it can be viewed as a special case of the general form. Edges in the deterministic format are labeled by an instruction which must have one of the following forms:

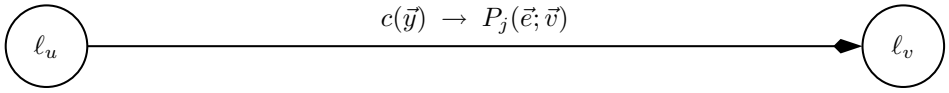
- An *assignment* $c(\vec{y}) \rightarrow [\vec{v} := f(\vec{y})]$, where the left-hand side variables $\vec{v} \subseteq \{\vec{u}, \vec{z}\}$ may not include any member of $\vec{x}$.



  An assignment can be viewed as a special case of a local-change statement where

$$d(\vec{y}, \vec{y}') \quad = \quad c(\vec{y}) \wedge \vec{v}' = f(\vec{y}) \wedge \mathit{pres}(\vec{y} - \vec{v})$$

- A *deterministic procedure call* $c(\vec{y}) \ \rightarrow \ P_j(\vec{e}; \vec{v})$, where $\vec{e}$ is a list of expressions over $\vec{y}$, and $\vec{v} \subseteq \{\vec{u}, \vec{z}\}$ is a list of distinct variables not including any member of $\vec{x}$. We refer to $\vec{e}$ and $\vec{y}$ as the actual arguments of the call.



  This statement can be viewed as a special case of a procedure call, where

$$d_{in}(\vec{y}, \vec{x}_2) \ = \ c(\vec{y}) \wedge \vec{x}_2 = \vec{e}(\vec{y}), \qquad d_{out}(\vec{y}, \vec{z}_2, \vec{y}') \ = \ \vec{v}' = \vec{z}_2 \wedge \mathit{pres}(\vec{y} - \vec{v})$$

Example 3.4. Fig. 3.3 shows the procedural version of F_{91} given in deterministic form.

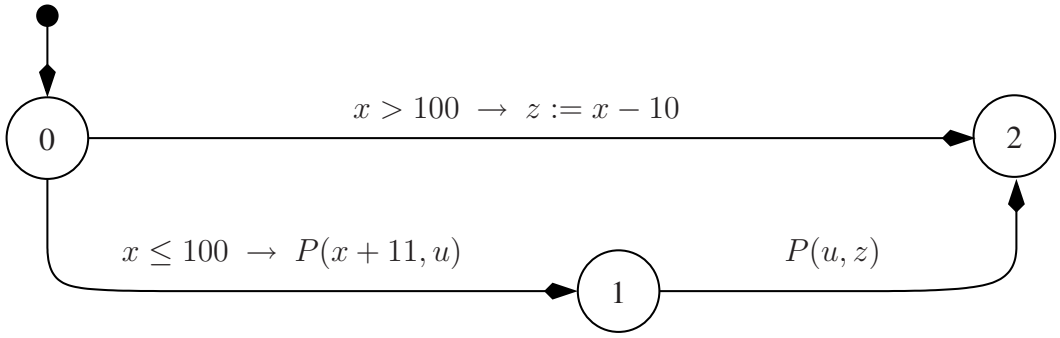


Figure 3.3: Procedural program F_{91} in deterministic form.

3.1.3.3 Computations

A *computation* of a program P is a maximal (possibly infinite) sequence of states and their labeled transitions:

$$\sigma : \langle \ell_0^0; (\xi, \vec{\perp}, \vec{\perp}) \rangle \xrightarrow{\lambda_1} \langle \ell^1; \vec{\nu}_1 \rangle \xrightarrow{\lambda_2} \langle \ell^2; \vec{\nu}_2 \rangle \cdots$$

where each $\vec{\nu}_i = (\xi_i, \eta_i, \zeta_i)$ is an interpretation of the variables $(\vec{x}, \vec{u}, \vec{z})$. The values $\vec{\perp}$ denote uninitialized values. Labels in the transitions are either names of edges in the program or the special label *return*. Each transition $\langle \ell; \vec{\nu} \rangle \xrightarrow{\lambda} \langle \ell'; \vec{\nu}' \rangle$ in a computation must be justified by one of the following cases:

Assignment: There exists an assignment edge e, such that $\ell = \ell_a$, $\lambda = e$, $\ell' = \ell_c$ and $\langle \vec{\nu}, \vec{\nu}' \rangle \models d(\vec{y}, \vec{y}')$.

Procedure Call: There exists a call edge e, such that $\ell = \ell_a$, $\lambda = e$, $\ell' = \ell_0^j$, and $\vec{\nu}' = (\xi', \vec{\perp}, \vec{\perp})$, where $\langle \vec{\nu}, \xi' \rangle \models d_{in}(\vec{y}, \vec{x}_2)$.

Return: There exists a procedure P_j (the procedure from which we return), such that $\ell = \ell_t^j$ (the terminal location of P_j). The run leading up to $\langle \ell; \vec{\nu} \rangle$ has a suffix of the form

$$\langle \ell_1; \vec{\nu}_1 \rangle \xrightarrow{\lambda_1} \underbrace{\langle \ell_0^j; (\xi; \vec{\perp}; \vec{\perp}) \rangle \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_k} \langle \ell; (\xi; \eta; \zeta) \rangle}_{\sigma_1}$$

such that the segment σ_1 is *balanced* (has an equal number of *call* and *return* labels), $\lambda_1 = e$ is a call edge, where $\ell' = \ell_c$, $\lambda =$ *return*, and $\langle \vec{\nu}_1, \zeta, \vec{\nu}' \rangle \models d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$.
This definition uses the computation itself in order to retrieve the context as it was before the corresponding call to procedure P_j.

