

Using Formal Methods to Verify and Derive Open Nested Locking Protocols

Trek Palmer

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003

May 23, 2008

Abstract

Transactional memory systems (TM) offer an alternative to standard lock-based programming. However, the convenience and composability of a transactional memory system often come at the price of efficiency. TM systems detect and repair conflicts at the memory level, which can lead to spurious aborts (and expensive rollbacks). Open nesting is an extension to standard closed nesting TM that raises the level of abstraction and repair in order to avoid costly spurious rollbacks. However, much of the work that was automatically done by the TM system must now be performed by the programmer. In particular, the programmer will have to specify a locking protocol as well as compensating actions (to undo in case of an abort). Although this looks no better than standard lock-based programming, it is a far more structured approach. This structure can be exploited (I claim) to verify automatically and derive locking protocols for open nested data structures. This structure may be rich enough, in fact, to be able to verify and propose simple compensating actions (i.e., proving that inverse actions are actually inverses). I propose that an abstract state specification will allow automatic verification of concurrency control and compensating actions and furthermore may enable heuristic derivation of locking protocols and inverse actions.

1 Introduction

Concurrent programming is hard. However, manufacturers are turning to multi-threaded and multi-core chips to keep up with Moore's law. This sea change in commodity architecture is pushing concurrency to the fore. Unfortunately, standard lock-based coding techniques are complicated to reason about, implement, and debug. If concurrency is going to become a wide-spread programming technology, then easier models are needed. One of the more attractive approaches (to both industry and academia) is Transactional Memory [25].

Transactional Memory is not a silver bullet, however. As Section 2 will show, standard closed-nested TM has problems with efficiency, I/O, and integration with non-transactional systems. Open Nesting is an extension to conventional TM that allows expert programmers to overcome these limitations in a structured way. However, Open Nesting is more burdensome than conventional (closed) nesting. The programmer must provide code to undo operations as well as a locking protocol to ensure correctness of the transactional data. These locks are actually locks over the abstract state of the data structure (hence, they are called abstract locks) and are meant to prevent logically conflicting operations from executing concurrently. This differs from conventional TM which uses physical memory to over-approximate abstract conflicts (which reduces concurrency).

Open Nesting is a way for programmers to raise the level of abstraction in their code to improve efficiency and expressiveness. However, constructing a correct locking protocol can be tedious and subtle. I propose to adapt and modify verification techniques in order to verify proposed locking protocols against an abstract specification, and to attempt to automatically derive correct protocols from descriptions alone. This is possible because open nesting compartmentalizes the concurrency decisions a programmer has to make, which reduces a complex problem like lock correctness to a simple commutativity test. Given a description of the abstract state of a data structure and operations over that state, my system will translate that into a set of SAT variables and clauses. The locking protocol will also be translated into SAT, with the end result being a SAT problem that tests pairwise commutativity of operations in the presence of the locking protocol.

A verifier alone would be a wonderful tool for programmers, and it would eliminate a portion of the additional work required by open nesting, but a verifier that generates counter-examples could be used to drive a system for automatically generating locking protocols. This system would propose a protocol, verify it, and use any counter-examples generated by an incorrect protocol to further refine the locking predicates. After completing the verifier, I intend to investigate methods for inducing functions from counter-examples and to produce a tool that can infer reasonable protocols from an abstract state description.

The rest of the paper is organized as follows. Section 2 is an overview of transactional memory and provides a context for my proposed research. Section 3 covers related work on both transactional systems and software verification systems. Section 4 is a description of the language I have developed to describe the abstract state of a data structure. Section 5 describes my proposed approach to lock protocol verification, automatically generating locking protocols and verifying inverse actions. Section 6 describes the current state of my work along with some preliminary results, and lastly section 7 describes my plan for actually accomplishing this work in a reasonable amount of time!

2 Transactional Memory and Open Nesting Overview

Transactional Memory is a vibrant and growing field of research. In this section I attempt to describe some of the more important aspects of TM.

2.1 Transactional Memory

Transactional memory is a concurrent programming model that reinterprets transactional semantics from the database community in a programming language context. A transaction is simply a set of operations that are required to execute atomically. As in databases, the transaction can complete successfully (commit), or fail (abort) and undo its actions (rollback, which is then usually followed by a restart). Much of this infrastructure is provided by the transactional memory run-time which handles all the transaction processing details (thus easing the burden of the programmer). Ideally, the programmer needs to just label the sections of code that should execute atomically and the TM run-time will take care of the rest (of course, some changes may need to be made to improve efficiency). Transactional Memory systems are interested in preserving the so-called ACID (atomicity, consistency, isolation, and durability) properties first described in the database community [19, 53]. *Atomicity* is ensuring that the transaction appear to happen atomically (the all-or-nothing property). *Consistency* ensures that the transaction performs a consistent transformation of the state. In short, *consistency* ensures that a correct program running under a TM system will not behave incorrectly. From a programming language point of view, this means that the TM system must implement and respect the memory model of the language being transactionalized. *Isolation* implies that transactions cannot observe each other executing concurrently. More formally, transactions appear to execute in some serial order and some transaction T_1 will see other transactions executing either before or after T_1 , but not both. *Durability* is more commonly called persistence in programming languages. A durable system is one that retains successful state changes through failures. This is of less importance in transactional memory. In fact, current TM systems [43, 23, 24] are not durable at all (they do not commit program state to persistent media such as a disk, and we will work within that model).

Databases are interested in tracking accesses and modifications to tables and fields, and by logging these operations databases can ensure transactional semantics. TM systems have a more open-ended domain and so have to track individual memory accesses. There are several competing implementation strategies, but abstractly a TM run-time associates two sets with each running transaction. One set records all the read accesses in program order (the read set), the other set records all the writes as well as the overwritten values (the write set). Conflict detection then checks two properties: whether a given transaction's read set overlaps with another transaction's write set (possibly read a stale value), or whether a given transaction's write set overlaps with another transaction's write set (another kind of data race). Whether or not one performs this check when logging each new read/write or waits until the end of the transaction is an implementation issue. Abstractly, if a conflict is detected, one of the transactions is chosen to be aborted and restarted. Rollback is simple because the TM run-time has access to the old values in the write set, the run-time simply writes out the old values back into memory as it traverses the write log in reverse order (e.g., last write first). Obviously, there are many complex details involved in implementing this abstraction efficiently.

2.2 Closed Nesting

Transaction nesting is almost essential in a TM system, executing one transaction within another maps very nicely onto standard function calls (where the functions may contain their own transactions). In terms of the read set/write set abstraction, a nested transaction maintains its own read and write sets while executing. Conflict detection is a little different for nested transactions. Basically, a nested transaction cannot conflict with its parent. To ensure this the parent's read/write sets must be considered part of the child's for the purposes of conflict detection. On abort, a nested transaction simply rolls back its own write set (note that this property means that nested transactions are a way of implementing partial rollback). On commit, the nested transaction appends its read/write sets to its parent's sets. This is necessary for correct conflict detection and rollback when the parent transaction tries to commit.

Nested transactions have an additional benefit. They allow atomic actions to be trivially composed. One simply calls the actions one wishes to compose from inside a higher level transaction. This is a huge advantage over lock-based systems, where it can be impossible to implement additional functionality on top of a concurrent data structure (without raising the possibility of deadlock). In a transactional system, one simply lumps together the actions in a larger, higher-level transaction.

2.3 Open Nesting

In the database community (where most of the transactional work was done), open nesting is a broad term that encompasses any multi-level transactional scheme where nested transactions relax one of the transaction properties. In this work, open nesting means relaxing the isolation property [39]. More specifically, this means that an open nested transaction's memory actions become visible to the whole world when it commits. This means that the TM run-time is unaware of the memory operations, and therefore cannot use them to trigger spurious aborts (which is good). Unfortunately, this also means that the TM run-time can no longer automatically roll back a transaction or detect a conflict between open-nested actions. This is why the programmer must provide locking semantics as well as inverse actions (to undo the transaction if the enclosing action aborts). However, this added responsibility can be more cleanly encapsulated than in standard locking.

Closed nested actions simply require the programmer to label the code as being atomic. Open nested actions need two more pieces of information, namely: compensating actions to execute on commit or abort, and abstract locks to acquire. Compensating action is a technical term that basically means inverse. The open nested developer must specify inverse actions for each forward going action; these will allow the TM system to restore the data structure state if the parent transaction (the caller of the open nested action) aborts. Abstract locks are a more complicated issue. Because the programmer is working without the automatic conflict detection of the TM run-time, there needs to be some way to prevent conflicting actions from executing concurrently. An abstract lock is actually more like a predicate than it is like a classical mutex/semaphore. The predicate is evaluated in the context of the current state of the data structure and the arguments to the open nested action. If the predicate is true, then the open nested action acquires the lock and proceeds

(otherwise the lock manager takes over and makes the action wait until it can proceed or chooses a victim transaction to abort).

