USING FORMAL METHODS TO VERIFY TRANSACTIONAL ABSTRACT CONCURRENCY CONTROL

A Dissertation Presented

by

TREK PALMER

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 25, 2014

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

USING FORMAL METHODS TO VERIFY TRANSACTIONAL ABSTRACT CONCURRENCY CONTROL

A Dissertation Presented

by

TREK PALMER

Approved as to style and content by:

J. Eliot B. Moss, Chair

Jack Wileden, Member

Neil Immerman, Member

George Avrunin, Member

Lori A. Clarke, Professor and Chair
School of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003

ABSTRACT

USING FORMAL METHODS TO VERIFY TRANSACTIONAL ABSTRACT CONCURRENCY CONTROL

August 25, 2014

TREK PALMER

Bachelor's of Science, UNIVERSITY OF NEW MEXICO

Master's of Science, UNIVERSITY OF NEW MEXICO

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

Concurrent application design and implementation is more important than ever in today's multi-core processor world. Transactional Memory (TM) [33] has emerged as a promising technique for vastly simplifying the task of implementing concurrent applications. Transactional semantics in a programming language context has the potential to greatly improve the correctness of concurrent code, and improve the productivity of those that write it. However, standard transactional memory systems have several deficiencies. It is difficult to integrate transactional and non-transactional code, the TM run-time may make sub-optimal decisions that can significantly impact performance, and some actions (such as I/O) are inherently non-transactional. To remedy these shortcomings, several extensions to ordinary closed-nested TM have been proposed [51, 19]. Each has its own particular advantages and disadvantages. However, these techniques each need some extra information to 'glue' the non-transactional operation into a transactional context. At the

most general level, non-transactional code must be decorated in such a way that the TM run-time can determine how those non-transactional operations commute with one another, and how to 'undo' the non-transactional operations in case the run-time needs to abort a software transaction.

The TM run-time trusts that these programmer-provided annotations are correct. Therefore, if an implementor needs to employ one of these transactional 'escape hatches', it is crucially important that their concurrency control annotations be correct. However, reasoning about the commutativity of data structure operations is often challenging, and increasing the burden on the programmer with a proof requirement does not simplify the task of concurrent programming. There is a way to leverage the structure that these TM extensions require to reduce greatly the burden on the programmer. If the programmer could describe the abstract state of the data structure and then reason about it with as much machine assistance as possible, then there would be much less opportunity for error. Abstract state is preferable to a more concrete state, because it permits the programmer to use different concrete implementations of the same abstract data type. Also, some TM extensions such as open nesting [51] can handle concrete state conflicts without programmer intervention (making the abstract state the appropriate state for reasoning about commutativity). A solution to the problem of specifying and verifying the concurrency properties of abstract data structures is the subject of this thesis.

We will describe a new language, ACCLAM, for describing the abstract state of a data structure and reasoning about its concurrency control properties. This thesis also describes a tool that can process ACCLAM descriptions into a machine verifiable form (they are converted to a SAT problem). We will also provides a more detailed overview of transactional memory and the more popular extensions, a detailed semantic description of ACCLAM and a set of example data structure models and the results of processing those examples with the language processing tool.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Multi-core architectures are prevalent in both high-end and commodity hardware. Now only applications that can exploit increases in core counts will get faster with newer generations of hardware. Exploiting multi-core architectures requires programmers to design applications to exploit concurrency. However, concurrent programming is hard. Reasoning about standard lock-based multi-threaded code is difficult and error-prone. In addition to all the normal correctness concerns, issues of memory consistency, deadlock, and atomicity make concurrent programming considerably more difficult than normal serial coding. Much effort is being devoted to making concurrent design and implementation simpler and Transactional Memory [33] (TM) is an intriguing programming model that can greatly reduce the complexity of multi-threaded application development.

TM is no panacea, however. The Transactional Memory run-time can prevent data races and deadlock, however this comes at the cost of performance and flexibility. For example, it is very challenging to integrate I/O into a transactional context where transactions may have to be aborted and have their mutations undone (or redone). Often data structures where operations traverse shared objects will suffer performance degradation as the TM run-time's conservative concurrency control may abort many transactions. For example, a simple linked list will often degrade to effectively single-threaded performance as each operation has a high probability of reading list nodes that another operation will later desire to mutate. Transactional semantics are also challenging to integrate with standard lock-based code as the transactional run-time will have to know about memory operations performed by library functions invoked within transactions. This is commonly done by requiring the programmer to assert certain transactional properties of non-transactional code and implement additional functionality (such as inverse methods). This approach is known as Boosting [19]. In managed environments (e.g., a Java virtual machine) it is possible to integrate transactional and non-transactional code automatically by intrusively tracking all memory operations [61] (known as strong atomicity). A third approach, Open nesting [51], is a general-purpose 'escape hatch' that lets expert programmers relax transactional semantics in order to overcome these performance and integration issues in a general, consistent fashion.

Open nesting is an extension to the more traditional closed nesting approach to Transactional Memory. In closed nesting, the child transaction executes and adds its changes to the parent transaction, whereas an open nested transaction's changes are not published to its parent on successful completion. This allows the expert developer to use open nesting to selectively remove certain operations from the TM run-time's control. Performance can be improved by selectively removing spuriously conflicting memory accesses from the run-time, which could decrease the number of aborts triggered by the run-time and/or increase the number of transactions the run-time can execute concurrently (the details depend upon the particular TM run-time implementation). Open nesting can also be leveraged to invoke non-transactional external code. This could be used to perform irrevocable operations such as I/O, or to call into pre-existing concurrent code that is non-transactional. This flexibility incurs a cost. Because an open nested transaction has removed information from the TM run-time's purview, the programmer must 'fill in the gaps' in order to implement correct concurrency control. In particular, the implementor must specify undo operations and abstract locks. Undo operations are invoked in the event that the containing parent transaction has to abort. Abstract locks are programmer specified to detect conflict conditions for the open nested transaction. Because the memory operations will be unavailable to the TM run-time, it will be unable to perform automatic conflict detection. Therefore, the programmer must specify conflict conditions in the form of a locking protocol that guards potentially conflicting operations. At first it appears as though Open Nesting has all the complexity of traditional lock-based approaches to concurrent design. It is difficult to reason about locks, locking, and inverse operations. Even though open nesting is, in some sense, for experts only, subtle concurrency bugs are notoriously difficult to track down and fix. Thus it appears that any TM application that leverages open nested code could be vulnerable to all the pernicious bugs of standard lock-based code. Appearances would be wrong in this case because unlike traditional locking code, open nested transactions take place in a transactional context, and we can leverage that to automatically prove correctness of locking and inverse operations for transactional data types.

Transactional data structures greatly simplify the correctness conditions for both locking and inverse operations by reducing everything to verifying commutativity of the transactional operations. This thesis describes the conditions needed to prove an open nested data structure is correct, as well as a language and tool that will automatically prove these properties via a reduction to SAT problems. 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 question being asked (for example, 'Is the following lock protocol correct?') is also translated into SAT form and the whole combined problem is fed into a standard SAT solver. SAT solvers inherently search for answers to existential questions and my system leverages that so that a 'yes' answer evaluates to UNSAT while a 'no' answer produces a SAT result, with a solving assignment. This assignment can then be translated into a specific counter-example, which will aid the programmer in constructing a correct refinement.

CHAPTER 2

TRANSACTIONAL SEMANTICS

Transactional Memory builds upon decades-long work on transactions in the database community. This chapter is intended to be an overview of salient transactional concepts that are applicable to transactional memory systems in general and open nesting systems in particular. Much of the work described in this chapter condenses many years of incremental results, excellent discussions of which can be found in the following definitive text by Gray and Reuter [27].

2.1 Transactions as Abstractions

A transaction is an abstraction of an atomic action. A transaction is a collection of operations that need to execute atomically in a consistent fashion, and potentially while other transactions are also in process. A transaction executes in the context of a transactional run-time system that provides functions for maintaining transaction state and for detecting and resolving transaction conflicts. A transaction starts, performs its operations and then either completes successfully (a commit), or fails (an abort). On commit, a transaction's changes are now visible to the system as a whole. On abort, a transaction's changes are not visible at all and the system should behave as though the transaction never executed.

Transactions obey several important properties, commonly abbreviated as *ACID* [27, 68]. This stands for *atomicity*, *consistency*, *isolation* and *durability*. Durability refers to storing transactional state on durable media so that the system state can be reconstructed after a hard crash. This is a property of some importance in databases; however, transactional memory systems tend to ignore that requirement. Therefore we won't spend much time discussing it.

Atomicity is the all-or-nothing property. A transaction appears to happen all at once or not at all. A successful transactional execution will install all the updated state at once, and an unsuccessful transaction won't leave behind any side-effects. This means that a transaction has a simple set of states, it can be actively executing, completed successfully (committed), or failed (aborted).

Consistency is the property that a transaction's view of the system state is consistent throughout the transaction's execution. For reads, this means that if a transaction reads a value twice in a row (without mutating it in between), the second read will produce the same result as the first. This property allows programmers to construct transactions using intuitive straight-line code semantics. However, the programmer must trust the transactional run-time to handle all the nitty-gritty details.

Isolation is the property that any transaction's internal states are not visible to another transaction. A transaction is an ongoing execution of a sequence of operations, so it will produce many intermediate states between the start and the end. An isolated transaction's intermediate states won't be visible to another transaction.

2.1.1 Transactions as Histories

A common abstraction for modeling transactions is to treat them as a *history*. A history is simply an ordered list of operations. The operations are ordered in time. The operations are also assumed to be individually atomic, by which I mean that they execute atomically and their atomicity is ensured by some lower level system (such as the underlying hardware). A transaction's history is a predictable affair. As operations are executed in the transaction, they are appended to the transaction's history. In a system that obeys full *ACID* semantics, a transaction's history will contain only operations issued by that transaction. The transactional system as a whole has its own history. This is a more complicated affair and it represents the interleaving of the operations in flight for however many transactions are executing simultaneously. It is the unenviable task of the transactional run-time to execute as many operations simultaneously as possible, but to keep the execution consistent.

A *serial ordering* is a history where all the operations are in transaction order, and apart from the initial and final operation, all prior and succeeding operations are from the same transaction. In short, a serially ordered history is one where each transaction is executed one-at-a-time one after another. It is trivially apparent that a serial ordering is consistent. Therefore, many transactional run-times attempt to maintain consistency by allowing only those interleavings that preserve the illusion of serial order. This is known as *serializability*. At its heart, a serializable order is one where operations from different transactions commute around each other. By repeatedly applying these commutations until all the operations for a transaction are grouped together, a serial ordering emerges [64]. Although commutativity is a property of pairs of operations, serializability depends upon the entire series of commutations that each operation must undergo in order to form a serial ordering. What this means is that even though a given operation may commute with its immediate neighbor in the history, in order to be serializable, it must commute with all the subsequent intervening operations separating it from the rest of its transaction. Likewise, this property must hold for all operations in the history not already in a serial ordering in order to declare the history serializable.

Therefore, a run-time that is interested in preserving serializability needs to be able to reason about the commutativity of the operations for each active transaction. For most practical systems, ‘operation’ really refers to reads and writes of some quantum of data. For example, databases are often concerned with reads and writes to individual table rows.

2.1.2 Read sets and Write sets

Reasoning about transactions can be difficult. One of the more convenient mechanisms for abstracting the state of an ongoing transaction is as a pair of sets of data. The read set contains every datum that was read by the transaction, and the write set contains every datum that was written. The data itself is highly contextual. For example, in a database the data themselves would be rows from tables; however, in a programming language context they could be memory words, objects, or fields in objects. The assumption is that the transaction-processing run-time is consistent about how it records any particular datum, so we can ignore the specifics.

The transaction run-time needs to be able to determine if and when transactions interfere with one another. Consider two transactions T1 and T2 that are both attempting to commit. The circumstances under which T1 and T2 interfere correspond to relationships between their respective read sets and write sets. For example, if one transaction's read set overlaps with another transaction's write set then they interfere, as one transaction has read a value written by another active transaction. Also, if one transaction's write set overlaps another transaction's write set, then one transaction's write will interfere with another.

Once the transactional run-time detects such interference, it must make a decision. The run-time could chose to do nothing and let both transactions complete. Many database systems allow the programmer to relax the consistency requirements to permit certain interfering transactions to commit (this often increases concurrency). If the run-time determines that the interference is fatal, then it must fail one of the transactions so that the other can complete. This whole process is usually referred to as concurrency control. In lock-based multi-threaded programming models, the focus is on preventing conflicting operations a priori, and it is the responsibility of the programmer to do so. In transactional systems, conflict detection/prevention is a function of the transactional run-time.

2.2 Concurrency Control

Transactions are an abstraction of compound atomic actions. A transaction executes in the context of a transactional run-time, which is the system that handles the nuts and bolts of transactional processing. In the abstract, a user submits transactions to the run-time for execution, and relies on the run-time to handle decisions about concurrent execution and coordination of shared resources and data. It is important for a transactional run-time to perform well, and in general this means exploiting concurrent execution to ensure that as many transactional operations are being successfully executed as possible. The topic of executing transactions concurrently, and managing synchronization among transactions and resources is known as *c*oncurrency control.

2.2.1 Synchronization

If multiple threads are going to be executing operations from different transactions in an interleaved fashion, some coordination is necessary in order to prevent or handle inconsistencies or deadlock. The usual way to think about this is in terms of reading and writing individual data. The task of concurrency control mechanisms is then to synchronize these reads and writes so that the transactions maintain consistent views of system state.

Pessimistic Concurrency Control refers to techniques of ensuring consistent access by excluding later transactions from modifying the data. For example, locks from everyday thread-based programming are a form of pessimistic concurrency control because they allow only one thread to access the datum at a time. Consistency is ensured by preventing conflicting modifications outright. However, the standard limitations of mutual exclusion apply to pessimistic concurrency control. Namely, locks may reduce concurrency and can create deadlock.

Optimistic Concurrency Control refers to techniques that ensure consistent access by optimistically performing operations against system state, and then verifying the changes as part of transaction commit. In the case of reads, this could mean recording the values that were read and then verifying that they still hold as part of commit. If read values had changed out from under the transaction it would fail (the transactional run-time could then restart the transaction, fail it outright, or prompt the user for input). Writes are more complicated because the mutation has to be inverted if the transaction fails. One technique is to make the mutations in a transaction-local copy of the datum and the publish these updated values as part of a successful commit (publication is usually carried out by atomically swapping in the updated value). Another tactic is to optimistically install a new version of the overwritten datum. In this scenario a successful commit would update the publicly visible version rather than the data itself (failure then involves either uninstalling the failed version, or ensuring that newer versions will overwrite the failed version). Optimistic techniques avoid artificially reducing concurrency as caused by over-locking. The disadvantages include maintaining the (possibly large) bookkeeping information for validation, and in highly contentious circumstances optimistic systems may end up doing a lot of work that will end up being discarded and resubmitted.

It is possible to have systems that employ both optimistic and pessimistic techniques. One may choose to be optimistic for reads and pessimistic for writes, or to be optimistic most of the time but switch to a pessimistic mode when some contention metric exceeds some threshold.

2.2.2 Transaction Failure

Closely related to the mechanisms of concurrency control are the mechanisms for handling transaction failures. A *write-in-place* or *undo-based* system overwrites data in place. Therefore, if the transaction fails, the run-time must replace the failed value with the old value (the run-time ‘undoes’ the forward-going mutation). This requires the run-time to record the prior version. A *non-blocking* or *redo-based* system makes mutations in some private memory or otherwise non-visible location. In the event of transaction failure, the failed writes can just be discarded and the transaction started over.

Transaction failure handling can take multiple forms. In some cases, the transaction run-time will automatically restart a failed transaction. This makes sense in non-interactive scenarios where the primary interaction with the transactional run-time is by concurrent programs. Another approach is to just halt transaction execution and require the user or client program to decide whether to abandon the transaction totally or to restart. In general Transactional Memory systems choose the automatic approach, while databases choose the interactive approach.

2.3 Deadlock Detection and Avoidance

Transactional run-times alleviate many of the burdens that face a designer of a concurrent system, but there is still the potential for deadlock. There are two main techniques for coping with deadlock: detection and avoidance. Deadlock detection discovers deadlock after it's established and then decides to abort one or more of the involved transactions to break the deadlock and allow the other transactions to continue. Detection is feasible in transactional systems because the run-time often has enough information available to track data dependencies accurately. Deadlock avoidance takes an alternate approach, and structures the synchronization protocols to prevent most or all possibilities for deadlock. For example, a pessimistic system may impose a global lock ordering 'under the covers' which would prevent deadlock. Another approach is two-phase locking [6], which ensures that aborting will work. Therefore, deadlock detection can pick any two-phase locking transaction to abort to break any deadlock that emerges. A hybrid approach is often possible, where the avoidance technique employed has the added benefit of simplifying the detection algorithm needed to detect the smaller set of remaining deadlock cases.

2.4 Nesting Transactions

Up to this point, the discussion of transactions has been assuming that all the transactions are 'top-level', by which I mean there is no hierarchical relationship between different transactions. As each transaction executes, operations are added to its history and if the transaction is aborted, all of its constituent operations are aborted as well. Sometimes, it is desirable to have more structure and to be able to nest smaller transactions within a larger one [49].

A nested transaction executes in the context of its parent (outer) transaction. Concurrency control decisions are made against the state of the parent transaction at the point the nested transaction began executing. This means that nested transactions can read the changes made by the parent transaction. Nested transactions can fail without causing the parent to fail, and can be rolled back within the context of the parent transaction. This means that the failure of a nested transaction may induce rollback only for the subset of operations that were part of the nested transaction.

There are several reasons why nested transactions may be desirable. The first is that it may aid in *semantically chunking* the larger parent transaction into logical sub-units. The second is that nested transactions can enable limited *partial rollback* in the event of transaction abort. Consider the case of a long-running transaction that needs to increment a counter. A failure to increment the counter because another transaction updated it first would be cause for aborting the transaction. Without nesting, this could mean undo-ing a large amount of work. However, with nesting, it may be possible to retry the increment operation in isolation and thereby avoid having to redo many expensive operations. The third reason is that transactions (unlike lock-protected critical sections) are *composable* [27], and nesting sub-transactions is a convenient way to assemble a compound atomic operation from a collection of smaller individually atomic operations. Lastly, nesting transactions may allow for *concurrent execution within a transaction*. Normally, operations within a transaction are executed serially with respect to the transaction context. By grouping subsets of the operations into nested transactions, the programmer indicates to the run-time that there is additional structure that can be exploited by the run-time to execute several pieces of the parent transaction in parallel.

There are also many different ways of implementing nesting transactions. However, there are broadly two categories. Closed nesting, in which the sub-transaction executes in much the same way as the top-level transactions would; and open nesting in which the run-time cedes concurrency control to the programmer for the purposes of increasing performance and flexibility.

2.4.1 Closed Nesting

Closed nesting is the most straightforward approach. The nested transaction executes normally, and on a successful commit, adds its read and write sets to the parent transaction's sets. This means that the parent transaction's read and write sets grow as if it were just a simple top-level transaction, and from the point of view of the run-time, after a nested commit any abort will require rolling back the entire transaction up to that point.

Nested transactions may also fail/abort, so those possibilities have to be handled. Therefore, while the transaction is executing, the run-time does need to know that it is in a nested context. Partial rollback is simply the normal undo/retry logic. However, it's applied just to the subset of the read and write sets that the nested transaction generated. A convenient way to think about it is as a log, where a marker is put down when a nested transaction begins and the run-time performs its rollback function on the portion of the log from the marker to the end.

Visibility and observability are more complex. The nested transaction's context is, in some sense, inherited from its parent. In practical terms this means that a parent transaction's writes are visible to the nested transaction, and that if the nested transaction aborts, it must abort to the state of the parent transaction at the point the nested transaction began. The run-time must adjust its concurrency control mechanisms to account for this. For example, in a system with pessimistic writes, a transaction would attempt to write a value by first locking it. Without nested transactions, the lock attempt succeeds only if the value was unlocked or the lock was already held by the requesting transaction. With nesting, the relationship between nested and parent transactions must be taken into account. The lock will succeed if the value is unlocked, or locked by the nested transaction, or by its parent. Note, that this may involve multiple levels of nesting so multiple 'parent locks' may need to be compared against. Another approach would be to have the nested transaction do the locking, but with the identity of the top-most ancestor (this only works if there is no concurrency within the transaction, and additional bookkeeping must be employed to ensure that all locks are properly re-entrant). Again, this would require changing the run-time to furnish this information to transactions, and to adapt the locking infrastructure to permit ‘multiple identities’.

Although there are some complications, most of the added complexity is in the implementation of the run-time to accommodate notions of multiple transactions having the same view of a particular value. At the high level of abstraction we’ve been using, closed nesting is just a straightforward re-application of transactional behavior to a portion of a transaction’s operations.

2.4.2 Open Nesting

Open nesting is quite different from closed nesting. An open-nested transaction executes in the context of its parent like a closed nested transaction. However, the open-nested transaction is not operating under the standard concurrency control mechanism that any closed transaction would. To maintain correctness the programmer must augment the open nested transaction with information so the run-time can still do conflict detection and retry/undo if the nested transaction aborts. There are two reasons why one would need to escape the mediation of the transactional run-time. The first is that the sub-transaction may need to do something inherently non-transactional (such as execute an irrevocable action like performing I/O). The second is that the concurrency control of the run-time may be conservatively declaring conflicts and over-aborting transactions (and thereby decreasing concurrency and therefore performance).

Escaping Transactional Semantics: Certain operations are inherently non-transactional. The standard example is I/O, where some external state is being manipulated or read outside of the control of the transactional run-time. Many useful programs must occasionally be able to interact with external, non-transactional processes (or invoke non-transactional code), and open nesting provides a formalized mechanism for doing so. Other mechanisms for 'stitching together' transactional and non-transactional worlds have been proposed [19, 63]. However, open nesting is not exclusively a mechanism for interacting with non-transactional processes and the focus of this thesis is the performance and correctness implications of using Open Nesting. Therefore, most of the discussion will not be about using Open Nesting to interact with external processes.

Increasing Concurrency: The transactional run-time must execute against the concrete state that the transactions themselves are manipulating. This concrete state is an implementation of some abstract state. For example, an array of integers may concretely implement an abstraction: a fixed size set of integers. The concrete state may not be a precise enough approximation of the abstract state to allow for accurate conflict detection. Continuing the example, the order of integers within the array doesn't change the abstract state being represented (the specific set of integers). However, two different concrete states may represent the same abstract state. The run-time doesn't know about the abstraction. Therefore it must decide on the basis of concrete states, and could therefore fail a transaction that is abstractly safe to commit, but whose concrete state differs from that produced by running transactions in a different order. Obviously this will reduce concurrency because at least one fewer transaction could complete in a given time interval. It also will reduce performance because all the work performed by the aborted transaction will have to be repeated.

An example will illustrate this more clearly. Continuing with the array of integers as the concrete representation of a set of integers, let's consider a starting state: `[1, 2, 3]`, and two operations: `find` and `remove`. Let's assume the operations are implemented as a linear scan, and that the value `e` is special and represents an empty slot. We can then illustrate the concrete state transitions that both operations represent (as operations against indices in the array):

```
remove(2): [1, 2, 3] => [1, e, 3] ([read 0, read 1, write 1])
```

```
find(3): [1, 2, 3] => [1, 2, 3] ([read 0, read 1, read 2])
```

Because `find` has to read the second element to inspect it, it will be in the read set of the transaction executing `find`. However, because `remove` overwrites the second element, it will be in the write set of the transaction executing `remove`. Therefore, in this case, the run-time will have to declare a conflict between the `find` transaction and the `remove` transaction and abort one of them. Now consider the same operations, working in the same way, but executing on a slightly different concrete initial state for the same abstract set: `[1, 3, 2]`. This change will induce the following read and write set changes:

```
remove(2): [1, 3, 2] => [1, 3, e] (rd 0, rd 1, rd 2, wr 2)
find(3): [1, 3, 2] => [1, 3, 2] (rd 0, rd 1)
```

Now, the read set for the `find` transaction has no overlap with the `remove` transaction's write set. The run-time can let both transactions complete. In a broad sense, the second concrete state allows twice as much concurrency as the first. It is certainly undesirable and possibly prohibitive to have the concurrency and performance of a system be so sensitive to a particular concrete representation of the state of an abstract data type. Therefore, an implementor of a transactional form of an abstract data type would need to use open nesting in order to raise the level of abstraction being used for concurrency control. The programmer knows the abstraction being coded against and can make more precise determinations of conflicts. In the example above, the programmer knows that an array is being used to represent a small set of integers. Therefore, the programmer may also know that the only true conflicts are when the membership of that set changes. Furthermore, the only way for a `remove` and `find` operation to conflict (i.e., be non-commutative) is for both operations to be operating against the same set element. In this example, the programmer would be able to greatly increase the concurrency of the array manipulation code by somehow informing the run-time's concurrency controller that `remove(x)` and `find(y)` conflict only if `x == y`.

2.4.2.1 Abstract Concurrency Control

Central to a full understanding of open nesting is abstract concurrency control. Because an open-nested action is executing outside of oversight by the transactional run-time, open-nested actions can't rely on the run-time's concurrency control mechanism for safety guarantees. In fact, often the action is being implemented in an open-nested fashion specifically to bypass restrictions imposed by the transactional run-time's concurrency control process. However, open-nested actions may still interfere and do require concurrency control. Therefore a run-time that supports open nesting must also support abstract concurrency control, which is a mechanism for ensuring safety in the presence of concurrent updates given some additional programmer inputs.

When a closed-nested transaction commits, it effectively appends its read sets and write sets to the parent transaction's. This means that the transactional run-time has all the information needed to both detect conflicts and to undo any changes (assuming 'undo' semantics). Open nested transactions explicitly do not modify the parent transaction's read and write sets when committing, therefore additional information must be provided by the programmer. Specifically, the programmer needs to provide an *inverse* operation that will be executed in the event that the open-nested transaction is aborted. The *inverse* is so-called because it is meant to invert all the open-nested transaction's changes (and thereby undo them). Additionally, the programmer must also provide conflict information so that the abstract concurrency controller can determine when another open-nested operation conflicts with the effects of the committed open-nested action (this other operation could be executing concurrently or subsequently). The conflict information can take many forms. One option is to literally specify the exact conflict predicate that logically describes the set of conflicting states. These predicates may be written to a log, and when a new action is starting up, the log of conflict predicates is compared against the action's state to determine if it conflicts with committed actions. This is the approach used by the Galois system [40], and it has certain performance implications. In the example of the integer array implementing an integer set, the predicate would be exactly `x == y` for `find(x)`. When `remove(y)` is executed as an open-nested transaction the predicate would be evaluated to determine if `remove` conflicted.

Another approach is abstract locking [51, 49], where a lock protocol over the abstract state is used to summarize the open-nested changes. An abstract lock resembles a more conventional lock in that it has a context (the states over which it is locking) and it has a mode (e.g., Shared/Exclusive). The lock protocol dictates the procedure to determine if two contexts interfere and if two modes interfere. In general, two actions interfere only if their abstract lock contexts overlap and their modes conflict. In the integer array as set example, `find(x)` would grab an abstract Read/Write lock with context `x` and mode `Read`. `remove(y)` would have the context `y` and the mode `Write`. In this case, the contexts overlap only if they are identical. However, one could easily imagine a case involving ranges or subsets of values, where overlap would be determined by non-empty set intersections. Using abstract locks for concurrency control, `find(x)` and `remove(y)` conflict only if `x == y` (the locks' contexts overlap). Because the modes conflict (as a standard Read/Write conflict), any overlap is also a conflict. An advantage to abstract locking over explicit predicate tracking is that it is possible to trade off precision with performance. With explicit predicates, each must be evaluated in turn regardless of the cost. Locks allow the programmer the option to reduce abstract conflict accuracy in order to increase the performance of the abstract concurrency controller within the run-time. Furthermore, a locking protocol may be able to exploit lower-cost algorithms and data structures than general predicate evaluation. For example, the locks for operations over a set of integers could be organized within a hash table (keyed by the set member). This would mean that determining conflict could be done at $O(1)$ cost, while the cost of a predicate-based scheme would depend on the number of elements accessed by running transactions.

CHAPTER 3

TRANSACTIONAL MEMORY

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.

3.1 Transactional Memory

Transactional memory (TM) is a concurrent programming model that reinterprets transactional semantics from the database community in a programming language context. As in databases, a transaction can complete successfully (commit), or fail (abort). TM systems can be optimistic and pessimistic as well. 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 only to 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 [27, 68]. *Atomicity* in TM means that other transactions can't see partial state updates (all the transactions changes are visible or none of them are). *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 [56, 31, 32, 15] 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.

3.2 Data tracking

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 several strategies have evolved to track changes to memory. Low-level systems such as TL2 [15] track memory accesses directly. Hardware and software/hardware systems can track changes to cache lines [57]. TM systems for higher-level languages have more options. In languages with explicit objects (such as Java or `C#`), it is possible to track changes to objects or fields of objects. This is especially attractive if the language has no concept of atoms that aren't objects themselves (such as Java). Whether a TM system tracks objects directly, or can also track object fields, is often referred to as the 'granularity' of the system (or of its read/write logging). In general a coarse-grained system tracks changes at the object level, and a fine-grained system tracks changes at the field level.

For a TM system, the read set represents all the read accesses in program order and the write set records all the writes as well as the overwritten values. 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). In a fully optimistic system (for historical reasons, TM implementors refer to optimistic systems as non-blocking or non-locking), changes are made to a scratch workspace. On commit, that workspace is compared to the target memory/objects (this process is usually called validation). If no read or written value changed out from under the transaction, then the commit can proceed by having the TM run-time atomically swap the workspace versions of objects into 'global' memory. In a pessimistic implementation (often called a 'blocking' or 'locking' implementation), the object (or field) is locked when read or written, and then the locks are released on commit (no validation required). In practice, hybrid approaches that mix locking with validation tend to out-perform strictly pessimistic and strictly optimistic approaches.

Abstractly, if a conflict is detected, one of the transactions is chosen to be aborted and restarted. Optimistic systems just have to discard the workspace data and start over. Pessimistic systems may have to rollback changes. Rollback is straightforward because the TM run-time has access to the old values in the write set: the run-time simply writes the old values back into memory as it traverses the write log in reverse order (i.e., last write first). Obviously, there are many complex details involved in implementing this abstraction efficiently.

3.3 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. Composability is a powerful property. Implementors of libraries don't have to expose the innards of their concurrent creations, which allows them to preserve the abstraction of the libraries' interface. This in turn allows the benefits of proper encapsulation to be applied to highly concurrent code.

3.4 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 [51]. 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 not only locking semantics but also inverse actions (to undo the transaction if the enclosing action aborts). However, this added responsibility can be more cleanly encapsulated than in standard locking.

In a TM system, an open-nested transaction grabs its abstract locks, executes and then registers the locks and the inverse action with the parent transaction. If the parent aborts, the inverse action is executed. Locks are used for conflict detection; however, in TM systems the context of an abstract lock is a collection of memory locations. When the outer-most open transaction commits, the abstract locks can be totally released, as this signifies the actual publication of the state changes to globally visible memory.

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. Abstract state corresponds more closely to the abstract data type implemented by concrete code. By focusing on abstract state, the programmer is free to use any concrete implementation. 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 the remove as an additional forward-going action. This may result in a different low-level memory state; but, 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.

3.5 Transactional Boosting

Transactional boosting is another extension to standard closed nesting that allows an expert programmer to take a linearizable data structure and ‘transactionalize’ it. This means that if there is a pre-existing concurrent code-base, it can be ‘boosted’ into a transactional form. Similarly to open nesting, a boosted structure must provide additional concurrency control information. In practice this means that a programmer must specify under which conditions and with which actions any particular action conflicts, as well as specifying any compensating actions to take in case of an abort.

Although it seems like the programmer's duties are identical under open nesting and transactional boosting, there are important differences. A boosted data structure must be handled opaquely and therefore none of the inner state is available to the run-time. Consequently, this means that the conflict predicates may have to be overly conservative. This also implies that the only state that can be evaluated in a boosted structure is globally committed. Whereas open nesting permits rich abstract locking, a boosted data structure must make simple conflict decisions before beginning any action [1].

3.6 TM implementation

Implementation of a TM system is a complex engineering task. In recent years it has become commonplace to build a library [28, 31, 56, 32] 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 run-time system. The McRT-STM system, for instance uses the ORP [10] JVM to automatically transactionalize loaded classes and to respect TM references during garbage collection. Another approach [31, 32] 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 then 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 the new value

[1] with the caveat that a boosted system may be able to relax the locks slightly once the run-time sees the result (if any) of the boosted action

(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 a 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 much less clear how to conveniently modify a non-blocking system to support open-nesting. Therefore, for the duration of this thesis, we will assume that the TM system in use is for a strongly-typed, object oriented language (such as Java), and that the TM system supports optimistic reads and pessimistic writes. This assumption has the advantage that it resembles those systems currently known to be performant and scalable in arguably real-world scenarios [56].

3.7 Integrating Transactional and Non-transactional Systems

Programmers may want or need to be able to integrate pre-existing concurrent code within a transactional context. Sometimes, it could just be that there is a library of code that one would like to re-use. Other times the programmer may be implementing a system that will be used by others, perhaps in a non-transactional context (e.g., the programmer is implementing a library and can't control how the end user will deploy it). For correctness's sake, it is important that the TM run-time be able to reason about the memory operations being performed by other threads. This has been termed the *privitization problem*, and a decent overview can be found in [63]. Several approaches have been taken. I have already discussed transactional boosting, whose primary goal is to integrate lock-based code libraries with a transactional application. Open nesting can also be used to ensure transactional consistency even when escaping the TM run-time's concurrency control. Both boosting and open nesting provide mechanisms to wrap a concurrent thing in transactional garb to present it as a transactional thing. Another approach is known as *strong atomicity* [61], where the run-time aggressively tracks all memory operations by all threads and effectively transactionalizes all operations. This certainly solves the privitization problem; however, the performance costs can be high. Also, such an approach is infeasible for languages with limited run-times (e.g., C++) or languages whose run-times cannot be modified.

CHAPTER 4

RELATED WORK

This work occurs in the context of intense academic interest in concurrent programming generally 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 database community. Third, I will describe related efforts in software verification and program analysis and compare them to my proposed work.

4.1 Transactional Memory

Transactional memory was first formulated as a hardware concept in 1993 by Herlihy and Moss [33]. 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: it supported only fixed-size transactions [33] or had no support for standard closed-nesting [60]. Recent hardware [47] and software [32, 56] 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 [56, 29] and compiler work [2]. However, neither of these approaches can detect 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 [61]. Some recent work on programming language semantics [46] 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 addresses the question of irrevocable actions (e.g. an I/O operation that cannot be undone) occurring within a transaction. Transactions with isolation and cooperation (TIC) [62], 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 [19], 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 run-time to determine if two lock-based actions can occur simultaneously (if they commute, they can be concurrently executed). Initially only 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 [50], but boosting lacks the ability to support rich state-based locking protocols. Boosting also does not address the safety concerns of integrating transactional and non-transactional code (boosting assumes that the concurrent system being boosted is safe and won’t corrupt transactional data).