For a run $\sigma_1 : \langle \ell_0^0; (\xi, \vec{\perp}, \vec{\perp}) \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \langle \ell; \vec{\nu} \rangle$, we define the *level* of state $\langle \ell; \vec{\nu} \rangle$, denoted *Lev*$(\langle \ell; \vec{\nu} \rangle)$, to be the number of "call" edges in σ_1 minus the number of "return" edges.

3.2 Verifying Termination

This section presents a method for verifying termination of procedural programs. Initially, the system is augmented with well-founded ranking components. Then a finitary state abstraction is applied, resulting in a finite-state procedural program. Procedure summaries are computed over the abstract, finite-state program, and a procedure-free

FDS is constructed. Finally, infeasibility of the derived FDS is checked, showing that it does not possess a fair divergent computation. This establishes the termination of the original program.

3.2.1 A Proof Rule for Termination

The application of a ranking abstraction to procedures is based on a rule for proving termination of loop-free procedural programs. We choose a well founded domain $(\mathcal{D}, \succ)$, such that for each procedure P_i with input parameters $\vec{x}$, we associate a *ranking function* δ_i that maps $\vec{x}$ to $\mathcal{D}$. For each edge e in P_i, labeled by a procedure call we generate the descent condition $D_e(\vec{y}) : d_{in}(\vec{y}, \vec{x}_2) \implies \delta_i(\vec{x}) \succ \delta_j(\vec{x}_2)$.
The soundness of this proof rule is stated by the following claim:

Claim 3.2 (Termination)**.** If the descent condition $D_e(\vec{y})$ is valid for every procedure call edge e in a loop-free procedural program P, then P terminates.

Proof. A non-terminating computation of a loop-free program must contain a subsequence of the form

$$\langle \ell_0^0; (\xi_0, \vec{\perp}, \vec{\perp})\rangle, \ldots, \langle \ell_{i_0}^0; (\xi_0, \eta_0, \zeta_0)\rangle, \langle \ell_0^{j_1}; (\xi_1, \vec{\perp}, \vec{\perp})\rangle, \ldots, \langle \ell_{i_1}^{j_1}; (\xi_1, \eta_1, \zeta_1)\rangle,$$
$$\langle \ell_0^{j_2}; (\xi_2, \vec{\perp}, \vec{\perp})\rangle, \ldots, \langle \ell_{i_2}^{j_2}; (\xi_2, \eta_2, \zeta_2)\rangle, \langle \ell_0^{j_3}; (\xi_3, \vec{\perp}, \vec{\perp})\rangle, \ldots$$

where, for each $k \geq 0$, $\mathit{Lev}(\langle \ell_0^{j_k}; (\xi_k, \vec{\perp}, \vec{\perp})\rangle) = \mathit{Lev}(\langle \ell_{i_k}^{j_k}; (\xi_k, \eta_k, \zeta_k)\rangle) = k$. If the descent condition is valid for all call edges, this leads to the existence of the infinitely

descending sequence

$$\delta_0(\xi_0) \succ \delta_{j_1}(\xi_1) \succ \delta_{j_2}(\xi_2) \succ \delta_{j_3}(\xi_3) \succ \cdots$$

which contradicts the well-foundedness of the δ_i's. □

3.2.2 Ranking Augmentation of Procedural Programs

Ranking augmentation was suggested in [KP00b] and used in [BPZ05] in conjunction with predicate abstraction to verify liveness properties of non-procedural programs. In its application here we require that a ranking function be applied only over the input parameters. Each procedure is augmented with a *ranking observer* variable that is updated at every procedure call edge e, in a manner corresponding to the descent condition D_e. For example, if the observer variable is *inc* then a call edge

$$d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2; \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$$

is augmented to be

$$d_{in}(\vec{y}, \vec{x}_2) \wedge \mathit{inc}' = \mathit{sign}(\delta(\vec{x}_2) - \delta(\vec{x})); \; P_j(\vec{x}_2; \vec{z}_2); \; d_{out}(\vec{y}, \vec{z}_2, \vec{y}') \wedge \mathit{inc}' = 0$$

All local assignments are augmented with the assignment *inc* $:= 0$, as the ranking does not change locally in a procedure. Well foundedness of the ranking function is captured

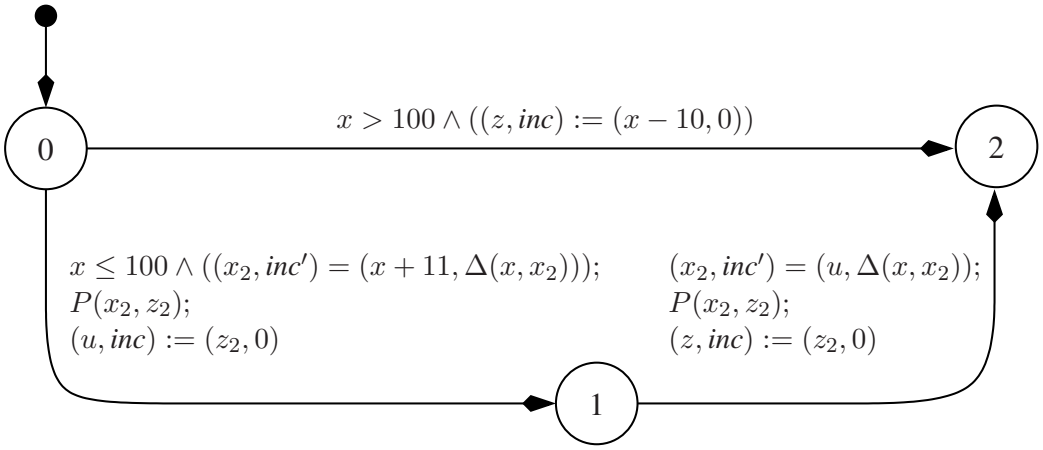


Figure 3.4: Program F_{91} augmented by a ranking observer. The notation $\Delta(x_1, x_2)$ denotes the expression $\mathit{sign}(\delta(x_2) - \delta(x_1))$

by the compassion requirement $(\mathit{inc} < 0, \mathit{inc} > 0)$ which is being imposed only at a later stage.