Open nesting offers an ‘escape hatch’ by which experienced programmers can implement their own, more sophisticated, locking schemes and rollback procedures. Open nesting is useful if one is a competent programmer with a much better grasp of the abstract semantics of a data structure than the TM run-time has. Seen in this light, open nesting is a way of raising the level of abstraction in a concurrent system (above the raw memory level where the standard TM run-time operates). It makes sense then to think about abstract state rather than pure memory state. It is also important to remember that it is possible that many concrete memory states may all correspond to the same abstract state. For instance, given an open-nested implementation of a B-tree, the inverse of an insert operation is a remove operation. In a closed nested scheme, rolling back an aborted insert would undo all of the memory operations returning the data structure to its initial memory state. In an open scheme, one simply runs remove as an additional forward-going action. This may result in a different memory layout; however, the abstract state of the tree will be identical to the closed nested case. This ability to distinguish memory conflicts from abstract state conflicts is the source of the efficiency of open-nested actions.

2.4 TM implementation

Implementation of a TM system is a complex engineering task. In recent years it has become common-place to build a library [20, 23, 43, 24] implementing methods to track memory accesses and detect conflicts. A compiler that understands the transactional syntax extensions will then generate normal code decorated with appropriate calls to the TM library system. Depending on the language, a TM library may need to interface with the runtime system. The McRT-STM system, for instance uses the ORP [6] JVM to automatically transactionalize loaded classes and to respect TM references during garbage collection. Another approach [23, 24] is to require the programmers to ‘transactionalize’ an object (usually by passing it through a factory). The library generates a transactional wrapper around the non-transactional object.

Implementations also differ as to whether they modify the data in place or first make a copy which is modified. This difference is also related to whether or not the system supports write locking or is geared towards a more non-blocking style. In terms of implementation, a modify-in-place system will log the old value and actually write out the new value (these systems usually also lock objects for writing). Non-blocking systems will first copy the entire object, modify the private copy and then atomically switch the global pointer on commit. If write conflicts are relatively rare, then all the additional copying is an unnecessary cost. Additionally, it is much easier to extend a modify-in-place system to support open-nesting. Open-nested actions need to become visible to the world when they commit, which in a modify-in-place system is automatic (if commit releases locks acquired by the open-nested action). It is not at all clear how to conveniently modify a non-blocking system to support open-nesting.

2.4.1 XJ

XJ is the tentative name for a platform-neutral Java TM runtime. It is implemented as a library that performs read and write logging to track memory operations. By default, XJ supports optimistic reads (no locking) and pessimistic writing (locking). This means that read conflicts are detected at commit time and write conflicts are detected immediately. This mixture was inspired by the McRT-STM work whose performance numbers indicated that this was the best configuration for Java code. Because XJ has no access to JVM internals, I am currently working on implementing a rewriting class-loader to automatically instrument Java classes to support transactional semantics. This is complicated by some of the oddities of Java class loading [15], but I am using a technique known as the Twin Class Hierarchy to avoid many of the pitfalls. I wrote XJ because none of the other freely available Java transactional systems provided support for modification in-place (which is vital for Open Nesting work).

3 Related Work

This work occurs in the context of intense academic interest in concurrent programming and transactional memory in particular. However, as the proposed work seeks to automatically verify and derive locking protocols it has a strong relation to topics in Software Verification and Program Analysis. This section will try to cover topics both transactional and formal. First, I will discuss directly related work from other Transactional Memory researchers. Second, I will describe the foundations of transactional research established by the data base community. Third, I will describe related efforts in Software Verification and Program Analysis and compare them to my proposed work.

3.1 Transactional Memory

Transactional memory was first formulated as a hardware concept in 1993 by Herlihy and Moss [25]. Since then, both hardware and software implementations of transactional semantics have been introduced. That initial work was itself grounded in the previous two decades of database-oriented transaction research (described later). Early TM work had many limitations: either it supported only fixed-size transactions [25] and/or had no support for standard closed-nesting [46]. Recent hardware [35] and software [24, 43] systems support both unbounded transactions as well as closed-nesting. The two current challenges attracting much attention are TM performance and integrating transactional and non-transactional code. Researchers have started to address performance questions with STM implementation [43, 21] and compiler work [1], however neither of these techniques can catch and avoid the spurious conflicts that open nesting addresses.

Integration of transactional and non-transactional systems has been more problematic. The primary issue is making sure that non-transactional code does not trample transactional meta-data or operate on partial state (uncommitted transactional data). One approach, dubbed ‘strong atomicity’ has been to make sure that all memory accesses are logged so that the TM system is aware of them and can abort/commit transactions appropriately. This increases the overhead of

standard memory operations but this additional cost can be reduced somewhat [47]. Some recent work on programming language semantics [34] has revealed that as long as transactional and non-transactional code don't share data, then there is no difference between strongly and weakly atomic systems. Note that neither of these results address the question of irrevocable actions (e.g. an I/O that cannot be undone) occurring within a transaction. Transactions with isolation and cooperation (TIC) [48], is an interesting effort to address these problems. In TIC, a transaction is effectively split around an irrevocable action, with the before portion committing before the I/O and the after portion proceeding as a normal transaction. Interestingly, TIC requires the programmer to annotate I/O actions so that they restore invariants expected by the transactional code. This resembles open nesting, and in fact, TIC is implemented in terms of open nesting.

Transactional Boosting [12], like open nesting, attempts to address performance and communication simultaneously. Boosting works by annotating normal lock-based code with commutativity rules. These rules are used by the TM runtime to determine if two lock-based actions can occur simultaneously (if they commute, they can be concurrently executed). Initially only the commutativity annotations were used, however as boosting was extended, compensating actions were found to be required. The annotations needed are basically the same as those in open nesting [38], but boosting lacks the ability to support rich state-based locking protocols. Additionally, at a recent presentation of the boosting work, questions about the safety of the method (similar to weak vs. strong atomicity) were raised.

As I was performing the literature search to compose this related work section, I encountered many efforts to address the shortcomings of transactional memory [10, 5, 42, 48, 12]. I was most struck by the fact that unlike the specialized mechanisms often employed to solve a particular problem, Open Nesting provided a generalized TM escape hatch that can be used both to speed up transactional code as well as communicate with non-transactional code in a structured, analyzable way.

3.2 Database Research

Transactions as programming constructs have existed in the database community for decades, and much of the formal analysis and practical implementation issues were first worked out in databases years ago. Much of this foundational work is summarized nicely in the textbooks by Gray and Reuter [19] and Weikum and Vossen [54]. Although transactional memory borrows heavily from the prior work of the database community, there are subtle differences. First, in databases, it is customary to think of transactions as the fundamental unit of concurrency. Programmers submit transactions to the database, which internally processes them. In this model, threads are used to process transactions and threads may move from one outstanding transaction to another. The database uses threads to enhance the concurrency of transactions. In a programming language context, it is often the case that threads are considered fundamental and therefore transactions are associated with specific threads. Threads use transactions to aid concurrency. This difference leads to different uses of transactional semantics. Nesting transactions [37] maps nicely onto function calls in a PL context, and many systems intend for them to be used primarily in that way. In databases, nested transactions often correspond to different levels of abstraction and allow one to isolate low level operations (such as 'subtract \$500 from field: balance') from higher level

operations (such as ‘process debit card transaction’).

Transactions were first formally analyzed by database researchers. Primarily, a concept of abstract serializability was derived. Through this formal structure, one could prove that transactions could be safely interleaved by mapping any interleaving onto some legal serial ordering. Importantly (for this work), it was discovered that if the operations of an outstanding transaction commuted with the operations from another transaction that they could be interleaved safely (while still preserving the transaction-internal order of operations). In an effort to overcome the efficiency burdens of classical transactions, database researchers attempted to move to locking data at finer levels of granularity (down to the field level). However, this increased the burden of the programmer and led to complicated locking issues. For instance, if a transaction needed to grab all rows in a table with a field of a certain value, it also had to block other transactions from adding fields which would contain that value. These were termed ‘phantom’ entries and abstractly could be blocked by obtaining a predicate lock on the data. A predicate lock is a predicate that would inform other transactions which data was locked within the table. However, predicate-locking (even with restricted operations) is in NP (it can be reduced to SAT) [27, 54] and so it hasn’t achieved much acceptance in the database community. However, the problem is somewhat different in the TM case. Whereas, in databases, the database must be able to handle essentially arbitrary predicates flowing in from users (in the form of SQL statements), a TM system has only to work with statically known values which can be analyzed and processed ahead of time as part of compilation. I claim this difference is sufficient to allow us to utilize a limited form of predicate locking for open nested actions.

Commercial Databases have long had high-performance extensions to standard transactions. Like open-nesting, these schemes were intended to be used only by expert programmers, who would then present a more traditional interface to people writing normal queries (similar to a library procedure in a modern programming language). At a very low level, databases use techniques like index logging [22] to track physical layout changes internal to the DB. By giving programmers access to such low-level information, standard concurrency bottlenecks can be bypassed. A standard example from the DB community is hand-over-hand locking (also known as ‘crabbing’) for B-trees rather than locking everything from the root down. Additionally, most commercial databases also include support for stored procedures. A stored procedure is a block of code (usually not written in SQL) that executes within the DB and with access to the DB internals. Again, these potentially unsafe techniques exist primarily because databases are focused on providing extremely high performance to experienced developers.