As I was performing the literature search to compose this related work section, I encountered many efforts to address the shortcomings of transactional memory [17, 9, 55, 62, 19]. 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.

4.2 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 [27] and Weikum and Vossen [69]. 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 [49] 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’). A crucial feature of nested transactions is commutativity [21], which allows the database to reorder nested operations with respect to one another or to provision threads to execute commuting nested actions in parallel.

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 then 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) [35, 69] 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 code 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 [30] or locking 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 ‘lock coupling’) 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.

4.3 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 [34, 20, 53, 45, 22]. 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: program analysis, algebraic (or abstract) data types, and model checking. Program analysis covers 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. I further divide analysis techniques into dynamic and static methods. Dynamic analysis operates on the concrete state of an actual system, but often has limited generality. Static analysis operates on the program text and may be more general than dynamic approaches, but must operate with the more limited information available in a static context. 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.

4.3.1 Program Analysis: 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 [58], which 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 [24] (i.e., 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 [44] or structured into a happens-before graph [41]. This analysis is then used to see if the code could be interleaved to violate serializability. Agarwal et al. [3] 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 [22] 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. [66] 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.

Burnim et al. [8] use an interesting approach where the programmer annotates the code with atomic blocks, and then specifies *bridge predicates.* These predicates specify post conditions that are true for commutative atomic actions, and allow the programmer to specify equivalence between data structures that may be physically distinct in memory. Their tool can execute a small subset of the possible interleavings of atomic actions to verify that certain actions are semantically linearizable. Although not designed for transactional code, their approach solves a problem similar to proving abstract concurrency annotations correct. However, it relies on execution of concrete implementations and can explore only a small number of interleavings.

4.3.2 Program Analysis: Static Methods

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 concurrency bugs statically. Locksmith [53] 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 works only on code that explicitly acquires and releases locks. Autolocker [45] 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 annotate all shared data explicitly with information indicating the protecting lock. Autolocker also was restricted to standard locking protocols (mutex or read/write), and was a whole program analysis (in the sense that all atomic sections accessing the shared data need to be available to the analysis). Lock Allocation [20] was 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 [24, 23]. 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. Much of the static analysis work for parallelizing compilers focuses on loop restructuring. However, Rinard [54] explicitly focused on analysis to discover and exploit commutativity (although not in a transactional context). The difficulty of discerning and exploiting fine-grained commutativity in general purpose code has restricted the utility of these techniques. Fortunately, the nature of transactional memory simplifies many of these problems to more tractable forms.

Recent improvements in the performance of SAT-solvers (such as Chaff [48]) has increased the popularity of using constraint solvers in program analysis. Tools like Saturn [71] 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 [18, 14] is to have the programmer describe program properties in a relational logic (Alloy [37] 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 [38] or a separation logic coupled with Lipton's reduction. For example, Wang et al. [65] 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 [72] or completely manual [52, 64].

4.3.3 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 Astesiano [4] 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 [52, 64] or by strictly specifying legal abstract schedules for operations [67, 59]. 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.

Burnim et al. [8] use an interesting approach where the programmer annotates the code with atomic blocks, and then specifies *bridge predicates.* These predicates specify postconditions that are true for commutative atomic actions, and allow the programmer to specify equivalence between data structures that may be physically distinct in memory. Their tool can execute a small subset of the possible interleavings of atomic actions to verify that certain actions are semantically linearizable. Although not designed for transactional code, their approach solves a problem similar to proving abstract concurrency annotations correct. However, it relies on execution of concrete implementations and can explore only a small number of interleavings.

The most directly related work is Rinard and Kim [39], in which the authors leveraged their Jahob [73] language to prove soundness and completeness properties for conflict predicates and inverses for a set of 18 operations (over 6 data types). Although there is significant overlap in terms of the problems addressed, the approach is very different. First, the work analyzes concrete implementations of abstract types using programmer-provided annotations and supplied projection functions to infer an abstract model. This approach requires considerably more programmer leg work in specifying the data type (the models appendix exceeds 600 pages) as well as the conflict predicates (up to three predicates for each operation combination). Second, the system is not fully automated and the underlying theorem prover required human intervention to complete 57 proofs. Lastly, their work is done outside of a transactional context, and therefore is unsuited to addressing the problems related to abstract locking and open nesting in general. However, because this work is done in the context of proving things about concrete implementations, it would be a powerful complement to ACCLAM.

4.3.4 Model Checking

Model Checking [13, 5] has been used to verify multi-threaded code. Model checkers such as Spin [34], NuSMV [11], and CBMC [12] 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). Second, 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. Fourth, the ACCLAM language and system is meant to address questions about abstract state only, and therefore doesn't need to deal with the complexities of proving a particular implementation correct. Although such verification is interesting and important, the approach I've taken is essentially 'implementation agnostic' and is therefore capable of generating results that are meaningful across multiple implementations and implementation languages. 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.

CHAPTER 5

ABSTRACT STATE SPECIFICATION LANGUAGE

5.1 Problem Domain

Open nesting extends conventional transactional memory semantics to allow the programmer to escape full oversight by the TM run-time. As discussed previously, this is done to enhance performance or to interact with external non-transactional processes. Other systems, such as boosting, allow similar (if more limited) forms of escape. All of these approaches work by hiding information from the TM run-time. Therefore, they all end up requiring the programmer to specify additional information so that the TM run-time can maintain the invariants necessary in order to preserve transactional semantics. Namely, the programmer must specify conditions under which the open actions conflict and how to invert the action if the parent transaction aborts. For performance reasons, abstract locks may be preferred over conflict predicates, in which case the programmer may have to specify the locks as well as (or instead of) conflict predicates. The programmer has chosen to nest this action in an open fashion, therefore maximizing concurrency is important to them. However, there is often a tension between the precision of a conflict detector and the simplicity of the specification. Generally, more precise conflict predicates are more complicated. Reasoning about concurrent systems is well-known to be difficult. In general, when requiring a programmer to input specifications it would be great if those specifications could be formally verified to be correct.

Of course, people have been trying to reason about concurrent systems for as long as there have been concurrent systems [16, 44, 41]. And reasoning, in a useful way, about general-purpose threaded lock-based code is very difficult. Open nesting has a distinct advantage in this regard. Because open nested code executes in a transactional context, a lot more can be assumed by a formal tool. In particular, transactional atomicity means that transaction boundaries can be used to eliminate having to reason about interleaving of operations, and transactional semantics greatly simplify conflict detection (by reducing it to a commutativity check; see Chapter 8). Therefore, the tools required to prove useful correctness properties of open nested code are much more tractable than a general purpose ‘concurrent code proof system.’

5.2 ACCLAM’s Audience

When we were designing the ACCLAM language, we assumed that the average user of ACCLAM would be using open nesting to enhance the performance of their code. As such, we can assume that they are an expert programmer and that it’s likely that their code is performance critical. Therefore, the transactional correctness of the code is even more important, and a programmer would welcome some automated safety checks even if there were some up front specification work. We designed ACCLAM to have a syntax similar to conventional systems languages (such as C or Java), and to resemble, as much as possible, conventional imperative system code. We attempted to apply the principle of least surprise to the language design in a conscious effort to make ACCLAM appealing to programmers who would be likely to use Open nesting or transactional boosting.

Formal systems for program analysis come in all shapes and sizes. ACCLAM uses a modeling approach, where the programmer specifies the abstract state being reasoned about, and explicitly specifies the state transformations that result from all supported operations. This approach was chosen over a more ‘conventional’ approach (such as a formal logic or some kind of model checking) because it was likely to result in specifications that resembled code for the data structure being reasoned about. Of course, the state is specified in an abstract fashion and there are several operators in ACCLAM that are useful for proving properties and wouldn't be used in a practical system. But the general idea was to have ACCLAM specifications be more 'code-like' and less 'proof-like'.

5.3 Language Overview

ACCLAM is a description language meant to specify transactional properties of data structures implemented in a language similar to Java. As such the language itself is designed to resemble Java as much as possible. ACCLAM descriptions are organized into packages of *models*, and models are structured similarly to Java classes. Models have names, can be parameterized by type, and can inherit from other models. Models also contain members and methods, however these differ from Java. Unlike Java, ACCLAM is not intended to be an executable language. Members of an ACCLAM model are descriptions of abstract state, and methods are descriptions of changes to the state caused by invoking the method. The difference can sometimes be subtle, but although ACCLAM method's are describing state transformations imperatively, they will be converted into pure expressions by the language processing tool so that they can eventually become SAT problems. In practical terms, the differences stem from the fact that when implementing a data structure in Java, a programmer is concerned with correctness and efficiency; but when describing a datatype in ACCLAM, a modeler is concerned with correctness and generality.

5.3.1 Language Pipeline

ACCLAM is a descriptive tool meant to drive an automated proof system. The proof system will respond with either a 'correct' token or an 'incorrect' token, and, one hopes, an example of a set of abstract states that violate the property being checked (for example, the operation doesn't commute in a given state, with the given arguments). The ACCLAM processing pipeline is as follows:

1. Programmer specifies a model and/or conflict predicates or abstract locks

2. ACCLAM description is parsed and resolved

3. Model methods are translated into pure expressions in terms of a base state

4. Conflict predicates or abstract locks are translated into pure expressions in terms of the same state

5. Pure expressions are translated into a SAT problem

6. SAT problem is fed into a SAT solver

Ideally, processing an ACCLAM model should take a matter of seconds so that the modeler can iterate rapidly in order to produce a correct locking protocol (or conflict predicate specification or inverse specification).

5.4 Notable Features

The differences between Java and ACCLAM are more apparent at the statement level. The following short descriptions provide an informal overview of notable language features. These descriptions are intended to give the reader a feel for the language and so they include examples. The formal specifications occur later (see Chapter 7).

5.4.1 Top-level Declarations

Relations: Relations in ACCLAM describe mathematical functions between elements of finite sets.[1] 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 over finite domains). 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:

[1] The name, relation, is unfortunate, however function has a different conventional meaning in programming languages and *map* has special meaning in the Java community.

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

Note that this differs from standard Java array declarations. This was done intentionally, to preserve standard Java syntax for array declarations (so that model writers would not be surprised by their arrays behaving like relations). Evaluating `isEven` at a given value `x` looks 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.

A declared relation is normally assumed to be uninitialized/empty. However it is possible to copy the state of a relation into a newly declared relation at declaration time. Using the = operator, a declared relation will start with a snapshot of the copied relation's state. From that moment on, they are separate entities (updates to one do not effect the other). This is useful in any situation where one wishes to snapshot or 'freeze' the state of the model (for instance, to construct iterators). The syntax is similar to that used when declaring a primitive state variable with an initial value. For example:

```
int instanceCount[T];
int copyOfCount[T] = instanceCount;
```

Relations are convenient for describing collections of state without having to worry about any specific implementation details. For example, if one were to model a set of integers, one could do something like:

```
boolean in[int];
boolean contains(int val) { return in[val]; }
void add(int val) { in[val] = true; }
```

In a practical system, representing a set as an array is usually not optimal. However, it is very convenient for modeling the abstract state of a set. The modeler can eschew implementation details for the most direct representation of the state. Of course, this means the language processing tool has to deal with relations in a way that makes them manageable and tractable.

Reductions: Reductions in ACCLAM are a way of describing a reducing function that is continuously computed over a set of relations. For example, many data structures keep track of the number of data elements they contain. It would be possible to explicitly model this in ACCLAM (for example, with an `int` that is explicitly updated on insert/delete). However, such explicit descriptions would constrain the implementation strategies. For example, when the concurrent variations of the `java.util.Map` interface were being designed they had to change the size calculation from a single `int` that was being maintained inside the data structure to an array of integers to increase the maximum concurrency permitted by the implementation. Reductions in ACCLAM are meant to abstract away any particular implementation strategies for maintaining such 'bookkeeping' data. ACCLAM reductions are declared with a primitive variable serving as the target storage for the reduction value as well as a set of relations that the reduction is operating over and a predicate that is true for values to be included in the reduction calculation. For example, to count the number of true instances in a boolean relation:

```
boolean rel[T];
int ct = count(T idx ; rel[idx];1;0);
```

Because simple summation came up regularly while developing the ACCLAM language, the keyword `count` is a shorthand for the summation reduction. In the example above, the integer `ct` stores the 'current' number of positive instances in the relation `rel`. The `count` declaration specifies locally bound index variables, a boolean expression that can use those index variables, and an expression to use for true instances and an expression to use for false instances (in this case, `1` and `0`, respectively). ACCLAM takes care of updating/recomputing reduction variables.

Reductions are a convenient way of calculating aggregate values over a set of relations, and like relations their main purpose is to make the meaning of the aggregate value explicit while abstracting away any particular implementation considerations. This allows the model to represent a larger range of implementations, at the cost of having the language processing tool handle reduction calculations in a tractable fashion.

Models: 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, standard Java data (primitives and objects), and reductions over the model's relations. Models themselves bear a syntactic resemblance to Java class definitions, and in this sense the abstract state functions as the data members of the model. As a motivating example, consider a description of a set data structure. An abstract model for a `Set` 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.

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.

Model Inheritance: Models (like classes) can inherit from other models. As there are no protection keywords in ACCLAM, this inheritance causes the child model to include all of the parent's state variables. Method overloading basically blocks the inclusion of the parent's methods (that have the same signature). At the moment, ACCLAM supports only single inheritance, although there is no fundamental reason why multiple inheritance cannot be supported (due to the highly static nature of the descriptions). However, as Java supports only single inheritance, multiple model inheritance may have limited utility.

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 inline the parent model's definitions.

ACCLAM is like a more conventional object-oriented language in that descendant models can be substituted for references to their parents. Therefore, a correctly inheriting submodel must be substitutable. Informally, this means that submodels can't violate any of the properties provable relative to the parent model. In particular, submodels must not conflict where the parent model had no conflicts (the set of conflicting states must be a subset of the parent's conflicting state set). Also, if a submodel overrides a method definition, it must perform the same state transformation as the parent's method (this is explained in more formal detail later).

Nested Models: Models can also be nested (as in Java). The nested (inner) model has access to its parent (outer) model's state. This can be useful when modeling auxiliary data structures that are dependent upon the outermost definition (for example, an iterator). For example, to declare an iterator inside the ordered set:

```
model OrderedSet<T implements Comparable<T> >
extends Set<T> { ...
  model OrderedSetIterator<T> { ...
    boolean hasNext() { ... }
    T next() { .... }
    ...
  } }
```

Methods: In addition to state variables, models can include methods. A model method describes an allowed change in 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. 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 indicates whether or not the `Set` was modified (as in Java). 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).

Constructors: As in object-oriented languages, constructors are specialized methods invoked only at object creation time to initialize the object state. In ACCLAM constructors have two purposes: to model the constructors of the data structure, and to allow models to create instances of other models. The first case interacts heavily with *invariants* (described below), as a correct constructor will always establish the invariants of the data structure. The second case is for modeling cases in Java where a data structure has methods that return other data structures (e.g., `java.util.Map.keySet()` is defined in `java.util.Map`, but returns a `java.util.Set` that contains all the keys in the map). Constructors are vital for modeling these cases, as correctness conditions often rely on equivalence of return values. Therefore, accurate modeling of the internal state of a returned data structure is extremely important. In terms of syntax, constructors in ACCLAM resemble their Java equivalents (no return type, and the method name is identical to the model name). In practice, they are often defined with the `forall` statement to initialize member relations.

```
model Set(){
  forall(T elem ; true) { in[T] = false; }
}
```

Method Handlers: Transactional systems have a more nuanced view of operations than explicit locking systems. Because conflicts may be detected after a method has begun, transactional methods must support aborts/undos. In practice, this means that each forward-going operation may have to specify a block of code to be executed in the event that an abort is generated. In ACCLAM this is accomplished with *method handlers*, which are labeled blocks of code appended to a method definition. In the case of abort, the handler is labeled `onabort` and is meant to model the undo action. For example, our Set model has an add method that inserts an element into the set. If an add instance is aborted, it must undo the changes to the abstract state thus:

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

In this case, the abort handler is just one line that undoes the second line of the add method. Note that this is not actually correct, in the case where `obj` is already in the set (before the invocation of `add`), the abort handler does not return the `Set` to the initial state. To do that, one would need to somehow cache the value of `ret` and allow it to be visible inside the abort handler. This is possible using *prevals* (discussed below).

In general, it may be useful to provide handlers for situations other than aborting. Three other handlers are available: `onvalidate`, `oncommit`, and `ontopcommit`. `onvalidate` is applied during validation of a transaction (after the forward-going operation is complete, but before the run-time has determined commit/abort status). `oncommit` is applied after a transaction is validated and set to commit. `ontopcommit` is applied after the top-level (outermost) transaction has been validated and set to commit.

Prevals: As mentioned in the discussion of method handlers, sometimes it is necessary to cache values computed at the beginning of a forward operation. These cached values are known as *prevals* (because they are computed before the ‘execution’ of the forward-going operation). Prevals are purely stored values. No mutation is allowed in generating prevals and no mutation of prevals is allowed in the body of a method or method handlers. Prevals are declared in a block delimited by square brackets after the method header. The prevals themselves are declared just like any other variables. For example, a correct definition of an abort handler for Set.add would look like:

```
model Set<T>{
  boolean in[T];
  boolean add(T obj) [boolean present = in[obj]; ]{ //preval
    in[obj] = true;
    return !present;
  } onabort { in[obj] = present; }
}
```

In this case, a preval (present) is declared to capture the incoming value of in at the point where it may be overwritten. Because present also captures the same information as ret in the previous example, there is no need to declare ret, and !present is returned instead. Now the abort handler correctly undoes add, and restores in to its initial value (even if in[obj] was true before invoking add).

Prevals also allow for the capture/copying of a relation's state. When declaring a new relation, it can be assigned the value of a pre-existing relation. This has the semantic effect of copying the entire relation into the newly declared relation. This allows for a user to 'snapshot' relation state. For example, in constructing derived or inner models it may be useful to capture the outer model's state at a particular point in time (this is especially useful when defining iterators over a particular model). Consider a method clone() on Set that is intended to return a copy of the Set itself:

```
Set clone()
[boolean copyOfIn[T] = in; ]
{ ...
  return new Set(copyOfIn);
}
```

Abstract methods: ACCLAM methods can be declared abstract. An abstract method is one that specifies only a type signature and leaves implementation up to child models. In Java, an abstract method declaration is an implementation obligation for inheriting classes. An abstract ACCLAM method is modeled as an uninterpreted function. No implementation is specified, so ACCLAM will allow the return values to range over all allowed values for the type. If some constraint on the return value is desired, constraints can be specified with *invariants* (described below). To declare a method abstract in ACCLAM one simply uses the `abstract` keyword similarly to Java. For example:

```
boolean isEmpty() {
  return size() > 0; //not abstract
}
abstract int size(); //abstract
```

Abstract methods, when combined with invariants allow for some flexibility when specifying a method's behavior without having to specify an explicit implementation.

Invariants: An ACCLAM description may include invariants. An invariant is a top-level declaration inside a model that constrains the space of legal values. In practice, invariants are useful for excluding illegal configurations of state. For instance, without clauses to constrain values, a SAT solver will potentially assign any value to an array of unbound literals. In the case of an ordered set, this may mean that the SAT solver will generate initial states where the stored keys are not ordered (this happened during the construction of the ACCLAM processing tool). Therefore, an invariant is needed to exclude situations where the keys aren't correctly ordered. Given an ordered set definition like this:

```
model OrderedSet<T>{
  boolean in[T];
  int rank[T];
  T next[T]; ... }
```

If `next` is intended to map its input to the next higher object in the `OrderedSet`, and if that ordering respects the `rank` relation, then a useful invariant would be:

```
invariant
forall(T x, T y ; (in[x] & y = next[x]) →
                  (y = null || rank[y] > rank[x]));
```

This invariant states that if an object (`x`) is in the set then its next field is either the special value `null`, or an object with higher rank. One can see that this invariant, applied to floating values of the initial state (i.e., values that the SAT solver can assign), will ensure that only valid initial states are generated by the SAT-solver (this avoids garbage-in-garbage-out problems).

Invariants are used to constrain the state of a model and are often specified in terms of universal quantifiers (`forall`). However, it is completely legal to specify an existential invariant. For example:

```
invariant exists(T x ; in[x] & next[x] = null)
```

This specifies that there is at least one `T` in the ordered set such that its next value is `null`. Depending upon the meaning of `null` this could be the greatest value currently in the set. Invariants can also be used to constrain abstract methods.

Abstract Locks: ACCLAM is a tool for describing several things: the abstract state of a data type, abstract operations on that data type, predicates over sets of instances of that data type, and abstract locking protocols. Abstract locking is the mechanism used by open nesting to ensure that conflicting actions don't occur concurrently. The usage pattern for abstract locking is that a thread invokes an open-nested action from within a transaction. The action specifies one or more abstract locks to acquire. The locks themselves are specified with a context and a mode. The context is essentially the data type over which the lock pertains, and the mode specifies a type of access that the transaction is requesting (e.g., read mode or write mode). The point of lock acquisition is to identify any potentially conflicting actions and to manage the concurrent execution accordingly. For example, if one transaction is attempting to add a string to a set while another transaction is attempting to remove a string from the same set, the run-time's concurrency management mechanism will have to know if the remove and add conflict before it can schedule the operations. Abstract locks conflict if they refer to the same context (e.g., add and remove are operating on the same set), the specific operation invocations overlap (e.g., add and remove are operating on the same string data) and the lock modes conflict (e.g., add and remove are both exclusive mode operations).

Currently, ACCLAM allows the specification writer to define abstract locks in a separate lock definition file. A lock definition file can be grouped under a `package` (like a `model`) and contains zero or more lock definitions. A lock definition can be either a lock predicate or a lock space. At a very high level, a lock predicate is simply a predicate that compares two requested locks and returns true if they overlap and false otherwise. In the context of abstract locking, 'overlap' means that their domains of concern overlap, which means that semantically, the locks are potentially referring to the same abstract state. Consider a set of strings, and two transactions that are attempting to add a string to the set. The lock predicate in this case would simply be string comparison. If the strings were equal, then the two transactions are talking about the same abstract state and may conflict. Otherwise, the two transactions are attempting to manipulate disjoint state and may execute concurrently. A lock space is a named collection of lock predicates, possibly parameterized by type. Lock spaces are a way of grouping together related lock definitions. For example, a point-like lock may guard operations around a single value (a specific integer in a set of integers, say) while a range-like lock may guard a whole range of values. They are different lock *types* but are obviously related. For example, we would like ACCLAM to be able to reason about one operation locking the value **14** and another locking the range **[5,20]**. Therefore, in ACCLAM we say that these locks are drawn from the same *lock space*. More concretely, all lock types defined in a lock space may potentially overlap; locks defined in different spaces can't overlap.

Locks are used/declared within a model as part of method declaration. A lock may be defined and have any free variables in the predicate definition bound to method parameters and/or method prevals, as well as a mode specified. A lock predicate can be regarded as a pure function that is evaluated at method invocation time. A lock is acquired (i.e., the operations don't conflict) if the predicate is false *or* the modes do not conflict.

Lock Predicates: A lock predicate is defined similarly to a model, but using the `lockpredicate` keyword. Lock predicates can be doubly parameterized: first by type (similarly to a type-parameterized model), and secondly by values. The value parameters are free symbols within the predicate definition and are bound to defined state variables at lock use time. The body of a lock predicate is a boolean function, defined similarly to a method within a model. The predicate argument is the other lock instance against which to compare the current bound instance. For example, the following lock predicate would work for our simple set example:

```
lockpredicate ValPred<T><T val> {
 overlap(ValPred other) {
   (val = other.val) || (val.equals(other.val))
} }
```

The predicate is parameterized over the type `T` and has one free variable, `val`, that will be bound at lock use time.

Lock Spaces: A lock space is a way of grouping together a bunch of related lock predicates and giving that grouping a name. Lock spaces can also further constrain the type parameters of lock predicates. For example, given the lock predicate definition above a second predicate and lock space might be defined as follows:

```
lockpredicate RangePred<T><T leftend, T rightend>{
 overlap(RangePred<T> other) {
  (leftend ≤ other.leftend) && (rightend ≥ other.leftend)
} }


lockspace ValAndRange<S implements Comparable<S>> {
 ValPred<S> point;
 RangePred<S> range;
}
```

This lock space contains predicates for determining if pairs of range-like locks overlap and pairs of point-like locks overlap. For completeness, we must also define a range/value predicate:

```
lockpredicate RangeValPred<T><T leftend, T rightend>{
 overlap(ValPred<T> other) {
  other.val ≤ rightend && other.val ≥ leftend
 }
}
```

Lock Modes: Abstract locking also depends upon a set of defined lock modes and a conflict matrix describing the compatibility of the modes. A mode itself is simply a symbol specified by the user, and is used to reference a conflict predicate from the mode definition matrix. The mode matrix itself is defined one entry at a time, using the $\star$ symbol to represent the intersection of two modes. For example, the mode definition for the standard shared/exclusive (read/write) lock could be defined as (with S for shared and X for exclusive):

```
modenames S, X;


S ⋆ S { false }

S ⋆ X { true }

X ⋆ S { true }

X ⋆ X { true }
```

The predicate definition is delimited by braces ({,}) and specifies the conditions under which the modes are compatible. In the example, shared mode is compatible with shared all the time, and exclusive mode is never compatible with another lock.

Using Locks: A model is verified against zero or more lock spaces and mode definitions that are used to define individual locks, and each method will define the specific locks that it will take. Each lock definition must bind the value parameters to either model state, method arguments or prevals. Additionally a lock mode must be specified. For example, to define locks for the set example mentioned above:

```
model Set<T implements Comparable<T>> {
  void add(T obj)
  @lock(point<T>(obj), X)
  {
      ...
  }
}
```

5.4.2 Statements and Expressions

If/Conditional: ACCLAM 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;
}
```

Assignment: ACCLAM 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. The syntax is similar to Java (= is the assignment operator). Here are examples of assigning a value to an element of an ACCLAM relation:

```
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 (at the index that was updated).

Forall: ACCLAM is designed to answer existentially-qualified questions. However, it is sometimes necessary to change multiple values, or to ‘filter’ values over an entire relation. Therefore there is an operator that supports a limited kind of parallel update masquerading as iteration, `forall`. `forall` takes typed symbols (to bind), a boolean expression (known as the `forall` predicate, and in terms of the bound variables), and a body of code to evaluate. The boolean expression controls which values of the typed symbols are actually used. 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 (conceptually) in parallel and therefore cannot refer to each other. Because the only way to affect other iterations is through assignment, a restriction applies to assignment operations (the specifics are given in the Chapter 7). 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<T>`:

```
forall(T 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 possible only because the domains being operated on are finite.

The restrictions on the body of the `forall` basically ensure that across all ‘iterations’ of the `forall` body, if any two index expressions evaluate to the same value, then all values assigned at that index within a relation are the same. The ACCLAM tool is capable of generating side-proofs that ensure that all uses of `forall` obey this restriction. This is possible because ACCLAM is built on top of an automated prover (a SAT solver). Because these side-proofs can be generated separately from other proofs, all other proof questions will assume that the `forall` correctness condition has been verified.

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. Every method can be seen to have a return expression, and this expression is like a bus where we guarantee that only one value is gated onto the bus. This value will be one of all return statements contained within the method. For example, the following method definition:

```
int meth(int arg){
  if(E(arg)){
    return 0;
  }else{
    return −1;
} }
```

The return statements are nested within a conditional that depends on some boolean expression E in terms of the method argument. As a SAT problem, the return expression for meth would be $E(arg) \rightarrow 0\ ,\ -1$. This return expression could itself be a subexpression of a larger return expression if meth was more complicated.

Non-deterministic Choice: When specifying the model for data structures it is sometimes convenient to be able to pick a value from the structure without having to specify any particular policy or logic implementing the ‘picking’. Consider an abstract set. Both a search tree and hash table are valid implementations of that set. Now consider an iterator over a set of values. The implementation of the iterator would necessarily have to consider the implementation of the set. However, that logic has nothing to do with the abstract operation of iterating over a set of data. Forcing the modeler to chose an iteration order would over-constrain the model and would restrict its usefulness. To be more general, an iterator should just specify that it picks a not-yet-picked value. To support situations where the modeler would prefer not to have to specify a particular mechanism for choosing data, ACCLAM supports a non-deterministic choice operator, choose. A choose statement specifies one or more locally-scoped variables to ‘choose’ and a predicate that constrains the particular values chosen. A body is provided to perform any additional operations on the chosen values. For example, assume that there’s some iterator model nested inside an abstract set and that we wish to use non-deterministic choice to abstract away the particular picking mechanism.

```
model setIterator<T> {
  boolean alreadyPicked[T];
  int remaining = count(T v ; !alreadyPicked[v] && in[v]);
  T next() {
    if (remaining = 0) {
      //nothing left to iterate over, an error
      throw NoSuchElementException;
    }

    choose(T next ; !alreadyPicked[next] & in[next])
    {
      alreadyPicked[next] = true;
      return next;
    } } }
```

In this example non-deterministic choice is used to grab a value from the enclosing set such that it hasn't already been visited by the iterator but is in the set. Which specific value is chosen is unspecified (and actually up to the vagaries of the SAT solver). This means that proofs involving this iterator specification are true over all valid set iterators and are not tied to any particular iteration order.

Non-deterministic choice *cannot* fail. This means that there must always be at least one assignment of values that satisfies the predicate. This is a requirement of the underlying proof system (a SAT solver). However the ACCLAM tool is capable of generating a side-condition proof that uses the SAT solver to prove that `choose` is used only in cases where there will always be a 'pickable' value. In the example above, the `remaining` reduction is used to guard the choose, by throwing an exception if there is nothing to pick (this is standard Java Iterator semantics).

Other Expressions: ACCLAM supports all the standard Java logical and bitwise operations. ACCLAM also supports (syntactically) all the arithmetic operations. Simple arithmetic, such as + and − is eventually implemented directly in terms of SAT clauses, but more complex operations (such as division) may be supported only as uninterpreted functions.

ACCLAM also supports function invocation, which is identical to Java syntax for function calls, but is carried out by inlining the called function at that point in the caller.

5.4.3 Atoms and Literals

ACCLAM supports the literal types of Java, and also treats certain concepts more concretely. Object references in ACCLAM are eventually modeled as indexes into the heap, and can be treated as a 32-bit value (although typing rules prevent integer operations from being used on references). Additionally, null has a specific concrete value at the SAT level, as does void (in order to allow the comparison of the return values of two void methods).

ACCLAM supports the typing system of Java, but unlike Java, all data in ACCLAM is boiled down to bit-vectors manipulated at the SAT level. Therefore a concrete meaning (in terms of bit-width) has to be assigned to all types. Fortunately the Java language spec [26] provides such assignments for all the primitive types, and also mandates that object references be at least 32 bits. Therefore, references can be considered 32-bit index values into an array of objects without losing any power to describe Java data structures. This also establishes a width for null. Another unique type is void, and the bit-width is essentially arbitrary (however, as a point of implementation it would probably be prudent to assign it a size and value guaranteed to be unique).

5.4.4 Model Inheritance

A model can be derived from another model. The derived model is said to be a submodel of the parent model. However, submodeling requires that the submodel be fully substitutable for its parent. This is to ensure that any reasoning done about an instance of the parent model as the parent model will apply to the submodel as well.

5.4.4.1 Conflicts in Sub Models

To be fully substitutable, any reasoning done about the parent model's conflict states must also apply to the submodel. In particular, if two methods in a state were proved not to conflict, the submodel's versions must also not conflict. If $P(\sigma)$ is the conflict predicate of the parent model and $S(\sigma')$ is the conflict predicate of the submodel, a submodel is *conflict-substitutable* for the parent model if: $\neg P(\sigma) \rightarrow \neg S(\sigma)$. Note that this means a submodel can have fewer conflict states than the parent. However, that knowledge can't be used by a model that's only reasoning over references to the parent.

5.4.4.2 Overriding Methods

An implementor of a submodel can do three things: they can add new state variables, add new methods, and override methods defined in the parent model. New state and new methods can't have been used to reason about parent models, by definition. So they are inherently substitutable. Overridden methods may have been reasoned about in the context of a parent model, and so they must not violate any properties that are exposed in order to be substitutable. Outside of a model definition, methods are very much like 'black boxes'. The only things visible to an invoker are the input state, the output state, and any abstract locks that may have been acquired. Therefore, any overridden method in a submodel must transform the same parent model input states into the same parent model output states. Of course, if the submodel is defined to include additional state variables, the input state may include values for those new state variables (M.M. for the output state). If σ_{in} is the input state of the parent model, and σ_{out} is the output state (both are sets of values for the actual state variables), and if Σ_i is the input state in the submodel and Σ_o is the output state; then a submodel's overridden method must obey: $\forall(\sigma_{in}, \sigma_{out}), (\Sigma_i, \Sigma_o)\ \sigma_{in} \subset \Sigma_i \rightarrow \sigma_{out} \subset \Sigma_o$. That is, a submodel's overridden method must map the same parent input states onto the same parent output states. A submodel that obeys this *state-mapping property* is *method-substituteable*.

Of course, a submodel may have more state variables than the parent, so a submodel may have an effectively larger state. This is why we use the subset relationship in the definition of method substitutability. To be fully rigorous, we must define $\subset$ for instance states. If we assume that a state is a set of mappings from symbols to expressions, then $\sigma_1 \subset \sigma_2 \rightarrow \forall(s \mapsto e) \in \sigma_1 \exists(s \mapsto e') \in \sigma_2$.

5.4.4.3 Invariants

The last visible model property we discuss is invariants. If a submodel is conflict-substitutable, and any overridden methods are method-substitutable, then the overridden methods won't violate the parent's invariants. However, any new methods must also obey the parent's invariants. A submodel whose new methods obey the parent model's invariants is *invariant-substitutable*.

5.4.4.4 Substitution

Any submodel that is conflict-substitutable, method substitutable and invariant substitutable is fully substitutable for any reference to the parent model. For the rest of this work, we assume that any submodel is fully substitutable with its parent models. Although it would be possible to extend the language processing tool to verify this, the current system does not and it was considered outside the scope of this thesis.

CHAPTER 6

ACCLAM EXAMPLES AND MODEL EXCERPTS

This section includes example models and discussions to motivate the structure of the language. The examples are descriptions of data types modeled from the main types of the Java Collections Framework [43].

6.1 Set

This example represents a simple set data type. A set contains at most one instance of an object, but makes no guarantees as to ordering among items. It's just a mutable set of objects (in the mathematical sense). The Set model (Figure 6.1) is type parameterized, ensuring that it is a homogeneous set (up to sub-typing). The model supports four basic operations: `add`, `remove`, `find`, and `size`. The `add` and `remove` methods return a boolean value indicating whether or not the operation caused the set to change. The state for Set is: a single relation, `in`, which maps objects to a membership flag, and a reduction `sz`, over `in` that counts the number of distinct member objects in the `Set`.

`add:` The `add` method insures an element is in the set. This is intended to model the way a Java programmer would interact with a data type. So rather than returning a new set that includes the new value, the `add` method mutates the `in` relation and returns a boolean to indicate whether or not the element was already present in the set. Of note in this description is the use of the preval `present`, which is just a name to reference a snapshot of `in`. However, `present` makes it nice and easy to express the conditional undo action without having to use a conditional statement.