Unlike the termination proof rule, the ranking function need not decrease on every call edge. Instead, a program can be augmented with multiple similar components, and it is up to the feasibility analysis to sort out their interaction and relevance automatically.

Example 3.5 (Ranking Augmentation of Program F_{91}). We now present an example of ranking abstraction applied to program F_{91} of Fig. 3.3. As a ranking component, we take

$$\delta(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } 0 \textbf{ else } 101 - x$$

Fig. 3.4 presents the program augmented by the variable *inc*. □

3.2.3 Predicate Abstraction of Augmented Procedural Programs

We consider the application of finitary abstraction to procedural programs, focusing on predicate abstraction for clarity. We assume a predicate base that is partitioned into $\vec{T} = \{\vec{I}(\vec{x}), \vec{W}(\vec{y}), \vec{R}(\vec{x}, \vec{z})\}$, with corresponding abstract (boolean) variables $\vec{b}_{T} = \{\vec{b}_{I}, \vec{b}_{W}, \vec{b}_{R}\}$. For each procedure the input parameters, working variables, and output parameters are $\vec{b}_{I}$,$\vec{b}_{W}$, and $\vec{b}_{R}$, respectively.

An abstract procedure will have the same control-flow graph as its concrete counterpart, where only labels along the edges are abstracted as follows:

- A local change relation $d(\vec{y}, \vec{y}')$ is abstracted into the relation

$$D(\vec{b}_{T}, \vec{b}'_{T}) : \quad \exists \vec{y}, \vec{y}' \; : \; \vec{b}_{T} = \vec{T}(\vec{y}) \wedge \vec{b}'_{T} = \vec{T}(\vec{y}') \wedge d(\vec{y}, \vec{y}')$$

- A procedure call $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$ is abstracted into the abstract procedure call $D_{in}(\vec{b}_{T}, \vec{b}^2_{I}); P_j(\vec{b}^2_{I}, \vec{b}^2_{R}); D_{out}(\vec{b}_{T}, \vec{b}^2_{R}, \vec{b}'_{T})$, where

$$\begin{array}{lcl} D_{in}(\vec{b}_{T}, \vec{b}^2_{I}) & : & \exists \vec{y}, \vec{x}_2 \; : \; \vec{b}_{T} = \vec{T}(\vec{y}) \wedge \vec{b}^2_{I} = \vec{I}(\vec{x}_2) \wedge d_{in}(\vec{y}, \vec{x}_2) \\ D_{out}(\vec{b}_{T}, \vec{b}^2_{R}, \vec{b}'_{T}) & : & \exists \vec{y}, \vec{x}_2, \vec{z}_2, \vec{y}' \; : \; \left(\begin{array}{l} \vec{b}_{T} = \vec{T}(\vec{y}) \wedge \vec{b}^2_{R} = \vec{R}(\vec{x}_2, \vec{z}_2) \wedge \vec{b}'_{T} = \vec{T}(\vec{y}') \wedge \\ d_{in}(\vec{y}, \vec{x}_2) \wedge d_{out}(\vec{y}, \vec{z}_2, \vec{y}') \end{array} \right) \end{array}$$

Example 3.6 (Abstraction of Program F_{91}). We apply predicate abstraction to program

F_{91} of Fig. 3.3. As a predicate base, we take

$$\vec{I} : \{x > 100\}, \qquad \vec{W} : \{u = g(x + 11)\}, \qquad \vec{R} : \{z = g(x)\}$$

where

$$g(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } 91$$

The abstract domain consists of the corresponding boolean variables $\{B_I, B_W, B_R\}$. The abstraction yields the abstract procedural program $P(B_I, B_R)$ which is presented in Fig. 3.5. □

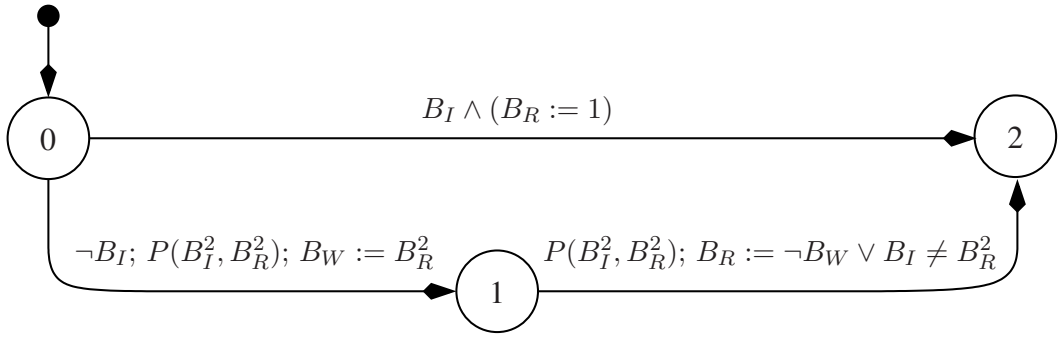


Figure 3.5: An abstract version of program F_{91}.

Finally we demonstrate the joint (predicate and ranking) abstraction of program F_{91}.