3.3 Formal Methods and Program Analysis

The work proposed in this document can be considered a kind of formal program analysis. Recently, work in program analysis and model checking has turned to analyzing concurrent code [26, 14, 41, 33, 16]. The amount of potential related work in these areas is quite large, so I have subdivided it into various types. Each type will be discussed along with some recent examples as well as how my work is distinct. First, I create a distinction between static and dynamic methods. My work is strictly static, in that it happens before run time. In the static space, I divide the related work into three sections: Algebraic (or Abstract) Data Types, Model Checking, and Program

Analysis. Algebraic Data Types covers the space of formal analysis that uses full-featured theorem provers to derive proofs of correctness from an abstract specification and a set of algebraic axioms. Model Checking refers to techniques that explore the program's state space and ensure that a set of temporal properties (e.g. temporal logic formulae) hold. Program Analysis covers other techniques, usually less mathematical, to analyze concurrent programs. This covers topics including type systems for concurrent code, techniques to derive lock sets and lock orders for concurrent code and others.

3.3.1 Dynamic Methods

Dynamic methods includes all analysis techniques that either occur at run time or rely upon traces generated at run time. Recently there have been several efforts employing dynamic techniques to detect errors in concurrent programs. Previously, much effort was focused on race detection, systems such as Eraser [44] sought to ensure that all accesses to a shared variable were protected by a lock, or set of locks (they pioneered the LockSet algorithm to track locks at run time). However, race freedom is neither necessary nor sufficient to prove program atomicity [18] (e.g. that it is, in fact, serializable). More recent efforts have focused on detecting atomicity violations. In general, program execution is traced and portions are lumped into transactions (this sometimes requires programmer annotation). These transactions are then analyzed with a version of Lipton's reduction [31] or structured into a happens-before graph [30]. This analysis is then used to see if the code could be interleaved to violate serializability. Agarwal et al [2] used a static first pass analysis to drive later dynamic trace gathering. In essence, they performed a partial type discovery to guide a dynamic detector (if their type system can prove a portion of code safe, then the dynamic checker need not inspect it). They grouped actions (heuristically) into transactions, which allowed them to analyze interleavings of transactions rather than interleavings of individual statements. This can make exhaustive inspection of interleavings tractable (this trick is used heavily by more recent work).

Atomizer [16] is a fully dynamic technique that detects potential atomicity violations given an execution. Operations are grouped into atomic transactions, which can either be specified by the programmer or derived automatically (via heuristics). Atomizer uses a version of the LockSet algorithm (from Eraser) to check for data races and to categorize data. This categorization allows Atomizer to determine if operations are *left-movers*, *right-movers*, *both-movers*, or *non-movers* (in accordance with Lipton's reduction). These commutativity rules allow Atomizer to check for atomicity violations by attempting to reduce an interleaving to a serial execution (by commuting operations into some serial order). Wang et al. [51] introduce an analysis to detect conflict and view serializability in dynamic traces of concurrent programs. They also group actions into transactions, but interestingly they use concepts from the data base community (conflict and view serializability) to define atomicity violations.

3.3.2 Program Analysis

Automatic analysis of concurrent programs has become quite important with the introduction of commodity multi-core processors, and many recent publications have attempted to catch concur-

rency bugs statically. Locksmith [41] analyzed a reasonable subset of C to infer correlations between locations and protecting locks. These correlations were then used to verify that the locations were consistently protected by their locks (thereby ensuring race-freedom). This approach was effective, however race-freedom does not ensure serializability, and the approach only works on code that explicitly acquires and releases locks. Autolocker [33] analyzed a program containing atomic sections and locks and inferred a global lock-acquisition order to prevent data races and deadlocks among atomic sections. These atomic sections can be considered pessimistic (they cannot abort or retry) and can therefore contain irrevocable actions (such as I/O). However, autolocker required the programmer to allocate global locks beforehand and to explicitly annotate all shared data with information indicating the protecting lock. Autolocker also was restricted to standard locking protocols (mutex or read/write), and was whole program (in the sense that all atomic sections accessing the shared data need to be available to the analysis). Lock Allocation [14] is a more general approach where the tool itself inferred which locks would be needed (in addition to where they should be acquired and released). However, Lock Allocation required that all shared data only be accessed in atomic sections and is a whole-program technique (in the same sense as Autolocker).

Flanagan et al. have developed a type system useful for inferring atomicity properties of programs [18, 17]. The basic idea is to infer types based upon Lipton’s reduction (i.e. left-movers, right-movers, both-movers, non-movers) from Java source code, and then use these types to verify that the program is serializable. The authors discovered several interesting properties. First, that race-freedom is insufficient to prove atomicity and second, that inferring this type system is NP-complete. Therefore, their analysis requires programmer annotations to work statically. However, this work has informed later dynamic tools (such as atomizer), which use dynamic traces to generate annotations.

Recent improvements in the performance of SAT-solvers (such as Chaff [36]) has increased the popularity of using constraint solvers in program analysis. Tools like Saturn [55] translate C code directly into SAT clauses (with a little hand-holding), and can be used to check locking properties of complex system code (the Linux kernel). Another approach [11, 9] is to have the programmer describe program properties in a relational logic (Alloy [28] is the currently popular choice) and then translate the logic descriptions to a SAT problem. This has the advantage of allowing modular analysis (a bug-bear of model checking), however in order for the problems to be tractable, the instances generated are extremely small (which is somewhat justified by the small-scope hypothesis). Additionally, this work is targeted more at verifying software contracts (in the software engineering sense), and it seems unscalable to a multi-threaded case.

Although promising, many of the approaches are restricted to analyzing standard explicit lock-based code. Transactional code requires different techniques (to analyze the implicit non-blocking behavior of transactions). These approaches use either rely-guarantee [29] or a separation logic coupled with Lipton’s reduction. For example, Wang et al. [50] describe a static analysis that operates on code containing low-level non-blocking primitives (e.g. compare-and-swap, load-linked and store-conditional) using a type system inspired by Flanagan’s. The analysis is incomplete, however, and very low-level. Analysis of non-blocking code is less developed than for locking code and tends to be low-level and only semi-automated [56] or completely manual [40, 49].

3.3.3 Abstract Data Types

The approach I propose shares much in common with what is usually known as Abstract or Algebraic Data Types. This approach to verification relies on an algebraic specification of a data type (this specification may include specialized axioms). These specifications are then processed by a theorem prover in order to verify desired properties. Because the theorem provers are semi automatic at best, these techniques tend to be semi automatic themselves. Verification of concurrency properties for algebraic data types is an even more difficult problem. Older work utilized extensions of the process algebra to reason about concurrent systems (see [3] for an overview) which is inappropriate for the analysis of highly concurrent non-blocking transactional data structures (which do not employ explicit message passing). Transactional data structures have been analyzed by hand using rely-guarantee reasoning [40, 49] or by strictly specifying legal abstract schedules for operations [52, 45]. My approach differs in that the property to be checked is fixed (e.g. correct abstract locking) and I am interested only in verifying the equivalence of abstract states.

3.3.4 Model Checking

Model Checking [13, 4] has been used to verify multi-threaded code. Model checkers such as Spin [26], NuSMV [7], and CBMC [8] are particularly useful for explicitly threaded code because they can exhaustively examine all possible interleavings of valid states. This differs from my approach in several respects. First, model checkers verify a model against an arbitrary formula in temporal logic (that might, for example, ensure that no deadlock occurs). My system is intended to check only one property (that the abstract lock prevents non-commutative orderings) which doesn't vary over time (and hence I don't need the full power of a temporal logic). Second, because the Transactional Memory system guarantees serializability of operations, my system doesn't need to check against all possible interleavings or all possible thread combinations. Third, model checkers (for reasons of completeness) often need access to the full program text (or spec) to render correct judgments. My system leverages the isolation afforded by transactions in order to analyze data structures in a modular fashion and so only needs the description for the data structure in question. In short, my system doesn't require the richness of a full model checker.

In summary, my approach differs from previous work in several ways. First, although much work has gone into verifying explicitly threaded code, comparatively little work has been done on transactional code (unsurprising, as TM is a relatively new concept). Secondly, although my approach utilizes an abstract description of a data type (similar to algebraic specifications), my system is not intended to answer arbitrary correctness questions and so I don't need the power of a full theorem prover or model checker. Third, my system can make simplifying assumptions (particularly regarding interleavings) by relying on the Transactional Memory infrastructure. In short, although portions of my approach are inspired by previous work, the problem domain and application of these approaches are significantly different from other work.

4 Abstract State Specification Language

Even with the higher level of abstraction offered by open nesting, it is still the case that not all operations can proceed concurrently across an abstract data structure. In order to preserve the data structure's invariants, some locking protocol must be provided to ensure correct access. It is in the programmer's interest to make this protocol as concurrent as possible. However, exotic locking schemes are difficult to concoct and verify by hand, which seems to indicate that making the programmer specify the locking protocol is a bad idea. Therefore, I propose instead that the programmer write a specification for the abstract state of the data structure. This specification can then be used to verify hand-rolled locking protocols as well as to generate valid locking protocols.