```
package examples;

model Set <T> {
  boolean in[T]; //membership relation
  int sz = count(T x ; in[x] ; 1 ; 0);

  int size() { return sz; }

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

  boolean add(T elem)
  [boolean present = in[elem]; ]{
    in[elem] = true;
    return !present;
  } onabort {
    in[elem] = present;
  }

  boolean remove(T elem)
  [boolean present = in[elem]; ]{
    in[elem] = false;
    return present;
  } onabort {
    in[elem] = present;
  }
}
```

Figure 6.1. An ACCLAM Set Example

`remove:` `remove` is very similar to `add`, it just uses a different boolean constant in the mutation.

`size:` The mutating methods, `add` and `remove`, don't adjust any counters to account for their changes to the membership of the set. This is ok because the system is modeling such changes with a reduction variable that will be kept in sync automatically by the verification tool. This is a reasonable thing to do. In practice, it is not uncommon to find that shared counters (like a size variable) end up as points of contention for concurrent threads [25]. Varied techniques exist for alleviating this. Sometimes, atomic operations may be used, or the counter is broken up into an array of partial values that is periodically aggregated. It is important to realize that all these techniques are implementations of atomically updating an abstract aggregating counter variable. Therefore, in ACCLAM we don't require the modeler to specify a particular implementation strategy. The model will just indicate the aggregation that is desired and assume that the operations are atomic as a whole (note that the restrictions on the `count` reduction ensure that the counter-level operations are commutative).

6.1.1 Conflict Predicates

Element-based operations on a set can conflict only if they are referring to the same element. Global operations on a set may conflict with any mutating operation. This is expressed in the table below.

	`add(y)`	`find(y)`	`remove(y)`	`size()`
`add(x)`	`x == y`	`x == y`	`x == y`	`!in[x]`
`find(x)`	`x == y`	`false`	`x == y`	`false`
`remove(x)`	`x == y`	`x == y`	`x == y`	`in[x]`
`size()`	`!in[y]`	`false`	`in[y]`	`false`

Table 6.1. Conflict predicate matrix for `Set`

All the element-wise operations (`add`, `find`, `remove`) have predicates that only depend upon the argument values. The global operation, `size`, requires global state information in order to be precise. It would be correct to have `size` always conflict with any mutating operation. However, that would include situations where there would be no actual change to the Set's abstract state (e.g., `add(x)` where `x` is already present in the set). The precise predicate indicates conflict only in those situations where the other operation will change the state of `in` and therefore the return value of `size()`.

A reasonable question at this point might be why the conflict predicate for `add(x) * add(y)` is `x == y`. One might reasonably argue that a more precise predicate would be something like `x == y & !in[x]`, where `in` is evaluated in the initial state (before either method has executed). This more complicated predicate is correct because conflict isn't just between the forward-going parts of the operation. Two operations will also conflict if the inverse operations conflict. If the initial state did *not* contain `x`, then the inverse operation for the first `add` invocation would be a `remove`, which does not commute if `x == y`. If the initial state already contained `x`, then both `add` operations would have a no-op as an inverse (which commutes trivially). So the more complex predicate is correct because it is false precisely when the inverse operations would commute. It is also more precise because it is true for fewer states than the simpler predicate. We chose to present the less precise (but still correct) predicate initially because: it is simpler and therefore intuitively easier to reason about; and more importantly, it is not clear what (if anything) the additional precision of the complex predicate is buying us. From a more pragmatic point of view there is always a question of trading off precision for resources (time and space).[1] A more complex predicate will require more resources to evaluate. The simple predicate just needs to examine the arguments. This is a straightforward thing to provide in a system (because a TM system would need those arguments to evaluate the methods anyway). The complex predicate requires an additional saved snapshot of prior state, which is not something one could reasonably expect a real system to provide. Therefore, because the simpler predicate is easier to explain and also more realistic, we have chosen to present it as the default predicate.

6.1.2 Abstract Locking

Using locks to detect conflicts with reasonable precision requires locks that can accept two levels of granularity. The finer grained locks will be over specific elements so that the element-wise operations will have lock conflicts only if the operations are referring to the same elements. The coarser grained locks will be more global. To represent these locking

[1] This is true for systems like boosting that explicitly use predicates for concurrency control. Locking-based systems do not have this restriction.

domains, we model the lock space as a collection of points. Each possible `T` is a separate point in this collection. There are two lock constructors: `Point1D` and `Everything`. `Point1D` locks are locks over a specific instance of `T`. `Everything` locks are locks over the entire collection (acting like a whole-object lock).

Figure 6.2 is a reduced version of the locking-based Set model. The element-wise operations all use point locks and are as precise as the conflict predicates in Table 6.1. However, the lock used for `size` is much less precise and causes all mutating operations to conflict. The precision could be increased by taking locks within the body of the mutating methods after determining whether or not the element is in the set, as in Figure 6.3.

```
model Set<T> {
   . . .
   //Lock table
   lockTable locks1D<T> setLocks;

   boolean add(T obj)
    @lock(setLocks, Point1D<T>(obj), X)

   boolean remove(T obj)
   @lock(setLocks, Point1D<T>(obj), X)

   boolean find(T obj)
    @lock(setLocks, Point1D<T>(obj), S)

   int size()
   @lock(setLocks, Everything<T>(), S)
}
```

Figure 6.2. An ACCLAM Set With Abstract Locks

The current design of the verification tool assumes that all locks are specified at method entry, so this specification is currently untestable. However, once this tool limitation is overcome, more precise locking will be easy to specify. [2]

[2] Another possibility would be to conditionalize the mode portion of the lock specification.

```
boolean add(T obj) {
  if (in[obj]) {
    @lock(setLocks, Point1D<T>(obj), S)
  } else {
    @lock(setLocks, Point1D<T>(obj), X)
  }
}
```

Figure 6.3. A more precise lock

6.2 Equivalence and Parameterizing Set

The model for Set was specified assuming that the elements of the set were equivalent only if they were equal (by ==). In general, that's not always the case, and in Java many object types allow equivalence without requiring references to be equal (i.e., they have an equals operation different from ==). We'd like to have a model for a Set that is parameterized by an equivalence relation so that we can prove properties that are true regardless of the specific mechanism used for determining object equivalence. We explored several techniques for generalizing equality.

Equivalence by relation: One approach was to abstract equivalence as a 2-parameter relation, boolean equ[T,T], that maps pairs of T's to true if they are equivalent (false otherwise). A set of invariants (shown in Figure 6.4) also needs to be specified constraining equ to obey the rules of an equivalence relation (transitivity, reflexivity, symmetry). Additionally, an invariant had to be specified to ensure that all members of an equivalence class were either in or out of the set (partial membership states are impossible). The mutating methods had to be changed to mutate in for all the members of an equivalence class. For example, the new code for add would include:

```
forall(T z; equ[z,x]) { in[z] = true; }
```

The transformation for remove is very similar. Additionally, the new equivalence relation and its invariants can also be encapsulated in a model (named Equivalence) and the parameterized set would simply have a member of the Equivalence type.

```
//equivalence class rules
invariant forall(T x; equ[x,x]); //reflexivity
invariant forall(T x, T y; equ[x,y] = equ[y,x]); //symmetry
//transitivity
invariant forall(T x, T y, T z;
        (equ[x,y] & equ[y,z]) → equ[x,z]);
//partial membership rule
invariant forall(T x, T y; equ[x,y] → (in[x] = in[y]));
```

Figure 6.4. Invariants for Set Member Equivalence

This approach demonstrates the power that invariants and relations give ACCLAM. This definition is just constraining `Equivalence` instances, which means that not only does the modeler not have to specify a concrete implementation of equivalence but also that multiple sets can each have their own independent equivalence definitions (i.e., the equivalence relation isn't fixed for all Sets or even for a particular type).

This approach has limitations, however. Size computation is no longer possible with a simple reduction. One could add an integer member and manually adjust it in each method. Another approach would be to introduce a new relation that maps elements of an equivalence class to a canonical element, and then define `in` in terms of canonical elements. That alternative approach to equivalence we describe later.

Another difficulty is in maintaining precise locks with the `equ` relation. A point lock is no longer correct, as every member of the equivalence class needs to be locked as well. Only after seeing a point-lock based model fail in the ACCLAM tool did I realize that it was still possible to add or remove other equivalent values. Demonstrating abstract locking properties is important, so I needed to change the modeling approach to accommodate locking. One could adopt the canonical element approach, or just resort to the old chestnut of reader/writer locking on the whole set.

Equivalence by canonical element mapping: The limitations of the basic relational equivalence approach led us to attempt modeling with equivalence defined as a relation mapping all elements of an equivalence class to a specific 'canonical' element within the equivalence class, such as `T canon[T]`. Now all the element-wise reasoning from the original Set model can be preserved; it is now in terms of the canonical element. We can recover the original definition of `size` and the locks, without losing any generality of expressing equivalence classes.

An interesting variation of the canonical element approach would be to have the canonical mapping actually map from type T to some other type (say T'), without losing any of the advantages of the canonical approach.

6.2.1 Iterators and Iteration

Iteration presents interesting problems of interpretation in a transactional context. Iterators can present problems in concurrent contexts because they examine the state of the data structure they're iterating over and they have two operations (`next` and `hasNext`) that are tightly coupled and can lead to atomicity violations. The logic of a Java iterator is that `next` is called after a successful call to `hasNext`. However, in a concurrent context, a large number of operations by other threads/processes may have operated on the data structure between when any particular thread calls `hasNext` and `next`. This operations may have changed the underlying state sufficiently that there is no valid value that `next` may return. Software transactions allow the caller to compose two atomic actions into a larger atomic action by wrapping them in a transaction. However forcing the programmer to remember always to do this is cumbersome. That problem is solved by merging the two operations into a `next` function that returns either the next element or some special value to indicate that iteration is over. Because ACCLAM models Java, the standard approach would be to throw an exception to indicate that there are no additional elements to iterate over. Simply returning `null` may be ambiguous, because Java permits collection types to contain `nulls`.[3]

[3] It might have been cleaner to return some pair type with a boolean value indicating if the `null` was a proper element. However, the Java standard does not do this (nor does Java have an option type which would also be cleaner).

In a transactional context, those problems are joined by a question of interpretation. It is perfectly legal to construct an iterator in one transaction, and use it in others. It is also legal for each iteration to happen within a different transaction. The question is which behaviors would make the most sense to the programmer? There are two behaviors that seemed reasonable. One, I call 'incremental' iteration. This iterator acts as if each `next` is in a separate transaction and therefore may allow some mutating transactions to change the data structure's state between invocations. Essentially, the incremental iterator remembers the values returned and returns some as yet unseen value (if there is one). The other approach I call 'snapshot' iteration. This iterator takes a snapshot of the data structure's state at the moment of iterator construction, and will iterate over that immutable copy of the data structure. This iterator acts as if no other mutating transactions modified the data structure between any invocations of `next`.

Both iterator models address different use cases for iteration. Incremental iteration is appropriate for circumstances where the data structure is shared between multiple communicating threads. An example is a global work queue. One could imagine that each of the consumer threads just needs to get the next work item, and that it is more efficient for the effects of other transactions to exclude work items that have already been dequeued. Snapshot iteration is appropriate for situations where the iterating transaction(s) need to process a single consistent version of the data structure. An example is converting a collection of Objects to a collection of Strings. Any situation where the programmer needs to read a holistically complete version of the data structure is a good use case for snapshot iteration.

Incremental Iterator: To implement the incremental iterator, we could use something like Figure 6.5. This example uses a relation `seenIt` to track values that have already been returned (assume that it is initialized to false when the iterator is constructed). In order to be as general as possible, we shouldn't specify a particular iteration mechanism or order. That way, any results obtained with an `IncrementalIterator` will be valid for any particular implementation. `next` is therefore implemented with a non-deterministic choice operator, so that any possible next value may be returned. This translates into a requirement on the verification tool that it prove things for all possible legal return values from `next`. The `remaining` reduction variable is used to determine if there is anything to iterate over.

```
model IncrementalIterator {
  boolean seenIt[T];
  int remaining = count(T z ; in[z] && !seenIt[z]);
  . . .
  T next() {
    if (remaining > 0) {
      choose (T idx ; in[idx] && (!seenIt[idx])) {
        seenIt[idx] = true;
        return idx;
    } else {
      throw new Exception();
} } } }
```

Figure 6.5. An Example Implementation of an Incremental Iterator

Snapshot Iterator: To implement a snapshot-based iterator, we could do something like the example in Figure 6.6. This iterator has a relation `contents` that stores a copy of the containing `Set`'s `in` relation. As elements are returned, the elements in `contents` are set to false. This iterator's `next` is a close model of the Java standard iterator because it throws an exception if there are no more elements to return.

6.3 Ordered Set

A set of ordered elements with operations that take that order into account covers a lot of data structure ground. It covers search trees and skip lists and binary heaps (and many more). Underlying all the implementation differences is an abstraction that is just a set of elements that maintains some ordering and can be accessed according to that order. We'd like to prove things about an ordered set without any dependence upon a particular ordering. Since ACCLAM has inheritance, we can start by defining a model of a partial

```
model SnapshotIterator {
 boolean contents[T];
 int remaining = count (T z ; contents[z] ; 1 ; 0);

 SnapshotIterator() {
  forall (T idx ; true) {
   contents[idx] = in[idx];
 } }

 T next() {
  if (remaining > 0) {
   choose (T idx ; contents[idx]) {
    contents[idx] = false;
    return idx;
   }
  } else {
   throw new Exception();
} } }
```

Figure 6.6. An Example Implementation of a Snapshot Iterator

order that uses a relation to model the ≤ operation (see Figure 6.7). For reasons of space,

```
model PartialOrder<T> {
 boolean leq[T,T];
// reflexivity
 invariant forall (T x; leq[x, x]);
// transitivity
 invariant forall (T x, T y, T z;
                (leq[x, y] → (leq[y, z] → leq[x, z]));
 ...
}
```

Figure 6.7. Partial Order Example

I have omitted definitions of the methods for comparison (e.g., `gt`, `lt`, `ge`, etc.). Now we can extend the partial order definition to model a total order (Figure 6.8): All that was needed was one additional invariant to exclude non-total orderings.

An ordered set is a set, parameterized by a total order, that can be traversed in ascending and descending order. The way we chose to model the ordering aspect was with two

```
model TotalOrder<T> extends PartialOrder<T> {
  invariant forall(T x, T y; leq[x, y] | leq[y, x]);
}
```

Figure 6.8. Total Order Example

relations: `next` and `prev`. `next` maps a value to the next higher element in the set (`null` if there is none higher). `prev` maps a value to the next lower element in the set (`null` if there is none lower). Note that the value itself need not be in the set. For example, using the natural ordering of integers, if 1 and 3 were in the set, `next[2]` should produce 3, even though 2 isn't in the set. Let's examine the definition of `ParamOrderedSet` in Figure 6.9 with two new methods, `lower` and `higher` that return the next lower and next higher elements: But how should we define the mutating methods to preserve ordering? The idea

```
model ParamOrderedSet<T> extends Set<T> {
  TotalOrder<T> order;
  T next[T]; // next higher T that is in the set
  T prev[T]; // next lower T that in in the set
  ...
  T higher (T x) { return next[x]; }
  T lower (T x) { return prev[x]; }
  ...
```

Figure 6.9. A Parameterized Ordered Set Example

here is to use the `forall` statement to allow us to modify the entire `next` and `prev` relations without having to be aware of their specific contents. Consider the definition of `add` in Figure 6.10. This definition adjusts `next` by ensuring that all values less than `x`, but greater than any prior element, will all end up considering `x` as their next highest neighbor in the set. `remove` can be similarly changed. This, it turns out is not quite enough. When attempting to validate this model, it became apparent that we needed some invariants to constrain `next` and `prev` so that they only expressed valid orderings. Otherwise, the SAT-solver was free to generate states where the ordering was not obeyed. For example,

```
void add (T x) {
 in[x] = true;
 forall (T z; order.lt(z,x) && (next[z] = next[x])) {
  next[z] = x;
 }
 forall (T z; order.gt(z,x) && (prev[z] = prev[x])) {
  prev[z] = x;
 }}
```

Figure 6.10. Ordered Set `add` Method

we added the following two invariants to ensure that `next` ordering was enforced for all legal states:

```
invariant forall(T x, T y;
         (x ≠ null && in[y] && order.lt(x,y)) →
           (next[x] ≠ null && order.le(next[x],y)));


invariant forall(T x;
         (x ≠ null && next[x] ≠ null) →
           (in[next[x]] && order.lt(x,next[x])));
```

The first invariant ensures that any value `x` less than a value in the set `y` will have a `next`-element that is less than or equal to `y`. The second invariant enforces `next`-ordering between a value and its `next`-element.

6.3.1 Conflict Predicates

Conflicts for an ordered set are more subtle than for a simple set. Rather than being concerned with specific elements alone, the ordered access operations (`higher` and `lower`) operate across intervals of elements. Mutations that would change the returned result from an ordered access operation obviously conflict. Table 6.2 describes the conflict predicates for the two mutation operations and two iteration operations (in the interests of space, the ordering functions have been shortened (e.g., `order.gt` is represented as `gt`).

	add(x)	w = higher(x)	w = lower(x)	remove(x)
add(y)	x == y	gt(y,x) & (w == null | le(w,y))	gt(x,y) & (w == null | gt(y,w))	x == y
remove(y)	x == y	gt(y,x) & (w == null | ge(w,y))	gt(x,y) & (w == null | ge(y,w))	x == y

Table 6.2. Example Conflict Predicates for Parameterized Ordered Set

These conflict predicates depend on the returned value of the ordered access operations. For example, the conflict predicate for add * higher is true if an element is being added that sits in the interval between the argument to higher and the value it would return without interference from add.

The conflict predicates also demonstrate that we are treating null as a special value. It is used to represent positive or negative infinity depending on context. Initially, I had neglected to include the null boundary in the conflict predicates, which means that the conflict predicate will not describe any situation where an add is inserting an element greater than any currently in the set while higher is being called on the largest value in the set.

6.3.2 Abstract Locking

The methods inherited from Set can use the same kind of point-based locking. The new ordered access operations have different semantics. For a method to conflict with higher, it would have to change the value that would be returned by dereferencing the next relation. This means any change to any values in the set between x and next[x]. To specify this constraint, a point-like lock is insufficient. Therefore, we extend the lock types with a Range type. Range locks can be interpreted geometrically as line segments along the number line. If higher(x) locks all points along the range from x to next[x] then only operations against values in that range will conflict with higher. The locks are shown in Figure 6.11.

```
T higher (T x)
@lock(orderedLocks, Range1D<T>(x, next[x]), S)
...
T lower (T x)
@lock(orderedLocks, Range1D<T>(prev[x], x), S)
```

Figure 6.11. Ordered Set Range Locks

6.4 Map

Mapping types are an important set of Java collection types. To model maps in ACCLAM we use a relation to model the mapping from keys to values. An additional relation from key types to booleans is needed to track the presence or absence of keys in the mapping. This is needed not only for operations like `containsKey`, but also to distinguish between keys that map explicitly to `null` values and keys that are not in the map. Figure 6.12 shows a map definition in ACCLAM.

6.4.1 Conflict Predicates

The state of the map is key based, so the conflict predicates are based off of key equality.

	`b = put(x,y)`	`b = size()`
`a = put(k, v)`	`x == k`	`!isMapped[k]`
`a = size()`	`!isMapped[x]`	`false`

Table 6.3. Example Map Conflict Predicates

6.4.2 Abstract Locks

`Maps` can use point locks to prevent conflict by locking on the key value.

6.5 MultiMap

A multimap is a map that can contain multiple value mappings for the same key. The standard Java collection types do not include a multimap. However, it is an interesting

```
model Map<K,V> {
  V mapping[K];
  boolean isMapped[K];
  int sz = count(K z ; isMapped[z] ; 1 ; 0);

  V put(K key, V newVal)
  [V oldVal = mapping[key];
    boolean wasMapped = isMapped[key]; ]{
    mapping[key] = newVal;
    isMapped[key] = true;

    return oldVal;
  } onabort {
    mapping[key] = oldVal;
    isMapped[key] = wasMapped;
  }

  V get(K key) {
    return mapping[key];
  }

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

 int size() {
   return sz;
} }
```

Figure 6.12. Map Definition

exercise to model one. Rather than using a relation to model the mapping from a key to a value, a relation from key/value pairs to a boolean is used to model the existence of the individual mapping. A separate relation (`mapCount`) from keys to integers is used to track the number of mappings for a given key. This is needed to determine when a key no longer maps to any value in the multimap. Figure 6.14 shows the model.

```
V put(K key, V newVal)
@lock(mapLocks, Point1D<K>(key), X)
...
V get(K key)
@lock(mapLocks, Point1D<K>(key), S)
...
boolean containsKey(K key)
@lock(mapLocks, Point1D<K>(key), S)
...
int size()
@lock(mapLocks, Everything<K>(), X)
```

Figure 6.13. Map Abstract Lock Examples

6.5.1 Conflict Predicates

With a MultiMap, conflict can depend on both the key and the value. Because a MultiMap can permit multiple mappings for the same key, two `put` operations will conflict only if they are adding the same mapping. Table 6.4 contains some example conflict predicates.

	`put(w, z)`	`r = getOne(w)`
`put(x,y)`	`w == x && y == z`	`x == w`
`s = getOne(x)`	`w == x`	`false`

Table 6.4. Example Conflict Predicates for `MultiMap`

6.5.2 Abstract Locks

`MultiMap` could use the same point-based locks as `Map`. However, it could be extended to two dimensional points (one dimension is the key, one the value). Some methods, like `containsKey` still need to lock a single key, which in a two dimensional interpretation would be a line running through the key value (it would overlap every 2D point with the same key). `getOne` still uses a 1D point, because the value parameter for a 2D

```
model MultiMap<K,V> {

  int mapCount[K];
  boolean mapping[K,V];
  ...
  boolean containsKey(K key) {
    return mapCount[key] ≠ 0;
  }

  void put(K key, V newVal)
  [boolean wasMapped = mapping[key, newVal]; ]{
    mapCount[key] = mapCount[key] + 1;
    mapping[key, newVal] = true;
  } onabort {
    mapCount[key] = mapCount[key] − 1;
    mapping[key, newVal] = wasMapped;
  }

  V getOne(K key) {
    if (containsKey(key)) {
      choose (V val ; mapping[key, val]) {
        return val;
      }
    } else {
      return null;
} } }
```

Figure 6.14. A Example Model of a Multi Map

point would be the return value, and therefore not available at method entry.[4] The locking protocol is illustrated in Figure 6.15

[4] An alternative would be to allow choosing locks at return points rather than method entry. However, the current ACCLAM tool does not support this.

```
boolean containsKey(K key)
@lock(mapLocks, Point1D<K>(key), S)

void put(K key, V newVal)
@lock(mapLocks, Point2D<K,V>(key, newVal), X)