Example 3.7 (Abstraction of Ranking-Augmented Program F_{91}). We wish to abstract the augmented program from Example 3.5. When applying the abstraction based on the

predicate set

$$\vec{I} : \{x > 100\}, \qquad \vec{W} : \{u = g(x + 11)\}, \qquad \vec{R} : \{z = g(x)\}$$

we obtain the abstract program presented in Fig. 3.6, where

$$\begin{array}{lllll} f(B_I, B_W, B_I^2) & = & \textbf{if} & \neg B_I \wedge (B_I^2 \vee \neg B_I^2 \wedge B_W) & \textbf{then} \quad -1 \\ & & \textbf{else if} & B_I \wedge B_I^2 & \textbf{then} \quad 0 \\ & & & & \textbf{else} \quad 1 \end{array}$$

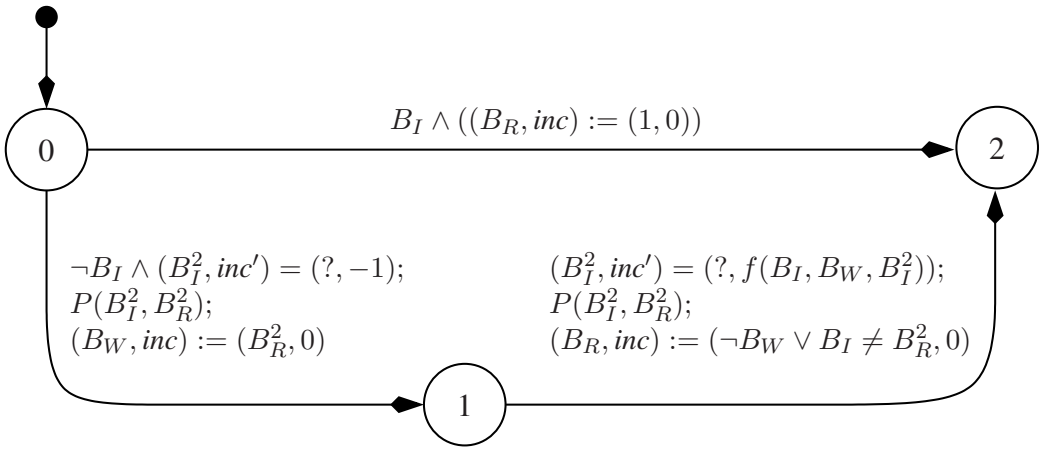


Figure 3.6: An abstract version of program F_{91} augmented by a ranking observer.

Note that some (in fact, all) of the input arguments in the recursive calls are left non-deterministically 0 or 1. In addition, on return from the second recursive call, it is necessary to augment the transition with an adjusting assignment that correctly updates the local abstract variables based on the returned result.

It is interesting to observe that all terminating calls to this abstract procedure return $B_R = 1$, thus providing an independent proof that program F_{91} is partially correct with respect to the specification $z = g(x)$.

The analysis of this abstract program yields that $\neg B_I \wedge B_W$ is an invariant at location 1. Therefore, the value of $f(B_I, B_W, B_I^2)$ on the transition departing from location 1 will always be -1. Thus, it so happens that even without feasibility analysis, from Claim 3.2 we can conclude that the program terminates. □

3.2.4 Summaries

A procedure *summary* is a relation between input and output parameters. A relation $q(\vec{x}, \vec{z})$ is a summary if it holds for any $\vec{x}$ and $\vec{z}$ iff there exists a run in which the procedure is called and returns, such that the input parameters are assigned $\vec{x}$ and on return the output parameters are assigned $\vec{z}$.

Since procedures may contain calls (recursive or not) to other procedures, deriving summaries involves a fixpoint computation. An *inductive assertion network* is generated that defines, for each procedure P_j, a summary q^j and an assertion φ_a^j associated with each location ℓ_a. For each procedure we construct a set of constraints according to the rules of Table 3.1. The constraint $\varphi_t^j(\vec{x}, \vec{u}, \vec{z}) \implies q^j(\vec{x}, \vec{z})$ derives the summary from the assertion associated with the terminating location of P_j. All assertions, beside φ_0^j, are initialized FALSE. φ_0^j, which refers to the entry location of P_j, is initialized TRUE, i.e. it allows the input variables to have any possible value at the entry location of proce-

dure P_j. Note that the matching constraint for an edge labeled with a call to procedure $P_i(\vec{x}_2; \vec{z}_2)$ encloses the summary of that procedure, i.e. the summary computation of one procedure comprises summaries of procedures being called from it.

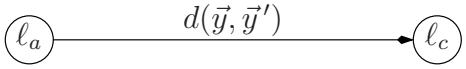
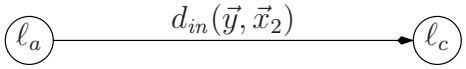
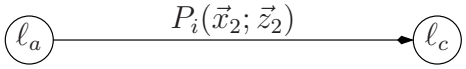
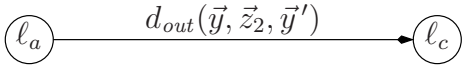
Fact	**Constraint(s)**
	$\varphi_0^j = \mathsf{true}$ $\varphi_t^j(\vec{x}, \vec{u}, \vec{z}) \implies q^j(\vec{x}, \vec{z})$
	$\varphi_a^j(\vec{y}) \wedge d(\vec{y}, \vec{y}') \implies \varphi_c^j(\vec{y}')$
	$\varphi_a^j(\vec{y}) \wedge d_{in}(\vec{y}, \vec{x}_2) \implies \varphi_c^j(\vec{y}, \vec{x}_2)$
	$\varphi_a^j(\vec{y}, \vec{x}_2) \wedge q^i(\vec{x}_2, \vec{z}_2) \implies \varphi_c^j(\vec{y}, \vec{z}_2)$
	$\varphi_a^j(\vec{y}, \vec{z}_2) \wedge d_{out}(\vec{y}, \vec{z}_2, \vec{y}') \implies \varphi_c^j(\vec{y}')$

Table 3.1: Rules for constraints contributed by Procedure P_j to the inductive assertion network.

An iterative process is performed over the constraints contributed by all procedures in the program, until a fixpoint is reached. Reaching a fixpoint is guaranteed since all variables are of finite type.