The specification describes (in terms of relations) the abstract state of the data structure, and the changes to the state caused by methods. At this point it is helpful to think of the methods less like actual functions and more like boolean circuits that take the initial state and parameter values and output a new state.

4.1 Relations

Relations in the modeling language describe functions between elements of finite sets. For instance, a predicate `isEven(x) : int -> boolean` relates 32-bit Java ints to boolean values. Because the domains of the relations are finite sets, relations can be modeled as large arrays (similar to treating them as uninterpreted functions). For instance, the `isEven` predicate above can be modeled as a giant array of boolean values. Consequently, in the language syntax, relation declaration and evaluation resemble Java array declaration and dereferencing, respectively. The declaration of the `isEven` relation above would be:

```
boolean isEven[int];
```

To evaluate `isEven` at a given value `x` would look like: `isEven[x]`. Relations can have many arguments (and resemble multi-dimensional arrays in syntax), and can be modeled as multi-dimensional arrays. For brevity's sake, relations from domains onto booleans are often referred to as predicates.

4.2 Models

4.2.1 Overview

A model describes the abstract state and abstract operations of a data structure. The abstract state itself is described as a collection of relations over finite sets. Models themselves bear a syntactic resemblance to Java class definitions, and in this sense the relations function as the data members of the Model. An abstract model for a Set data structure needs only one relation, the predicate `in:Object->boolean` which abstractly describes membership in the set. Models themselves are declared similarly to Java classes, so a declaration for the Set data type would be:

```
model Set{
```

```

    boolean in[Object];

    //...operations...
}

```

Models can contain multiple relations, which may be needed to track more complex state. Consider a simple graph representation using an adjacency matrix to represent the graph itself:

```

model SimpleGraph{
    boolean adjacent[Node, Node];
    int connectivity[Node];
    int weight[Node, Node];
}

```

This definition contains both a 2-argument relation (`adjacent`) that represents the graph itself, a connectivity relation that records the number of edges incident to a node, and a weighted adjacency matrix.

4.2.2 Parameterized Models

Models can be parameterized by type similar to Java 5 generics. This is useful if you are trying to describe a data structure that is type generic. For example, the abstract Set model described above specifically used Java Objects. It would allow for more flexibility if one description could be used to analyze sets of Objects as well as sets of Strings. To declare such a set, one would write:

```

model Set<T> {
    boolean in[T];
}

```

Now we have a type-generic set description. As in Java generics, one can specify constraints over the type parameter using the Java keywords `extends` and `implements`. Consider an extension of Set that maintains an order over its elements. Java programming practice requires that this data structure accept only `Comparable` types. To specify that in the Model, one would write:

```

model OrderedSet<T implements Comparable<T> > {
    ...
}

```

One can specify multiple type parameters (each separately constrained), delimited by commas.

4.2.3 Model Inheritance

Models (like classes) can inherit from other models. As there are no protection keywords in the language, this inheritance causes the child model to fully include the parent's relations. Method overloading basically blocks the inclusion of the parent's methods. At the moment, the language supports only single inheritance, although there is no particular reason why multiple inheritance

cannot be supported (due to the highly static nature of the descriptions). Model inheritance is syntactically identical to Java class inheritance. For example, if the ordered set model were to inherit from the plain set model, it would look like:

```
model OrderedSet<T implements Comparable<T> > extends Set<T> {  
    ...  
}
```

Note that the type parameter is shared in the reference to the parent model. This basically allows the system to directly in-line the parent model's definitions.

4.2.4 Methods

In addition to relations, models can include methods. A model method describes an allowed change to the abstract state of the model. However, even though the syntax resembles that of Java methods, a model method is a description rather than a chunk of executable code. It is better to think of the method itself as a function that transforms an initial state into a final state rather than as a series of imperative statements that side-effect memory. This distinction is important as Java methods are meant to be compiled (eventually) into machine code and model methods are transformed into SAT circuits that transform state descriptions. A model method is an abstraction of an actual Java implementation and as such must describe the possible parameters and return value of the actual method, consequently model method declarations resemble Java method declarations (without protection keywords). Consider the simple set example, say we add descriptions for two abstract operations, `add` and `remove`:

```
model Set<T> {  
    boolean in[T];  
  
    boolean add(T elem) {  
        boolean ret = in[elem];  
        in[elem] = true;  
        return ret;  
    }  
  
    boolean remove(T elem) {  
        boolean ret = in[elem];  
        in[elem] = false;  
        return ret;  
    }  
}
```

The two methods take an element and either add or remove it from the set. The return value distinguishes between cases where the element was in the set before the operation or not. The bodies of the methods are statements that reflect the total change to the abstract state that results from executing the abstract operation. Note too that it is important for the abstract methods to have return values as these represent data communication (an escape of state information to the caller).

4.2.5 Constructors and Initial State

This section describes two special forms used to construct and verify the validity of states. However, at the time of this writing, the exact syntax of these forms is still in flux and may change as the language becomes fully defined.

Constructors As in object-oriented languages, constructors are specialized methods invoked only at object creation time to initialize the object state. In the modeling language, the purpose of the constructor is similar. Constructors are used to supply initial values to the relations that describe the state. In some cases the default Java values may be fine. In those situations where the abstract state must hold another value (inserting sentinel values, for instance), a constructor should be used. The modeling language uses the constructor to generate a valid initial state. The syntax is meant to be identical to Java's, and the full range of statements may be used. For example, this code snippet describes a constructor for `Set<T>`, except that the `in` predicate is prepopulated with `true` values.

```
Set<T> () {
    forall(T elem ; true) {
        in[T] = true;
    }
}
```

Invariants Constructors tell the modeling system how to form a valid initial state, however it may also be useful to specify the validity of a state in the form of an invariant. Using invariants to filter good states from the universe of possible states is a standard approach, however the inherently universal nature of the invariant may compromise the tractability of the SAT problem (SAT solvers answer existential questions efficiently). An invariant would then act as a guard over the unbound state variables that the solver would be exploring. Consider the case of a data structure meant only to store positive integers (a hash table, where the keys should be positive). Java's type system is not expressive enough to represent this restriction (all `ints` are effectively signed). This invariant would be used by the resulting circuit to guard against false positives. That is, satisfying assignments derived from invalid states that disprove the correctness of the locking protocol. Invariants are used to generate clauses over the initial state values that are satisfiable only for correct values.

4.3 Informal Grammar and Semantics

Model methods need to describe how to transform abstract state. The modeling language includes a variety of operations to describe this transformation. Similar to methods, these operations resemble Java-style imperative code but are, in fact, describing a transformation of an abstract state. Each operation will be described in turn (informally) with the semantics given in an unusual fashion. Rather than expressing a mathematical denotation, or an operation to a memory/environment, these semantics are expression-oriented. They are directly equivalent to boolean circuits (and

therefore very amenable to SAT conversion). Rather than imagining the method body being evaluated one expression at a time, these expressions should be seen as generating an ever-growing chain of boolean circuits. These circuits take as their (initial) input: the method arguments and the initial state (the initial values for the model's relations). The semantics can then be grouped into two categories, semantics that take a state and produce a value (corresponding to normal notions of an expression) and semantics that wrap a state to produce an effectively modified state (this corresponds to normal notions of statements).

In the circuit model, statements are effectively filtering or wrapping the initial state. These statements can then be chained to represent the total state changes of a method (as expressions over the initial state).

4.3.1 Arrays/Relations

The semantics of relations are simple. Assume that there is a state function σ such that given a relation it returns a function that maps relation arguments (array indices) to variables. For example given the two relations: `boolean in[Object]` and `int weight[Node, Node]`, $\sigma in \Rightarrow \lambda index : Object \dots$ and $\sigma weight \Rightarrow \lambda index1 : Node. \lambda index2 : Node \dots$. The actual bodies of the index functions are left unspecified, but could be considered a chain of conditionals testing index values.

4.3.2 Dereference/Index

The dereference/index operation is used to evaluate a relation with a specific value. The syntax resembles array dereferencing in Java. For example,

```
in[x];
```

This code evaluates the predicate `in` for the value `x` at that point in the method.

Semantics The semantics of dereferencing are more subtle than one would initially suspect. The subtlety arises from the process of *scalarization*. Although not described here in detail (see Section 4.4), scalarization is the process of decomposing a relation (considered as an array) into a set of scalar variables, each one of which represents a specific value (at a specific offset) from the original relation/array. In the non-scalarized case, dereferencing is simply: $\llbracket in[x] \rrbracket \Rightarrow apply(\sigma in)[x]$. With scalarization, `in[x]` is replaced by its scalar if `x` is statically known, or with a chain of conditionals testing `x`. For example if `in` is scalarized to two scalars `ina` and `inb` (for indices `a` and `b` respectively), then: $\llbracket in[x] \rrbracket \Rightarrow if(x == a) then in_a else in_b$.

Circuit Semantics The dereference circuit takes a state and index as inputs and produces a value as output (state is not modified). Although this could be modeled by a multiplexor, it is represented by a chain of conditionals. In general, this chain could be quite large (one conditional for each index), however scalarization should reduce this to a more manageable number.