V getOne(K key)
@lock(mapLocks, Point1D<K>(key), S)
```

Figure 6.15. Lock Declarations for `MultiMap`

CHAPTER 7

FORMAL DESCRIPTION OF ACCLAM

This chapter describes the formal semantics of the ACCLAM language. It follows the conventional steps of specifying an abstract syntax and symbol domains, then static semantics and finally dynamic semantics. Where things differ is in the interpretation of the dynamic semantics. For a conventional 'executable' language, the dynamic semantics are basically the formalization of the run-time behavior of programs. Since ACCLAM is more of a modeling and description language, run-time behavior is a more distant concept. However, there is descriptive utility in examining ACCLAM in a conventional fashion. Since the actual intended use of ACCLAM is to produce expressions describing program behavior, there is a fourth section describing the dynamic semantics in terms of the *circuit expressions* that ACCLAM may produce. Finally, there is a proof that the *executable* dynamic semantics are equivalent to the *circuit* semantics. In practice, the actual ACCLAM verification tool implements the circuit-oriented semantics, additionally converting said pure expressions into CNF form for processing by a SAT solver.

7.1 Abstract Syntax

An ACCLAM program is broadly a collection of model definitions, a collection of instances of models, a collection of named states with statements showing how to produce those states, and a predicate expression in terms of the global instances and named states. ACCLAM is intended to model transactional data types specified in a language like Java, so it includes statements and expressions, objects and methods, and employs a type system very similar to the Java type system. Due to this similarity to Java, the formal semantics of ACCLAM were based on lighter-weight versions of Java semantics [7, 36]. We will be adopting their convention of using an overbar to represent tuples of entities. For example $C < \bar{U} >$ means a class type C parameterized by zero or more type parameters, $\bar{U}$. This has the nice effect of reducing the number of special case rules without reducing generality (it also looks nicer). The full concrete syntax appears in Appendix B.

7.1.1 Type Domains:

P	$\leftarrow$ `boolean	byte	short	int`	
	`| long	float	double	void`	*primitives*
X	$\leftarrow Id$	*type parameters*
U	$\leftarrow P \mid X \mid N$	*non-relation types*
C	$\leftarrow Id$	*model names*
N	$\leftarrow C{<}\bar{U}{>}$	*models*
R	$\leftarrow U[\bar{U}]$	*relations*
B	$\leftarrow P \mid N \mid R$	*non-parameterized/base types*
T	$\leftarrow U \mid R$	*all types*

It is worth noting that, unlike Java, ACCLAM will permit primitive types to be concrete type parameters.

7.1.2 Metavariables

x - ranges over variables

f - ranges over model fields

m - ranges over methods

c - ranges over literal constants

7.1.3 Metafunctions

FIELDS: A map of models to maps of names to field declarations

METHODS: A map of models to maps of names and types to a set of method bodies

INITS: A map of models to maps of types to a set of method bodies

A method body is a pair $(\bar{x}, \overline{stmt})$ of argument names and statements.

7.1.4 Models

The abstract syntax uses the inheritance operator $\triangleleft$ in place of the `extends` or `implements` keywords. Since extension and implementation are the same in ACCLAM, there's no need to distinguish the two cases. Using the $\triangleleft$ operator is a convention employed in formal Java semantics as well. In the abstract syntax, the process of object construction has been broken up into two phases. Each model has one constructor that just initializes the fields to the default values. There are multiple initialization functions that can set the model's fields to different values. This decomposition was done to simplify the semantics of `new`, and is similar to the technique employed in Middleweight Java [7]. Decomposing object construction into two phases does not change the semantics of the program at all, because allocation and initialization still occur, and in the correct order. However, in the abstract syntax, the initialization occurs because of an explicit invocation of an initialization method, rather than implicitly within the concrete syntax.

Rather than making a distinction between interfaces and classes when specifying inheritance, these descriptions will use the inheritance operator $\triangleleft$. For example, if the concrete syntax were `A extends B implements C`, the abstract syntax would be $A \triangleleft B, C$.

$$
\begin{array}{ll}
\mathit{Program} & \leftarrow \overline{\mathit{ModDef}}\ ;\ \overline{T\ x}\ ;\ \overline{x\{\overline{\mathit{stmt}}\}x}\ ;\ e \\
\mathit{ModDef} & \leftarrow \mathbf{model}\ C\ < \overline{X \triangleleft N} > \triangleleft\ \overline{N}\ \{\overline{\mathit{Field}}\ ;\ \mathit{Ctor}\ ;\ \overline{\mathit{Init}}\ ;\ \overline{\mathit{Method}}\} \\
\mathit{Field} & \leftarrow T\ f; \\
 & \mid \text{int x} = \mathbf{count}(\overline{T\ x}\ ;\ e\ ;\ e\ ;\ e) \\
\mathit{Ctor} & \leftarrow N(\overline{T\ x})\{\mathbf{super}(\bar{e})\ ;\ \overline{\mathit{stmt}}\} \\
\mathit{Init} & \leftarrow N(\overline{T\ x})\{\mathbf{super}(\bar{e})\ ;\ \overline{\mathit{stmt}}\} \\
\mathit{Method} & \leftarrow < \overline{X \triangleleft N} >\ T\ m(\overline{T\ x})\{\overline{\mathit{stmt}}\}
\end{array}
$$

7.1.5 Expressions

$$\begin{array}{rcll}
e & \leftarrow & x & \mid c \\
 & \mid & e.f & \mid (U)e \\
 & \mid & e. < \overline{T} > m(\overline{e}) & \mid \textbf{new}\, N(\overline{e}) \\
 & \mid & e[\overline{e}] & \mid \textbf{exc}\, \overline{e} \\
 & \mid & \textbf{primop}(\overline{e}) &
\end{array}$$

7.1.6 Statements

$$\begin{array}{rcll}
\mathit{stmt} & \leftarrow & ; & \mid e; \\
 & \mid & e.f = e & \mid T\ x = e \\
 & \mid & x = e\ | e[\overline{e}] = e & \mid \textbf{return}\ e \\
 & \mid & \textbf{if}(e)\{\overline{\mathit{stmt}}\}\ \textbf{else}\ \{\overline{\mathit{stmt}}\} & \\
 & \mid & \textbf{forall}(\overline{T\ x}\ ;\ e)\{\overline{\mathit{stmt}}\} & \\
 & \mid & \textbf{choose}(\overline{T\ x}\ ;\ e)\{\overline{\mathit{stmt}}\} &
\end{array}$$

7.1.7 Well-formedness

An ACCLAM program can consist of multiple models, so we assume there's a mapping structure, **MT** (model table), that maps names to models, and every model in the program is within **MT**. A well-formed ACCLAM program has:

1. No duplicate model definitions (two different models with the same name)

2. No duplicate field names within a given model

3. No cycles in the inheritance graph

4. All types are well-formed (TYPE OK)

5. All statements are well-formed (STATEMENT OK)

ACCLAM is a Java-like language, and as such requires that all objects have a parent type/model, except for the two built-in model types: `Object` and `Throwable` (the parent type of all exceptions). Also, like Java, ACCLAM models define a scoped name space and so the requirement that there are no duplicate field names really only applies to each model singly and need not account for the chain of inheritance. Detecting duplicate models and cycles in the inheritance graph can be done incrementally as the model table is populated. Likewise, detecting duplicate fields can be done incrementally in a straightforward fashion. Determining that all types and type usages are well-formed, and that all statements are well-formed requires a few additional rules.

7.1.7.1 Type well-formedness

These rules are for enforcing well-formed type usage. They prevent misuse of parameterized types, duplicate method signatures, and uses of undefined types. These rules define a new judgment `TYPE OK`, which depends on a model table (`MT`).

Models:

$$\frac{\begin{array}{l}\textbf{model}\, C < \overline{X \lhd N} > \lhd \overline{N_2}\{\overline{Field}\ ;\ Ctor\ ;\ \overline{Init};\ \overline{Method}\ \},\\ \overline{N}\ \text{TYPE OK in MT}, \overline{N_2}\ \text{TYPE OK in MT}, \overline{Field}\ \text{TYPE OK in MT},\\ Ctor\ \text{TYPE OK in MT}, \overline{Init}\ \text{TYPE OK in MT}, \overline{Method}\ \text{TYPE OK in MT}\end{array}}{C\ \text{TYPE OK in MT}} \tag{7.1}$$

Fields: A field declaration is well-formed if the type and initializing expression are well-formed.

$$\frac{T\ \text{TYPE OK in MT}, e\ \text{TYPE OK in MT}}{T\ f\ =\ e\ \text{TYPE OK in MT}} \tag{7.2}$$

Constructors and Initializers:

$$\frac{\overline{T}\ \text{TYPE OK in MT},\ \overline{stmt}\ \text{TYPE OK in MT},\ \overline{e}\ \text{TYPE OK in MT}}{N(\overline{T\ x})\{\textbf{super}(\overline{e}); \overline{stmt}\}\text{TYPE OK in MT}} \tag{7.3}$$

Methods:

$$\frac{\begin{array}{l}\overline{N} \text{ TYPE OK in MT}, \; T' \text{ TYPE OK in MT}, \\ \overline{T} \text{ TYPE OK in MT}, \; \overline{stmt} \text{ TYPE OK in MT} \\ m \in \text{METHOD}(C), C = \textbf{model } C < \overline{X \triangleleft N} > \ldots, \\ \overline{X'} \subset \overline{X}, \; \overline{N'} \triangleleft \overline{N}, \\ \overline{X'} \text{ TYPE OK in MT}, \; \overline{N'} \text{ TYPE OK in MT}\end{array}}{< \; \overline{X' \triangleleft \; N'} \; > \; T' \; m(\overline{T \; x})\{\overline{stmt}\} \text{ TYPE OK in MT}} \tag{7.4}$$

Expressions:

$$\frac{U \text{ TYPE OK in MT}}{(U)e \text{ TYPE OK in MT}} \tag{7.5}$$

$$\frac{e' \text{ TYPE OK in MT}, \; \overline{T} \text{ TYPE OK in MT}, \; \overline{e} \text{ TYPE OK in MT}}{e'. < \overline{T} > \; m(\overline{e}) \text{ TYPE OK in MT}} \tag{7.6}$$

$$\frac{N \text{ TYPE OK in MT}, \; \overline{e} \text{ TYPE OK in MT}}{\textbf{new } N(\overline{e}) \text{ TYPE OK in MT}} \tag{7.7}$$

$$\frac{}{c \text{ TYPE OK in MT}}, \; \frac{}{x \text{ TYPE OK in MT}}, \; \frac{e \text{ TYPE OK in MT}}{e.f \text{ TYPE OK in MT}, \; \textbf{exc } e \text{ TYPE OK in MT}} \tag{7.8}$$

$$\frac{e \text{ TYPE OK in MT}, \; \overline{e} \text{ TYPE OK in MT}}{e[\overline{e}] \text{ TYPE OK in MT}, \textbf{primop}(\overline{e}) \text{ TYPE OK in MT}} \tag{7.9}$$

7.1.8 Static Semantics Typing Rules:

The typing rules for ACCLAM are similar to the rules for Java. In addition to a model table (MT), the semantics require a typing environment, Γ.

Type environment

The type environment will be represented by the Greek letter Γ. The environment is a mapping from type names to types. Sometimes it is convenient to discuss an extension to the type environment, and that is represented by adding an explicit mapping after the Γ.

For example, to add a mapping that the symbol X now maps to type T', we would write: $\Gamma[X \mapsto T']$.

Inheritance Operator, $\lhd$:

ACCLAM supports limited inheritance, which these semantic descriptions capture. However, the standard syntax is cumbersome so we adopted the practice of others in the Java semantics community and use the $\lhd$ operator to mean ‘inherits from’. $X \lhd\ Y$ means that X inherits from Y, and $\overline{A} \lhd\ \overline{B}$ means that each element in $\overline{A}$ inherits from $\overline{B}$ pair-wise (and therefore $\overline{A}$ must be the same size as $\overline{B}$). Inheritance is also transitive, so:

$$\frac{A \lhd\ B,\ B \lhd C}{A \lhd\ C} \tag{7.10}$$

Next, we more formally define the inheritance operator $\lhd$

Model types

$$\frac{\Gamma \vdash A\ :\ C < \overline{U} >, \Gamma \vdash B\ :\ C' < \overline{U'} >,\ C \lhd\ C',\ \overline{U} \lhd\ \overline{U'}}{A \lhd B} \tag{7.11}$$

Primitive types

$$\frac{\Gamma \vdash A\ :\ P, \Gamma \vdash B\ :\ P',\ P = P'}{A \lhd\ B} \tag{7.12}$$

Substitutions: To express a substitution within a statement or expression, we use square brackets and the $\mapsto$ operator. For example, to substitute the variable name y for x in expression `exp`: `exp`$[x \mapsto y]$

7.1.8.1 Type Schema

relation types $\tau[\overline{\tau}]$

method types $\overline{\tau} \rightarrow \tau$

For typing models it will be convenient to have several helper functions for looking up the types of members and methods. For the purpose of these definitions, assume $C \lhd C'$.

$$\begin{aligned}
\mathit{fieldType}(C)(f) &= \begin{cases} T & f \in \mathrm{FIELDS}(C) \wedge \mathrm{FIELDS}(C)(f) = T\ f \\ fieldType(C')(f) & otherwise \end{cases} \\
\mathit{methType}(C)(m) &= \begin{cases} \overline{T} \to T' & m \in \mathrm{METHODS}(C) \\ & \wedge\ \mathrm{METHODS}(C)(m) = T' < \overline{X} > m(\overline{T\ x}) \\ \mathit{methType}(C')(m) & \mathit{otherwise} \end{cases}
\end{aligned} \tag{7.13}$$

Note that we assume overloaded methods (i.e., methods with the same name but different type signatures), have been rewritten to use different names. This is just a convenience and does not limit what the language can express.

Initializers aren't inherited, so they can just be looked up in the model directly.

$$\mathit{initType}(C)(\overline{T''}) = \overline{T''} \text{ if } C(\overline{T''}) \in \mathrm{INITS}(C) \tag{7.14}$$

Top Type: We use a place holder type, top ($\top$), for unbounded types. This can occur with an unbound type parameter in a model or method definition. $\top$ unifies thus:

$$\begin{aligned}
\mathit{unify}(\top, P) &= P \\
\mathit{unify}(\top, N) &= N \\
\mathit{unify}(\top, \top) &= \top
\end{aligned} \tag{7.15}$$

It extends the inheritance operator $\lhd$ thus:

$$\frac{T \neq \mathbf{void}}{T \lhd \top} \tag{7.16}$$

7.1.8.2 Method Overload Well-formedness

ACCLAM, like Java, requires that at each method invocation site, the specific method being invoked is statically knowable from the method name and the types of the arguments at the invocation site. This puts some restrictions on the declaration of methods that overload each other (same name, different signatures). Simply having a different type signature at the definition site in the model is insufficient, the argument list must also not subtype or super type any other argument list in the set of defined overloaded methods for a given method name. I define a predicate, *noIntersection* that formalizes this requirement by being true if two argument lists have at least one position whose types do not intersect.

$$\frac{\#\overline{T_1} \neq \#\overline{T_2}}{\mathit{noIntersection}(\overline{T_1}, \overline{T_2}, \Gamma)} \tag{7.17}$$

$$\frac{\overline{T_1} = \{T_{1,1}, \ldots, T_{1,k}, \ldots, T_{1,n}\}, \overline{T_2} = \{T_{2,1}, \ldots, T_{2,k}, \ldots, T_{2,n}\}, \mathit{unify}(T_{1,k}, T_{2,k}, \Gamma) = \emptyset}{\mathit{noIntersection}(\overline{T_1}, \overline{T_2}, \Gamma)} \tag{7.18}$$

The *noIntersection* predicate can be used to define the rules for legally overloading methods in a model. In this case, it is part of the definition for a well-formedness condition, OVERLOAD OK (that is in terms of a particular model C and type environment).

$$\frac{\begin{array}{c} m_1 =<\overline{X} \lhd \overline{N}> m(\overline{T_1\ x_1}),\ m_2 =<\overline{Y} \lhd \overline{M}> m(\overline{T_2\ x_2}),\ m_1 \neq m_2, \\ \overline{T_1'} = \overline{T_1}[\overline{X} \mapsto \overline{N}],\ \overline{T_2'} = \overline{T_2}[\overline{Y} \mapsto \overline{M}],\ \mathit{noIntersection}(\overline{T_1'}, \overline{T_2'}, \Gamma) \end{array}}{m_1, m_2 \text{ OVERLOAD OK in } C, \Gamma} \tag{7.19}$$

$$\frac{\forall m_1, m_2 \in \text{METHOD}(C)(m)\ :\ m_1 \neq m_2 \implies m1, m2 \text{ OVERLOAD OK in } C, \Gamma}{m \text{ OVERLOAD OK in } C, \Gamma} \tag{7.20}$$

$$\frac{\forall m \in \text{METHODS}(C),\ m \text{ OVERLOAD OK in } C, \Gamma}{C \text{ OVERLOAD OK in } \Gamma} \tag{7.21}$$

A model that is OVERLOAD OK has defined methods that can be directly inferred from their name and the type of the arguments at any call site. The way this is modeled in the formal semantics is to do a little name mangling and create new name bindings in the METHOD map for each of the overloaded method definitions. Then a pass through the program will replace all call sites to an overloaded method with a call to the mangled name version of the function. We assume that the name mangling function doesn't alias methods (that is, if two method's mangled names are equal, then the methods must have the same fully-qualified name and type signature). This means that the rest of the formal description doesn't have to deal with resolving overloaded methods and can treat each call site as being fully resolved. Rule 7.22 describes name mangling more formally, and rule 7.23 describes rewriting the call sites to refer to the unambiguously named methods (by using the mangled names instead). The rewriting rule has to account for the fact that the concrete types at the call site may not be identical to the method's declared types. But, because the type vectors for two overloaded method declarations must be non-overlapping in at least one position, we know that there can be at most one type vector for which the call site's types are a subtype.

$$\frac{m1 \in \mathrm{METHOD}(C)(m),\ m1 =<\overline{X} \lhd \overline{N}>\ m(\overline{T\ x}),\ \mathit{mangle}(m1) \notin \mathrm{METHOD}(C)}{\mathrm{METHOD}(C)(\mathit{mangle}(m1)) = m1} \tag{7.22}$$

$$\frac{\begin{array}{l}\Gamma \vdash e\ :\ C,\ \Gamma \vdash \overline{e'}\ :\ \overline{T'},\ m1 =<\overline{X} \lhd \overline{N}>\ m(\overline{T_1 x_1}),\\ m1 \in \mathrm{METHOD}(C)(m),\ \mathit{mangle}(m1) \in \mathrm{METHOD}(C),\\ \overline{T^*} = \overline{T_1}[\overline{X} \mapsto \overline{N}],\ \overline{T'} \lhd \overline{T^*},\ \overline{T} \lhd \overline{N}\end{array}}{e.<\overline{T}>\ m(\overline{e'}) \mapsto e.<\overline{T}> \mathit{mangle}(m1)(\overline{e'})} \tag{7.23}$$

A similar set of typing rules and name-mangling transformations can be done with the model initializer methods.

The remainder of this section will be a discussion of the rules describing a correctly typed ACCLAM program. I will take a bottom-up approach, describing the typing rules by starting with the simpler language elements and showing how they can be built into the more complex rules until we can fully describe a correctly typed model. We begin with expressions.

7.1.8.3 Expression Typing

Field Dereference:

$$\frac{\Gamma \vdash e \ : \ C,\mathit{fieldType}(C)(f) = T}{\Gamma \vdash e.f \ : \ T} \tag{7.24}$$

Casts:

$$\text{downcast:}\frac{\Gamma \vdash e \ : \ U'',U \lhd U''}{\Gamma \vdash (U)e \ : \ U} \text{ , upcast:}\frac{\Gamma \vdash e \ : \ U'',U'' \lhd U}{\Gamma \vdash (U)e \ : \ U} \tag{7.25}$$

Relation Dereference:

$$\frac{\Gamma \vdash e \ : \ U[\overline{U}], \ \Gamma \vdash \overline{e'} \ : \ \overline{U'}, \ \overline{U'} \lhd \overline{U}}{\Gamma \vdash e[\overline{e'}] \ : \ U} \tag{7.26}$$

Exceptions:

$$\frac{\Gamma \vdash e \ : \ N, \ N \lhd \textbf{Throwable}}{\Gamma \vdash \textbf{exc}\, e \ : \ N} \tag{7.27}$$

Primitive Ops:

ACCLAM supports the Java primitive operations (equality, boolean operators, arithmetic operators and primitive type conversion). Rather than list them all out, we use a place holder operation **primop**.

$$\frac{\Gamma \vdash \overline{e} \ : \ \overline{P}, \ \Gamma \vdash \textbf{primop} \ : \ \overline{P} \rightarrow P'}{\Gamma \vdash \textbf{primop}(\overline{e}) \ : \ P'} \tag{7.28}$$

Method Invocation ACCLAM, like Java, permits sub typing. Therefore, the type-correctness of a particular method invocation requires unification rather than simple type equality to check if the argument type vector is compatible with the declared parameter type vector. ACCLAM also permits parameterized types, as well. So we have two main requirements: first, the call site's arguments (type parameters and all) must unify with the method's declared parameter types; second, all the parameterized types must subtype their appropriate formal parameter type. The second requirement ensures that no legal program will contain a type substitution for a type parameter that will be incompatible with any parameterized types used in the declared method.

$$\frac{\begin{array}{l}\Gamma \vdash e\ :\ C,\ \text{METHOD}(C)(m) =< \overline{X} \lhd \overline{N} >\ m(\overline{Tx}),\\ \mathit{methType}(C)(m) = \overline{T} \rightarrow T'',\ [\overline{X} \mapsto \overline{\alpha}] = \mathit{unify}(\overline{N}, \overline{T'}),\\ \Gamma \vdash \overline{e^*}\ :\ \overline{T^*},\ \overline{T^*} \lhd \overline{T}[\overline{X} \mapsto \overline{\alpha}]\end{array}}{\Gamma, [\overline{X} \mapsto \overline{\alpha}] \vdash e. < \overline{T'} >\ m(\overline{e^*})\ :\ T''} \tag{7.29}$$

Allocation (`new`) We assume that the name-mangling for initializers has already happened, so the call to a particular initialization method is unambiguous here.

$$\frac{\mathit{initType}(N) = \overline{T_i},\ \Gamma \vdash \overline{e'}\ :\ \overline{T'},\ \overline{T'} \lhd \overline{T_i}}{\Gamma \vdash \textbf{new}\, N(\overline{e'})\ :\ N} \tag{7.30}$$

7.1.8.4 Statement Types

We chose to describe well-typed ACCLAM statements with a STATEMENT OK judgment, rather than by assigning types to non-`return` statements. STATEMENT OK determines statement-by-statement if statements are well-typed. RETURN OK is a judgment especially for methods that is true only if a return statement exists on all control paths through the method body. A method body is well-typed if each statement in it is well-typed, and there is a return on every path.

Empty Statement

$$\frac{\Gamma \vdash ;\ :\ \textbf{void}}{;\text{STATEMENT OK in }\Gamma} \tag{7.31}$$

Expressions

$$\frac{\Gamma \vdash e\ :\ T}{e\ \text{STATEMENT OK in }\Gamma} \tag{7.32}$$

Conditionals

$$\frac{\begin{array}{l}\overline{\mathit{stmt}_1}\ \text{STATEMENT OK in }\Gamma,\ \overline{\mathit{stmt}_2}\ \text{STATEMENT OK in }\Gamma,\\ \Gamma \vdash e\ :\ \texttt{boolean}\end{array}}{\textbf{if}(e)\{\overline{\mathit{stmt}_1}\}\textbf{else}\{\overline{\mathit{stmt}_2}\}\ \text{STATEMENT OK in }\Gamma} \tag{7.33}$$

Field Assignment:

$$\frac{\Gamma \vdash e\ :\ N,\ \Gamma \vdash e'\ :\ N', \mathit{fieldType}(N)(f) = N'', N' \lhd N''}{e.f = e' \text{ STATEMENT OK in } \Gamma} \tag{7.34}$$

Local Variable Assignment:

$$\frac{\Gamma \vdash x\ :\ T,\ \Gamma \vdash e\ :\ T',\ T' \lhd T,\ x \neq \textbf{this}}{x = e \text{ STATEMENT OK in } \Gamma} \tag{7.35}$$

Relation Assignment:

$$\frac{\Gamma \vdash e\ :\ U[\overline{U^*}],\ \Gamma \vdash \overline{e'}\ :\ \overline{U'},\ \Gamma \vdash e''\ :\ U'',\ \overline{U'} \lhd \overline{U^*},\ U'' \lhd U}{e[\overline{e'}] = e'' \text{ STATEMENT OK in } \Gamma} \tag{7.36}$$

forall:

$$\frac{\Gamma[\overline{x} \mapsto \overline{T}] \vdash e\ :\ \texttt{boolean},\ \overline{\mathit{stmt}} \text{ STATEMENT OK in } \Gamma[\overline{x} \mapsto \overline{T}]}{\textbf{forall}(\overline{T\ x}\ ;\ e)\{\overline{\mathit{stmt}}\} \text{ STATEMENT OK in } \Gamma} \tag{7.37}$$

choose:

$$\frac{\Gamma[\overline{x} \mapsto \overline{T}] \vdash e\ :\ \texttt{boolean},\ \overline{\mathit{stmt}} \text{ STATEMENT OK in } \Gamma[\overline{x} \mapsto \overline{T}]}{\textbf{choose}(\overline{T\ x}\ ;\ e)\{\overline{\mathit{stmt}}\} \text{ STATEMENT OK in } \Gamma} \tag{7.38}$$

return:

$$\frac{\Gamma \vdash @\textbf{return}\ :\ \textbf{void}}{\textbf{return}; \text{ STATEMENT OK in } \Gamma} \tag{7.39}$$

$$\frac{\Gamma \vdash e\ :\ T,\ \Gamma \vdash @\textbf{return}\ :\ T', T \lhd T'}{\textbf{return}\ e; \text{ STATEMENT OK in } \Gamma} \tag{7.40}$$

$$\frac{\Gamma \vdash e\ :\ N,\ N \lhd \textbf{Throwable},\ \Gamma \vdash @\textbf{throws}\ :\ \overline{T^*},\ \exists T' \in \overline{T^*}.N \lhd T'}{\textbf{return exc}\ e; \text{ STATEMENT OK in } \Gamma} \tag{7.41}$$

sequence with no declaration:

$$\frac{\mathit{stmt}_1 \text{ STATEMENT OK in } \Gamma,\ \overline{\mathit{stmt}} \text{ STATEMENT OK in } \Gamma}{\mathit{stmt}_1; \overline{\mathit{stmt}} \text{ STATEMENT OK in } \Gamma} \tag{7.42}$$

sequence with a declaration:

$$\frac{\overline{stmt} \text{ STATEMENT OK in } \Gamma}{T\ x;\ \overline{stmt} \text{ STATEMENT OK in } \Gamma[x \mapsto T]} \tag{7.43}$$

RETURN OK

$$\frac{}{\textbf{return}; \text{ RETURN OK}} \tag{7.44}$$

$$\frac{}{\textbf{return } e; \text{ RETURN OK}} \tag{7.45}$$

$$\frac{}{\textbf{return exc } e; \text{ RETURN OK}} \tag{7.46}$$

$$\frac{\overline{stmt};\ stmt',\ stmt' \text{ RETURN OK}}{\overline{stmt};\ stmt' \text{ RETURN OK}} \tag{7.47}$$

$$\frac{\overline{stmt} \text{ RETURN OK}}{\textbf{choose}(\overline{T\ x}\ ;\ e)\{\overline{stmt}\} \text{ RETURN OK}} \tag{7.48}$$

$$\frac{\overline{stmt_1} \text{ RETURN OK, } \overline{stmt_2} \text{ RETURN OK}}{\textbf{if}(e)\{\overline{stmt_1}\} \textbf{ else } \{\overline{stmt_2}\} \text{ RETURN OK}} \tag{7.49}$$

`return` is not allowed in a `forall` statement, so there is no `forall` case for the RETURN OK judgment.

Method Definitions: Here we use the definitions of both STATEMENT OK and RETURN OK to define the rules for correct method definitions. Basically, a method is statically ok if all of the statements are well-typed and every control path ends in a well-typed return.

$$\frac{\begin{array}{l} \text{METHOD}(C)(m) = T' < \overline{X} >\ m(\overline{T\ x})\{\overline{stmt}\}, \\ \overline{stmt} \text{ STATEMENT OK in } \Gamma[\overline{x} \mapsto \overline{T}, @\textbf{return} \mapsto T'], \\ \overline{stmt} \text{ RETURN OK} \end{array}}{m \text{ STATEMENT OK in } \Gamma,\ C} \tag{7.50}$$

Model Definitions: Given that we now can specify well-typed expressions, statements and methods, we can now define well-typed models by a MODEL OK judgment. Note that the types of inherited methods and fields are already handled by the auxiliary functions *fieldType* and *methType*. For this rule, we assume that the type parameters for the model can be separated out into unconstrained parameters ($\overline{W}$) and constrained parameters ($\overline{X}$).

$$\frac{\begin{array}{l}\Gamma' = \Gamma[\overline{W} \mapsto \top,\ \overline{X} \mapsto \overline{N}],\\ \overline{T\ x\ =\ e};\ \text{STATEMENT OK in}\ \ \Gamma',\\ \Gamma^{+} = \Gamma'[\overline{x} \mapsto \overline{T}],\\ \overline{\mathit{Init}}\ \text{STATEMENT OK in}\ \Gamma^{+},\\ \overline{\mathit{Method}}\ \text{STATEMENT OK in}\ \Gamma^{+},\\ \overline{N'}\ \text{TYPE OK in}\ \Gamma\end{array}}{\textbf{model}\ C < \overline{W},\ \overline{X} \lhd \overline{N} > \ \lhd \overline{N'}\ \{\overline{T\ x = e};\ \overline{\mathit{Init}}\ \overline{\mathit{Method}}\}\ \text{MODEL OK in}\ \Gamma} \tag{7.51}$$

If every model in a model table is TYPE OK and MODEL OK, then the model table is well-typed.

Programs: We now know how to specify well-typed models and model tables. An ACCLAM program is just a model table, a set of named state transitions and an expression. Since we also know how to specify well-typed expressions that just leaves named states. Named state transitions are defined as a starting state (with σ_i meaning the initial state), a sequence of statements, and a new state (the named state). The rule is essentially that a named state is well-typed if the statements are well-typed and the starting state is known. For the second requirement, a new environment, STATES, will be introduced that contains mappings from names to states. State names must also be unique, to avoid the possibility of colliding with other states as well as any user-defined symbol in the program. We assume there's a naming environment, Σ, for state names.

$$\frac{\begin{array}{c}\text{STATES} \vdash \sigma,\ \sigma \in \Sigma, \sigma' \notin \Sigma,\\ \overline{\mathit{stmt}}\ \text{STATEMENT OK in}\ \Gamma\end{array}}{\sigma\ \{\overline{\mathit{stmt}}\}\ \sigma'\ \text{OK in}\ \Sigma[\sigma'],\ \Gamma} \tag{7.52}$$

$$\frac{\begin{array}{c}\overline{\sigma_{in}\{\overline{stmt}\}\sigma_{out}} \text{ OK in } \Sigma,\Gamma,\\ \sigma\{\overline{stmt}\}\sigma' \text{ OK in } \Sigma[\sigma'],\Gamma\end{array}}{\overline{\sigma_{in}\{\overline{stmt}\}\sigma_{out}};\sigma\{\overline{stmt}\}\sigma' \text{ OK in } \Sigma[\sigma'],\Gamma} \tag{7.53}$$

Now all that remains is to specify an overall rule for ACCLAM programs. The top-level expression is just like a normal expression, except that portions of it can be evaluated in terms of different named states (drawn from Σ). Because the program is well-typed, the names will still refer to the same types even if the values may be different in different states. Therefore, we can just type the expression as a normal ACCLAM expression.

$$\frac{\begin{array}{l}\overline{C} \text{ MODEL OK in } \Gamma,\\ \overline{T\ x = e;} \text{ STATEMENT OK in } \Gamma,\\ \Gamma' = \Gamma[\overline{x} \mapsto \overline{T}],\\ \overline{\sigma_{in}\{\overline{stmt}\}\sigma_{out}} \text{ OK in } \Sigma,\Gamma',\\ \Sigma,\Gamma' \vdash b \ \texttt{: boolean}\end{array}}{\overline{C};\overline{T\ x = e;}\ \overline{\sigma_{in}\{\overline{stmt}\}\sigma_{out}};b \text{ OK}} \tag{7.54}$$

At this point we have formally specified well-formed and well-typed ACCLAM programs.

7.2 Operational Semantics

ACCLAM was designed initially to be a modeling language, used primarily as input to automated theorem proving systems. However, since ACCLAM models operational languages, it makes sense to have a notional operational specification for the meaning of ACCLAM programs. This section will formally describe an operational semantics for ACCLAM. The semantics will be in terms of configurations, which are tuples describing a state. This approach was used in Middleweight Java [7], and it maps reasonably directly onto the Java state model.

A configuration consists of a Heap, a Stack, a term being evaluated and a set of terms remaining to be evaluated (the continuation or frame stack). The terms are also called frames, and I will use these words interchangeably. The heap is a mapping from ids to heap objects (model instances and relation instances). A model instance is a pair of a type and a mapping from field names to values. A relation is a lambda function from indexes to values. The stack is a list of maps of variables to expressions (the list is used to handle pushing and popping). A term is a bit of program syntax, and the continuation/frame stack is a list of terms. The empty list is represented as []; $\circ$ is the list cons operator.

Term frames come in two flavors, closed and open. A closed frame is one that has been completely evaluated and needs no additional values to be computed for its effects to be known. An open frame is one that requires one or more values to be substituted before the entire term can be evaluated. In the semantics the 'hole' that is being evaluated will be represented by a bullet point, $\bullet$. All this is summarized in the grammar we now describe:

We will start with the heap and stack portions of the configuration. The stack will be referred to as *S* or *VS* (for value stack). This is the stack up to this point. *MS* is a mnemonic for method stack, and it is the stack frame that is tacked onto the top of the stack on method entry. *BS* stands for block stack, and is the mnemonic for a stack frame associated with a block scope (e.g., the `else` block of a conditional).

$$\begin{array}{ll} \mathit{config} & := (H, VS, CF, FS) \end{array}$$

$$\begin{array}{ll} H & := \{\mathrm{oid} \mapsto o, \mathrm{rid} \mapsto r\} \end{array}$$

$$\begin{array}{ll} VS & := MS \circ VS \mid [] \\ MS & := BS \circ MS \mid [] \\ BS & := \{x \mapsto (v,\ T)\} \end{array}$$

Next we'll describe the terms. *FS* is the frame stack, which is the continuation of the current term. *CF* is a closed frame, and is a frame where all expressions have been evaluated so the statement rule can be applied (a closed frame is a statement continuation because it may change the heap and/or stack state). *OF* is an open frame, a term in which some subexpressions have to be evaluated before a statement rule may apply (an open frame is therefore an expression continuation). For model types, we'll use the math bold $\mathbb{F}$ to stand for the mapping from field name to a value. An instance of a model in the heap will map to a pair of model type and field mapping.

$$\begin{aligned}
FS \quad &:= F \circ FS \mid [] \\
F \quad &:= CF \mid OF
\end{aligned}$$

$$\begin{aligned}
CF \quad &:= \overline{stmt} \mid \textbf{return}\ e; \mid \textbf{return exc}\ e; \\
&\mid \{\} \mid e \mid \textbf{super}(\bar{e}) \\
OF \quad &:= \textbf{if}(\bullet)\{\overline{stmt}\}\ \textbf{else}\ \{\overline{stmt}\} \mid \textbf{primop}(\bar{v}, \bullet, \bar{e}) \\
&\mid \bullet.f \mid \bullet.f = e \mid o.f = \bullet \\
&\mid \bullet[\bar{e}] \mid r[\bar{v}, \bullet, \bar{e}] = e \mid r[\bar{v}] = \bullet \\
&\mid (U)\bullet \mid \bullet.m(\bar{e}) \mid v.m(\bar{v}, \bullet, \bar{e}) \\
&\mid \textbf{new}\ N(\bar{v}, \bullet, \bar{e}) \mid \textbf{super}(\bar{v}, \bullet, \bar{e}) \\
&\mid x = \bullet \mid \textbf{return}\ \bullet
\end{aligned}$$

$$\begin{aligned}
v \quad &:= \textbf{null} | \textbf{void} | \ o \mid p \mid r \\
o \quad &:= (N, \mathbb{F}) \\
r \quad &:= \lambda \bar{i}.f(\bar{i}) \to v,\ r[\bar{i}] \\
&\mid \lambda \bar{i}.(\bar{i} = \bar{v}) \to v,\ r[\bar{i}] \\
&\mid \lambda \bar{i}.v
\end{aligned}$$

7.2.1 Metafunctions

For the sake of convenience, I'll define several metafunctions to read values out of the stack, to update values in the stack, and to produce initial values. If S is a stack, then $\mathrm{dom}(S)$ is the set of variables in the stack.

eval: Eval takes a variable that is in a stack and produces the expression to which that variable currently maps.

$$\frac{x \in \mathrm{dom}(BS)}{\mathrm{eval}((BS \circ MS), x) = BS(x)} \quad \frac{x \notin \mathrm{dom}(BS)}{\mathrm{eval}((BS \circ MS), x) = \mathrm{eval}(MS, x)} \tag{7.55}$$

update: Update changes the mapping of a variable to a new expression.

$$\frac{BS(x) = (v, T)}{\mathrm{update}((BS \circ MS), (x \mapsto v')) = BS[x \mapsto (v', T)] \circ MS} \quad \frac{x \notin \mathrm{dom}(BS)}{\mathrm{update}((BS \circ MS), (x \mapsto v)) = BS \circ \mathrm{update}(MS, (x \mapsto v))} \tag{7.56}$$

init: *Init* returns the initial value for the type provided.

$$\frac{U = C < \overline{U} >}{init(U) = \mathbf{null}} \tag{7.57}$$

$$\frac{U = \text{boolean}}{init(U) = \mathbf{false}} \tag{7.58}$$

$$\frac{\begin{array}{l} U = \texttt{int} \vee U = \texttt{long} \vee U = \texttt{float} \vee U = \texttt{double} \\ \vee\, U = \texttt{short} \vee U = \texttt{char} \vee U = \texttt{byte} \end{array}}{init(U) = 0} \tag{7.59}$$

7.2.2 Computation Steps

The bulk of the operational semantics will be defined as single-step reductions that will transition from one configuration to the new, updated configuration.

7.2.2.1 Operations on Local Variables

The stack is the part of the configuration where local variable information is stored, so they need to be treated distinctly from heap objects.

variable lookup

$$\frac{\mathrm{eval}(MS,x) = (v,T)}{(H,MS \circ VS,x,FS) \to (H,MS \circ VS,v,FS)} \tag{7.60}$$

relation dereference

$$\frac{v' = \mathrm{eval}(MS,r)(\overline{V})}{(H,MS \circ VS,r[\overline{v}],FS) \to (H,MS \circ VS,v',FS)} \tag{7.61}$$

variable update

$$\frac{x \in \mathrm{dom}(MS)}{(H,MS \circ VS,x = v,FS) \to (H,\mathrm{update}(MS,x \mapsto v) \circ VS,;,FS)} \tag{7.62}$$

relation update

$$\frac{r_{old} = \mathrm{eval}(MS,r),\ r' = \lambda\bar{i}.(\bar{i} = \overline{v}) \to v,\ r_{old}(\bar{i})}{(H,MS \circ VS,r[\overline{v}] = v,FS) \to (H,\mathrm{update}(MS,r \mapsto r') \circ VS,;,FS)} \tag{7.63}$$

local variable declaration

$$\frac{BS' = BS[x \mapsto (v,\ U)]}{(H,(BS \circ MS) \circ VS,\ U\ x = v,FS) \to (H,(BS' \circ MS) \circ VS,;,FS)} \tag{7.64}$$

local relation declaration

$$\frac{H' = H[r \mapsto \lambda\overline{U'\ i}.init(U)],\ BS' = BS[x \mapsto r]}{(H,(BS \circ MS) \circ VS,Ux[\overline{U'}],FS) \to (H',(BS' \circ MS) \circ VS,;,FS)} \tag{7.65}$$

7.2.2.2 Blocks and Scope

ACCLAM employs explicit scopes, which are modeled by the stack. Adding a new scope is simply a matter of tacking a new map onto the front of the stack list. Popping a scope is complicated by the fact that we need to pop the scope off the stack at the correct point in the sequence of statements. I've adapted a trick employed by MJ and introduced a special 'scope popping' statement, $\{\}$.

scope start

$$\frac{}{(H, MS \;\circ\; VS, \{\overline{stmt}\}, FS) \to (H, ([] \circ MS) \circ\; VS, \overline{stmt}, \{\} \;\circ\; FS)} \tag{7.66}$$

scope exit

$$\frac{}{(H, (BS \circ MS) \circ VS, \{\}, FS) \to (H, MS \circ VS, ;, FS)} \tag{7.67}$$

return from scope

$$\frac{}{(H, MS \circ VS, \textbf{return}\ v;, \{\} \circ FS) \to (H, VS, v, FS)} \tag{7.68}$$

$$\frac{}{(H, (BS \circ MS) \circ VS, \textbf{return}\ v;, \{\} \circ FS) \to (H, MS \circ VS, \textbf{return}\ v;, FS)} \tag{7.69}$$

We must also support a return that happens in blocks nested within methods (for example, a return in the `else` clause of a conditional within a method body). In that case, the return's effect is to bypass all the remaining statements in the continuation until we hit a scope statement.

$$\frac{F \neq \{\}}{(H, MS \circ VS, \textbf{return}\ v;, F \circ FS) \to (H, MS \circ VS, \textbf{return}\ v;, FS)} \tag{7.70}$$

7.2.2.3 Simple Expressions and Statements

Here I'm gathering together all the operational semantics for the simpler, straightforward expressions and statements.

Conditionals:

$$\frac{v = \text{true}}{(H, S, \text{if}(v)\{\overline{stmt_1}\}\text{else}\{\overline{stmt_2}\}, FS) \to (H, S, \{\overline{stmt_1}\}, FS)} \tag{7.71}$$

$$\frac{v = \text{false}}{(H,S,\text{if}(v)\{\overline{stmt_1}\}\text{else}\{\overline{stmt_2}\},FS) \to (H,S,\{\overline{stmt_2}\},FS)} \tag{7.72}$$

Model Field Statements:

field read:

$$\frac{o \in \text{dom}(H),\ H(o) = (N,\mathbb{F}),\ f \in \text{dom}(\mathbb{F}),\ \mathbb{F}(f) = v}{(H,S,o.f,FS) \to (H,S,v,FS)} \tag{7.73}$$

`null` dereference:

$$\frac{o \in \text{dom}(H),\ H(o) = \textbf{null}}{(H,S,o.f,FS) \to (H,S,\textbf{return exc}\ \texttt{NullPointerException},FS)} \tag{7.74}$$

update field:

$$\frac{\begin{array}{l} o \in \text{dom}(H),\ H(o) = (N,\mathbb{F}), \\ f \in \text{dom}(\mathbb{F}),\ \mathbb{F}' = \mathbb{F}[f \mapsto v], \\ H' = H[o \mapsto (N,\ \mathbb{F}')] \end{array}}{(H,S,o.f = v,FS) \to (H',S,;,FS)} \tag{7.75}$$

`null` update:

$$\frac{o \in \text{dom}(H), H(o) = \textbf{null}}{(H,S,o.f = v,FS) \to (H,S,\textbf{return exc}\ \texttt{NullPointerException},FS)} \tag{7.76}$$

relation update

$$\frac{r_{old} = H(r), r' = \lambda \bar{i}.(\bar{i} = \bar{v}) \to v',\ r_{old}(\bar{i}),\ H' = H[r \mapsto r']}{(H,S,r[\bar{v}] = v',FS) \to (H',S,;,FS)} \tag{7.77}$$

Casts:

Object cast

$$\frac{H(o) = (N_1,\mathbb{F}), N_1 \triangleleft N_2}{(H,S,(N_2)o,FS) \to (H,S,o,FS)} \tag{7.78}$$

`null` cast

$$\frac{}{(H,S,(N_2)\textbf{null},FS) \to (H,S,\textbf{null},FS)} \tag{7.79}$$

Primitive casts are just like one would expect in Java. Since the code was type correct, we know the cast is legal, and therefore the semantics of primitive casts involve conversion of numerical representations and/or truncating or extending a value. For the sake of brevity, I have omitted the specific rules for each legal primitive cast type pair.

Allocation/`new`

The challenge here is in ensuring that the correct `init` method is called after the new heap object is created. However, if we assume that the overloaded method disambiguation step from Section 7.1.8.2 has happened, then we can safely assume that $\mathit{init}(C)(\overline{T})$ is defined for these arguments and will return exactly one method body.

$$\frac{\begin{array}{c} o \notin \mathrm{dom}(H),\ \mathbb{F} = [f \mapsto \mathit{init}(T)]\ \forall f \in \mathit{FIELDS}(C), \\ \mathit{MS} = [\mathbf{this} \mapsto (o,C), \overline{x} \mapsto \overline{(v,\ T)}], \mathit{INITS}(C)(\overline{T}) = (\overline{x}, \overline{\mathit{stmt}}) \end{array}}{(H,S,\mathbf{new}\ C(\overline{v}),FS) \rightarrow (H[o \mapsto (C,\mathbb{F})], \mathit{MS} \circ S, \overline{\mathit{stmt}}, (\mathbf{return}\ o;) \circ \{\} \circ FS)} \tag{7.80}$$

Method Invocations

Normal Method Invocation:

$$\frac{\begin{array}{l} H(o) = (N,\mathbb{F}), \\ \mathit{METHOD}(N)(m) = (\overline{x}, \overline{\mathit{stmt}}), \\ \mathit{methType}(N)(m) = N' \rightarrow N'', \\ \mathit{MS} = \{\mathbf{this} \mapsto (o,N),\ \overline{x} \mapsto (\overline{v},\ \overline{N'})\} \end{array}}{(H,S,o.m(\overline{v}),FS) \rightarrow (H, \mathit{MS} \circ S, \overline{\mathit{stmt}}, \{\} \circ FS)} \tag{7.81}$$

For invocation of super-model methods, the important thing is to ensure that we're evaluating the immediate parent model rather than any model on the inheritance chain from `Object`.

super invocation

$$\frac{\begin{array}{l} MS(\mathbf{this}) = \ (o,C), \\ C \lhd C' \ \wedge \ \forall\, K \ : \ (C \lhd K \ \wedge C \neq K) \implies C' \lhd K, \\ INITS(C')(\overline{T}) = (\bar{x}, \overline{stmt}), \\ MS' = \{\mathbf{this} \mapsto (o,C'),\ \bar{x} \mapsto \overline{(v,\ T)}\} \end{array}}{(H, MS \circ VS, \mathbf{super}(\bar{v}), FS) \to (H, ([\,] \circ MS') \circ (MS \circ VS), \overline{stmt}, (\mathbf{return}\ o;) \circ \{\} \circ FS)} \tag{7.82}$$

null invocation

$$\frac{H(o) = \mathbf{null}}{(H,S,o.m(\bar{v}),FS) \to (H,S,\mathbf{return\ exc}\ \texttt{NullPointerException};,FS)} \tag{7.83}$$

sequence

$$\frac{}{(H,S,;,F \circ FS) \to (H,VS,F,FS)} \tag{7.84}$$

evaluated expressions

$$\frac{}{(H,S,v,F \circ FS) \to (H,VS,F[v] \circ FS)} \tag{7.85}$$

7.2.2.4 Forall and Choose

The `forall` and `choose` statements define a predicate over which their body is evaluated. That predicate must capture the state of the model at that point, so we need to define a closure function (*CLOSE*) to evaluate the predicate in terms of the actual model state. The predicate may be defined over an arbitrary set of relations, and in the operational context is effectively ‘evaluated’ for each relation element combination. This means that the predicate cannot mutate any data defined outside the predicate functions. Because ACCLAM has an imperative flavor, this restriction doesn’t apply to variables declared within any predicate function’s local scope. This excludes model member assignment as well as relation assignment from the types of statements allowed in this context (`pureStmts`).

CLOSE

Heap Values

$$\frac{\mathrm{eval}(S,x) = o, o \in \mathrm{dom}(H)}{\mathit{CLOSE}(x,H,S) = o} \tag{7.86}$$

Field Read

$$\frac{o \in \mathrm{dom}(H),\ H(o) = (N,\mathbb{F}),\ \mathrm{f} \in \mathrm{dom}(\mathbb{F}),\ v = \mathbb{F}(\mathrm{f})}{\mathit{CLOSE}(o.f,H,S) = v} \tag{7.87}$$

$$\frac{}{\mathit{CLOSE}(e.f,H,S) = \mathit{CLOSE}(e,H,S).f} \tag{7.88}$$

Relations

$$\frac{r \in \mathrm{dom}(H),\ H(r) = (R, r_{fn})}{\mathit{CLOSE}(r[\overline{e}],H,S) = r_{fn}(\mathit{CLOSE}(\overline{e},H,S))} \tag{7.89}$$

$$\frac{}{\mathit{CLOSE}(e[\overline{e'}],H,S) = \mathit{CLOSE}(\mathit{CLOSE}(e,H,S)[\overline{e'}],H,S)} \tag{7.90}$$

Scalar Values

$$\frac{\mathrm{eval}(S,x) = v}{\mathit{CLOSE}(x,H,S) = v} \tag{7.91}$$

Primops

$$\frac{}{\mathit{CLOSE}(\mathbf{primop}(\overline{e}),H,S) = \mathbf{primop}(\mathit{CLOSE}(\overline{e},H,S))} \tag{7.92}$$

Casts

$$\frac{}{\mathit{CLOSE}((T)e,H,S) = (T)(\mathit{CLOSE}(e,H,S))} \tag{7.93}$$

Method Invocation

Method invocation requires carefully substituting for `this` and arguments. In addition any local variables declared in the method body will need to be converted into expressions and handled correctly. Finally, the return is handled differently from normal method invocation, and becomes a big expression-producing statement whose value is the value of the invocation.

method call:

$$\frac{\begin{array}{l}\Gamma \vdash o : C, \\ \mathit{METHOD}(C)(m) = (\overline{x}, \overline{\mathit{pureStmt}}), \\ \overline{e'} = \mathit{CLOSE}(\overline{e}, H, S), \\ S' = \{\mathbf{this} \mapsto o, \overline{x} \mapsto \overline{e'}\} \circ S\end{array}}{\mathit{CLOSE}(o.m(\overline{e}), H, S) = \mathit{CLOSE}(\text{valOf}(\overline{\mathit{pureStmt}}), H, S)} \tag{7.94}$$

A couple of new functions will be handy for processing methods: `valOf` takes a statement and produces an expression that represents the run-time value of that statement. `resultIs` takes an expression and produces a statement. Furthermore: $\text{resultIs}(\text{valOf}(\mathit{stmt})) = \mathit{stmt}$, and $\text{valOf}(\text{resultIs}(e)) = e$. The specific transformations from statements to expressions basically walk down through the expressions and statements and place `resultIs` at return points (returning values or exceptions). The transformation is described by the following rules:

Conditional

$$\frac{\begin{array}{l}e' = \mathit{CLOSE}(e, H, S), \\ \mathit{condEx} = \text{valOf}(\mathbf{if}(e')\{\mathit{CLOSE}(\overline{\mathit{pureStmt}_1}, H, S)\}\mathbf{else}\{\mathit{CLOSE}(\overline{\mathit{pureStmt}_2}, H, S)\}) \\ \text{eval}(e', H', S') = \mathbf{true}\end{array}}{\text{eval}(\mathit{condEx}, H', S') = \text{eval}(\mathit{CLOSE}(\overline{\mathit{pureStmt}_1}, H, S), H', S')} \tag{7.95}$$

$$\frac{\begin{array}{l}e' = \mathit{CLOSE}(e, H, S), \\ \mathit{condEx} = \text{valOf}(\mathbf{if}(e')\{\mathit{CLOSE}(\overline{\mathit{pureStmt}_1}, H, S)\}\mathbf{else}\{\mathit{CLOSE}(\overline{\mathit{pureStmt}_2}, H, S)\}) \\ \text{eval}(e', H', S') = \mathbf{false}\end{array}}{\text{eval}(\mathit{condEx}, H', S') = \text{eval}(\mathit{CLOSE}(\overline{\mathit{pureStmt}_2}, H, S), H', S')} \tag{7.96}$$

Local Variable Declaration:

$$\frac{e' = \text{eval}(\mathit{CLOSE}(e, H, S), H, S),\ S' = \{x \mapsto e'\} \circ S}{\text{valOf}(\mathit{CLOSE}(T\ x = e'; \overline{\mathit{stmt}}, H, S)) = \text{valOf}(\mathit{CLOSE}(\overline{\mathit{stmt}}, H, S'))} \tag{7.97}$$

Local Variable Assignment:

$$\frac{e' = \text{eval}(\mathit{CLOSE}(e,H,S),H,S),\ S' = \{x \mapsto e'\} \circ S}{\text{valOf}(\mathit{CLOSE}(x = e';\overline{\mathit{stmt}},H,S)) = \text{valOf}(\mathit{CLOSE}(\overline{\mathit{stmt}},H,S'))} \tag{7.98}$$

Return:

$$\begin{aligned} \mathit{CLOSE}(\textbf{return}\ e;,H,S) &= \mathit{CLOSE}(\text{ResultIs}(\mathit{CLOSE}(e,H,S)),H,S) \\ \mathit{CLOSE}(\textbf{return}\ e;\overline{\mathit{stmt}},H,S) &= \text{ResultIs}(\mathit{CLOSE}(e,H,S)) \end{aligned} \tag{7.99}$$

Forall Enter Scope: To properly evaluate a `forall`, the predicate must be closed, and then two special variables will be bound in the new stack frame. `@forallVars` will capture the forall scoped variables, and `@forallPred` will bind to the closed predicate expression. These two values are needed when evaluating the body of the `forall`.

$$\frac{\begin{array}{l} e' = \mathit{CLOSE}(e,H,MS \circ S), \\ \mathit{FREEVARS}(e') \cup \mathit{FREEVARS}(\mathit{CLOSE}(\overline{\mathit{stmt}},H,S)) \subseteq \overline{x} \end{array}}{\begin{array}{l} (H,MS \circ S,\textbf{forall}(\overline{T\ x};e)\{\overline{\mathit{stmt}}\},FS) \rightarrow \\ (H,(\{\textbf{@forallVars} \mapsto \overline{(x,\ T)},\textbf{@forallPred} \mapsto e'\} \circ MS) \circ S,\overline{\mathit{stmt}},\{\} \circ FS) \end{array}} \tag{7.100}$$

7.2.2.5 Updates in `forall` bodies

`forall` is a construct used to specify bulk parallel predicated updates. Because no ordering is specified, each update must be independent or idempotent. For scalar variables (locals and fields), idempotency means that the assignment must always be of the same value (e.g., for all predicate-satisfying values, the right-hand side of the assignment must be the same). For relation variables, idempotency means that for every update to a relation, if the index variables are equal for different updates, the update values must also be equal. These requirements for idempotency are required properties, and are a proof requirement of the modeler. Chapter 8 discusses how proof of this requirement can be automated.

Relation updates have a further complication, which is that the update may be specified in terms of expressions referring to the `forall` variables conditioned by the predicate.

Relations themselves are functions of their own index variables, so in order to produce a valid expression for the new value of the relation, the expressions in terms of `forall` variables will need to be transformed into equivalent expressions but in terms of the relation's index variables. Therefore, we assume the existence of a binding function *BIND*, that will take the predicate and relation index expressions in terms of the `forall` variables and produce a set of expressions for the predicate arguments in terms of the relation function's index variables. Consider the following example:

```
forall (int x ; P(x)) {
  rel[x + 1] = e;
}
```

We would like the new function for the relation (assuming the old one was named rel_{old}), to look like: $\lambda i \ . \ P'(i) \rightarrow e, \ \text{rel}_{old}(i)$. Because the relation function is in terms of index variables, the converted predicate P' will have to be in terms of the index variable but still equivalent to P. In this case, $P'(x) = P(x-1)$. The *BIND* function's job is to produce a set of substitutions that map the `forall` variables onto functions of the relation's index variables. Therefore, for any relation, $P'(\bar{i}) = P(\bar{x})[\bar{x} \mapsto \overline{fn_i(i)}]$, and therefore *BIND* must produce a substitution from `forall` variables onto expressions in terms of relation index variables. In general, computing an 'inverse' function for an arbitrary expression is a difficult problem. Practical systems will be able to automatically invert only a subset of expressions. Therefore, for the purposes of these semantics, *BIND* will be specified by constraining its properties. First, *BIND* must produce a substitution whose expressions have no free variables other than the relation's index variables.

Let $\bar{x}$ be the forall variables, and $\bar{i}$ be the relation index variables.

$$\forall x \in \bar{x} \ : \ \exists f(\bar{i}) \ : \ [x \mapsto f(\bar{i})] \in \mathit{SUB} \wedge \mathit{FREEVARS}(f(\bar{i})) \subseteq \bar{i} \tag{7.101}$$

Assume $f(\bar{i})$ is produced by *BIND* as a valid substitution.

$$\forall \bar{x},\ \bar{i}\ :\ [P(\bar{x}) \rightarrow (\overline{f(\bar{i}) = x})] \iff P'(\bar{i}) \tag{7.102}$$

A sample BIND function: Consider the following example *BIND* function, that can only bind relation updates whose index expressions are each a function of a single `forall` variable. Furthermore, each relation update must use all the `forall` variables. With those restrictions, we can simplify the relation update expression to something like: $r[\overline{f(x)}] = e$. Assuming that each f is invertible, we can define a bind function as: $bind(\bar{i},\bar{x},\overline{f(x)}) = [\bar{x} \mapsto \overline{f^{-1}(\bar{i})}]$. The overlining convention is hiding an important ordering requirement here, which is that the ordering of the $\bar{x}$ variables is not necessarily the same as the order in which they were declared (the x that occurs in the n^{th} position in the array of index expressions will map to the n^{th} relation index variable). This is actually very close to the *BIND* function that exists in the current implementation. Apart from the restrictions on the form of relation updates, the power of this *BIND* is limited by the ability of the implementation to invert the functions. In the current implementation it is limited to boolean functions and simple arithmetic (addition or subtraction of constant values).

Given a suitable *BIND* function, we can define the semantics for a relation update inside a `forall`:

$$\frac{\begin{array}{l} \text{eval}(S, \textbf{@forallVars}) = \overline{(x,T)},\ \text{eval}(S, \textbf{@forallPred}) = e_{pred}, \\ \overline{expr'} = CLOSE(\overline{expr},H,S),\ e' = CLOSE(e_{pred},H,S),\ r \in \text{dom}(H), \\ H(r) = r_{old} = \lambda\bar{i}.\ldots,\ r' = \lambda\bar{i}.(BIND(\bar{i},\bar{x},\overline{expr'},e) \rightarrow e',\ r_{old}(\bar{i})),\ H' = H[r \mapsto r'] \end{array}}{(H,S,r[\overline{expr}] = e,FS) \rightarrow (H',S,;,FS)} \tag{7.103}$$

Reduction Updates:

Whenever a relation is updated, any reductions defined in terms of that relation may also have changed value. A reduction is defined by a predicate over a relation and two values (one if the predicate is true, and one if the predicate is false). For the purposes of the operational semantics, we'll be specifically discussing the `count` reduction which produces an integer value. In general, it should be possible to use any commutative group to define a reduction. But, we have found that `count` seems to suffice for all the data types we've explicitly modeled. When the relation that the reduction is defined over is updated, the reduction's current value can be updated incrementally by subtracting the relation's contribution before the update and adding it back after. This two step process handles all possible permutations of the predicate's value. Reduction predicates share many similarities with `forall` predicates. They are defined in terms of scoped variables declared just for the reduction variable. To be useful, the reduction's predicate needs to be reformulated into a function of relation index variables. This is precisely what the *BIND* function does for `forall` predicates. Therefore, we can use an appropriate *BIND* function to produce a modified predicate, P'. In the interest of brevity, the rule below assumes that the updated version of the relation r is r' and that H' is the updated heap with the new value of r.

For the following rule, let $\oplus$ be the group operator ($+$ for `count`), and let 0 be the identity, $-$ be the group inverse, and let v be a value drawn from the group.

$$\frac{\begin{array}{l} \mathit{reduxVar} = (\overline{T\ x}, P(\bar{x}), v, v'), \\ H(r) = \lambda \bar{i}\ .\ \ldots,\ H(\mathit{reduxVar}) = e_{\text{init}},\ P' = \mathit{BIND}(\bar{i},\bar{x},\bar{e}), \\ e_{\mathit{before}} = (\text{eval}(P',H) \to -v,\ -v'), \\ e_{\mathit{after}} = (\text{eval}(P',H') \to v,\ v'), \\ e'_{\mathit{redux}} = (e_{\mathit{init}} \oplus e_{\mathit{before}}) \oplus e_{\mathit{after}},\ H'(\mathit{reduxVar}) = e'_{\mathit{redux}} \end{array}}{(H,S,r[\bar{e}] = v, FS) \to (H',S,;,FS)} \tag{7.104}$$

Relation updates within a `forall` are more complex. It's not possible to update incrementally; however, in an operational context it is possible just to re-evaluate the reduction after the `forall`. This is by no means efficient, but it will result in a correct concrete value. As for the incremental case, we assume that r' is the post-`forall` version of the relation, and H' is the post-`forall` version of the heap state.

$$\frac{\begin{array}{c} \mathit{reduxVar} = (\overline{T\ x}, P(\bar{x}), \mathit{trueVal}, \mathit{falseVal}),\ P' = \mathit{BIND}(\bar{i},\bar{x},\bar{e}), \\ e'_{\mathit{redux}} = \Sigma_{\bar{i}}\ (\mathrm{eval}(P',H') \rightarrow \mathit{trueVal}, \mathit{falseVal}),\ H'(\mathit{redux}) = e'_{\mathit{redux}} \end{array}}{(H,S,\mathbf{forall}(\overline{T\ x},P_{\mathit{forall}})\{\ldots\},\mathit{FS}) \rightarrow (H',S,;,\mathit{FS})} \tag{7.105}$$

7.2.2.6 Choose

`choose` is a statement that non-deterministically picks a set of values that satisfy its predicate, and evaluates a body in terms of that particular value vector. It is a requirement that the modeler only write `choose` statements where the predicate can, in fact, be true. This can be proven automatically as discussed in Chapter 8. These semantics assume that there will be some value vector that will satisfy the predicate.

$$\frac{\begin{array}{c} e' = \mathit{CLOSE}(e,H,S),\ \mathit{FREEVARS}(e') \subseteq \bar{x}, \\ \exists\ \overline{T\ v}\ :\ \mathrm{eval}(\{\bar{x} \mapsto \overline{(v,\ T)}\} \circ S, e') = \mathbf{true} \end{array}}{(H,S,\mathbf{choose}(\overline{T\ x};e)\{\overline{\mathit{stmt}}\},\mathit{FS}) \rightarrow (H,\{\bar{x} \mapsto \bar{v}\} \circ S, \overline{\mathit{stmt}}, \{\} \circ \mathit{FS})} \tag{7.106}$$

7.2.2.7 Ordering Rules

This section contains all the open frame rules that formalize the order of evaluation of parts of complex statements and expressions (for example, evaluating arguments to a function call in a left-to-right fashion). These rules use the • symbol to represent the term currently under evaluation.

sequence:

$$\frac{}{(H,S,\mathit{stmt}_1;\ \ldots\ ;\ \mathit{stmt}_n;,\mathit{FS}) \rightarrow (H,S,\mathit{stmt}_1;,(\mathit{stmt}_2;\ \ldots\ \mathit{stmt}_n;) \circ \mathit{FS})} \tag{7.107}$$

$$\frac{}{(H,S,e_1,\ldots,e_n,\mathit{FS}) \rightarrow (H,S,e_1,(e_2,\ldots,e_n) \circ \mathit{FS})} \tag{7.108}$$

return:

$$\frac{}{(H, MS \circ VS, \textbf{return } e;, FS) \rightarrow (H, MS \circ VS, e, (\textbf{return } \bullet;) \circ FS)} \tag{7.109}$$

conditionals:

$$\frac{}{\begin{array}{l}(H, MS \circ VS, \textbf{if}(e)\{\overline{stmt_1}\} \textbf{ else } \{\overline{stmt_2}\}, FS)\\ \rightarrow (H, MS \circ VS, e, \textbf{if}(\bullet)\{\overline{stmt_1}\} \textbf{ else } \{\overline{stmt_2}\}, FS)\end{array}} \tag{7.110}$$

field dereference:

$$\frac{}{(H, S, e.f, FS) \rightarrow (H, S, e, (\bullet.f) \circ FS)} \tag{7.111}$$

relation dereference:

$$\frac{}{(H, S, e_1[\overline{e_2}], FS) \rightarrow (H, S, e_1, (\bullet[\overline{e_2}]) \circ FS)} \tag{7.112}$$

$$\frac{}{(H, S, r[\overline{e_2}], FS) \rightarrow (H, S, \overline{e_2}, (r[\bullet]) \circ FS)} \tag{7.113}$$

casts:

$$\frac{}{(H, S, (T)e, FS) \rightarrow (H, S, e, ((T)\bullet) \circ FS)} \tag{7.114}$$

field writes:

$$\frac{}{(H, S, e_1.f = e_2, FS) \rightarrow (H, S, e_1, (\bullet.f = e_2) \circ FS)} \tag{7.115}$$

$$\frac{}{(H, S, o.f = e_2, FS) \rightarrow (H, S, e_2, (o.f = \bullet) \circ FS)} \tag{7.116}$$

relation update:

$$\frac{}{(H, S, e_1[\overline{e_2}] = e_3, FS) \rightarrow (H, S, e_1, (\bullet[\overline{e_2}] = e_3) \circ FS)} \tag{7.117}$$

$$\frac{}{(H, S, r[\overline{e_2}] = e_3, FS) \rightarrow (H, S, \overline{e_2}, (r[\bullet] = e_3) \circ FS)} \tag{7.118}$$

$$\frac{}{(H, S, r[\overline{v}] = e_3, FS) \rightarrow (H, S, e_3, (r[\overline{v}] = \bullet) \circ FS)} \tag{7.119}$$

local variable update:

$$\frac{}{(H,S,x=e,FS) \to (H,S,e,(x=\bullet)\circ FS)} \tag{7.120}$$

new:

$$\frac{}{(H,S,\mathbf{new}\,N<\cdots>(e_0,\overline{e}),FS) \to (H,S,\overline{e},(\mathbf{new}\,N<\cdots>(\bullet,\overline{e}))\circ FS)} \tag{7.121}$$

$$\frac{}{(H,S,\mathbf{new}\,N<\cdots>(\overline{v},e_n,\overline{e}),FS) \to (H,S,\overline{e},(\mathbf{new}\,N<\cdots>(\overline{v},\bullet,\overline{e}))\circ FS)} \tag{7.122}$$

super invocation:

$$\frac{}{(H,S,\mathbf{super}(\overline{e}),FS) \to (H,S,\overline{e},(\mathbf{super}(\bullet))\circ FS)} \tag{7.123}$$

method invocation:

$$\frac{}{(H,S,e.m(\overline{e'}),FS) \to (H,S,e,(\bullet.m(\overline{e'}))\circ FS)} \tag{7.124}$$

$$\frac{}{(H,S,o.m(\overline{e'}),FS) \to (H,S,\overline{e'},(o.m(\bullet))\circ FS)} \tag{7.125}$$

7.2.2.8 State Transition Rules

ACCLAM programs can specify a set of named states. There is an automatically supplied named state `init`; all other states are specified as the result after a series of statements have been executed against a known state. The semantics of the named state transitions require another environment *STATE* that's a mapping from a symbol to a heap and an initial stack (the stack contains the values for all the top-level variables). *STATE* is also needed for the top-level expression which may evaluate expressions in terms of different states, or two compare two states for equality.

$$\frac{\begin{array}{l} STATE(\sigma_1) = (H, VS),\ \sigma_2 \notin \mathrm{dom}(STATE), \\ (H, VS, \overline{stmt}, \{\} \circ FS) \to^* (H', VS', \{\}, FS), \\ STATE' = STATE[\sigma_2 \mapsto (H', VS')] \end{array}}{(STATE, \sigma_1\{\overline{stmt}\}\sigma_2, FS) \to (STATE', ;, FS)} \tag{7.126}$$

$$\frac{}{\begin{array}{l} (STATE, \sigma_{1,1}\{\overline{stmt}\}\sigma_{1,2} \ldots \sigma_{n,1}\{\overline{stmt}\}\sigma_{n,2}, FS) \\ \to (STATE, \sigma_{1,1}\{\overline{stmt}\}\sigma_{1,2}, (\sigma_{2,1}\{\overline{stmt}\}\sigma_{2,2} \ldots \sigma_{n,1}\{\overline{stmt}\}\sigma_{n,2}) \circ FS) \end{array}} \tag{7.127}$$

Equality Between States

State equality is the pairwise equality between entries in heaps and stacks. The following rules define equality between stacks, heaps and models.

State Equality

$$\frac{\sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2),\ H_1 \text{ EQHEAP } H_2,\ S_1 \text{ EQSTACK } S_2}{\sigma_1 = \sigma_2} \tag{7.128}$$

Stack Equality

$$\frac{\sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2),\ \text{vars} = \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2),\ \forall v \in \text{vars}\ :\ S_1(v) = S_2(v)}{\sigma_1 \text{ EQSTACK } \sigma_2} \tag{7.129}$$

Heap Equality

$$\frac{\begin{array}{l} \sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2), \\ \text{instances} = \mathrm{dom}(H_1) \cap \mathrm{dom}(H_2), \\ \forall m \in \text{instances}\ :\ H_1(m) \text{ EQINSTANCE } H_2(m) \end{array}}{\sigma_1 \text{ EQHEAP } \sigma_2} \tag{7.130}$$

Instance Equality:

$$\frac{\begin{array}{l}\sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2),\\ H_1(o) = o_1 = (N, \mathbb{F}_1),\ H_2(o) = o_2 = (N, \mathbb{F}_2),\\ o_1 \text{ EQSCALAR } o_2 \ \vee\ o_1 \text{ EQREF } o_2 \ \vee\ \ o_1 \text{ EQREL } o_2\end{array}}{o_1 \text{ EQINSTANCE } o_2} \tag{7.131}$$

For the following rules, assume the existence of three new metafunctions: *SCALARS*, *MODELS*, and *RELATIONS*, that are functions over the field mapping for an instance ($\mathbb{F}$). They return the set of field names that map to the appropriate kinds of fields (*SCALARS* returns all the scalar fields, *MODELS* all the model reference fields, and *RELATIONS* returns all the relation reference fields).

Equality of Scalar Instance Fields

$$\frac{\begin{array}{l}\sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2),\\ H_1(o) = o_1 = (N, \mathbb{F}_1),\ H_2(o) = o_2 = (N, \mathbb{F}_2),\\ \forall f \in \text{dom}(\mathbb{F}_1) \wedge f \in \mathit{SCALARS}(\mathbb{F}_1)\ :\ \mathbb{F}_1(f) = \mathbb{F}_2(f)\end{array}}{o_1 \text{ EQSCALAR } o_2} \tag{7.132}$$

Equality of Model References

$$\frac{\begin{array}{l}\sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2),\\ H_1(o) = o_1 = (N, \mathbb{F}_1),\ H_2(o) = o_2 = (N, \mathbb{F}_2),\\ \forall f \in \text{dom}(\mathbb{F}_2) \wedge f \in \text{dom}(\mathit{MODELS}(\mathbb{F}_1))\\ \quad :\ m_1 = \mathbb{F}_1(f) \wedge m_2 = \mathbb{F}_2(f) \wedge m_1 \text{ EQINSTANCE } m_2\end{array}}{o_1 \text{ EQREF } o_2} \tag{7.133}$$

Equality of Relations

$$\frac{\begin{array}{l}\sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2),\\ H_1(r) = r_1 = \lambda\bar{i}\ .\ \ldots,\ H_2(r) = r_2 = \lambda\bar{i}\ .\ \ldots,\\ H_1(o) = o_1 = (N, \mathbb{F}_1),\ H_2(o) = o_2 = (N, \mathbb{F}_2),\\ \mathbb{F}_1(r) = r_1, \mathbb{F}_2(r) = r_2, \forall\, \bar{x}\ :\ r_1(\bar{x}) = r_2(\bar{x}),\end{array}}{r_1 \text{ EQREL } r_2} \tag{7.134}$$

$$\frac{\begin{array}{l}\sigma_1 = (H_1, S_1),\ \sigma_2 = (H_2, S_2),\\ H_1(o) = o_1 = (N, \mathbb{F}_1),\ H_2(o) = o_2 = (N, \mathbb{F}_2),\\ \forall r \in \mathit{RELATIONS}(\mathbb{F}_1)\ :\ r_1 = \mathbb{F}_1(r) \wedge r_2 = \mathbb{F}_2(r) \wedge r_1 \text{ EQREL } r_2\end{array}}{o_1 \text{ EQREL } o_2} \tag{7.135}$$

7.3 Soundness

With a description of operational semantics and typing, we can prove that ACCLAM is a sound language. The approach is to define typing of an operational configuration piece-wise, and then to demonstrate that the operational transition rules preserve the typing. The typing rules in this section will usually be in terms of the model table (MT) and a configuration element (such as the heap).

7.3.1 Well-typed Heap

A well-typed heap is one in which all fields of all objects in the heap are within the same heap, and the heap and model table show the correct type for the heap objects.

Models Well-typed

$$\frac{H(o) = (N, \mathbb{F}),\ N \lhd T,\ N \text{ OK in MT}}{\text{MT}, H \vdash o\ :\ T} \tag{7.136}$$

Null

$$\frac{N \text{ OK in MT}}{\text{MT}, H \vdash \textbf{null}\ :\ N} \tag{7.137}$$

Field References are Valid

$$\frac{\begin{array}{c}H(o) = (N, \mathbb{F}),\ \mathrm{dom}(\mathbb{F}) = \mathrm{dom}(\mathrm{FIELDS}(N)) = \overline{f},\\ \forall f \in \mathrm{dom}(\mathbb{F})\ :\ (\mathrm{MT}, H \vdash \mathbb{F}(f)\ :\ \mathrm{FIELDS}(N)(f))\end{array}}{\mathrm{MT}, H \vdash o\ \mathrm{OK}} \tag{7.138}$$

Field Relation References are Valid

$$\frac{H(o) = (N, \mathbb{F}),\ \mathit{RELATIONS}(\mathbb{F}) = \{r_1, \ldots, r_n\}}{\mathrm{MT}, H \vdash o\ \mathrm{OK}} \tag{7.139}$$

Heap Well-typed A well-formed well-typed heap will be HEAP OK in terms of a model table.

$$\frac{\begin{array}{l}\mathrm{dom}(H) = \{o_1, \ldots, o_n\} \cup \{r_1, \ldots, r_m\},\\ \forall i \in \{1 \ldots n\}\ :\ \mathrm{MT},\ H \vdash o_i\ \mathrm{OK},\\ \forall i \in \{1 \ldots m\}\ :\ \mathrm{MT},\ H \vdash r_i\ \mathrm{OK},\end{array}}{\mathrm{MT} \vdash H\ \mathrm{HEAP\ OK}} \tag{7.140}$$

7.3.2 Well-typed Variable Stack

A well-typed variable stack must contain only valid or `null` values.

Variables Well-typed

$$\frac{S = \overline{x \mapsto (v, T)},\ \forall x \in S\ :\ \mathrm{MT},\ H \vdash x\ :\ T}{\mathrm{MT},\ H \vdash S\ \mathrm{OK}} \tag{7.141}$$

Empty Stack

$$\frac{\mathrm{MT}, H \vdash []\ \mathrm{OK},\ \mathrm{MT}, H \vdash S\ \mathrm{OK}}{\mathrm{MT}, H \vdash [] \circ S\ \mathrm{OK}} \tag{7.142}$$

Compound Stack

$$\frac{\mathrm{MT}, H \vdash \mathit{VS}\ \mathrm{OK},\ \mathrm{MT}, H \vdash \mathit{BS}\ \mathrm{OK}}{\mathrm{MT}, H \vdash\ (\mathit{BS} \circ \mathit{VS})\ \mathrm{OK}} \tag{7.143}$$

7.3.3 Well-typed frame stack

Typing a full configuration requires that we extend the basic type environment Γ to include information about heap and stack variables. We define the following extend function to augment Γ.

$$\begin{array}{ll}
\text{extend}(\Gamma,\{\},[]) & = \Gamma \\
\text{extend}(\Gamma,\{\},(\{\}\circ MS)\circ VS) & = \text{extend}(\Gamma,\{\},MS\circ VS) \\
\text{extend}(\Gamma,\{\},(BS[x\mapsto(v,\ T)]\circ MS)\circ VS) & = \begin{cases} \text{extend}(\Gamma,\{\},(BS\circ MS)\circ VS) & x\in\Gamma \\ \text{extend}(\Gamma[x\mapsto T],(BS\circ MS)\circ VS) & x\notin\Gamma \end{cases} \\
\text{extend}(\Gamma,H[o\mapsto(N,\mathbb{F})],S) & = \begin{cases} \text{extend}(\Gamma,H,S) & x\in\Gamma \\ \text{extend}(\Gamma[o\mapsto N],H,S) & x\notin\Gamma \end{cases}
\end{array} \tag{7.144}$$

To properly type a frame stack requires handling open and closed frames, all the statement varieties, scope changes and control flow altering statements (`return`).

7.3.3.1 Typing Substitution Holes

$$\frac{\Gamma\vdash o\ :\ N,\ \text{MT}\vdash\Gamma\,\text{OK}}{\text{MT},\Gamma\vdash o\ :\ N} \tag{7.145}$$

$$\frac{\Gamma\vdash\bullet\ :\ N,\ \text{MT}\vdash\Gamma\,\text{OK}}{\text{MT},\Gamma\vdash\bullet\ :\ N} \tag{7.146}$$

7.3.3.2 Case-by-case Typing of Statements

empty stack (empty stacks don’t change the types)

$$\frac{}{\text{MT},\ H,\ (BS\circ[])\circ[]\vdash[]\ :\ \tau\to\tau} \tag{7.147}$$

new scope

$$\frac{\text{MT},\ H,\ MS\circ VS\vdash FS\ :\ \textbf{void}\to\tau}{\text{MT},\ H,\ (BS\circ MS)\circ VS\vdash(\{\})\circ FS\ :\ \tau'\to\tau} \tag{7.148}$$

method with hole

$$\frac{\text{MT},\ H,\ VS\vdash FS\ :\ \tau\to\tau',\ \tau''\lhd\tau}{\text{MT},\ H,\ (BS\circ[])\circ VS\vdash(\textbf{return}\ \bullet;)\circ FS\ :\ \tau''\to\tau'} \tag{7.149}$$

local variable declaration

model and primitive types:

$$\frac{\begin{array}{l} L \in N \cup P, \\ \text{MT}, H, (BS[x \mapsto (init(L), L)] \circ MS) \circ VS \vdash FS \ : \ \textbf{void} \rightarrow \tau, \\ x \notin \text{dom}(BS \circ MS) \end{array}}{H, \ (BS \circ MS) \circ VS \vdash (L\,x;) \circ FS \ : \ \tau' \rightarrow \tau} \tag{7.150}$$

relation types:

$$\frac{\begin{array}{l} R' \in R, \\ \text{MT}, \ H, \ (BS[x \mapsto (e, R')] \circ MS) \circ VS \vdash FS \ : \ \textbf{void} \rightarrow \tau, \\ \text{MT}, \ H, \ (BS[x \mapsto (e, R')] \circ MS) \circ VS \vdash e \ : \ \tau', \\ x \notin \text{dom}(BS \circ MS) \end{array}}{H, \ (BS \circ MS) \circ VS \vdash (R'\,x;) \circ FS \ : \ \tau' \rightarrow \tau} \tag{7.151}$$

sequencing:

$$\frac{\text{MT}, \ H, \ S \ \vdash (stmt_1) \circ (\overline{stmt}) \circ FS \ : \ \tau \rightarrow \tau'}{\text{MT}, \ H, \ S \ \vdash (stmt_1; \overline{stmt}) \circ FS \ : \ \tau \rightarrow \tau'} \tag{7.152}$$

open stack frame: This rule is for all open frames not handled by the explicit rules above (non-return statements)

$$\frac{\begin{array}{l} \text{MT}, \text{extend}(\Gamma, H, S), \bullet \ : \ T \vdash OF \ : \ \tau, \\ H, \ S \vdash FS \ : \ \tau \rightarrow \tau' \end{array}}{\text{MT}, \ H, \ S \vdash OF \circ FS \ : \ T \rightarrow \tau'} \tag{7.153}$$

closed stack frames: This rule handles all closed frames not handled above (excluded are returns, local variable declarations, and scope entry/exit).

$$\frac{\text{MT}; \text{extend}(\Gamma, \ H, \ S) \vdash CF \ : \ \tau, \ H, \ S \vdash FS \ : \ \tau \rightarrow \tau'}{\text{MT}, \ H, S \vdash CF \circ FS \ : \ \tau'' \rightarrow \tau'} \tag{7.154}$$

7.3.4 Soundness

To prove soundness we need to prove that progress is guaranteed (e.g., there is no configuration where the program is stuck) and that all possible transitions preserve type. With that, we can prove that all possible transitions are sound in terms of the sub typing relationship $\triangleleft$.

$$\begin{aligned}
\text{terminals} \quad & = \textbf{return}\ v \mid \{\} \mid v \\
\text{non-terminals} \quad & = \overline{stmt} \mid \textbf{primop} \\
& \mid o.f \mid o.f = e \\
& \mid r[\overline{e}] \mid r[\overline{e}] = e \\
& \mid (U)e \mid o.m(\overline{e}) \mid \textbf{new}\ N(\overline{e}) \mid x = e \\
& \mid \textbf{if}(e)\{\overline{stmt}\}\ \textbf{else}\ \{\overline{stmt}\} \\
& \mid \textbf{forall}(\overline{T\ x};\ e)\{\overline{stmt}\} \\
& \mid \textbf{choose}(\overline{T\ x};\ e)\{\overline{stmt}\}
\end{aligned} \tag{7.155}$$

7.3.4.1 Progress

If the semantics ensure progress, then there is always a legal well-typed configuration reachable from any non-terminal configuration. Formally: if (H,S,F,FS) is non-terminal $\wedge$ MT $\vdash (H,S,F,FS)\ :\ T \implies \exists\ (H',S',F',FS')$ such that $(H,S,F,FS) \rightarrow (H',S',F',FS')$. The proof is a series of small proofs for each non-terminal statement type. The general format of each case is to state the prior configuration, then prove that there is a successor configuration, and a description of the successor configuration. In the interest of brevity, the configurations are usually just the current frame being evaluated F, and the successor frame $F'$and possibly frame stack FS' (i.e., only those things that change from one configuration to the next will be mentioned).

A note about statements within forall bodies: the only statement type that has a different posterior configuration within a `forall` body is the relation update. Non-relation updates have the requirement that they are idempotent, and so for any value of the `forall` variables the right hand side must be identical. Local variable declarations will have been closed over and will have been incorporated as expressions in updates.

primop: $F = \mathbf{primop}(\overline{v})$

By well typing, $\Gamma \vdash F : T$, then by rule 7.93, F reduces to v and $\Gamma \vdash v : T$ therefore $F' = v$.

sequence/skip $(F =;)$ By rule 7.84, there exists another frame F'.

conditional $F = \mathbf{if}(v)\{\overline{stmt_1}\}$ **else** $\{\overline{stmt_2}\}$

By well-typing, $\Gamma \vdash v :$ boolean. Therefore if $v = \mathbf{true}$, $F' = \overline{stmt_1}$; otherwise $F' = \overline{stmt_2}$.

field dereference $F = e.f$

case 1 $e = \mathbf{null}$, reduces by rule 7.74, $F' = \mathbf{return\ exc}$

case 2 $e = o$, reduces by rule 7.73, $F' = v$

case 3 $e \neq o$, reduces by rule 7.111, $F' = e, FS' = (\bullet.f) \circ FS$

field assignment 1 $F = (e.f = e')$

Reduces by rule 7.115, $F' = e, FS' = (\bullet.f = e') \circ FS$.

field assignment 2 $F = (v.f = e')$

case 1 $v = \mathbf{null}$ reduces by rule 7.74, $F' = \mathbf{return\ exc}$

case 2 $v = o$ reduces by rule 7.116, $F' = e', FS' = (v.f = \bullet) \circ FS$

field assignment 3 $F = v.f = v'$

case 1 $v = \mathbf{null}$ reduces by rule 7.74, $F' = \mathbf{return\ exc}$

case 2 $v = o$ reduces by rule 7.75, $F' =;$

relation read

case 1 $F = e_1[\overline{e}]$, reduces by rule 7.112. $F' = e_1,\ FS' = (\bullet[\overline{e_2}]) \circ FS$

case 2 $F = r[\overline{e}]$, reduces by rule 7.113. $F' = \overline{e},\ FS' = (r[\bullet]) \circ FS$

case 3 $F = r[\overline{v}]$, reduces by rule 7.61, $F' = v'$

relation assignment

case 1 $F = e_1[\overline{e_2}] = e_3$, reduces by rule 7.117, $F' = e_1, FS' = (\bullet[\overline{e_2}] = e_3) \circ FS$

case 2 $F = r[\overline{e_2}] = e_3$, reduces by rule 7.118, $F' = \overline{e_2}, FS' = (r[\bullet] = e_3) \circ FS$

case 3 $F = r[\overline{v_1}] = e_3$, reduces by rule 7.119, $F' = e_3, FS' = (r[\overline{v_1}] = \bullet) \circ FS$

case 4 $F = r[\overline{v_1}] = v_2$, reduces by rule 7.77, $F' = ;$

case 5 **@forallVars** $\in \mathrm{dom}(S), F = r[\overline{e_2}] = e_3$

$H(r) = r_{old} = \lambda \bar{i}. \ldots,$

$r' = \lambda \bar{i}.(BIND(\bar{i},\ \bar{x},\ \overline{e_2},\ \mathrm{eval}(S,\ \textbf{@forallPred})) \rightarrow e_3,\ r_{old}(\bar{i}))$

This is a relation update within a forall, which reduces by rule 7.103,

$F' =;,\ H' = H[r \mapsto r']$

cast

case 1 $F = (U)e$. Reduces by rule 7.114, $F' = e,\ FS' = ((U)\bullet) \circ FS$.

case 2 $F = (U)v$. Reduces by rule 7.78 or 7.79. $F' = v$

method invocation

case 1 $F = e_1.m(\overline{e_2})$, reduces by rule 7.124, $F' = e_1,\ FS' = (\bullet.m(\overline{e_2})) \circ FS$

case 2 $F = o.m(\overline{e_2})$, reduces by rule 7.125, $F' = \overline{e_2},\ FS' = (o.m(\bullet)) \circ FS$

case 3 $F = o.m(\overline{v})$, reduces by rule 7.81, $S' = MS \circ S,\ F' = \{\overline{stmt}\}$

block statement $F = \{\overline{stmt}\}$

Because F is well typed, we know that $S = MS \circ VS$, and therefore this configuration reduces by rule 7.66, $F' = \overline{stmt},\ FS' = \{\} \circ FS, S' = ([] \circ MS) \circ VS$

local variable read

$F = x$, by well-typing $x \in \mathrm{dom}(S)$, therefore this configuration is reducible by rule 7.60. $F' = v$.

local variable update

case 1 $F = (x = e)$, reduces by rule 7.120, $F' = e,\ FS' = (x = \bullet) \circ FS$

case 2 $F = (x = v)$, reduces by rule 7.62, $F' = ;$

object creation

case 1 $F = \textbf{new}\ T(\overline{e})$, reduces by rule 7.121, $F' = \overline{e},\ FS' = \textbf{new}\ T(\bullet) \circ FS$

case 2 $F = \textbf{new}\ T(\overline{v})$, reduces by rule 7.80, $F' = \overline{stmt},\ FS' = (\textbf{return}\ o;) \circ FS$,

$H' = H[o \mapsto (T, \mathbb{F})]$

forall

$F = \textbf{forall}(\overline{T\ x}; e)\{\overline{stmt}\}$, reduces by rule 7.100, $S' = (\{\textbf{@forallVars} \mapsto \overline{(x,\ T)},$ $\textbf{@forallPred} \mapsto e'\} \circ MS) \circ S, F' = \overline{stmt}, FS' = \{\} \circ FS$

choose

$F = \mathrm{choose}(\overline{T\ x}; e)\{\overline{stmt}\}$, reduces by rule 7.106 $F' = \overline{stmt}, FS' = \{\} \circ FS$

block exit

$F = \{\}$, reduces by rule 7.67. By the well-typing of the prior configuration, $S = (BS \circ MS) \circ VS$. $F' = ;, S' = MS \circ VS$

return

case 1 $F = \textbf{return}\ e$, reduces by rule 7.109, $F' = e, FS' = (\textbf{return}\ \bullet) \circ FS$

case 2 $F = \textbf{return}\ v, FS = \{\} \circ FS'$, this reduces by rules 7.68, 7.69. $F' = v$

case 3 $F = \textbf{return}\ v, FS = F'' \circ FS'', F'' \neq \{\}$, this reduces by rule 7.70. $F' = F,\ FS' = FS''$

7.3.4.2 Type Preservation

The formal requirements for preserving well-typing are, for typed frames (expressions):

$$\begin{aligned}&\text{If MT, } \Gamma \vdash (H,S,F,FS) \ : \ T \wedge (H,S,F,FS) \rightarrow (H',S',F',FS') \\ &\text{then } \exists\, T' \text{ such that MT, } \Gamma \vdash (H',S',F',FS') \ : \ T'\end{aligned} \tag{7.156}$$

and for untyped (statement) frames:

$$\begin{aligned}&\text{If MT, } \Gamma \vdash (H,S,F,FS) \text{ OK} \wedge (H,S,F,FS) \rightarrow (H',S',F',FS') \\ &\text{ then MT, } \Gamma \vdash (H',S',F',FS') \text{ OK}\end{aligned} \tag{7.157}$$

Any typed frame that obeys rule 7.156 is also an OK frame. Therefore 7.156 is a subset of 7.157.

For ACCLAM I have adopted the approach used by Middleweight Java, for which many of the type-preservation rules apply with little or no change. One difference is that in MJ, the statements have type (most of them have type `void`). The ACCLAM semantics do not assign a type to statements; rather they are judged as being OK. For most statements, there is no real difference whether they type as `void` or are OK. This section will consist of the cases where there is a difference.

Covariant Typing Lemma A useful lemma used by MJ is the covariant typing lemma, which states: Given an open frame *OF*

$$\Gamma[\bullet \ : \ T'] \vdash OF \ : \ T_1 \implies \forall T_2 \ : \ T_2 \lhd T' \implies \exists\, T_3 \ : \ (\Gamma[\bullet \ : \ T_2] \vdash OF \ : \ T_3) \wedge (T_3 \lhd T_1) \tag{7.158}$$

This lemma greatly reduces the number of cases that need to be examined when a sub typing relationship is present when typing open frames. Rather than having to prove four cases for each open frame (base type and sub-type for each side of the →), the covariant typing lemma lets us prove one. For ACCLAM, we just need to extend the lemma with cases for relation operations. After this, most of the MJ type-preservation rules apply to ACCLAM with the extended covariant typing lemma.

relation read ($OF = r[\bullet]$)

Since $T_3 \lhd T_1$, if $\Gamma \vdash r[\bullet] \ : \ T_3 \wedge T_3 \lhd T_1$ then $\Gamma \vdash r[\bullet] \ : \ T_1$. By substituting into the typing rule for relation read, we get:

$$\frac{\Gamma \vdash r \ : \ T_1[\overline{T'}],\ \Gamma \vdash \bullet \ : \ \overline{T_2},\ \overline{T_2} \lhd \overline{T'}}{\Gamma \vdash r[\bullet] \ : \ T_1} \tag{7.159}$$

relation update ($OF = r[\overline{e}] = \bullet$)

$$\frac{\begin{array}{c}\Gamma \vdash r \ : \ T_1[\overline{T'}],\ \Gamma \vdash \overline{e} \ : \ \overline{T_2},\\ \Gamma \vdash \bullet \ : \ T_3,\ \overline{T_2} \lhd \overline{T'},\ T_3 \lhd T_1\end{array}}{r[\overline{e}] = \bullet \text{ OK}} \tag{7.160}$$

Now we will examine the cases where we have to extend MJ's type preservation rules to cover ACCLAM's semantics.

non void return

assume: $\Gamma \vdash (H, MS \circ VS, \textbf{return } v, FS) \ : \ T, H \text{ OK},\ MS \circ VS \text{ OK},\ \text{FS OK}$

prove: $\Gamma \vdash (H, VS, v, FS) \ : \ T$

$MS \circ VS \rightarrow$ VS OK by well typing of stacks, which leaves the type of v. By typing rule 7.41, if $\Gamma \vdash v \ : \ T'$, then $T' \lhd T$. Therefore $\Gamma \vdash (H, VS, v, FS) \ : \ T$.

void return

assume: $\Gamma \vdash (H, MS \circ VS, \textbf{return}, FS) \ : \ \textbf{void},\ H \text{ OK},\ MS \circ VS \text{ OK}, \text{FS OK}$

prove: $\Gamma \vdash (H, VS, ;, FS) \ : \ \textbf{void}$

trivially true as $\Gamma \vdash ; \ : \ \textbf{void}$

exceptions

assume: $\Gamma \vdash (H, MS \circ VS, \textbf{return exc } v, FS) \ : \ \textbf{Throwable}, H \text{ OK}, MS \circ VS \text{ OK}, \text{FS OK}$

prove: $\Gamma \vdash (H, VS, \textbf{exc } v, FS) \ : \ \textbf{Throwable}$

By typing rule 7.27, $\Gamma \vdash \text{exc } v \ : \ \textbf{Throwable}$, therefore $\Gamma \vdash (H, VS, \textbf{exc } v, FS) \ : \ \textbf{Throwable}$.

relation read

assume: $\Gamma \vdash (H, VS, r[\overline{v}], FS) \ : \ T, \ \text{eval}(r, H) = \lambda \overline{x}.\ f(\overline{x}), \ f(\overline{v}) = v'$

prove: $\Gamma \vdash (H, VS, v', FS) \ : \ T$

By rule 7.26, $\Gamma \vdash r \ : \ T[\overline{T'}]$ and therefore $\Gamma \vdash v' \ : \ T$, so therefore $\Gamma \vdash (H, VS, v', FS) \ : \ T$.

relation update

assume: $\Gamma \vdash (H, S, r[\overline{v}] = v', FS)$ OK, $H' = H[r \mapsto r']$, $(\text{MT}, \ \Gamma) \vdash H$ OK

prove: $\Gamma \vdash (H', S, ;, FS)$ OK

By typing rule, 7.36, $\Gamma \vdash r \ : \ T'[\overline{T''}]$, $\Gamma \vdash \overline{v} \ : \ \overline{T''}$, $\Gamma \vdash v' \ : \ T'$, therefore $\Gamma \vdash r' \ : \ T'[\overline{T''}]$, therefore $\Gamma \vdash H'$ OK.

relation declaration

assume: $\Gamma \vdash (H, (BS \circ MS) \circ VS, U[\overline{U}] \ r;, FS)$ OK

prove: $\Gamma \vdash (H', (BS' \circ MS) \circ VS, ;, FS)$ OK

By typing rule 7.65, $\Gamma \vdash r \ : U[\overline{U}]$ and $r \in \text{dom}(H')$, therefore $\text{MT}, \Gamma \vdash H'$ OK.

By typing rule 7.65, $BS' = BS[r \mapsto H'(r)]$, therefore $\text{MT}, \Gamma, H' \vdash BS'$ OK, therefore $\Gamma \vdash (H', (BS' \circ MS) \circ VS, ;, FS)$ OK.

7.3.4.3 Soundness

By the progress proof, $(H, S, F, FS) \rightarrow^* (H', S', F', FS')$ where (H', S', F', FS') is terminal. By the type preservation proof, if (H, S, F, FS) OK then (H', S', F', FS') OK. Therefore all well-typed methods and well-typed models in ACCLAM are sound.

A well-typed ACCLAM program is sound if its model table's models are sound, if the named state transitions are sound, and the top-level expression is sound. A well-typed ACCLAM program can't use non-well-typed statements in the named state transitions, therefore they are sound. A well-typed ACCLAM program must also have a well-typed top-level expression, therefore that is sound. Therefore a well-typed ACCLAM program is sound.

7.4 Circuit Semantics

ACCLAM is a modeling language. An operational description was illustrative and necessary for a soundness proof. However, the actual ACCLAM processing tool converts models into a pure expression-based circuit that's appropriate for conversion into a SAT problem. This circuit form is so called because the inputs to the circuit are the initial state and function arguments, and the outputs are the final state and return values. The circuits themselves are expressions that transform the initial state values into final state values. Since there is no mutation in the circuit form, these expressions can be converted into SAT clauses in a straight-forward fashion. This section describes that transformation using the same kinds of formalism as the previous semantic descriptions.

7.4.1 Circuit Constructs

There are a couple of new value types that make sense in the circuit context that will be introduced here.

Circuit Variables: A circuit variable is a value of a specific type that is an input to the circuit. A circuit variable's value is unknown and may assume any legal value for its type.

Circuit Values: A circuit value is an encoding of an ACCLAM type into a set of bits, and is analogous to wires in a more conventional circuit.

Circuit Constant: A circuit constant is a circuit value that is hard-wired to produce a constant value.

The semantics will be expressed in terms of a series of transforms that will accept some portion of an ACCLAM program and produce a pair of an expression and a set of circuit variables.

These transforms will operate on instances with named scoped state, so we'll need a Values environment that will map a name to an expression. For convenience, we'll also maintain a heap environment, H that will also map names to expressions.

7.4.2 Circuit Expressions

This is the abstract grammar of the valid output of the circuit transformation functions.

$$\begin{array}{lcl} \mathit{expr} & = & \mathit{CircuitValue} \\ & | & \mathit{lambda} \\ & | & \mathit{application} \\ & | & \mathit{multiplexor} \\ & | & \mathit{op}(\overline{\mathit{expr}}) \\ \mathit{lambda} & = & \lambda \overline{T v} \;.\; \mathit{expr} \\ \mathit{application} & = & \mathrm{lambda}(\overline{\mathit{expr}}) \\ \mathit{multiplexer} & = & \mathit{expr} \rightarrow \mathit{expr},\; \mathit{expr} \end{array} \tag{7.161}$$

The set of expressions is values, lambda forms, primitive operations, and multiplexors. The final expressions will not contain any unapplied lambda forms; however, intermediate results will. Primitive operations is a catch-all for unary and binary operations on primitive values as well as == for reference values. A multiplexor is just a circuit-friendly form of a conditional.

7.4.2.1 Lambda Application Rule

Lambda application is done with substitution.

$$\frac{f = \lambda\ \overline{T\ x}\ .\ e,\ \Gamma \vdash \overline{e'}\ :\ \overline{T'},\ \overline{T'} \triangleleft \overline{T}}{f(\overline{e'}) = e[\overline{e'}\ /\ \overline{v}]} \tag{7.162}$$

7.4.3 Transforms

The transform functions operate on a configuration tuple similarly to the operational semantics. The configuration may contain a stack mapping, a heap mapping, a term under evaluation, a continuation, and a control flow expression. The control flow expression is the way the circuit transformation handles control flow, by generating a boolean expression that is true if control would have reached the term and false otherwise.

There is a transform function for every major type of syntactic element (values and variables, expressions, statements and the program as a whole). The functions were defined separately (rather than as one big function) to present the transformations in an incremental fashion. The transform functions may invoke one another.

$$
\begin{array}{ll}
\textbf{TransformV} & : \mathit{value} \rightarrow \mathit{circuit\ value} \\
\textbf{TransformC} & : \mathit{constant} \rightarrow \mathit{circuit\ constant} \\
\textbf{TransformE} & : \mathit{expr} * \mathit{stack} * \mathit{heap} * \mathit{control\text{-}flow\ expr} \rightarrow \mathit{circuit\ expr} * \mathit{heap} \\
\textbf{TransformS} & : \mathit{stmt} * \mathit{stack} * \mathit{heap} * \mathit{control\text{-}flow\ expr} * \overline{\mathit{stmt}} \\
 & \rightarrow \mathit{expr} * \mathit{stack} * \mathit{heap} * \mathit{control\text{-}flow\ expr} \\
\textbf{TransformP} & : \mathit{program} * \mathit{state\ map} \rightarrow \mathit{expr}
\end{array}
$$

7.4.4 State in the Circuit

The circuit transforms will be written in terms of a heap and stack state. However, they are just mappings from names to expressions. The circuit expressions produced by the transforms must produce all the same values that the statements/expressions would under a more conventional operational approach. The way this is accomplished is by conditionalizing state by the control-flow expression. Updates do not overwrite/shadow concrete values. In circuit form, an update wraps the old expression with a conditional expression. So the new expression is something like: $e_{\mathit{control\text{-}flow}} \rightarrow e_{\mathit{new}}, e_{\mathit{old}}$. The circuit interpretation of this is that all possible state expressions are ‘evaluated’ and fed into a multiplexor. The control flow expression is also evaluated and is used as the selector input to the multiplexor. A state element that is updated multiple times becomes a chain of multiplexors+inputs.[1]

7.4.5 Utility Functions

Lookup Lookup is used to look up the value that a name maps to in a stack.

[1] Another formulation is to combine the control flow expressions and feed their multi-bit output into a large multiplexor.

$$\frac{\{name \mapsto val\} \in S}{\text{lookup}(S \circ S', name) = val} \tag{7.163}$$

$$\frac{\{name \mapsto val\} \notin S}{\text{lookup}(S \circ S', name) = \text{lookup}(S', name)} \tag{7.164}$$

Stack Update Update replaces the mapping for a name at the appropriate nesting level for the stack.

$$\frac{BS(v) = e}{\text{update}(v, x, c, BS \circ S) = BS[v \mapsto (c \to x, e)] \circ S} \tag{7.165}$$

$$\frac{v \notin \text{dom}(BS)}{\text{update}(v, x, c, BS \circ S) = BS \circ \text{update}(v, x, c, S)} \tag{7.166}$$

Heap Update Definitions

Instance Update

$$\frac{H(o) = (N, \mathbb{F})}{\text{update}(o, (N, \mathbb{F}'), c, H) = H[o \mapsto (c \to (N, \mathbb{F}'), (N, \mathbb{F}))]} \tag{7.167}$$

Relation Update

$$\frac{H(r) = e}{\text{update}(r, e', c, H) = H[r \mapsto (c \to e', e)]} \tag{7.168}$$

$$\frac{r \in \text{dom}(S)}{\text{update}(r, e', c, S, H) = \text{update}(r, e', c, S), H} \tag{7.169}$$

$$\frac{r \notin \text{dom}(S)}{\text{update}(r, e', c, S, H) = S, \text{update}(r, e', c, H)} \tag{7.170}$$

7.4.6 Value Transform

Creates a new mapping for the variable that refers to an expression that produces the initial value for the variable's type.

$$\frac{}{\textbf{TransformV}(Tx) = (x, \textbf{TransformC}(\text{init}(T)))} \tag{7.171}$$

7.4.7 Expression Transform

Expression transform functions produce an expression and a heap map. In these rules, c is the control-flow expression.

Constants

$$\frac{}{\textbf{TransformE}(k,S,H,c) = \textbf{TransformC}(k),H} \tag{7.172}$$

Variables

$$\frac{}{\textbf{TransformE}(x,S,H,c) = \text{lookup}(S,c),H} \tag{7.173}$$

Field Access

$$\frac{\begin{array}{c}\textbf{TransformE}(e,S,H,c) = o,H', \\ o = (N,\mathbb{F}),\ e' = \mathbb{F}(f)\end{array}}{\textbf{TransformE}(e.f,S,H,c) = e',H'} \tag{7.174}$$

Cast

$$\frac{\Gamma \vdash e\ :\ T, T \lhd N}{\textbf{TransformE}((N)e,S,H,c) = \textbf{TransformE}(e,S,H,c)} \tag{7.175}$$

Good downcast

$$\frac{\begin{array}{c}\Gamma \vdash e\ :\ T, N \lhd T, \\ e \text{ instanceof } N', N' \lhd N\end{array}}{\textbf{TransformE}((N)e,S,H,c) = \textbf{TransformE}(e,S,H,c)} \tag{7.176}$$

Bad downcast

$$\frac{\Gamma \vdash e\ :\ T,\ N \lhd T,\ e \text{ instanceof } N', \neg(N' \lhd N)}{\begin{array}{l}\textbf{TransformE}((N)e,S,H,c) \\ = \textbf{TransformE}(\textbf{return exc } \texttt{ClassCastException},S,H,c)\end{array}} \tag{7.177}$$

‘Stupid’ cast

$$\frac{\Gamma \vdash e\ :\ T,\ \neg(T \lhd N),\ \neg(N \lhd T)}{\begin{array}{l}\textbf{TransformE}((N)e,S,H,c) \\ = \textbf{TransformS}(\textbf{return exc } \texttt{ClassCastException},S,H,c)\end{array}} \tag{7.178}$$

Primitive cast

Java defines rules for transforming primitive expressions from one primitive type to another. Those rules are represented by the *CONVERT* function (*CONVERT* : *type* ∗ *type* ∗ *e* → *e*).

$$\frac{\Gamma \vdash e\ :\ P',\ e', H' = \mathbf{TransformE}(e, S, H, c)}{\mathbf{TransformE}((P)e, S, H, c) = \mathit{CONVERT}(P', P, e'), H'} \tag{7.179}$$

Relation Dereference

$$\frac{\begin{array}{l}\mathbf{TransformE}(e, S, H, c) = r, H',\\ r = \lambda\ \bar{i}\ .\ e_r,\\ \overline{e''}, H'' = \mathbf{TransformE}(\overline{e'}, S, H', c)\end{array}}{\mathbf{TransformE}(e[\overline{e'}], S, H, c) = \mathbf{TransformE}(e_r[\overline{e''}/\bar{i}], S, H'', c), H''} \tag{7.180}$$

Primitive Operations

For primitive operations (e.g., integer addition), we assume there's a function PRIMOP that produces a circuit-equivalent version of the ACCLAM primitive operation. For example, PRIMOP$(+, \mathrm{int}_1, \mathrm{int}_2))$ could produce a 32-bit wide ripple-carry adder circuit. It could also produce a fancier carry-lookahead circuit. Rather than get bogged down in specifying exact circuits for each of the well-specified Java primitive operations, we assume that there is a straightforward conversion from a primitive operation to a circuit operating on circuit values and leave the specific circuit used as an implementation decision.

$$\frac{\Gamma \vdash \bar{e}\ :\ P,\ \mathbf{TransformE}(\bar{e}, S, H, c) = \overline{e'}, H'}{\mathbf{TransformE}(\mathbf{primop}(\bar{e}), S, H, c) = \mathrm{PRIMOP}(\mathbf{primop}, \overline{e'}), H'} \tag{7.181}$$

Method Invocation

Method invocation is not a valid circuit expression, therefore the transform needs to inline the method definition by producing an expression that generates the possible return values for the method. For this we'll use the statement transform applied to the body of the method being invoked. This transformation assumes that exceptions, if thrown, are not being thrown up multiple levels of function invocation. At the moment ACCLAM has no support for specifying full `try-catch` semantics, therefore an exceptional return is a return of an exception value.

Note that the stack value that captures the return expression (@**return**) is explicitly set to `void`. All non-void methods (because they are well-formed) will update this value before returning, so this is safe. This rule also assumes that the type of the instance as stored in the heap (N) is the most specific type. This means that the value from METHOD will be the most specific method defined for that type. Therefore, this rule implements Java-style `invokevirtual` method invocation.

$$\frac{\begin{array}{l}\textbf{TransformE}(\bar{e},S,H,c) = \overline{e'},H, \\ H(o) = (N,\mathbb{F}),\ \text{METHOD}(N)(m) = (\overline{T\ x},\overline{stmt}), \\ MS = \{\textbf{this} \mapsto o, \textbf{@return} \mapsto \textbf{void}, \bar{x} \mapsto \overline{e'}\}, \\ \textbf{TransformS}(;,S',H,c,\overline{stmt} \circ \{\}) = e_{ret},S'',H',c' \\ S' = MS \circ S\end{array}}{\textbf{TransformE}(o.m(\bar{e},S,H,c) = e_{ret},H'} \tag{7.182}$$

New Instance Creation Creating a new instance means producing all the expressions for the initially constructed object, while also creating a new top level circuit variable. This is necessary because object construction is extending the state, and since the circuit form is mutation-free, that extension just means that the new instance was always part of the variable set, it just wasn't referenced in any expression until this point in the circuit.

$$\frac{\begin{array}{l}\textbf{TransformV}(N) = (N,\mathbb{F}),\ o \notin \text{dom}(H),\ H^{+} = H[o \mapsto (N,\mathbb{F})], \\ \textbf{TransformE}(\bar{e},S,H^{+},c) = \overline{e'},H',\ \Gamma \vdash \bar{e}\ :\ \bar{T}, \\ \text{INIT}(N)(\bar{T}) = (\overline{T\ x},\overline{stmt}),\ MS = \{\textbf{this} \mapsto o, \bar{x} \mapsto \overline{e''}\}, \\ \textbf{TransformS}(;,MS \circ S,H',c,\overline{stmt} \circ \{\}) = \textbf{void},S'',H'',c''\end{array}}{\textbf{TransformE}(\textbf{new}\ N(\bar{e}),S,H,c) = H''(o),H''} \tag{7.183}$$

7.4.8 Statement Transform

The statement transform takes a statement, a stack, a heap, a control flow expression, and a statement continuation and produces an expression, a stack, and a heap. As in the expression transform, c will stand for the boolean control-flow expression. K will stand for the statement continuation (analogous to FS from the operational configurations).

expression

$$\frac{}{\mathbf{TransformS}(e;,S,H,c,K) = \mathbf{TransformS}(;,S,H,c,K)} \tag{7.184}$$

sequence

$$\frac{}{\mathbf{TransformS}(;,S,H,c,stmt \circ K) = \mathbf{TransformS}(stmt,S,H,c,K)} \tag{7.185}$$

Exit Scope (scope entry will be handled for each scope-creating statement)

$$\frac{}{\mathbf{TransformS}(\{\},BS \circ S,H,c,K) = \mathbf{TransformS}(;,S,H,c,K)} \tag{7.186}$$

Exit Method Scope

$$\frac{}{\mathbf{TransformS}(;,MS \circ S,H,c,\{\}) = BS(@\mathbf{return}),S,H} \tag{7.187}$$

Local Variable Assignment

$$\frac{e',\ H' = \mathbf{TransformE}(e,S,H,c),\ S' = \text{update}(x,e',c,S)}{\mathbf{TransformS}(x = e;,S,H,c,K) = \mathbf{TransformS}(;,S',H',c,K)} \tag{7.188}$$

Heap Variable Assignment

$$\frac{\begin{array}{l} \mathbf{TransformE}(e,S,H,c) = e',H', \\ H(o) = (N,\mathbb{F}),\ \mathbb{F}' = \mathbb{F}[f \mapsto e'], \\ H'' = \text{update}(o,(N,\mathbb{F}'),c,H) \end{array}}{\mathbf{TransformS}(o.f = e;,S,H,c,K) = \mathbf{TransformS}(;,S,H'',c,K)} \tag{7.189}$$

Local Variable Declaration For circuit semantics, the declaration `T x = e;` is broken down into a declaration `T x;`, followed immediately by an assignment `x = e;`

$$\frac{\mathbf{TransformV}(T) = v,\ S' = S[x \mapsto v]}{\mathbf{TransformS}(T\ x, S, H, c, K) = \mathbf{TransformS}(;, S', H, c, K)} \tag{7.190}$$

$$\frac{\mathbf{TransformV}(T) = v,\ BS' = BS[x \mapsto v]}{\mathbf{TransformS}(T\ x, BS \circ S, H, c, K) = \mathbf{TransformS}(;, BS' \circ S, H, c, K)} \tag{7.191}$$

Normal Relation Assignment (not in a `forall`)

$$\frac{\begin{array}{l}\mathbf{TransformE}(e, S, H, c) = r, H', \\ \mathbf{TransformE}(\overline{e'}, S, H', c) = \overline{e''}, H'', \\ \mathbf{TransformE}(e^{+}, S, H'', c) = e^{++}, H^{++}, \\ r' = \lambda\ \bar{i}.(\bar{i} = \overline{e''}) \rightarrow e^{++}, r(\bar{i}), \\ S''', H''' = \text{update}(r, r', c, S, H^{++})\end{array}}{\mathbf{TransformS}(e[\overline{e'}] = e^{+}, S, H, c, K) = \mathbf{TransformS}(;, S''', H''', c, K)} \tag{7.192}$$

Normal Reduction Update

Reduction updates are performed implicitly when the relation they depend on is updated. Assuming the relation update rule 7.192, we extend it with the following (assuming that the reduction is defined as: int redux = $\mathbf{count}(\overline{T\ x}; P(\bar{x}); \mathit{trueVal}; \mathit{falseVal})$

$$\frac{\begin{array}{l}P' = \mathit{BIND}(\bar{i}, \bar{x}, \overline{e'}), \\ \mathbf{TransformE}(P', S, H, c) = P_{before}, H \\ \mathbf{TransformE}(P', S''', H''', c) = P_{after}, H''' \\ H(\text{redux}) = e_{old}, \\ \text{redux}' = (e_{old} + (P_{before} \rightarrow -\mathit{trueVal},\ -\mathit{falseVal})) + (P_{after} \rightarrow \mathit{trueVal},\ \mathit{falseVal}), \\ \text{update}(\text{redux}, \text{redux}', c, S''', H''') = S^{r}, H^{r}\end{array}}{\mathbf{TransformS}(e[\overline{e'}] = e^{+}, S, H, c, K) = \mathbf{TransformS}(;, S^{r}, H^{r}, c, K)} \tag{7.193}$$

This is not completely correct. $H(\text{redux})$ is incorrect: since reductions are part of a model's state, they need to be looked up indirectly through the model's reference. Doing that precisely in the rule would have complicated it without adding anything to the discussion of reduction value generation. For the sake of brevity, we assume that the compiler that is generating this code will remember the model that the relation update was associated with, and that $H(\text{redux})$ is a shorthand for the reduction's value in the heap.

return

The normal return case (the return is at the end of a block).

$$\frac{\begin{array}{l}\mathbf{TransformE}(e, MS \circ S, H, c) = e', H',\\ MS' = \text{update}(MS, \mathbf{@return}, (c \to e', MS(\mathbf{@return})))\end{array}}{\begin{array}{l}\mathbf{TransformS}(\mathbf{return}\ e, MS \circ S, H, c, \{\} \circ K)\\ = \mathbf{TransformS}(;, MS' \circ S, H', c, \{\} \circ K)\end{array}} \tag{7.194}$$

The case where the return is in the middle of a block.

$$\frac{\begin{array}{l}\mathbf{TransformS}(\mathit{stmt}, S, H, \mathbf{false}, \{\}) = \mathbf{void}, S', H', c',\\ \mathit{stmt} \neq \{\}\end{array}}{\mathbf{TransformS}(\mathbf{return}\ e, S, H, c, \mathit{stmt} \circ K) = \mathbf{TransformS}(\mathbf{return}\ e, S', H', c, K)} \tag{7.195}$$

conditional

$$\frac{\begin{array}{l}\mathbf{TransformE}(b, S, H, c) = b', H',\\ \mathbf{TransformS}(;, S, H', b' \wedge c, \overline{\mathit{stmt}_{then}} \circ \{\}) = e_1, S_{then}, H_{then}, c_{then},\\ \mathbf{TransformS}(;, S, H', (\neg b') \wedge c, \overline{\mathit{stmt}_{else}} \circ \{\}) = e_2, S_{else}, H_{else}, c_{else},\\ \forall\, x\ :\ S'(x) = b' \to S_{then}(x), S_{else}(x),\\ \forall\, x\ :\ H''(x) = b' \to H_{then}(x), H_{else}(x)\end{array}}{\begin{array}{l}\mathbf{TransformS}(\mathbf{if}\ (b)\{\overline{\mathit{stmt}_{then}}\}\ \mathbf{else}\ \{\overline{\mathit{stmt}_{else}}\}, S, H, c, K)\\ = \mathbf{TransformS}(;, S', H'', c_{then} \vee c_{else}, K)\end{array}} \tag{7.196}$$

forall Much of the work done for the operational description can be re-used here. We still need to *CLOSE* the `forall` predicate, and *BIND* any relation index variables. Since both *CLOSE* and *BIND* produce expressions without method invocation, they are suitable as circuit expressions.

$$\frac{\begin{array}{l} e' = CLOSE(e,H,S),\ \overline{stmt'} = CLOSE(\overline{stmt},H,S), \\ S' = \{@\mathbf{forallVars} \mapsto \overline{(T,x)}, @\mathbf{forallPred} \mapsto e'\} \circ S \end{array}}{\begin{array}{l} \mathbf{TransformS}(\mathbf{forall}(\overline{T\ x};e)\{\overline{stmt}\} \circ \{\},S,H,c,K) \\ = \mathbf{TransformS}(\overline{stmt'},S',H,c,K) \end{array}} \tag{7.197}$$

Relation Assignment in a Forall

$$\frac{\begin{array}{l} \mathbf{TransformE}(e,S,H,c) = r,H, \\ \mathbf{TransformE}(\overline{e'},S,H,c) = \overline{e''},H'', \\ \mathbf{TransformE}(e^{+},S,H'',c) = e^{++},H^{++}, \\ S(@\mathbf{forallPred}) = e_{pred},\ S(@\mathbf{forallVars}) = \overline{T\ x}, \\ r' = \lambda\ \bar{i}\ .\ BIND(\bar{i},\bar{x},\overline{e''},e_{pred}) \to e^{++}, r(\bar{i}), \\ S''',H''' = \text{update}(r,r',c,S,H^{++}) \end{array}}{\mathbf{TransformS}(e[\overline{e'}] = e^{+},S,H,c,K) = \mathbf{TransformS}(;,S''',H''',c,K)} \tag{7.198}$$

Reduction Update in a Forall

Reductions are implicitly updated within a `forall`. It is not possible to produce incremental updates for the reduction. As in the operational semantics, the value of the reduction after a `forall` can be explicitly calculated.

$$\frac{e_{new} = \Sigma_{\bar{x}}\ P(\bar{x}) \to trueVal,\ falseVal,\ H'''' = H'''[\text{redux} \mapsto e_{new}]}{\mathbf{TransformS}(e[\overline{e'}] = e^{+},S,H,c,K) = \mathbf{TransformS}(;,S''',H'''',c,K)} \tag{7.199}$$

choose the circuit form of `choose` binds the `choose` variables to circuit variables. The predicate is conjoined with the current control-flow variable to produce the control-flow expression for the body of the `choose`.

$$\frac{\begin{array}{l}\textbf{TransformV}(\overline{T\ x}) = \overline{v},\\ \textbf{TransformE}(\mathit{CLOSE}(e,H,S),S,H,c) = e',H,\\ \overline{\mathit{stmt}'} = \mathit{CLOSE}(\overline{\mathit{stmt}},H,S),\\ S' = \{\overline{T\ x} \mapsto \overline{v}\} \circ S,\\ \textbf{TransformS}(\overline{\mathit{stmt}'},S',H,e' \wedge c,\{\}) = e'',H'',S'',c'\end{array}}{\textbf{TransformS}(\textbf{choose}(\overline{T\ x};e)\{\overline{\mathit{stmt}}\},S,H,c,K) = \textbf{TransformS}(;,S'',H'',c',K)} \tag{7.200}$$

7.4.9 Program Transform

An ACCLAM program is a 4-tuple of: a model table, variable declarations, named states, and a top-level expression. We will need an additional environment *STATES* to map state names to actual states. A state is a mapping from names to expressions, and is therefore the heap produced by a statement transform.

Variable Declaration

$$\frac{\begin{array}{l}\mathit{STATES}(\text{@init}) = H,\\ \textbf{TransformV}(\overline{T\ x}) = \overline{v},\\ H' = H[\overline{T\ x} \mapsto \overline{v}],\\ \mathit{STATES}' = \mathit{STATES}[\text{@init} \mapsto H']\end{array}}{\textbf{TransformP}(\overline{T\ x},\mathit{STATES}) = \mathit{STATES}'} \tag{7.201}$$

Named States Transforming the named states portion of a program is simply transforming the statement portion and extracting out the resulting heap expressions and extending the *STATES* mapping. We are assuming that there will always be at least the initial state within the *STATES* mapping and that it will be named some well-known reserved keyword (e.g., `@init`). For readability's sake, in these rules all states will be named σ_i for some i.

$$\frac{\begin{array}{l} H = STATES(\sigma_1), \\ \mathbf{TransformS}(;, H, [], \mathbf{true}, \overline{stmt} \circ \{\}) = e, H', S', c', \\ STATES' = \{\sigma_2 \mapsto H'\} \circ STATES \end{array}}{\mathbf{TransformP}(\sigma_1\ \{\overline{stmt}\}\ \sigma_2, STATES) = STATES'} \tag{7.202}$$

$$\frac{\mathbf{TransformP}(\sigma_1\ \{\overline{stmt}\}\ \sigma_2, STATES) = STATES'}{\begin{array}{l} \mathbf{TransformP}(\sigma_1\ \{\overline{stmt}\}\ \sigma_2; \overline{(\sigma_n\{\overline{stmt}\}\sigma_{n+1})}, STATES) \\ = \mathbf{TransformP}(\overline{(\sigma_n\{\overline{stmt}\}\sigma_{n+1})}, STATES') \end{array}} \tag{7.203}$$

Top-level Expression The top-level expression is a normal expression, except that it supports two additional forms. One allows an expression to be evaluated in terms of a named state (rather than the default, which is the initial state). This is not a fully general expression and can only distribute around primitive operations. The transform function that can handle this additional syntax is **TransformEE**. The other additional form is equality of states, which is syntactically distinct from other forms of equality.

Expressions in Terms of a Named State

$$\frac{STATES(\sigma_n) = H',\ \mathbf{TransformE}(\overline{e}, [], H', \mathbf{true}) = \overline{e'}}{\mathbf{TransformEE}(\sigma_n(\mathbf{primop}(\overline{e})), STATES) = \mathbf{TransformE}(\mathbf{primop}(\overline{e'}), [], H', \mathbf{true})} \tag{7.204}$$

$$\frac{STATES(\sigma_n) = H'}{\mathbf{TransformEE}(\sigma_n(e), STATES) = \mathbf{TransformE}(e, [], H', \mathbf{true})} \tag{7.205}$$

Equality of Named States