Claim 3.3 (Soundness)**.** Given a minimal solution to the constraints of Table 3.1, q_j is a summary of P_j, for each procedure P_j.

Proof. In one direction, let $\sigma : s_0, \ldots, s_t$ be a computation segment starting at location ℓ_0^j and ending at ℓ_t^j, such that $\vec{x}[s_0] = \vec{v}_1$ and $\vec{z}[s_t] = \vec{v}_2$. It is easy to show by induction on the length of σ that $s_t \models \varphi_t^j(\vec{x}, \vec{u}, \vec{z})$. From Table 3.1 we obtain $\varphi_t^j(\vec{x}, \vec{u}, \vec{z}) \implies$

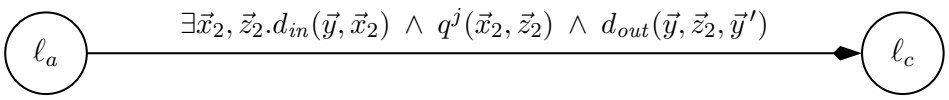
$q^j(\vec{x}, \vec{z})$. Therefore $s_t \models q^j(\vec{x}, \vec{z})$. Since all edges satisfy $\vec{x} = \vec{x}'$, we obtain $[\vec{x} \mapsto \vec{v}_1, \vec{y} \mapsto \vec{v}_2] \models q^j(\vec{x}, \vec{y})$.

In the other direction, assume $[\vec{x} \mapsto \vec{v}_1, \vec{y} \mapsto \vec{v}_2] \models q^j(\vec{x}, \vec{y})$. From the constraints in Table 3.1 and the minimality of their solution, there exists a state s_t with $\vec{x}[s_t] = \vec{v}_1$ and $\vec{z}[s_t] = \vec{v}_2$ such that $s_t \models \varphi_t^j$. Repeating this reasoning we can, by propagating backward, construct a computation segment starting at ℓ_0 that initially assigns $\vec{v}_1$ to $\vec{x}$. □

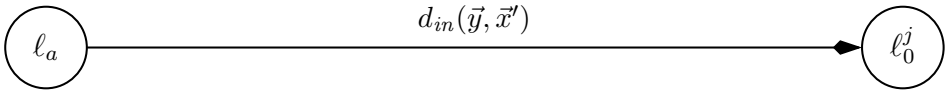
3.2.5 Deriving a Procedure-Free FDS

Using summaries of an abstract procedural program P_A, one can construct the *derived* FDS of P_A, labeled $\mathsf{derive}(P_A)$. This is an FDS denoting the set of *reduced computations* of P_A, a notion formalized in this section. The variables of $\mathsf{derive}(P_A)$ are partitioned into $\vec{x}, \vec{y}$, and $\vec{z}$, each of which consists of the input, working, and output variables of all procedures, respectively. The FDS is constructed as follows:

- Edges labeled by local changes in P_A are preserved in $\mathsf{derive}(P_A)$
- A procedure call in P_A, denoted by a sequence of edges of the form $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$ from a location ℓ_a to a location ℓ_c, is transformed into the following edges:
  - A *summary* edge, specified by



– A *call* edge, specified by



- All compassion requirements, which are contributed by the ranking augmentation and described in Subsection 3.2.2, are imposed on $\mathsf{derive}(P_A)$.

The reasoning leading to this construction is that summary edges represent procedure calls that return, while call edges model non-returning procedure calls. Therefore, a summary edge leads to the next location in the calling procedure while modifying its variables according to the summary. On the other hand, a call edge connects a calling location to the entry location of the procedure that is being called. Thus, a nonterminating computation consists of infinitely many call edges, and a call stack is not necessary.

We now prove soundness of the construction. Recall the definition of a computation of a procedural program given in Subsection 3.1.3.3. A computation can be terminating or non-terminating. A terminating computation is finite, and has the property that every computation segment can be extended to a *balanced* segment, which starts with a calling step and ends with a matching return step. A computation segment is *maximally balanced* if it is balanced and is not properly contained in any other balanced segment.

Definition 3.4. Let σ be a computation of P_A. Then the *reduction* of σ, labeled $\mathsf{reduce}(\sigma)$, is a sequence of states obtained from σ by replacing each maximal balanced segment by a summary-edge traversal step.

Claim 3.5. For any sequence of states σ, σ is a computation of $\mathsf{derive}(P_A)$ iff there exists σ', a computation of P_A, such that $\mathsf{reduce}(\sigma') = \sigma$.

Proof of the claim follows from construction of $\mathsf{derive}(P_A)$ in a straightforward manner. It follows that if σ is a terminating computation of P_A, then $\mathsf{reduce}(P_A)$ consists of a single summary step in the part of $\mathsf{derive}(P_A)$ corresponding to P_0. If σ is an infinite computation of P_A, then $\mathsf{reduce}(\sigma)$ (which must also be infinite) consists of all assignment steps and calls into procedures from which σ has not returned.

Claim 3.6 (Soundness – Termination)**.** If $\mathsf{derive}(P_A)$ is infeasible then P_A is a terminating program.

Proof. Let us define the notion of abstraction of computations. Let $\sigma = s_0, s_1, \ldots$ be a computation of P, the original procedural program from which P_A was abstracted. The abstraction of σ is a computation $\alpha(s_0), \alpha(s_1), \ldots$ where for all $i \geq 0$, if s_i is a state in σ, then $\alpha(s_i) = [\vec{b}_I \mapsto \vec{I}(\vec{x}), \vec{b}_W \mapsto \vec{W}(\vec{y}), \vec{b}_R \mapsto \vec{R}(\vec{x}, \vec{z})]$.