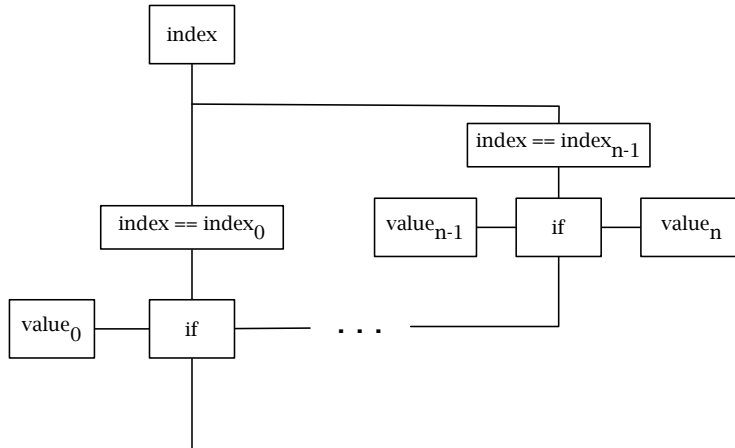


Figure 1: Array Dereferencing Circuit

4.3.3 If/Conditional

The modeling language supports standard Java-style conditionals. Like Java, the conditional must be an expression that evaluates to a boolean. The syntax is very similar to Java, for example:

```
if (expr) {
    stmt1;
} else {
    stmt2;
}
```

This example ‘evaluates’ `stmt1` if `expr` is true otherwise `stmt2` is ‘evaluated’.

Semantics If statements map nicely onto a boolean conditional circuit, however, they must be folded into the state. Because different arms of a conditional may assign different values to the same variable, later uses of that variable will need to reference the conditional. Therefore: $\llbracket \text{if}(E) S1 \text{ else } S2 \rrbracket \Rightarrow \text{if}(\llbracket E \rrbracket) \text{ then } \llbracket S1 \rrbracket \text{ else } \llbracket S2 \rrbracket$. Although it could also be represented directly as ANDs and ORs, this representation loses some information about the circuit structure and therefore `if-then-else` structure is preserved at this level.

Circuit Semantics An if-then-else circuit is a state modifying circuit, and so takes a state (used in computing the conditional expression). The result of an if is a modified state, so the circuit corresponding to an if-then-else takes an index input as well as an input state. Figure 2 corresponds to `if (E) {S1;}else{S2;}`

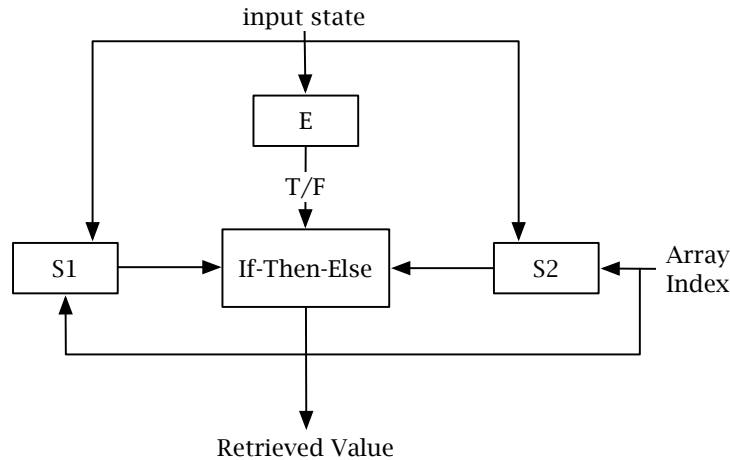


Figure 2: If-Then-Else Circuit

4.3.4 Assignment

The modeling language permits an operation that resembles assignment from an imperative language. Interpreted in an imperative fashion, assignment would overwrite the value of a relation/array at a particular index. However, in the circuit world, assignment simply hides the old value from subsequent circuits. The syntax is similar to Java (= is the assignment operator). In the modeling language to assign a value to a relation, one would do something like:

```

in[x] = False;
//OR
weight[n1, n2] = weight[n1, n2] - 1;

```

Semantically, this means that any subsequent statements in the method body will see the new value when they evaluate the relation.

Semantics Assignment becomes a conditional over the indices of the effected relation. For example, if $R[\alpha]$ is a relation of one variable and $\sigma R = \lambda index. \dots$, then the result of assignment to R is to wrap σ with an augmented state σ' that contains a definition of R that corresponds to the assignment. For example, given the statement $R[E1] = E2$, the semantics would need to wrap the definition of R and then wrap σ to produce the new definition for subsequent operations. The new definition of R (R'), would be:

$$R' = \lambda index . \text{if}(index == [[E1]]\sigma) \text{ then} \\ \quad [[E2]]\sigma \\ \text{else} \\ \quad R[index]$$

Then the wrapper for σ would be:

$$\sigma' = \lambda \text{ name } . \text{if}(\text{name} == R) \text{ then}$$

$$\quad R'$$

$$\text{else}$$

$$\quad \sigma \text{ name}$$

Circuit Semantics In a circuit, assignment wraps a conditional around the input state. Figure 3 illustrates a dereference of relation A after the statement: $A[E1] = E2$;

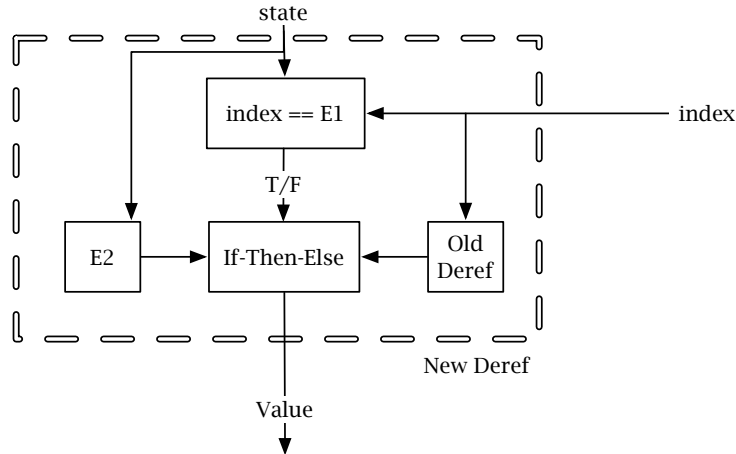


Figure 3: Assignment Circuit

4.3.5 For

Although the modeling language is not intended to express arbitrary iteration it may be convenient to mimic iteration. `for` is used when the user needs to express a block of code to be evaluated over a finite and predefined set of values. `for` takes a symbol for the index variable and a list of index values. The effect of this is to effectively unroll the loop for each index value. For example, the following code flags all the composite integers less than 10:

```

boolean isPrime[int];

for(x; 1, 4, 6, 8, 9) {
  isPrime[x] = false;
}

```

Semantics `For` is simply iteration over a predetermined set and so the semantics are simply an unrolling specified in an induction-like manner. Specifically:

$$\llbracket \text{for}(\alpha; E_1, \dots, E_m)\{S_1\} \rrbracket \Rightarrow \text{apply } \llbracket \text{for}(\alpha; E_2, \dots, E_m) \rrbracket (\text{apply } \llbracket S_1 \rrbracket \sigma[\alpha / \llbracket E_1 \rrbracket])$$

where $\sigma[\alpha / \llbracket E_1 \rrbracket]$ is σ wrapped so that α is given the value of evaluating E_1 .

Circuit Semantics For is basically an unrolling of the loop. Because the bodies are ‘executed’ in sequence they wrap the input state in that sequence. Therefore the outer-most state circuit corresponds to the last index of the `for`.

4.3.6 Forall

The modeling language is designed to answer existentially-qualified questions. However, it is sometimes necessary to change multiple values. Therefore there is an operator that supports a limited kind of parallel update masquerading as iteration, `forall`. `forall` takes a symbol, a boolean expression (known as the `forall` predicate), and a body of code to evaluate. The boolean expression forms a kind of ad-hoc boolean relation. For each value such that the boolean expression is true, the body of the `forall` is evaluated. Because these updates are considered to happen in parallel there are some special scoping issues. The symbol and boolean expression refer to the enclosing scope (i.e., the body of the `forall` cannot change the value produced by evaluating the predicate). The bodies are evaluated in parallel and therefore cannot refer to each other. Because the only way to affect other iterations is through assignment, the restriction applies to assignment operations (the specifics are given in the semantics section). The updates are considered to happen in parallel, effectively invisible to each other, and take effect immediately after the `forall`. Syntactically, a `forall` resembles a `for`. For example, the following code ‘removes’ all Strings from a `Set<Object>`:

```
forall(x; in[x] && isString[x]){
    in[x] = False;
}
```

Note that this is legal, even though the body seems to be side-effecting a relation used in the predicate. These changes will be visible only after the execution of the `forall`. This kind of universal operation is only possible because the domains being operated on are finite (and hopefully small).