$$\frac{\begin{array}{l} STATES(\sigma_n) = H_n,\ STATES(\sigma_{n+1}) = H_{n+1}, \\ \text{NAMES} = \{\text{name} \mid \text{name} \in \text{dom}(H_n)\ \wedge\ \text{name} \in \text{dom}(H_{n+1})\} \end{array}}{\begin{array}{l} \mathbf{TransformEE}(\sigma_n == \sigma_{n+1}, STATES) \\ = \bigwedge \text{name} \in \text{NAMES}\ :\ H_n(\text{name}) = e_n \wedge H_{n+1}(\text{name}) = e_{n+1}\ \wedge \\ \mathbf{TransformE}(\mathbf{TransformE}(e_n, [], H_n, \mathbf{true}) == \\ \ \mathbf{TransformE}(e_{n+1}, [], H_{n+1}, \mathbf{true}), [], STATES(\sigma_{init}), \mathbf{true}) \end{array}} \tag{7.206}$$

Normal Expressions Fall-through to **TransformE**

$$\frac{STATES(\sigma_{init}) = H}{\textbf{TransformEE}(e, STATES) = \textbf{TransformE}(e, [], H, \textbf{true})} \tag{7.207}$$

Whole Program Transformation

$$\frac{\begin{array}{l}\textbf{TransformP}(\overline{T\ x}, \{\sigma_{init} \mapsto \{\}\}) = STATES',\\ \textbf{TransformP}(\overline{\sigma_1\ \{\overline{stmt}\}\ \sigma_2}, STATES') = STATES'',\\ \textbf{TransformEE}(e, STATES'') = e'\end{array}}{\textbf{TransformP}((MT; \overline{T\ x}; \overline{\sigma_1\ \{\overline{stmt}\}\ \sigma_2}; e), \{\}) = e'} \tag{7.208}$$

7.5 Equivalence of Circuit and Operational Semantics

Equivalence between operational and circuit descriptions will be demonstrated by proving the equivalence between states. Simplistically, given equivalent initial states, the circuit expression must produce the same values as the operational rule evaluates to. Since the operational semantics also encompass mutations of heap and stack state, equivalence must be done over all heap and stack values.

This proof will be a case-by-case proof for each of the expression and statement types. We will assume that there is a mapping from operational values/constants to circuit values/-constants that is correct, and the proofs will be demonstrating that the end states are still equivalent under the mapping. We assume that the mapping function preserves typing (it is sound) and that it is 1-to-1. We also assume that the mapping function distributes over primitive operations (i.e., we assume that the operational and circuit forms of primitive operations are equivalent). Therefore, given a mapping M, where $M(\textbf{primop}_{op}) = \textbf{primop}_{circ}$. $M(\textbf{primop}_{op}(\overline{e})) = \textbf{primop}_{circ}(M(\overline{e}))$.

As a reminder, the types of values in the operational state are: primitive and instance values, primitive and instance variables, conditionals, and lambda forms (for relations). The types of circuit state are: circuit values, circuit variables, multiplexors, lambda forms, and circuit expressions.

7.5.1 Equivalence of Basic Forms

Values and Variables:

An operational value is equivalent to a circuit expression if the circuit expression produces the equivalent mapped value whenever the control-flow expression is true. Given a mapping, M and a control flow expression c,

$$\frac{M(e_{circuit}) = e_{op},\; M(c_{circuit}) = c_{op},\; c_{op} \implies e_{op} = v_{op}}{v_{op} \text{ EQ } e_{circuit}} \tag{7.209}$$

$$e_{op} = v_{op} \implies (H,S,e_{op},FS) \rightarrow^{*} (H,S,v,FS) \tag{7.210}$$

An operational variable is equivalent to a circuit variable if the operational value of the variable is equivalent to the circuit expression of the circuit variable (conditioned by the control-flow expression).

Lambda Forms:

Two lambda forms are equivalent if for all mapped inputs the mapped lambda returns equivalent outputs. More formally, given a mapping M and two lambda forms: f_{op} and $f_{circuit}$,

$$f_{op} \text{ EQ } f_{circuit} \iff \forall \overline{y_{op}} \;:\; M(\overline{y_{op}}) = \overline{y_{circuit}} \implies M(f_{op}(y_{op})) = f_{circuit}(\overline{y_{circuit}}) \tag{7.211}$$

Conditional Forms:

A conditional is equivalent to a circuit multiplexor if, given a mapping M,

$$\frac{\begin{array}{c} M(c_{op}) = c_{circ},\; M(then_{op}) = then_{circ},\; M(else_{op}) = else_{circ}, \\ c_{op} \text{ EQ } c_{circ},\; then_{op} \text{ EQ } then_{circ},\; else_{op} \text{ EQ } else_{circ} \end{array}}{c_{op} \rightarrow then_{op},\; else_{op} \text{ EQ } c_{circ} \rightarrow then_{circ},\; else_{circ}} \tag{7.212}$$

Equivalence of Models:

We assume that the programs under consideration are well-formed and well-typed. Therefore we know that the models are of the same type. Equivalence of models then boils down to equivalence of fields. Given a mapping, M,

$$\frac{H_{op}(o_{op}) = (N, \mathbb{F}_{op}),\ H_{circ}(o_{circ}) = (N, \mathbb{F}_{circ}),\ \forall\ f \in \text{dom}(\mathbb{F}_{op})\ :\ \mathbb{F}_{op}(f)\ \text{EQ}\ \mathbb{F}_{circ}}{o_{op}\ \text{EQ}\ o_{circ}} \tag{7.213}$$

7.5.2 Equivalence of Stacks and Heaps

Stacks and heaps are just maps from names to states. In the operational context, state is either a primitive value, a model, or a relation lambda. In the circuit context, a state is either an expression, a model, or a relation lambda. Two stacks are equivalent if for each name in both stacks, the states mapped to by that name are equivalent (likewise for heaps).

Stack Equivalence

$$\frac{\forall\ \text{name} \in \text{dom}(S_{op}) \cap \text{dom}(S_{circ})\ :\ S_{op}(\text{name})\ \text{EQ}\ S_{circ}(\text{name})}{S_{op}\ \text{EQ}\ S_{circ}} \tag{7.214}$$

Heap Equivalence

$$\frac{\forall\ \text{name} \in \text{dom}(H_{op}) \cap \text{dom}(H_{circ})\ :\ H_{op}(\text{name})\ \text{EQ}\ H_{circ}(\text{name})}{H_{op}\ \text{EQ}\ H_{circ}} \tag{7.215}$$

7.5.3 Expressions

A circuit transform $\textbf{TransformE}(e, S, H, c) = e'_{circ},\ H'_{circ},\ c'_{circ}$ and an operational rule $(H, S, e, FS) \to (H'_{op}, S'_{op}, e'_{op}, FS)$ are equivalent if e'_{op} EQ e'_{circ} and S'_{op} EQ S'_{circ} when c is true, and H'_{op} EQ H'_{circ} when c is true. For all these cases, we assume that the starting states are already equivalent. For brevity's sake, we omit c' if it is identical to c (i.e., no control flow changes).

The general format of these rules will be a summary of the operational and circuit rules, followed by a proof of equivalence for parts of the final configuration that may differ from the initial configuration.

Field Reads

Operational: $(H_{op}, S_{op}, o.f, FS) \rightarrow (H_{op}, S_{op}, \mathbb{F}_{op}(f), FS)$

Circuit: $\textbf{TransformE}(o.f, S_{circ}, H_{circ}, c) = \mathbb{F}_{circ}(f), H_{circ}$

Heap and stack are unchanged in both cases, therefore they are equivalent. Therefore $H_{op}(o)$ EQ $H_{circ}(o)$, and $\mathbb{F}_{op}(f)$ EQ $\mathbb{F}_{circ}(f)$. □

Relation Dereference

Operational: $(H_{op}, S_{op}, r[\overline{v}], FS) \rightarrow (H_{op}, S_{op}, r_{op}(\overline{v}), FS$

Circuit: $\textbf{TransformE}(r[\overline{e'}], S_{circ}, H_{circ}, c) = r(\textbf{TransformE}(\overline{e'}, S_{circ}, H_{circ})), H_{circ}$

Let $H_{op}(r) = r_{op}$ and $H_{circ}(r) = r_{circ}$

Heap and stack are unchanged, therefore they are equivalent.

$\overline{e'}$ EQ $\overline{v}$, therefore $\textbf{TransformE}(\overline{e'}, S_{circ}, H_{circ})$ EQ $\overline{v}$. r_{op} EQ r_{circ},

therefore $r_{circ}(\textbf{TransformE}(\overline{e'}, S_{circ}, H_{circ}))$ EQ $r_{op}(\overline{v})$.□

Casts, Primops Trivially true.

Method Invocation

Let $H(o) = (N, \mathbb{F})$ and $\mathrm{METHOD}(N)(m) = (\overline{T\ x}, \overline{stmt})$

Operational:

$$\frac{MS_{op}=\{\textbf{this}\mapsto(o,N),\overline{x}\mapsto\overline{(v,\ N)}\}}{\begin{array}{l}(H_{op}, S_{op}, o.m(\overline{v}), FS)\\ \rightarrow (H_{op}, (MS_{op}) \circ S_{op}, \overline{stmt}, \{\} \circ FS)\\ \rightarrow^* (H'_{op}, S_{op}, v'_{op}, FS)\end{array}}$$

Circuit:

$$\frac{\begin{array}{l}\textbf{TransformE}(\overline{e}, S_{circ}, H_{circ}, c) = \overline{e'_{circ}}, H_{circ},\\ MS_{circ} = \{\textbf{this} \mapsto o, \overline{x} \mapsto \overline{e'_{circ}}\},\\ S'_{circ} = (MS_{circ}) \circ S_{circ}\end{array}}{\begin{array}{l}\textbf{TransformE}(o.m(\overline{e}), S_{circ}, H_{circ}, c)\\ = \textbf{TransformS}(;, (MS_{circ}) \circ S_{circ}, H_{circ}, c, \overline{stmt} \circ \{\})\\ = e_{ret}, S_{circ}, H'_{circ}\end{array}}$$

Assume $\overline{v}$ EQ $\overline{e}$, then MS_{op} EQ MS_{circ}. Therefore the stacks are equivalent.

Assume v'_{op} EQ $\textbf{TransformS}(;, (MS_{circ}) \circ S_{circ}, H_{circ}, c, \overline{stmt} \circ \{\})$ by induction. Then if c then v'_{op} EQ e_{ret}. Therefore the final terms are equivalent.

By assumption above, if v'_{op} EQ $\textbf{TransformS}(\ldots)$, then H'_{op} EQ H'_{circ}. Therefore the heaps are equivalent.□

New Model Construction Let $\text{INIT}(C)(\overline{T}) = (\overline{T\ x}, \overline{stmt})$,

Operational:

$$\frac{\begin{array}{l}\forall\ f \in \text{FIELDS}(C)\ :\ \mathbb{F}_{op}(f) = \text{init}(T),\\ MS_{op} = \{\textbf{this} \mapsto (o,C), \overline{x} \mapsto \overline{(v,\ T)}\}\\ H'_{op} = H_{op}[o \mapsto (C, \mathbb{F}_{op})]\end{array}}{\begin{array}{l}(H_{op}, S_{op}, \textbf{new}\ C(\overline{v}), FS)\\ \rightarrow (H'_{op}, (MS_{op}) \circ S_{op}, \overline{stmt}, (\textbf{return}\ o;) \circ FS)\\ \rightarrow^{*} (H''_{op}, S_{op}, v, FS)\end{array}}$$

Circuit:

$$\frac{\begin{array}{l}\textbf{TransformV}(C) = (C, \mathbb{F}_{circ}),\\ MS_{circ} = \{\textbf{this} \mapsto (o,C), \overline{x} \mapsto \overline{(v,\ T)}\},\\ H'_{circ} = H_{circ}[o \mapsto (C, \mathbb{F}_{circ})],\\ \textbf{TransformS}(;, (MS_{circ}) \circ S_{circ}, H'_{circ},\\ c, \overline{stmt} \circ \{\}) = S_{circ}, H''_{circ},\\ H''_{circ}(o) = (C, \mathbb{F}_{circ})\end{array}}{\begin{array}{l}\textbf{TransformE}(\text{new}\ C(\overline{v_{circ}}), S_{circ}, H_{circ}, c)\\ \quad = H''_{circ}(o), H''_{circ}\end{array}}$$

$\mathbb{F}_{circ}$ EQ $\mathbb{F}_{op}$ by the definition of **TransformV**, therefore MS_{op} EQ MS_{circ}.

Since H_{op} EQ H_{circ} as well, then we can infer that H'_{op} EQ H'_{circ}.

Because all the state components are EQ, we can now assume that:

$\left[(H'_{op}, (MS_{op}) \circ S_{op}, \overline{stmt}, (\textbf{return}\ o;) \circ FS)\right]$ EQ

$[\textbf{TransformS}(;, (MS_{circ}) \circ S_{circ}, H'_{circ}, c, \overline{stmt} \circ \{\})]$,

and therefore (S_{op}, H''_{op}) EQ (S_{circ}, H''_{circ}).

Because the final heaps are equivalent, and because $v = H''_{op}(o)$ we can conclude that v EQ $H''_{circ}(o)$. Therefore the final terms are also equivalent. □

7.5.4 Statements

Assume $\textbf{TransformS}(\dots) = e, S, H, c$ and a final configuration (H', S', v, FS). A circuit statement transform is EQ to the operational step if v EQ $e\ \wedge\ S'$ EQ $S\ \wedge\ H'$ EQ H. The biggest difference between statements and expressions is the control-flow variable. Before examining statements case-by-case, we'll need a lemma to help us along the way.

7.5.4.1 Control Flow Lemma

A control flow expression, c, will be true at the start of a statement if c is true on the control path reaching that statement. This means that if control reaches a statement in the operational context, the control flow expression will be true in the circuit context. For circuits, the statement transform operates over a continuation K, and there is an additional requirement which is that assuming the initial c is true, at most one continuation will have a c' that is also true. This ensures that the set of statements 'evaluated' in the circuit context is identical to the set of statements evaluated in the operational context. Both of these properties of control-flow are implicitly true for the operational semantics; the rules can only evaluate along a single control flow path. Therefore, we need to prove that the circuit semantics produce at most one continuation with a true control-flow expression.

Case 1: Straight-line code Trivially true, $c' = c$ and there is only one continuation. $\square$

Case 2: Conditional Statements:

Starting with $\textbf{if}(e)\{\overline{stmt_1}\}$ **else** $\{\overline{stmt_2}\}$, with control-flow variable c, and continuation K. We assume that e is transformed into b, which leaves us with two possible continuations:

$K_1 = \textbf{TransformS}(;, S, H, b \wedge c, \overline{stmt_1}) = e_1, S_1, H_1, c_1$

$K_2 = \textbf{TransformS}(;, S, H, (\neg b) \wedge c, \overline{stmt_2}) = e_2, S_2, H_2, c_2$

$c_1 \implies b \wedge c$, c is true, therefore $c_1 \implies b$ and $\neg b \implies \neg c_1$.

$c_2 \implies (\neg b) \wedge c$, c is true, therefore $c_2 \implies \neg b$ and $b \implies \neg c_2$.

After a little substitution, we have $c_1 \implies \neg c_2 \wedge c_2 \implies \neg c_1$. Therefore the control flow expression can only be true for one of K_1 or K_2. $\square$

Case 3: Choose

By the side condition for `choose`, we can assume that the choose predicate is true for at least one set of values. Therefore, the body will be executed (once), and therefore there is at most one continuation for which c' is true. $\square$

Case 4: Return

Normal case In the normal case, the `return` is at the end of a scope/block and the next statement is the scope exit $\{\}$. The transformation would be:

$\textbf{TransformS}(\textbf{return}\ v, S, H, c, \{\} \circ K) = \textbf{TransformS}(;, S', H', c, \{\} \circ K)$. There is only one continuation (K), therefore at most one continuation will have c true. $\square$

Abnormal Case In this case, the return is in the middle of a block of statements. The transformation is: $\textbf{TransformS}(\textbf{return}\, e, S, H, c, stmt \circ K) = \textbf{TransformS}(\textbf{return}\, e, S', H', c, K)$. c is unchanged, and there is only one continuation. □

7.5.4.2 Conditioned-State Lemma

The control-flow expression also manifests itself in the state mappings for the circuit transformations. Every update is conditioned by the control-flow expression at that program point. Thus every variable name maps to an expression of the value of that variable over all possible control paths in the model. The final state in the operational context will contain only one value for each state element. To demonstrate equivalence, we need to prove that the expressions for state elements in the final circuit state produce only the value that is visible at that program point. That is, only updates made on the control-path to that program point are visible. This proof will be done for each of the control-flow changing statements. In these cases, e_{old} is the expression for that state element before any update was processed.

Case 1: Straight-line Code

Assuming a control-flow expression c, all updates in straight-line code will be of the form $c \rightarrow e,\ e_{old}$. Therefore, if c is true, only the new value will be visible. If c is not true, then the new value will not be visible. □

Case 2: Conditional Statements

The final states of the transformed conditional are:

$S' = b' \rightarrow S_{then}; S_{else}$, $H' = b' \rightarrow H_{then},\ H_{else}$, where b' is the transformed condition that is used to select between the two state maps. The initial condition expression is c, and the final expression is $c' = (b' \wedge c) \vee (\neg b' \wedge c) = c$.

If b' is true, then $b' \wedge c = c$ and $S' = S_{then}$ and $H' = H_{then}$. Therefore none of the `else` state will be visible in the final state.

If b' is false, then $\neg b' \wedge c = c$ and $S' = S_{else}$ and $H' = H_{else}$. Therefore none of the `then` state will be visible in the final state.

Therefore if c is true, then only one of the `then` state or the `else` state will be visible. Also, if c is false neither the `then` nor `else` state will be visible. □

Case 3: Return

Normal case The return is at the end of a block. There are no subsequent statements, therefore the state is the same. □

Abnormal case The return is followed by some number of statements before the block exit. All the updates are conditioned by an explicitly false control flow expression, therefore none of those updates will be visible.□

New Instances New instances are not conditioned by a control-flow variable. However, by well-formedness no well-formed program may refer to a name outside of the scope in which it is defined. For a program to violate the conditioned state lemma, well-formed code would have to be able to distinguish between a state where the object was constructed and one where it wasn't. However, no well-formed program may refer to an instance that hasn't been constructed, and no well-formed program may refer to an instance outside its scope. Therefore new instance construction doesn't violate the conditioned state lemma.

7.5.5 Statement Equivalence

Now that we have proved the conditioned state lemma, we can assume that as long as the operational value is equivalent to the circuit expression, then the heap and stack states are equivalent. We can now do a case-by-case proof of the equivalence of circuit and operational semantics.

sequence Trivially equivalent.

Local Variable Assignment

$$Operational: \quad \frac{\text{update}(S, x \mapsto v) = S'_{op}}{(H_{op}, S_{op}, x{=}v, FS) \rightarrow (H_{op}, S'_{op}, ;, FS)} \qquad Circuit: \quad \frac{S'_{circ} = \text{update}(S_{circ}, x, (c \rightarrow e, S_{circ}(x)))}{\textbf{TransformS}(x = e;, S_{circ}, H_{circ}, c, K)}$$

$$= \textbf{TransformS}(;, S'_{circ}, H_{circ}, c, K)$$

H_{op} EQ H_{circ}, because neither is changed.

e EQ v, by initial assumption. $(c \to e, S_{circ}(x)) = e$, by the control-flow lemma. Therefore S_{op} EQ S_{circ}.

Therefore both rules are equivalent. □

Heap Variable Assignment

Operational:

$$\frac{H_{op}(o) = (N, \mathbb{F}),\quad \mathbb{F}'_{op} = \mathbb{F}_{op}[f \mapsto v],\quad H'_{op} = H_{op}[o \mapsto (N, \mathbb{F}'_{op})]}{(H_{op}, S_{op}, o.f{=}v, FS) \to (H'_{op}, S_{op}, ;, FS)}$$

Circuit:

$$\frac{\begin{array}{l}\textbf{TransformE}(e, S_{circ}, H_{circ}) = (e_{circ}, H'_{circ}),\\ H_{circ}(o) = (N, \mathbb{F}_{circ}),\\ \mathbb{F}'_{circ} = \mathbb{F}_{circ}[f \mapsto e_{circ}],\\ H''_{circ} = \text{update}(H'_{circ}, o, (N, \mathbb{F}'_{circ}))\end{array}}{\begin{array}{l}\textbf{TransformS}(o.f = e, S_{circ}, H_{circ}, c, K)\\ = \textbf{TransformS}(;, S_{circ}, H''_{circ}, c, K)\end{array}}$$

S_{op} EQ S_{circ} because they are unchanged.

v EQ e_{circ} by equivalence of expression transforms, therefore H'_{circ} EQ H_{op} and $\mathbb{F}_{op}$ EQ $\mathbb{F}_{circ}$, and $\mathbb{F}'_{op}$ EQ $\mathbb{F}'_{circ}$.

$H''_{circ}(o) = (c \to \mathbb{F}'_{circ},\ \mathbb{F}_{circ})$ by the definition of update.

By the control flow lemmas, c is true and therefore $H''_{circ}(o) = \mathbb{F}'_{circ}$ when control reaches the operational rule.

Therefore H'_{op} EQ H''_{circ}. Therefore both rules are equivalent. □

Local Variable Declaration

Operational:

$$\frac{BS'_{op} = BS_{op}[x \mapsto (\text{init}(T), T)]}{\begin{array}{l}(H_{op}, (BS_{op} \circ MS) \circ S, T\ x;, FS)\\ \to (H_{op}, (BS'_{op} \circ MS) \circ S, ;, FS)\end{array}}$$

Circuit:

$$\frac{\textbf{TransformV}(T) = v_{circ},\ BS'_{circ} = BS_{circ}[x \mapsto v_{circ}]}{\begin{array}{l}\textbf{TransformS}(T\ x;, (BS_{circ} \circ MS_{circ}) \circ S_{circ},\\ H_{circ}, c, K) = \textbf{TransformS}(;,\\ (BS'_{circ} \circ MS_{circ}) \circ S_{circ}, H_{circ}, c, K)\end{array}}$$

H_{op} EQ H_{circ} trivially.

v_{circ} EQ $\text{init}(T)$, therefore BS'_{op} EQ BS'_{circ}.

Therefore $(BS'_{circ} \circ MS_{circ}) \circ S_{circ}$ EQ $(BS'_{op} \circ MS) \circ S$.

Therefore the rules are equivalent. □

Relation Assignment

Operational: *Circuit:*

$$\frac{\begin{array}{l} r_{op} = H_{op}(r), \\ r'_{op} = \lambda\ \bar{i}.(\bar{i} = \bar{v}) \rightarrow v', r_{op}(\bar{i}), \\ H'_{op} = H_{op}[r \mapsto r'_{op}] \end{array}}{(H_{op}, S_{op}, r[\bar{v}] = v', FS) \rightarrow (H'_{op}, S_{op}, ;, FS)} \qquad \frac{\begin{array}{l} H_{circ}(r) = r_{circ}, \\ r'_{circ} = \lambda \bar{i}.(\bar{i} = \overline{e_{circ}}) \wedge c \rightarrow e', r_{circ}(\bar{i}) \\ H'_{circ} = \text{update}(r, r'_{circ}, H_{circ}) \end{array}}{\begin{array}{l} \textbf{TransformS}(r[\overline{e_{circ}}] = e', S_{circ}, H_{circ}, c, K) \\ = \textbf{TransformS}(;, S_{circ}, H'_{circ}, c, K) \end{array}}$$

Stacks are EQ trivially.

H_{op} EQ H_{circ} by initial assumption, therefore r_{op} EQ r_{circ}.

$\bar{v}$ EQ $\overline{e_{circ}}$ and v EQ e' by initial assumption, and $(\bar{i} = \overline{e_{circ}}) \wedge c$ EQ $(\bar{i} = \bar{v})$ by the control flow lemmas. Therefore r'_{op} EQ r'_{circ}. Therefore H'_{op} EQ H'_{circ}.

Therefore both rules are equivalent. □

Return

Normal Return (return statement at end of block)

Operational: *Circuit:*

$$\frac{}{\begin{array}{l} (H_{op}, MS_{op} \circ S_{op}, \textbf{return}\ v;, \{\} \circ FS) \\ \rightarrow (H_{op}, S_{op}, v, FS) \end{array}} \qquad \frac{\begin{array}{l} MS'_{circ} = \text{update}(MS_{circ}, \textbf{@return}, \\ \quad c \rightarrow e, MS_{circ}(\textbf{@return})) \end{array}}{\begin{array}{l} \textbf{TransformS}(\textbf{return}\ e, MS_{circ} \circ S_{circ}, \\ H_{circ}, c, \{\}) = \textbf{TransformS}(;, MS'_{circ} \circ S_{circ}, \\ H_{circ}, c, \{\}) = MS'_{circ}(\textbf{@return}), S_{circ}, H_{circ} \end{array}}$$

Before we demonstrate the equivalence of these rules, we must first argue for the generality of the circuit form. The statement continuation is the end-of-scope token $\{\}$. Does this cover all possible returns? By well-formedness we know that returns can occur only within a method context. We also know that method bodies are evaluated only via method invocation. Method invocation is an expression, and therefore in the circuit context there is a **TransformE** that will consume what is produced by the **TransformS**. Therefore, this form of the circuit transformation is general enough to cover all well-formed cases.

H_{op} EQ H_{circ}, S_{op} EQ S_{circ} trivially.

$MS'_{circ}(@\textbf{return}) = c \to e, MS_{circ}(@\textbf{return})$. e EQ v, by initial assumption.

Therefore v EQ $c \to e, MS_{circ}(@\textbf{return})$ by the conditioned state lemma.

Therefore $MS'_{circ}(@\textbf{return})$ EQ v. □

Abnormal Return (return statement in the middle of a block)

$$\begin{array}{cc}
\textit{Operational:} & \textit{Circuit:} \\[1em]
\dfrac{F \neq \{\}}{\begin{array}{l}(H_{op}, S_{op}, \textbf{return } v;, F \circ FS) \\ \to (H_{op}, S_{op}, \textbf{return } v;, FS)\end{array}} &
\dfrac{\begin{array}{l}\textbf{TransformS}(stmt, S_{circ}, H_{circ}, \textbf{false}, \{\}) \\ \quad = \text{void}, S'_{circ}, H'_{circ}, c', \\ stmt \neq \{\}\end{array}}{\begin{array}{l}\textbf{TransformS}(\textbf{return } e, S_{circ}, H_{circ}, c, stmt \circ K) \\ = \textbf{TransformS}(\textbf{return } e, S'_{circ}, H'_{circ}, c, K)\end{array}}
\end{array}$$

To begin, **return** e EQ **return** v trivially.

S'_{circ} is an extension of S_{circ} conditioned by the control flow expression, c'. For each symbol, x in S_{circ}, $(S'_{circ}(x) = S_{circ}(x)) \vee (S'_{circ}(x) = c' \to e', S_{circ}(x))$. For all mappings that weren't updated S'_{circ} is identical to S_{circ}. However, since c' is false, then for all the updated mappings $S'_{circ}(x) = \textbf{false} \to e', \ S_{circ}(x) = S_{circ}(x)$, and is therefore also identical. Therefore S'_{circ} EQ S_{circ}, and therefore S_{op} EQ S_{circ}. A very similar argument can be made for heap equivalence as well as the control flow variable being the same. □

Conditional Statements

Operational:

Condition true

$$\frac{v_{op}=\textbf{true}}{\begin{array}{l}(H_{op},S_{op},\textbf{if}(v_{op})\{\overline{stmt_1}\}\ \textbf{else}\ \{\overline{stmt_2}\},FS)\\ \rightarrow (H_{op},S_{op},\{\overline{stmt_1}\},FS)\\ \rightarrow (H^1_{op},S^1_{op},\{\},FS)\end{array}}$$

Condition false

$$\frac{v_{op}=\textbf{false}}{\begin{array}{l}(H_{op},S_{op},\textbf{if}(v_{op})\{\overline{stmt_1}\}\textbf{else}\{\overline{stmt_2}\},\\ FS) \rightarrow (H_{op},S_{op},\{\overline{stmt_2}\},FS)\\ \rightarrow (H^2_{op},S^2_{op},\{\},FS)\end{array}}$$

Circuit:

$$\frac{\begin{array}{l}\textbf{TransformS}(;,S_{circ},H_{circ},b \wedge c,\overline{stmt_1}\circ\{\})\\ = e^1_{circ},S^1_{circ},H^1_{circ},c_1\\ \textbf{TransformS}(;,S_{circ},H_{circ},\neg b \wedge c,\overline{stmt_2}\circ\{\})\\ = e^2_{circ},S^2_{circ},H^2_{circ},c_2\\ S'_{circ}(x) = b \rightarrow S^1_{circ}(x),\ S^2_{circ}(x)\\ H'_{circ}(x) = b \rightarrow H^1_{circ}(x),\ H^2_{circ}(x)\end{array}}{\begin{array}{l}\textbf{TransformS}(\textbf{if}(v_{op})\{\overline{stmt_1}\}\ \textbf{else}\ \{\overline{stmt_2}\},S_{circ},H_{circ},c,K)\\ = \textbf{TransformS}(;,S'_{circ},H'_{circ},c_1 \vee c_2,K)\end{array}}$$

We shall prove equivalence by proving the true and false conditions separately.

Condition true H^1_{op},S^1_{op} EQ $H^1_{circ}(x),S^1_{circ}(x)$ by the equivalence of statements. If the condition is true, then $b = \textbf{true}$, therefore $S'_{circ} = S^1_{circ}$ (likewise for H'_{circ}). Therefore if the condition is true, the two forms are EQ. □

Condition false H^2_{op},S^2_{op} EQ $H^2_{circ}(x),S^2_{circ}(x)$ by the equivalence of statements. If the condition is false, then $S'_{circ} = S^2_{circ}$ (likewise for H'_{circ}). Therefore the states are also EQ when the condition is false. □.

Because b EQ v_{op} by the initial assumption, conditional statements are EQ. □

Forall

Operational:

$$\frac{\begin{array}{l} e'_{op} = CLOSE(e, H_{op}, S_{op}), \\ \overline{stmt_{op}} = CLOSE(\overline{stmt}, H_{op}, S_{op}), \\ BS_{op} = \{\textbf{@forallVars} \mapsto \overline{(x\ T)},\ \textbf{@forallPred} \mapsto e'_{op}\} \end{array}}{\begin{array}{l} (H_{op}, S_{op}, \textbf{forall}(\overline{T\ x};\ e)\{\overline{stmt}\}, FS) \\ \rightarrow (H_{op}, BS_{op} \circ S_{op}, \overline{stmt_{op}}, \{\} \circ FS) \end{array}}$$

Circuit:

$$\frac{\begin{array}{l} e'_{circ} = CLOSE(e, H_{circ}, S_{circ}), \\ \overline{stmt_{circ}} = CLOSE(\overline{stmt}, H_{circ}, S_{circ}), \\ BS_{circ} = \{\textbf{@forallVars} \mapsto \overline{(x\ T)},\ \textbf{@forallPred} \mapsto e'_{circ}\} \end{array}}{\begin{array}{l} \textbf{TransformS}(\textbf{forall}(\overline{T\ x};\ e)\{\overline{stmt}\}, S_{circ}, H_{circ}, c, K) \\ = \textbf{TransformS}(\overline{stmt_{circ}}, BS_{circ} \circ S_{circ}, H_{circ}, c, K) \end{array}}$$

CLOSE is the same in both contexts, therefore e'_{op} EQ e'_{circ}, and $\overline{stmt_{op}}$ EQ $\overline{stmt_{circ}}$. Therefore BS_{op} EQ BS_{circ}, and therefore by the initial assumption $BS_{op} \circ S_{op}$ EQ $BS_{circ} \circ S_{circ}$. The heaps are trivially equivalent. Therefore both rules are equivalent. □

Relation Assignment in Forall

$$\begin{array}{cc} \textit{Operational}: & \textit{Circuit}: \\ \dfrac{\begin{array}{l} \text{eval}(S_{op}, \textbf{@forallVars}) = \overline{(x\ T)}, \\ \text{eval}(S_{op}, \textbf{@forallPred}) = e_{op}, \\ H_{op}(r) = r_{op} = \lambda\bar{i}\ .\ \ldots, \\ r'_{op} = \lambda\bar{i}\ .\ BIND(\bar{i}, \bar{x}, \bar{e}, e_{op}) \rightarrow v,\ r_{op}(\bar{i}), \\ H'_{op} = H_{op}[r \mapsto r'_{op}] \end{array}}{\begin{array}{l} (H_{op}, S_{op}, r[\bar{e}] = v, FS) \\ \rightarrow (H'_{op}, S_{op}, ;, FS) \end{array}} & \dfrac{\begin{array}{l} S_{circ}(\textbf{@forallPred}) = e_{pred}, \\ S_{circ}(\textbf{@forallVars}) = \overline{T\ x}, \\ H_{circ}(r) = r_{old} = \lambda\bar{i}\ .\ \ldots, \\ r'_{circ} = \lambda\bar{i}\ .\ BIND(\bar{i}, \bar{x}, \bar{e}, e_{pred}) \\ \quad \wedge c \rightarrow e_{circ},\ r_{circ}(\bar{i}), \\ H'_{circ} = \text{update}(r, r'_{circ}, H_{circ}) \end{array}}{\begin{array}{l} \textbf{TransformS}(r[\bar{e}] = e_{circ}, S_{crc}, H_{crc}, c, K) \\ = \textbf{TransformS}(;, S_{circ}, H'_{circ}) \end{array}} \end{array}$$

BIND is the same in both contexts, therefore $BIND(\bar{i},\overline{x},\overline{e},e_{op})$ EQ $BIND(\bar{i},\overline{x},\overline{e},e_{pred})$. Therefore by the conditioned state lemma, r'_{op} EQ r'_{circ}. Therefore H'_{op} EQ H_{op}. □

Choose

Operational:

$$\frac{\begin{array}{l} e'_{op} = CLOSE(e,H_{op},S_{op}), \overline{stmt'} = CLOSE(\overline{stmt},H_{op},S_{op}), \\ \exists \overline{T\ v_{op}}\ :\ eval(e'_{op}[\overline{v_{op}}\ \overline{x}]) = \textbf{true}, BS_{op} = \{\overline{x} \mapsto \overline{v_{op}}\} \end{array}}{\begin{array}{l} (H_{op},S_{op},\textbf{choose}(\overline{T\ x};e)\{\overline{stmt}\},FS) \\ \to (H_{op},BS_{op} \circ S_{op},\overline{stmt},\{\} \circ FS) \to (H'_{op},S'_{op},;,FS) \end{array}}$$

Circuit:

$$\frac{\begin{array}{l} \textbf{TransformV}(\overline{T\ x}) = \overline{v_{circ}}, \\ e_{circ} = CLOSE(e,H_{circ},S_{circ}), \\ \overline{stmt'} = CLOSE(\overline{stmt},H_{circ},S_{circ}), \\ BS_{circ} = \{\overline{T\ x} \mapsto \overline{v_{circ}}\}, \\ \textbf{TransformS}(\{\overline{stmt}\},BS_{circ} \circ S_{circ},H_{circ},c \wedge e_{circ},K) = e'',H'_{circ},S'_{circ},c' \end{array}}{\begin{array}{l} \textbf{TransformS}(\textbf{choose}(\overline{T\ x};e)\{\overline{stmt}\},S_{circ},H_{circ},c,K) \\ = \textbf{TransformS}(;,S'_{circ}H'_{circ},c,K) \end{array}}$$

e_{op} EQ e_{circ}, assuming that $\exists\ \overline{T\ v_{op}}$ such that e_{op} is true, then BS_{op} EQ BS_{circ}, therefore H'_{circ},S'_{circ} EQ H'_{op},S'_{op} by the equivalence of statement transformations. Therefore `choose` is equivalent □.

This proof requires that the `choose` be executable, that is, that there is a legal value assignment such that the `choose′s` predicate is true. This is a proof requirement for the modeler, and can actually be proved automatically (see Chapter 8).

7.5.6 Program Equivalence

An ACCLAM program is a set of models, some variable declarations, named state definitions and an expression. The models, variable declarations and expression are all trivially equivalent given the prior proofs. State definitions are just the evaluation of statement sequences, with the additional extension of a *STATES* mapping. Since the mapping is identical for both the circuit and operational context, the state definitions are also trivially equivalent.

CHAPTER 8

QUESTIONS ANSWERABLE WITH THE LANGUAGE AND TOOL

ACCLAM is meant to model general purpose data types so that programmers can do transactional data structure design with formal feedback. ACCLAM needs to be flexible so that programmers can describe as many data types as possible. However, the real utility of ACCLAM depends on the kinds of questions that an ACCLAM processing tool can answer (that is, prove). This chapter will discuss several such useful questions, why they would be interesting in a concurrent and transactional context, and how they are best represented for use by a SAT solver.

Throughout this section I'll use the word 'question' to refer to a specific proof condition that an ACCLAM tool can prove given a fully-specified model.

8.1 Correctness Questions

The first set of questions deal with general correctness issues. These are questions that address whether or not the data type being modeled is correct in a concurrent context.

8.1.1 Conflict Predicate Correctness

Given two methods op_1 and op_2, and a predicate expression, P, over the state, the ACCLAM tool can prove that the predicate correctly describes the states under which op_1 and op_2 conflict (cannot commute). In general, P may be a function of the initial state, the possible intermediate and final states, the method arguments, and their return values. Of course, practical systems may not be able to sample all those values, but ACCLAM permits the modeler to describe potentially impractical conflict predicates. Specifying a predicate to describe conflicting states is interesting because it may be possible to precisely describe the states of interest even though a practical method for enforcing concurrency control may be necessarily less precise. Some systems [19] explicitly require some kind of predicate be provided by the programmer (although in boosting, the predicate may only be a function of the initial state and the method arguments). Additionally, it can be enlightening to try to specify the exact states under which methods conflict, and this can lead to improvements to a given data type's concurrency.

Consider an initial state σ_i, and operations and states related as follows:

$$\begin{aligned}
&\sigma_i\ \{ret_{11} = op_1(\bar{x})\}\ \sigma_1\\
&\sigma_i\ \{ret_{21} = op_2(\bar{y})\}\ \sigma_2\\
&\sigma_1\ \{ret_{22} = op_2(\bar{y})\}\ \sigma_{12}\\
&\sigma_2\ \{ret_{12} = op_1(\bar{x})\}\ \sigma_{21}
\end{aligned}$$

We can say that, for a given set of arguments and initial state, op_1 and op_2 do not commute if:

$$(ret_{11} \neq ret_{12}) \vee (ret_{21} \neq ret_{22}) \vee (\sigma_{12} \neq \sigma_{21})$$

In this expression equality of primitive values is as you'd normally expect, equality of object types is the piecewise equality of their member values, and equality of states is the piecewise equality of each of the entries in either state. Equality of relations is defined as $r = r' \iff \forall x\ :\ r[x] = r'[x]$.

We can summarize the non-commutativity conditions as a predicate:

$$NonComm(ret_{11}, ret_{12}, ret_{21}, ret_{22}, \sigma_{12}, \sigma_{21})$$

A sufficient condition for proving the correctness of a given conflict predicate P is:

$$\forall_{\sigma_i,\bar{x},\bar{y}}\ NonComm(\dots) \rightarrow P(ret_{11}, ret_{12}, ret_{21}, ret_{22}, \sigma_{12}, \sigma_{21}, \sigma_1, \sigma_2, \sigma_i, \bar{x}, \bar{y})$$

Of course, this just proves that the supplied predicate is true when the methods wouldn't commute under a given state, and says nothing about how exact the predicate is.

Practically, conflict predicates might often be developed incrementally. The modeler would propose a predicate and then refine it iteratively to get a better idea of the commutativity behavior of the data type being modeled.

8.1.2 SAT-friendly form

SAT solvers are particularly good at exploring a space in a way that is convenient for answering existentially quantified expressions but not universally quantified ones. Since the correctness condition above is universally quantified, it would seem to be a bad fit for a SAT solver. The ACCLAM verification tool reformulates the condition so that an UNSAT result implies that the condition is met. The reformulation is based on inverting the problem and treating a satisfying assignment (that is a SATisfiable problem) as a rejection. Thus we negate the more intuitive expression above to create an existential form:

$$\exists \sigma_i,\ \bar{x},\ \bar{y}\ s.t.\ NonComm(\ldots) \wedge \neg P(\ldots)$$

A satisfying assignment for this form implies that there is a state for which the operations do not commute but for which the conflict predicate is false. This would contradict the correctness condition. Therefore if this existential form is fed into a SAT solver and it finds an assignment, the conflict predicate is not correct. Furthermore, the modeler has an assignment that is a counter-example, which may help the modeler to refine the predicate. If the SAT solver terminates with UNSAT, there is no valid assignment and thus the predicate is correct.

8.1.3 Lock Correctness

Conflict predicates can be (almost) arbitrarily complex [1] and depend on many different states. They can be incredibly precise, describing exactly those combinations of state and arguments that prevent commutativity . However, in real-world systems many of the state variables that a precise predicate would require may not be accessible. For example, possible intermediate states may require the partial execution of the methods in question, which may be impossible or inefficient. A computationally expensive expression that must be evaluated before every method invocation may not be practical, especially if performance is a factor. One proposal for a more practical conflict detection scheme is abstract locking [51]. Like more conventional locking, the locks and their lock modes are used to determine if locks conflict. Locks are acquired as a method proceeds and are held until the outer transaction commits. Abstract locks act as a summary of the state modifications performed by an open-nested action.

An abstract lock is created within a *lock space* and with a particular *mode*. A *lock space* is a collection of lock types with rules for determining if given instances of those lock types overlap. For example, one could imagine a space of two dimensional objects. It might have two types within it, points and line segments. The rules for determining overlap might be the conventional ones from two dimensional geometry. If the lock space were defined so that the points within it were integers or strings, one could imagine using such a lock space to model locks over elements and ranges of elements within a data type.

Locks are also instantiated with a particular mode. Modes are much as they are in the 'conventional' locking world. Mode conflict can be defined as a matrix of boolean values where the modes define the axes and at each cell within the matrix there is a boolean value determining whether those two particular modes conflict. In order for two locks to conflict, they must be from the same space, overlap within that space, and their respective modes

[1] Alternation of quantifiers is not allowed, and any predicate that causes the tool to exhaust memory and/or CPU resources is obviously impractical.

must conflict. Locks are constructed based on the state at a particular point of execution and the arguments to a method. Usually, they're constructed at method entry although they may also be acquired within a method as well. Because locks depend on the state at the point where they are constructed, method invocation order may affect lock instances or modes. For two methods op_1 and op_2, there are four potentially distinct sets of lock instances and two possible combinations:

$$Locks_1(\sigma_i,\bar{x}) * Locks_{21}(\sigma_1,\bar{y})$$
$$Locks_2(\sigma_i,\bar{y}) * Locks_{12}(\sigma_2,\bar{x})$$

If we define a lock as a tuple of a lock instance (which includes the mode) and a lock space, then we can define a function to determine overlap:

$$OVERLAP(< instance_1, space_1 >, < instance_2, space_2 >)$$
$$= (space_1 = space_2) \rightarrow overlap_{space_1}(instance_1, instance_2),\ false$$

In this definition, we assume that $overlap_{space_i}$ is the overlap function defined for the $space_i$ lock space. Using the lock space aware overlap function, we can define a lock conflict predicate $LockCon$:

$$LockCon =$$
$$OVERLAP(lock_1, lock_2) \wedge \text{conflict}(\text{Mode}(lock_1), \text{Mode}(lock_2))$$

We then generalize this definition to sets of locks, since a method may acquire multiple locks:

$$LocksCon =$$
$$\exists\ lock_i \in Locks_1(\sigma_i,\bar{x}),\ lock_j \in Locks_{21}(\sigma_1,\bar{y}),$$
$$lock_m \in Locks_2(\sigma_i,\bar{y}),\ lock_n \in Locks_{12}(\sigma_2,\bar{x})$$
$$:\ LockCon(lock_i, lock_j) \vee LockCon(lock_m, lock_n)$$

Using this lock conflict predicate, we can then define the lock correctness condition as:

$$\forall_{\sigma_i,\bar{x},\bar{y}}\, NonComm(\ldots) \implies LocksCon(\sigma_i,\sigma_1,\sigma_2,\bar{x},\bar{y},Locks_1,Locks_2,Locks_{12},Locks_{21})$$

This is not vastly different from the conflict predicate correctness condition. However, the locking conflict predicate is a very different animal. In particular the lock arguments themselves may be large expressions that depend on the state of the model.

8.1.3.1 SAT-friendly form

We invert the problem by treating UNSAT as correct and SAT as incorrect. We then negate the universally quantified form to obtain:

$$\exists \sigma_i,\bar{x},\bar{y}\ :\ NonComm(\ldots) \wedge \neg LockCon(\ldots)$$

If the SAT solver finds an assignment (a SAT result), then there is a combination of initial state and legal arguments that is non-commutative and for which the defined locks do not conflict. This result is proof of the incorrectness of the specified locking protocol, and it is also a specific failure case that the modeler can examine to help debug the locking protocol. If the SAT solver returns UNSAT, then no incorrect assignment is possible so the model is proved to specify a safe locking predicate.

8.1.4 Correctness of Inverses

Transactional programming involves reasoning not only about the commutativity of operations on a data type, but also about their invertability. A transactional run-time is free to fail and roll back a transaction at any point, so it is important that an open-nested data structure correctly rolls back any uncommitted changes that were made. Of course, since the data structure uses open nested actions, the inverse doesn't need to undo every single concrete operation that was performed up to the abort. The inverse's function is to undo the changes to produce an abstract state that is equivalent to the initial state just before the open-nested action was invoked.

While an open-nested action is executing, all the normal concurrency control mechanisms are in place, therefore an incomplete open-nested action can be undone in essentially the same way as a closed nested action. After the open nested action has committed, all the information that would allow the transactional run-time to undo automatically is discarded (to prevent spurious conflicts). If the outer transaction in which the open nested action is nested aborts, the run-time will invoke the programmer-provided inverse. Therefore, checking the correctness of the inverse means ensuring that for all legal initial and final states of the operation, the inverse transforms the final state back into the initial (abstract) state. The inverse operation itself has access to the arguments and initial state. ACCLAM itself allows the user to specify *prevals*, which are immutable variables initialized to expressions over the initial state. This is a convenience that makes it easier to write inverse operations. However, since all prevals are derived exclusively from the initial state and function arguments I won't include them specifically in the formal description of inverse operations.

Given an operation $op(\bar{x})$ we can define its inverse $op^{-1}(\bar{x}, \sigma)$. If an inverse is correct, then given an initial state σ_i:

$$\forall \sigma_i, \bar{x}, \sigma_{out} \ : \ \sigma_i \{op(\bar{x}); op^{-1}(\bar{x}, \sigma_i)\} \sigma_{out} \implies \sigma_i = \sigma_{out}$$

A correct inverse will always be able to produce a state equivalent to the initial state for all possible output states of the forward-going operation.

8.1.5 SAT-friendly form

An existential version of the correctness condition uses the SAT solver to hunt for a combination of initial state and arguments that the inverse does not successfully handle.

In this case, SAT is considered a failure, and a satisfying assignment indicates a state and arguments for which the inverse was not correct.

$$\exists \sigma_i, \bar{x}, \sigma_{out} \ : \ \sigma_i\{op(\bar{x}); op^{-1}(\bar{x}, \sigma_i)\}\sigma_{out} \implies \sigma_i \neq \sigma_{out}$$

8.2 Side-conditions

ACCLAM's semantics depend upon certain assumptions associated with some of the language constructs. Most of these can be checked statically ahead of time (e.g., type correctness). However, there are two conditions that require more sophisticated proofs to verify. Fortunately, the ACCLAM verification tool's job is to produce SAT problems for verification. We call these additional proofs 'side-conditions' and they fortunately can be proved as separate SAT problems.

8.2.1 Forall side condition

ACCLAM does not support unbounded looping or recursion, so modelers have to model this behavior with the `forall` statement. `forall` behaves like an atomic parallel update, therefore the modeler can't assume a particular order of evaluation. The transformation to a pure expression form requires that the effect of the `forall` statement be consistent. What this means is that the final state after the `forall` is the same regardless of the 'order of execution'. More specifically, it means that any relation update in the body of the `forall` must always assign the same value to the same index of the relation. A relation update may have expressions that produce index values, so it is possible that, for different values that satisfy the `forall` predicate, the index expressions will produce equal values. Similarly, the value being assigned to that index may also be the result of an expression. Therefore, this is a tricky condition to determine statically. So instead of attempting to determine `forall` correctness using conventional program analysis, ACCLAM just treats `forall` correctness as another correctness condition to prove.

A `forall` statement has some bound variables $\bar{z}$, a predicate $P(\bar{z})$, a control flow expression $C(\sigma)$, and a body that may contain multiple relation updates. Given a relation update in the body of the `forall`, `rel[e] = e'`, a correct `forall` will obey:

$$\forall \sigma, \bar{z_1}, \bar{z_2} \ : \ (C(\sigma) \wedge P(\bar{z_1}) \wedge P(\bar{z_2}) \wedge e[\bar{z_1}/\bar{z}] = e[\bar{z_2}/\bar{z}]) \implies e'[\bar{z_1}/\bar{z}] = e'[\bar{z_2}/\bar{z}]$$

A more SAT-friendly form is:

$$\exists \sigma, \bar{z_1}, \bar{z_2} \ : \ C(\sigma) \wedge P(\bar{z_1}) \wedge P(\bar{z_2}) \wedge e[\bar{z_1}/\bar{z}] = e[\bar{z_2}/\bar{z}] \wedge e'[\bar{z_1}/\bar{z}] \neq e'[\bar{z_2}/\bar{z}]$$

The ACCLAM verifier can exhaustively examine every possible valid combination of `forall` variables by extracting all possible variables from all possible states in which the `forall` can execute. In practice, any method calls within the `forall` will have been inlined so proof conditions can be generated for each `forall` independently.

8.2.2 Choose side condition

The `choose` statement in ACCLAM is correct only if there is always a valid choice that can be made. If the predicate within a `choose` is true for no elements of the model's state, then the `choose` must be on a code path that isn't logically executed. For each `choose` statement we have two expressions: the predicate of the `choose` itself, and the control-flow expression at that point in the method. Let's call the `choose` predicate $P(\bar{x}, \sigma)$ and the control flow predicate $C(\sigma)$. Then the correctness condition for a given choose is:

$$\forall \sigma \exists \bar{x} \ : \ C(\sigma) \rightarrow P(\bar{x}, \sigma)$$

The SAT formulation is:

$$\exists \sigma \ : \ \forall \bar{x} \ : \ C(\sigma) \wedge \neg P(\bar{x}, \sigma)$$

If the SAT-solver finds an assignment, that means there was a legal state in which the `choose` predicate is always false (along control paths that reached the `choose` statement).

8.2.2.1 SAT-friendly form

Although the states are explored dynamically by the SAT solver, we know ahead of time all the state variables that could possibly be accessed by the `choose` predicate. Therefore, much like universally-quantified invariant expressions, we can just expand the nested $\forall$ into a conjunction of all possible combinations. Then the condition simplifies to:

$$\exists\sigma \ : \ \bigwedge_{\bar{x}} C(\sigma) \wedge \neg P(\bar{x},\sigma)$$

8.2.3 Invariant Maintenance

The other proof conditions assume that the defined methods are correct with respect to the invariants. It is possible to prove that the methods maintain the invariants. A method maintains an invariant if, for all legal input states and arguments the output state doesn't violate the invariant. Assuming that INVARIANTS is the set of all defined invariants, and that an invariant, *inv* is a predicate that takes a state and arguments, and σ_{out} is the state after executing a method, we can formalize the invariant maintenance condition as:

$$\begin{aligned}&\forall\, m_j \in \text{METHODS} \ : \\ &\forall \sigma, \bar{x}, \sigma_{out} \ : \ \sigma[m_j(\bar{x})]\sigma_{out} \wedge \left[(\textstyle\bigwedge_{inv_i \in \text{INVARIANTS}} \ inv_i(\sigma,\bar{x})) \implies inv(\sigma_{out}) \right]\end{aligned}$$

8.2.3.1 SAT-friendly form

The SAT formulation will use the SAT solver to try to find an initial state or set of arguments that will cause the output state to violate the invariant. For a given method and invariant, the formula would be:

$$\exists \sigma,\, \bar{x} \ : \ (\textstyle\bigwedge_{inv_i \in \text{INVARIANTS}} \ inv_i(\sigma,\bar{x})) \wedge \neg inv(\sigma_{out})$$

Since the proofs of invariant maintenance are independent for each method, they can be developed as separate SAT problems and solved separately (and this is what the current prototype does).

8.3 Performance questions

Up to this point, I have described correctness conditions. It is actually possible to retool the ACCLAM verification machinery to answer questions more related to performance. Of course, for true performance information, we'd have to model the system that would be executing this model. As a proxy for more concrete performance details we'll use precision. A *precise* predicate is one that exactly describes the states for which it is true. Over-approximation may produce a correct predicate, but it may be true in many situations where it needn't be. Using "precision" this way is reasonable if one considers conventional locking methodologies. A standard mutex can correctly prevent concurrent modification, but it may be overly restrictive. Many practical systems use a more precise lock, the read-write lock. It has more complex semantics than a simple mutex, but it can allow reader parallelism, which can boost the performance of data structures where readers tend to outnumber writers. In the context of ACCLAM the read-write lock can be considered to be more precise than the simple mutex for read operations.

8.3.1 Conflict Predicate Precision

Conflict predicates allow the modeler to reason about the commutativity of the data structure as a function of state. A more precise conflict predicate more exactly describes the division between commuting and non-commuting operation invocations. To test for conflict predicate precision, we need a condition that describes invocations that commute but for which the conflict predicate is true. Going back to the SAT-friendly form of conflict predicate correctness:

$$\exists \sigma_i, \bar{x}, \bar{y} \ : \ \mathit{NonComm}(\ldots) \wedge \neg P(\ldots)$$

Remembering that $NonComm(\dots)$ is the raw state-comparison expression that is true if any part of the final state depends on the order of method invocation, we can infer that if the methods commute under the state, then *NonComm* will be false. We just need to rearrange the expression a little to get an expression that is true for all the commutative states in which the conflict predicate is conservatively true, something like:

$$\exists \sigma_i, \bar{x}, \bar{y} \ : \ P(\dots) \wedge \neg NonComm(\dots)$$

This statement is true in states where the conflict predicate is true, but the raw state-comparison commutativity function shows that the methods actually commute.

8.3.2 Lock Tightness

The more precise a lock is, the more concurrency it may allow. Therefore, in general terms, a more precise lock may permit higher performance. Of course, in practice there may be a trade-off between the computational overhead of performing highly precise lock calculations and the incremental improvement in concurrency. However, there is benefit in understanding those situations for which a model's locking protocol may be sub-optimal. At the very least, the modeler will be able to see the trade-offs made in the concrete design. More generally, a model may be iteratively adjusted over its lifetime to refine locking protocols when profiling reveals that reduced concurrency is responsible for performance degradation (and that the locks are not tight).

We can do the same kind of inversion of the correctness condition here to yield a precision condition. Revisiting the lock correctness condition:

$$\exists \sigma_i, \bar{x}, \bar{y} \ : \ NonComm(\dots) \wedge \neg LockCon(\dots)$$

We can do a little re-arranging to get:

$$\exists \sigma_i, \bar{x}, \bar{y} \ : \ \neg NonComm(\dots) \wedge LockCon(\dots)$$

This statement is true for states where the locks conflict, but the methods actually commute.

CHAPTER 9

TOOL DESIGN AND PERFORMANCE

The ACCLAM verification tool is a Java program that acts as a compiler for the ACCLAM language. Rather than compiling into an executable composed of machine code, the Verifier (the verification tool) converts a circuit representation of an ACCLAM model into a conjunctive normal form SAT problem. This SAT problem is then fed into a freely available SAT solving library (sat4j [42]).

9.1 The Verifier's Front End

The Verifier is structured similarly to many modern language processors. The front end is composed of phases that are executed in a pipelined fashion. The parser for ACCLAM is written in ANTLR (an LL(k) parser-generator). The resulting abstract syntax trees are fed into a name resolution pass that tags every symbol it can find with the declaration of that symbol. The next pass is a type-checker that implements the judgments described in the static formal semantics. The output from all this is a fully resolved, type correct ACCLAM program. It still contains statements, and still needs to be turned into a circuit-based program before it can be fed to the SAT solver.

9.1.1 Expansion into circuit expressions

The part of the tool that takes a statement-rich ACCLAM program and processes it into a collection of composable pure expressions is called the 'Expression Expander'. Its odd naming aside, it basically performs, in code, the steps laid out in the formal circuit semantics. The expander processes an ACCLAM description in a modular fashion, treating each model and method as an independent module. What this means is that it converts a method into an expression-based circuit with inputs and outputs so that they can be more readily stitched together. An important point to remember is that instance variables (the members of the description itself) may also have to be considered as inputs, because in general, it may be necessary to construct circuits that perform the same operation but on multiple different instances.

The expander does a form of flow-analysis to handle variable assignment, and this is also expressed in the circuit semantics. Method invocations are inlined directly (their circuit expansion is wired in), which is the main reason why ACCLAM doesn't have support for recursive functions. Interestingly enough, the lack of recursion didn't hamper our ability to describe non-trivial data types. The `forall` construct permitted us to express many of the semantics for which standard programming languages would use iteration or recursion.

The expander predates much of the formal understanding of the circuit-based semantics. I have found that I often need to build prototypes in order to explore the problem enough to get a feel for it. Therefore, much of the circuit semantics is a cleaner, more formal rendition of the internal processes within the expression expander. For example, the control-flow variable/bit is a fundamental part of how the expander works. As the expander flows through a statement block, it very carefully keeps track of the expression that represents the state needed to reach each point in the code. When a variable's value must be computed (when converting an assignment), it is wrapped in the control-flow expression.

9.1.2 Relations and scalarization

From the point of view of the modeler, and the language of ACCLAM itself, a relation can be viewed as a massive (possibly multi-dimensional) array. However, in order to practically represent a relation in a real system in a feasible way, the verifier employs a technique we call 'scalarization'. Scalarization is turning a relation of N-indexes into a function of

N arguments whose body is a chain of conditional expressions (if-then-else). A relation `boolean rel[T, T, T]` would become something like:

```
boolean rel(T idx1, T idx2, T idx3) {
  if (idx1 = <val1> && idx2 = <val2> && idx3 = <val3>) {
    rel_element1;
  } else if (...) {
    rel_element2;
  } else {
    rel_element_initial;
  }
}
```

After working on this for a while, we discovered that it is very similar to an older transformation known as 'ackermanization' (or the Ackerman reduction) [1]. Scalarization leverages the order of evaluation for a conditional expression to implement successive assignments. Each assignment to a relation essentially wraps the previous expression in a new conditional expression with the old expression as the *else* case. `rel[1, 2, 3] = true;` would become something like:

```
boolean rel'(T idx1, T idx2, T idx3) {
  if (idx1 == 1 && idx2 == 2 && idx3 == 3) {
     true;
  } else {
     rel(idx1, idx2, idx3);
  }
}
```

These transformations are formalized in the formal semantics (chapter 7), however I often find that an informal description with examples can be very beneficial to understanding a language processing system.

9.1.2.1 Relations within `forall`

`forall` is a special case. The main purpose of a `forall` statement is to model a parallel update that assigns values to all indexes in a relation that match some predicate function. This makes up for not having a more general recursion or iteration construct in ACCLAM. Like more pedestrian assignment, `forall` wraps the prior definition in a new condition. However, the conditional expression itself is a transformed version of the forall's predicate.