Assume that $\mathsf{derive}(P_A)$ is infeasible. Namely, every infinite run of $\mathsf{derive}(P_A)$ violates a compassion requirement. Suppose that P has an infinite computation σ. Consider $\mathsf{reduce}(\sigma)$ which consists of all steps in non-terminating procedure invocations within σ. Since the abstraction of $\mathsf{reduce}(\sigma)$ is a computation of $\mathsf{derive}(P_A)$ it must be unfair

with respect to some compassion requirement. It follows that a ranking function keeps decreasing over steps in $\mathsf{reduce}(\sigma)$ and never increases – a contradiction. □

3.2.6 Analysis

The feasibility of $\mathsf{derive}(P_A)$ can be checked by conventional symbolic model-checking techniques. If it is feasible then there are two possibilities: (1) The original system truly diverges, or (2) feasibility of the derived system is *spurious*, that is, state and ranking abstractions have admitted behaviors that were not originally present. In the latter case, the method presented here can be repeated with a refinement of either state or ranking abstractions. The precise nature of such refinement is outside the scope of this dissertation.

3.3 LTL Model Checking

In this section we generalize the method discussed so far to general LTL model-checking. To this end we adapt to procedural programs the method discussed in Section 1.3 for model-checking LTL by composition with temporal testers [KPR98]. We prepend the steps of the method in Section 3.2 with a *tester composition step* relative to an LTL property. Once ranking augmentation, abstraction, summarization, and construction of the derived FDS are computed, the resulting system is model-checked by conventional means, verifying absence of feasible initial states that do not satisfy the property.

The main issue is that synchronous composition of a procedural program with a

global tester, including justice requirements, needs to be expressed in terms of local changes to procedure variables. In addition, since LTL is modeled over infinite sequences, the derived FDS needs to be extended with idling transitions.

3.3.1 Composition with Temporal Testers

A temporal tester is defined by a unique global variable, here labeled t, a transition relation $\rho(\vec{z}, t, \vec{z}', t')$[1] over primed and unprimed copies of the tester and program variables, where t does not appear in $\vec{z}$, and a justice requirement. In order to simulate global composition with ρ, we augment every procedure with the input and output parameters t_i and t_o, respectively, as follows:

- An edge labeled by a local change is augmented with $\rho(\vec{z}, t_o, \vec{z}', t_o')$
- A procedure call of the form $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{x}_2, \vec{y}')$ is augmented to be $d_{in}(\vec{y}, \vec{x}_2) \ \wedge \ \rho(\vec{z}, t_o, \vec{x}_2, t_i^2); P_j((\vec{x}_2, t_i^2), (\vec{z}_2, t_o^2)); d_{out} \ \wedge \ \rho(\vec{z}_2, t_o^2, \vec{z}', t_o')$
- Any edge leaving the initial location of a procedure is augmented with $t_o = t_i$

Example 3.8. Consider the program in Fig. 3.7. Since this program does not terminate, we are interested in verifying the property $\varphi : (\diamondsuit z) \vee \square \diamondsuit at\_\ell_2$, specifying that either eventually a state with $z = 1$ is reached, or infinitely often location 2 of P_1 is visited. To verify φ we decompose its negation into its principally temporal subformulas, $\square \neg z$ and $\diamondsuit \square \neg at\_\ell_2$, and compose the system with their temporal testers. Here we

[1] We assume here that the property to be verified is defined over the output variable only.

demonstrate the composition with $T[\Box\,\neg z]$, given by the transition relation $t = \neg z \ \wedge \ t'$ and the trivial justice requirement true. The composition is shown in Fig. 3.8.

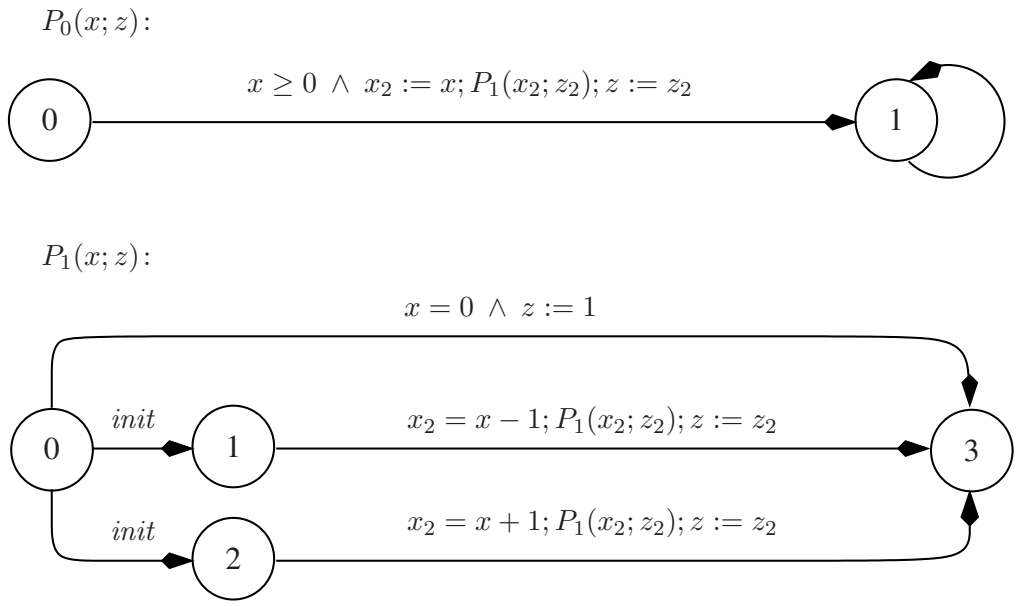


Figure 3.7: A divergent program. *init* represents $x > 0 \ \wedge \ z := 0$.

As a side remark, we note that our method can handle global variables in the same way as applied for global variables of testers, i.e., represent every global variable by a set of input and output parameters and augment every procedure with these parameters and with the corresponding transition relations.