Semantics Before a discussion of the general semantics, it will be useful to examine the details of the restriction on statements inside the body of a `forall`. The only operation of interest is assignment, and the assignments are only potentially dangerous if they refer to the predicate variable. Given:

```
forall(v; P(...)){
    ...
    x[E1(v)] = E2(v);
```

```

    ...
}

```

Then, this can cause problems if different ‘iterations’ would assign a different value to the same index. More formally, this statement is allowed only if:

$\forall v_1, v_2. P(v_1) \wedge P(v_2) \wedge (E1(v_1) == E1(v_2)) \rightarrow E2(v_1) == E2(v_2)$. This can be generalized to multiple assignments (to the same relation). Given two assignments $x[E1] = E2$ and $x[E3] = E4$ in the body of the same `forall`, then these would be allowed only if $\forall v_1, v_2. P(v_1) \wedge P(v_2) \wedge (E1(v_1) == E3(v_2)) \rightarrow (E2(v_1) == E4(v_2))$. Note that this covers the same iteration ($v_1 == v_2$) as well as distinct iterations.

The semantics of `forall` itself are expressed as a wrapper around an initial state that evaluates the predicate for incoming symbols. If the predicate is true, then the input is evaluated in terms of the state resulting from evaluating the `forall` body, otherwise the input is evaluated. This is possible because of the restrictions placed on statements in the `forall` body. More semi-formally:

```

[[forall( $\alpha$ ;  $E_1$ ) $S_1$ ]]  $\Rightarrow$   $\lambda id$  . let  $\sigma_2 = \sigma_{init}[id/\alpha]$ 
                        if(apply([[ $E_1$ ]],  $\sigma_2$ ))then
                            apply(apply([[ $S_1$ ]],  $\sigma_2$ ), id)
                        else
                            apply( $\sigma_{init}$ , id)

```

This effectively places a filter between subsequent statements and the initial state, such that if variables satisfy the predicate in terms of the initial state, they are filtered through the `forall` body before being passed to subsequent circuits.

Circuit Semantics The independence of the `forall` bodies means that the circuit corresponding to a `forall` statement just needs to test the input index against the predicate. Figure 4 shows a dereference of a relation previously assigned to with a `forall`.

4.3.7 Return

Return is used to signify abstract state values communicated back to the caller. These are important to note because for two operations to commute, not only must their state changes commute, but their return values must not be dependent on their execution order. In terms of syntax, it’s very Java-like:

```
return x;
```

where, `x` is of the appropriate type.

Semantics `return`’s semantics are straightforward. In the case of a top-level return (i.e. there is no calling method). The returned value is marked for later use by the verification system. In the case of a nested return (i.e. there is a calling method), the returned value is the result of

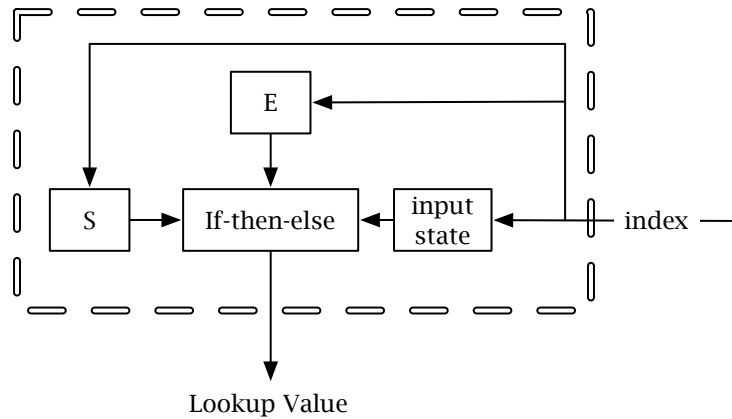


Figure 4: Forall Circuit

evaluating the call. In this case, the called method can be considered to have its entire circuitry ‘inlined’ between the call and its enclosing statement. The return value then flows as input into the enclosing statement.

Circuit Semantics In circuit terms, multiple `returns` present a few problems. Having multiple potential return sites can be handled by gating all their values onto a common bus (that then flows to subsequent expressions in the caller), however only one `return`’s actual return value must be sent through. In this case, the flow of control through the method can be modeled explicitly with a control bit (or token). This bit is then ANDed with each bit of the return expression to ensure that only a return statement that control reached is allowed to ‘fire’. Note that the only statement that can affect the value of the control bit is the conditional statement (where the `then` arm has the control bit set to the conditional expression and the `else` arm has the control bit as the complement of the conditional expression). Because of this, the control ‘wire’ has been omitted from the other circuit illustrations. Note too that the control bit can be used to model exceptional control flow (throwing an exception). In that case, a separate exception ‘bus’ could be used to indicate the type of exception. Figure 5 illustrates the statement `return E`, explicitly showing the control bit and the return ‘bus’.

4.3.8 Reductions

Reductions may be useful in the modeling language, but as the language is intended to be always reducible to SAT-circuits, general user reduction is probably impossible. While investigating graph data structures, the designers found it was useful to have pairwise arithmetic reduction over arrays/relations (consider tracking the in/out degrees of nodes in a graph, or the size of a map’s keyset).

In order for a reduction to be implementable as a simple feed-forward circuit the reduction needs to be able to be decomposed element-wise so that a running count can easily be maintained.

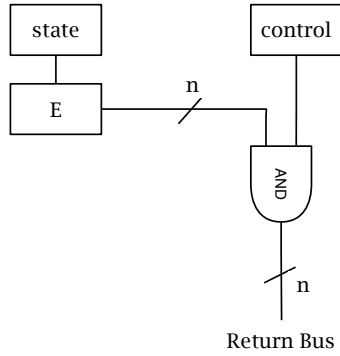


Figure 5: Return Circuit

For example, the summation reduction has the property that a change in an element can be used directly to generate the changed whole value. Set-based AND and OR do not have this property as the entire reduction must be recomputed if an element is changed (to account for the case where one particular bit flip changes the entire output). This requirement also imposes a requirement that the reduction is equivalent to repeatedly performing a primitive operation element-wise across the set. Algebraically, this means that the reduction operator itself must define an abelian group over its domain. This allows the system to track changes rather than having to recompute the entire reduction. More formally, given a reduction ρ_{op} across a set of elements S , then for an x , then the reduction is ‘modelable’ if $\rho_{op}(S \cup x) = op(\rho_{op}(S), x)$, for a ‘positive’ delta and if $\rho_{op}(S \setminus x) = op(\rho_{op}(S), x^{-1})$ for a ‘negative’ delta. Given this particular restriction, general reduction is not allowed in the language. Specific syntax will be included to represent reductions that obey this property (e.g., summation and parity).

4.4 Scalarization

Relations can, in principle, be huge. However, the number of distinct values that need to be examined to verify a description is generally small. Scalarization is a process that reduces a relation down to a small set of scalar elements that are then explicitly used by the description circuitry. At a dereference point, there are several possibilities: either the index can evaluate to a previously assigned index, or the index can dereference something not yet assigned. In the first case, the conditional circuits generated by assignment will handle it. In the second case, a new scalar must be generated to reflect this new value. This scalar is essentially unbound and allowed to take on any value (modulo `invariant` restrictions), this allows the SAT solver to fully explore value space. Now consider the case where a subsequent dereference uses the same index value. Correctness dictates that the circuit evaluate to the same scalar in that case. Therefore, scalarization must also generate a chain of conditionals (similar to assignments).

Consider the following code:

```
1:    boolean red[int];
```

```
2:    boolean a = red[x];
3:    boolean b = red[y];
```

Assuming that no assignments have been made to `red`, then after scalarizing line 2, `red` will effectively be:

```
red = λ z . if(z == x) {
    scalar1;
} else {
    scalar0;
}
```

Note that `scalar0` is the ‘initial’ value of `red`. After scalarizing line 3, `red` will be:

```
red = λ z . if(z == x) {
    scalar1;
} else if(z == y) {
    scalar2;
} else {
    scalar0;
}
```

The order of the conditionals is important. If `y == x` it is important (from a correctness standpoint) that `b` has value `scalar1`, rather than `scalar2` which would happen if one simply wrapped the previous expression with a new conditional. Scalarization effectively reduces a potentially large relation down into as many distinct values as may be needed (in the worst case) to cover every dereference.

4.5 Examples

This section includes example models and discussions to motivate the structure of the language.

4.5.1 Set

This example represents the simple set data type. The set contains at most one instance of an object, but makes no guarantees as to ordering, insert time, lookup time, or removal time. It’s just a big bag of objects. This set model is type parameterized, technically ensuring that it is a homogeneous set. The model supports three basic operations: `add`, `remove` and `find`. These methods all return a boolean value indicating whether or not the element was in the set (in the case of `add` or `remove`, before the operation was performed). The sole state of the set is the boolean relation, `in` which is true if an element is in the set.

```
model Set<T>{
    boolean in[T];
```

```

boolean contains(T elem){
    return in[elem];
}

boolean add(T elem){
    boolean ret = in[elem];
    in[elem] = True;
    return ret;
}

boolean remove(T elem){
    boolean ret = in[elem];
    in[elem] = False;
    return ret;
}
}

```

4.5.2 Ordered Set

This example is an extension of the simple set. The semantics are extended to incorporate ordering information. This model respects Java standards, namely that the set elements implement the `Comparable` interface. This model makes use of the built-in `rank` relation (of type `int rank[Comparable]`) which is a sort-of ‘super hash function’ that returns an integer value (which should be as large as the maximum reference size). This integer represents a pre-defined global ordering over the objects.

`OrderedSet` has two relations, `next` and `prev` mapping an object’s rank to the next and previous objects (that are actually in the set). Insertion and deletion are cases where parallel update (via `forall`) is extremely useful.

```

model OrderedSet<T implements Comparable<T> > extends Set<T>{
    T next[int]; //rank->next element
    T prev[int]; //rank->previous element

    boolean add(T elem){
        boolean ret = in[elem];
        in[elem] = true;

        int our_rank = rank[elem];
        T prev_guy = prev[rank[elem]];
        T next_guy = next[rank[elem]];
        int prev_rank = rank[prev_guy];
        int next_rank = rank[next_guy];
    }
}

```

```

//need to set the prev for all possible successors
//e.g. if z > y and we're adding y,
//      then we want: (y,z].prev = y
forall(int next_node ; next_node >  our_rank &
        next_node <= next_rank){
    prev[next_node] = elem;
}
//need to compute next nodes, basically we want [x,y) = y
forall(int prev_node ; prev_node >= prev_rank &
        prev_node <  our_rank){
    next[prev_node] = elem;
}
return ret;
}

```

```

boolean remove(T elem){
    boolean ret = in[elem];
    if(in[elem]){
        int our_rank = rank[elem];
        T prev_guy = prev[our_rank];
        T next_guy = next[our_rank];
        int prev_rank = rank[prev_guy];
        int next_rank = rank[next_guy];
        //[prev,us).next = us.next
        forall(int prev_node ; prev_node >= prev_rank &
                prev_node <  our_rank){
            next[prev_node] = next_guy;
        }
        //(us,next].prev = us.prev
        forall(int next_node ; next_node <= next_rank &
                next_node >  our_rank){
            prev[next_node] = prev_guy;
        }
    }

    in[elem] = false;
    return ret;
}

```

```

//one could also use this to fetch the next higher key,
// regardless of whether elem was in the set or not
T getNext(T elem){

```

```

    if(in[elem]){
        return next[rank[elem]];
    }else{
        return null;
    }
}

T getPrev(T elem){
    if(inelem){
        return prev[rank[elem]];
    }else{
        return null;
    }
}
}

```

4.5.3 Map

This example is an implementation of the abstract state for a Java-style Map. It contains the obvious relation mapping keys to values (`values`), and also contains two predicates to make calls to `containsKey` and `containsVal` simple to describe. Because a Map may contain multiple instances of a value, it is necessary to count the number of occurrences (this is what `valRefs` is for, note also that Java initializes `ints` to 0, so no constructor is needed to populate `valRefs`).

```

model Map<K, V>{
    V values[K];
    boolean keyPresent[K];
    boolean valPresent[V];
    int valRefs[V];

    boolean put(K key, V val){
        boolean ret = keyPresent[key];
        values[key] = val;
        keyPresent[key] = true;
        valPresent[val] = true;
        valRefs[val] = valRefs[val] + 1;
        return ret;
    }

    V get(K key){
        if(keyPresent[key]){
            return values[key];
        }
    }
}

```

```

    }
    return null;
}

boolean containsKey(K key) {
    return keyPresent[key];
}

boolean containsValue(V val) {
    return valPresent[val];
}

V remove(K key) {
    if(keyPresent[key]){
        keyPresent[key] = false;
        V ret = values[key];
        values[key] = null;
        int num = valRefs[ret];
        if(num <= 1){
            valRefs[ret] = 0;
            valPresent[ret] = false;
        }else{
            valRefs[ret] = num - 1;
        }
        return ret;
    }
    return null;
}
}

```

5 Proposed Work

5.1 Lock Protocol Verification

Given a specification of the abstract model of a data structure it should be possible to verify a given locking protocol (specified, say, as a conflict matrix) against the model. At this level, we're interested in pairwise conflicts between operations. Therefore conflict tests are really verifying that two operations commute on a given set of arguments. More formally, given two abstract operations $op_1(x)$ and $op_2(y)$ and an initial state σ_i , we can set up the following definitions. Let σ_{12} be the state after running op_1 followed by op_2 (that is, $\sigma_{12} = op_1(x);op_2(y)$), and let σ_{21} be the state after running op_2 followed by op_1 (that is, $\sigma_{21} = op_2(y);op_1(x)$). If the provided conflict matrix is correct, then whenever $\sigma_{12} \neq \sigma_{21}$ the predicate $P(x,y)$ specified in the conflict matrix should be

true for inputs x and y . Verification then becomes an existential statement, namely: $\exists x, y [(\sigma_{12} \neq \sigma_{21}) \wedge (\neg P(x, y))]$. The locking protocol is sufficiently strong if this existential statement is not true. Note that the locking protocol may be overly conservative; that is, the conflict predicate may be true for situations in which the resulting states are equivalent (this is correct, but not optimal in terms of allowing concurrent actions). Because return values represent a communication of state to the outside world, if two operations commute then their return values should be the same in either order. Therefore the operation's return values should also be tested for equality across both orderings.

The existential nature of this verification question suggests using a SAT solver automatically to verify user-provided protocols. Here the nature of the modeling language aids us in translating problems into SAT. Models are sets of boolean variables that are processed by the methods to produce an output value (the state). These methods can then be translated into boolean circuits that take inputs from the initial state (as well as method arguments) and produce new values. In this sense the circuits act as “filters” over the initial state. To simulate (in SAT) the sequence of operations $op_1(x); op_2(y)$, one essentially chains together the circuits for op_1 and op_2 (note that the scalarized representation of the relations may change depending upon which operations are being analyzed). Now that it is possible to create boolean circuits that simulate sequences of operations (and it is definitely possible to translate boolean circuits into standard CNF clauses), it is possible to use a SAT solver to answer the verification question.

5.1.1 Transformation Description

After parsing, the description is resolved and checked and then converted into a control-flow graph (CFG). The CFG form makes it convenient to process conditional statements (each arm can be visited normally). The CFG can then be abstractly interpreted step-wise to produce expressions representing the return values as well as the transformed state. Each method is visited individually, generating a simple summary (used for scalarization). These summaries are combined pairwise to generate the compound states. The output of each of these states is fed into a comparator to check for equality. The following diagram illustrates the high-level boolean circuit:

The leftmost AND tests the correctness condition. In this configuration, one would actually want the resulting problem to be UNSAT (that is, there are no inputs that the conflict predicate doesn't guard). However, any SAT result is a concrete counter-example witnessing an unguarded state. These counter-examples can be used to inform the programmer to amend their predicate, or to drive a function induction system to automatically refine a proposed predicate (described in more detail in subsection 5.2).

5.1.2 Opportunities for Optimization

Optimization in this case refers to generating SAT problems that are easier to solve. Initially, it should be possible to use more sophisticated techniques to convert boolean circuits to CNF expressions [32]. SAT solvers themselves can also be optimized to exploit a circuit's structural information. Using a specialized circuit solver rather than a generic SAT solver may speed up execution greatly. A more involved optimization is reducing the bit-width of variables (thereby

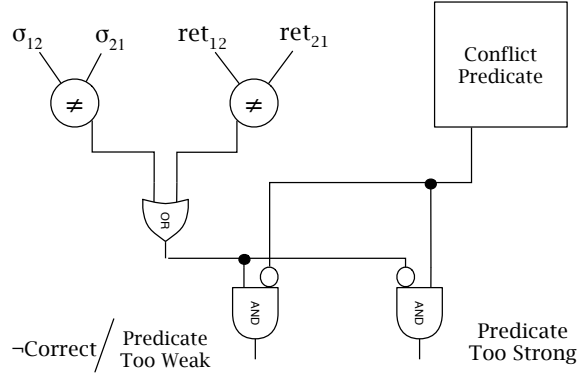


Figure 6: Boolean Circuit for Verifying and Inferring Protocols

reducing the number of literals and clauses in the output CNF). As an example, consider a relation that after scalarization has at most 5 distinct elements. In this case, the index variable really only needs to be 3 bits wide (as opposed to a more general 32 bits). An analysis that discovers these opportunities could ensure that even large problems remain feasible.

5.2 Inferring Lock Protocols

The main thrust of the proposed work is to derive, automatically, appropriate locking protocols given abstract state specifications. In figure 6, we show a graphical representation of the verification process. The leftmost AND gate represents the inverse of the correctness condition (if it fires, then the conflict predicate is not correct for that case). However, the output can also be interpreted as indicating that the predicate is too weak rather than simply incorrect. The rightmost AND gate has no verification function, but outputs a signal indicating that the conflict predicate has fired when, in fact, there is no conflict (signaling that the conflict predicate is too strong). We use these two signals to drive some kind of function induction (rule induction, for instance). In the simplest case, one could start out with a conflict matrix full of False (meaning no operations conflict) and strengthen the predicates, or start with a conflict matrix full of True values and weaken them. The function induction step is heuristic, and may not always produce a suitable result automatically, but verified protocols are guaranteed correct regardless.