3.3.2 Observing Justice

In order to observe justice imposed by a temporal tester, each procedure is augmented by a pair of *observer* variables that consists of a working and an output variables. Let

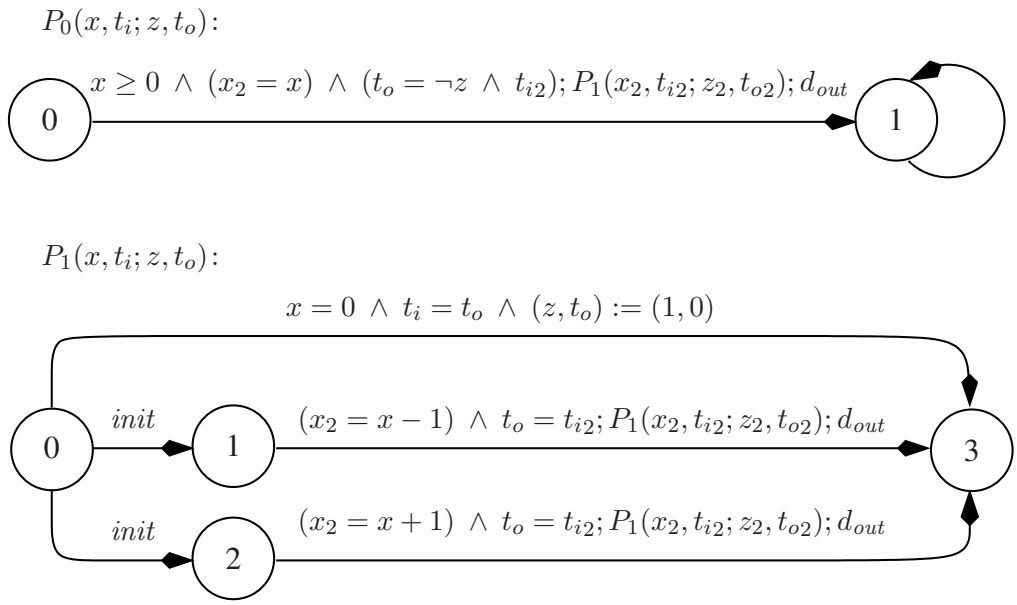


Figure 3.8: The program of Fig. 3.7, composed with $T[\Box \neg z]$. The assertion d_{out} represents $t_{o2} = (\neg z_2 \ \wedge \ t'_o) \ \wedge \ z := z_2$, and *init* represents $x > 0 \ \wedge \ t_i = t_o \ \wedge \ z := 0$.

J be a justice requirement, P_i be a procedure, and the associated observer variables be J_u and J_o. P_i is augmented as follows: On initialization, both J_u and J_o are assigned TRUE if the property J holds at that state. Local changes are conjoined with $J_u := J'$ and $J_o := (J_o \ \vee \ J')$. Procedure calls are conjoined with $J_u := (J' \ \vee \ J_o^2)$ and $J_o := (J_o \ \vee \ J' \ \vee \ J_o^2)$, where J_o^2 is the relevant output observer variable of the procedure being called.

While J_u observes J at every location, once J_o becomes TRUE it remains so up to the terminal location. Since J_o participates in the procedure summary, it is used to denote whether justice has been satisfied within the called procedure.

3.3.3 The Derived FDS

We use the basic construction here in deriving the FDS as in Section 3.2.5. In addition, for every non-output observer variable J_u we impose the justice requirement that in any fair computation, J_u must be TRUE infinitely often. Since LTL is modeled over infinite sequences, we must also ensure that terminating computations of the procedural program are represented by infinite sequences. To this end we simply extend the terminal location of procedure P_0 with a self-looping edge. Thus, a terminating computation is one that eventually reaches the terminal location of P_0 and stays idle henceforth.

3.4 Related work

Recent work by Podelski et al. [PSW05] generalizes the concept of summaries to capture effects of computations between arbitrary program points. This is used to formulate a proof rule for total correctness of recursive programs with nested loops, in which a program summary is the auxiliary proof construct (analogous to an inductive invariant in an invariance proof rule). The rule and accompanying formulation of summaries represent a framework in which abstract interpretation techniques and methods for ranking function synthesis can be applied. In this manner both [PSW05] and our work aim at similar objectives. The main difference from our work is that, while we strive to work with abstraction of predicates, and use relations (and their abstraction) only for the treatment of procedures, the general approach of [PSW05] is based on the abstraction of relations

even for the procedure-less case. A further difference is that, unlike our work, [PSW05] does not provide an explicit algorithm for the verification of aribtrary LTL properties. Instead it relies on a general reduction from proofs of termination to LTL verification.

Recursive State Machines (RSMs) [AEY01, ABE$^+$05] and Extended RSMs [ACEM05] enhance the power of finite state machines by allowing for the recursive invocation of state machines. More precisely, the RSMs model consists of a set of component machines where each has a set of nodes (atomic states), boxes (each mapped to a specific component machine), and edges which connect them. An edge entering a box models the invocation of the corresponding component, while an edge leaving a box models the return from that component. RSMs are used to model the control flow of programs containing recursive procedure calls, and to analyze reachability and cycle detection. In [ACEM05] they proposed algorithms for model checking and an implementation of an on-the-fly model checker using augmentation of RSMs, called Extended RSMs. Using this model checker they were able to check safety and liveness properties of recursive boolean programs. They are, however, limited to programs with finite data. On the other hand, the method that we present in this paper can be used to verify recursive programs with infinite data domains by making use of ranking and finitary state abstractions. In [AEM04] they introduced a temporal logic of calls and returns (CARET) for the specification and verification of structural programs. The basic concept of CARET is that the formulas are tagged with special symbols in order to allow a path to jump from a procedure call to its matching return. This feature can be used to specify regular properties of local runs within a procedure which skips over calls to other procedures.