5.3 Verifying and Generating Inverses

It may also be possible to verify inverse actions in almost all cases (compound forward actions may not be possible). In these cases one would ensure that the state before running the action and its inverse was equivalent to the state after running the action and its inverse. More formally, if σ_i is the initial state, and σ_f is the state after running $op(x); op^{-1}(x)$, verification is then answering the question $\exists x s.t. \sigma_i \neq \sigma_f$.

It may also be possible, in cases where inverse operations correspond to other forward going

actions (for instance, in a b-tree `add` and `delete` are both forward actions as well as each other's inverse), the system should automatically discover this and propose inverses for the programmer. This could be accomplished by chaining the two operation's circuit representations and verifying that the output state is equivalent to the initial state.

5.4 Expected Contributions

I have already designed an abstract state language and implemented a prototype TM run-time system and protocol verifier system. I intend to extend that work to more challenging and difficult cases (as well as to verifying inverses). I intend to generate a fully automatic protocol and inverse verifier. I intend to use this verifier to drive a system to heuristically generate potential locking protocols. My goal is for this system to also be fully automatic and general. Lastly, I intend to use similar techniques to discover and propose inverse actions. Although I would greatly like this last system to be fully automatic, I suspect that it may not be possible.

6 Current State of Work

Currently, I have implemented a reference transactional runtime system (XJ) and a prototype verification system. The prototype is actually capable of reading in simple descriptions and converting them into circuits and then handing them off to a SAT solving library (`sat4j`).

6.1 Preliminary Results

Recently I have been able to analyze the simple `Set<T>` example (described in section 4.5) and generate circuits for the `add` and `remove` methods. Although the current system is brittle and incomplete, with some hand-holding, I was able to verify that the `x==y` predicate preserved commutativity and that a 'bad' predicate (that asserted no conflict) was incorrect. I was able to compare the automatically generated 'good' and 'bad' circuits with a hand-rolled circuit (where I manually reduced bit-widths). The following table outlines these initial results (note that time is the estimated time reported by `sat4j`).

	Clauses	Literals	Time(ms)
Good	3962	1460	377 (UNSAT)
Bad	3174	1208	18 (SAT)
By Hand	2382	891	87 (UNSAT)

Table 1: Comparison of 3 Boolean Circuits

The manually constructed example (By Hand, in the table), differs from the Good example only in the bit-width of the indexing variable (used to dereference the `in` predicate), which was 2 in By Hand and 32 in Good. This one change was enough to noticeably reduce the problem size.

But the fact that the automatically generated problem was able to reach an UNSAT conclusion in less than a second is very encouraging.

7 Plan of Work

There are three main areas of future work: completing the run-time system, completing the Protocol Verifier, and extending the verifier to attempt protocol inference. Inferring inverses appears to be a much harder problem, and may not be possible.

7.1 Run-time system

The run-time library is basically done. It is feature-complete but certainly contains bugs. However, without a functioning compiler I have only been able to test it with manually generated code (which is very time intensive). However, I hope to be able to generate a simple compiler using a source-to-source translation tool (such as Polyglot). Additionally, a rewriting class loader needs to be completed so that pre-compiled code (such as the standard Java libraries) can be used inside transactions. I estimate that completing and testing the run-time system (at least to the point where I can run reasonable experiments) will take 3-6 months.

7.2 Protocol Verifier

A prototype lock verifier currently exists. However, it needs some work before it can be considered fully automatic (I had to manually wire up the lock predicate to the state circuits). The state description language also needs to be more rigorously formalized. A good understanding of the descriptive power of the language will be essential in precisely defining the limits of the language, which will inform the design and future extensions. In addition to completing the verifier (and extending it to handle verifying inverses), I intend to construct more complex state descriptions. Not only will this exercise the verification tool and demonstrate the validity of my work, but the more complex/exotic examples will be excellent test cases for a protocol inference system. This work can be pursued concurrently with the development of the run-time system, fortunately. I estimate that completing the verifier and writing more state descriptions will take 3-6 months.

7.3 Protocol Inference

Inferring a protocol from a set of counter-examples can leverage work on generic function inference. Additionally, because the system will have access to the abstract state description, it may be more effective to construct protocols symbolically. At the moment, it is unclear whether brute force (over concrete values) or a more algebraic approach (using term-rewriting, perhaps) would be more effective. The inference system would require a fully-functioning verifier, and I estimate that an inference system would take an additional 6-9 months.

7.4 Inverse Inference

Inferring inverses seems like a much harder problem than protocol inference (particularly in cases where there is no single operation corresponding to an inverse). This work could be carried out concurrently with protocol inference, however. I anticipate that this may take 6-9 months.

In total, I expect to spend 12-15 months on the proposed work. Currently, the verification work may develop into a strong paper at a programming language or software verification conference (deadlines usually mid to late fall). The run-time system may generate a publication at a transactional memory or Java language conference.

8 Bibliography

References

- [1] 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.
- [2] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–242, New York, NY, USA, 2005. ACM.
- [3] E. Astesiano, M. Broy, and G. Reggio. Algebraic specification of concurrent systems, 1999.
- [4] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen with P. McKenzie. *Systems and Software Verification*. Springer-Verlag, Berlin, Germany, 2001.
- [5] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, New York, NY, USA, 2006. ACM.
- [6] Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. The Open Runtime Platform: a flexible high-performance managed runtime environment. *Concurrency and Computation: Practice and Experience*, 17(5-6):617–637, 2005.
- [7] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.
- [8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction*

- and Analysis of Systems (TACAS 2004), volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
- [9] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 109–120, New York, NY, USA, 2006. ACM.
- [10] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. Lock Free Data Structures Using STM in Haskell. *Functional and Logic Programming*, 3945/2006:65–80, 2006.
- [11] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In *ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 195–204, New York, NY, USA, 2007. ACM.
- [12] Guy Eddon and Maurice Herlihy. Language Support and Compiler Optimizations for STM and Transactional Boosting. In *ICDCIT*, pages 209–224, 2007.
- [13] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [14] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 291–296, New York, NY, USA, 2007. ACM.
- [15] Michael Factor, Assaf Schuster, and Konstantin Shagin. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 288–300, New York, NY, USA, 2004. ACM.
- [16] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs (summary). In *IPDPS*. IEEE Computer Society, 2004.
- [17] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 47–58, New York, NY, USA, 2005. ACM.
- [18] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In J. Gregory Morrisett and Manuel Fähndrich, editors, *PLDI*, pages 338–349. ACM, 2003.
- [19] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.
- [20] Tim Harris and Keir Fraser. Language support for lightweight transactions. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA*, pages 388–402. ACM, 2003.

- [21] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In Michael I. Schwartzbach and Thomas Ball, editors, *PLDI*, pages 14–25. ACM, 2006.
- [22] Joseph M. Hellerstein and Michael Stonebraker. Anatomy of a database system. In *Readings in Database Systems*, pages 42–95. The MIT Press, 2005.
- [23] Maurice Herlihy. *SXM: C# software transactional memory*.
- [24] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.
- [25] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.
- [26] Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2004.
- [27] Harry B. Hunt and Daniel J. Rosenkrantz. The complexity of testing predicate locks. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 127–133, New York, NY, USA, 1979. ACM.
- [28] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
- [29] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
- [30] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [31] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
- [32] Panagiotis Manolios and Daron Vroon. Efficient Circuit to CNF Conversion. In *SAT 2007: The Tenth International Conference on Theory and Applications of Satisfiability Testing*, pages 4–9. Springer-Verlag, 2007.
- [33] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. *SIGPLAN Not.*, 41(1):346–358, 2006.
- [34] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL08*, New York, NY, USA, 2008. ACM.
- [35] 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 John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 359–370. ACM, 2006.

- [36] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM.
- [37] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, MA, 1985.
- [38] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.
- [39] Yang Ni, Vijay 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*, pages 68–78, 2007.
- [40] Matthew J. Parkinson, Richard Bornat, and Peter W. O’Hearn. Modular verification of a non-blocking stack. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 297–302. ACM, 2007.
- [41] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, New York, NY, USA, 2006. ACM.
- [42] Michael F. Ringenburt and Dan Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 92–104, New York, NY, USA, 2005. ACM.
- [43] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPOPP*, pages 187–197, 2006.
- [44] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
- [45] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, 1984.
- [46] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
- [47] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, New York, NY, USA, 2007. ACM.

- [48] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 191–210, New York, NY, USA, 2007. ACM.
- [49] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–136, New York, NY, USA, 2006. ACM.
- [50] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 61–71, New York, NY, USA, 2005. ACM.
- [51] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, New York, NY, USA, 2006. ACM.
- [52] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–282, 1989.
- [53] Gerhard Weikum and Hans-Jorg Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann Publishers, 1992.
- [54] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann Publishers, San Mateo, CA, 2001.
- [55] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.
- [56] Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In Chris Okasaki and Kathleen Fisher, editors, *ICFP*, pages 175–188. ACM, 2004.