In [BR00], an approach similar to ours for computing summary relations for procedures is implemented in the symbolic model checker Bebop. However, while Bebop is able to determine whether a specific program statement is reachable, it cannot prove termination of a recursive boolean program or of any other liveness property.

3.5 Conclusions

We have described the integration of ranking abstraction, finitary state abstraction, procedure summarization, and model-checking into a combined method for the automatic verification of LTL properties of infinite-state recursive procedural programs. Our approach is novel in that it reduces the verification problem of procedural programs with unbounded recursion to that of symbolic model-checking. Furthermore, it allows for application of ranking and state abstractions while still relegating all summarization computation to the model-checker. Another advantage is that fairness is being supported directly by the model, rather than being specified in a property.

We have implemented a prototype based on the TLV symbolic model-checker and tested several examples such as Ackerman's function, the Factorial function and a recursive formulation of the 91 function. We verified that they all terminate and model checked satisfiability of several LTL properties.

As further work it would be interesting to investigate concurrency with bounded context switching as suggested in [RQ05]. Another direction is the exploration of different versions of LTL that can relate to nesting levels of procedure calls, similar to the manner

in which the CARET logic [AEM04] expresses properties of recursive state machines concerning the call stack.

BIBLIOGRAPHY

[AAK+05] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005.

[ABE+05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T.W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[ACEM05] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *TACAS*, pages 61–76, 2005.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS*, pages 467–481, 2004.

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV*, pages 207–220, 2001.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AMP00] Rajeev Alur, Kenneth L. McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160(1-2):167–188, 2000.

[AR05] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct 2005.

[ATLM+06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press.

[BLM05] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

[BPZ05] I. Balaban, A. Pnueli, and L.D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'2005: Verification, Model Checking, and Abstraction Interpretation*, pages 164–180, 2005.

[BR00] T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[Car06] B. Carlstrom. The atomos transactional programming language, 2006.

[CPZ08] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Verifying transactional memories with TLPVS, 2008. Available at `http://cs.nyu.edu/acsys/tlpvs/tm.html`.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.

[DS06] David Dice and Nir Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[Fra03] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003.

[GC08] Justin Gottschlich and Daniel A. Connors. Extending contention managers for user-defined priority-based transactions. In *Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods April, 2008*. Apr 2008.

[GHJS08] R. Guerraoui, T. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *Programming Language Design and Implementation*, 2008. To Appear.

[GHKP05] Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon. Robust contention management in software transactional memory. In *Proceedings of the OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[GHP05] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, Sep 2005.

[GK08] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, 2008.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[HF03] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Anaheim, CA, October 2003.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522, Washington, DC, USA, 2003. IEEE Computer Society.

[HLMW03] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, Jul 2003.

[HM93a] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[HM93b] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[Hol98] Allen Holub. Programming java threads in the real world, part 2. World Wide Web electronic publication, 1998.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[HWC+04] L. Hammond, W.Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Herzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31st annu. Int. Symp. on COmputer Architecture*, page 102. IEEE Computer Society, June 2004.

[KCH+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, New York, NY, USA, 2006. ACM.

[KP00a] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 4(2):328–342, 2000.

[KP00b] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 1–16. Springer-Verlag, 1998.

[KPSZ02] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck. Network invariants in action. In *13th International Conference on Concurrency Theory (CONCUR02)*, volume 2421 of *Lect. Notes in Comp. Sci.*, pages 101–115. Springer-Verlag, 2002.

[KPV01] Yonit Kesten, Amir Pnueli, and Moshe Y. Vardi. Verification by augmented abstraction: The automata-theoretic view. *J. Comput. Syst. Sci.*, 62(4):668–690, 2001.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[Lam94] Leslie Lamport. Introduction to TLA. Technical Report 1994-001, Palo Alto, CA, 1994.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lie04] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, May 2004.

[LR07] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[MBM+06a] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Austin, TX, Feb 2006.

[MBM+06b] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370. ACM Press, New York, NY, USA, Oct 2006.

[MCC+06] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory, 2006.

[MH06] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MSH+06] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Jun 2006.

[MSS05] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*. LNCS, Springer, Sep 2005.

[NMA+07] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007.

[OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS *System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[PA03] A. Pnueli and T. Arons. TLPVS: A PVS-based LTL verification system. In *Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday*, Lect. Notes in Comp. Sci., pages 84–98. Springer-Verlag, 2003.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[Pnu05] Amir Pnueli. Verification of procedural programs. In *We Will Show Them! (2)*, pages 543–590. College Publications, 2005.

[PSW05] A. Podelski, I. Schaefer, and S. Wagner. Summaries for while programs with recursion. In *ESOP*, pages 94–107, 2005.

[RQ05] J. Rehof and S. Qadeer. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '06)*, pages 187–197. ACM, Mar 2006.

[Sco06] M.L. Scott. Sequential specification of transactional memory semantics. In *Proc. TRANSACT the First ACM SIGPLAN Workshop on Languages,*

Compiler, and Hardware Suppport for Transactional Computing, Ottawa, 2006.

[Sha00] E. Shahar. *The* TLV *Manual*, 2000. http://www.cs.nyu.edu/acsys/tlv.

[SMD+06] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear. Hardware acceleration of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Jun 2006.

[SP81] M. Sharir and A. Pnueli. Two approaches to inter-procedural data-flow analysis. In Jones and Muchnik, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[SS04] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[SS05a] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[SS05b] William N. Scherer III and Michael L. Scott. Randomization in stm contention management (poster). In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[SSH+07] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 104–115. ACM, Jun 2007.

[ST95] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. Princ. of Dist. Comp.*, pages 204–213, Ottawa Ontario CA, 1995. ACM Press.