```
boolean friends[T x, T y];

. . .
void unfriendTotally(T user) {
  ...
  forall (T x ; friends[x, user]) {
    friends[x, user] = false;
  }
  ...
}
```

Figure 9.1. An Example of Forall Usage

Given the example in Figure 9.1, if we assume that the prior definition of the `friends` relation was `friends_prior`, this would become:

```
boolean friends(T idx1, T idx2) {

  if (friends_prior(idx1, user)) {

     false;

  } else {

     friends_prior(idx1, idx2)

  } }
```

It's straightforward to extend this to handle predicates that mention completely different relations. For example:

```
boolean even[int];
boolean odd[int];

...

forall(int x ; even[x]) {
   odd[x] = false;
}
```

would become something like:

```
boolean odd(int idx1) {
  if (even(idx1)) {
     false;
  } else {
     odd_prior(idx1);
  }
}
```

As long as the relation being updated is referring directly to the `forall` variables, the substitution is a straightforward replacement of `forall` variables with relation index variables. However, if the relation assignment is indexed by a function of the forall variables, then the tool must be able to invert the function in order to recover the values. Consider:

```
forall (T x ; P[x]) {
  rel[f(x)] = <constant>;
}
```

In order to build a conditional expression, we'd need to be able to convert the forall predicate to an expression in terms of the index variables of the relation. As long as `f` can be inverted (as `f'`), we can have:

```
rel (T idx) {
  if (P (f'(idx))) {
     <constant>
  } else {
     rel_prior(idx)
  }
}
```

The ACCLAM tool is fairly limited in functions it knows how to invert so it is possible to express a sophisticated, but legal, `forall` statement that the tool won't be able to process. This is just a limitation of the current tool, however. In my modeling efforts I haven't found this limitation to be particularly restrictive.

There is an additional correctness constraint over `forall` statements. The right-hand-side of the assignment might itself be an expression in terms of variables. In order for the tool (and the model writer) to reason about the meaning of a forall, it is vital that for any assignments in the body: `rel[f(x1,..., xN)] = g(x1, ..., xN);` $\forall \bar{x} = (x_1,...,x_N), \bar{y} = (y_1,...,y_N). f(\bar{x}) = f(\bar{y}) \rightarrow g(\bar{x}) = g(\bar{y})$ Because of this requirement, the tool can assume that it can always safely do the following transformation:

```
rel (idx1, ..., idxN) {
  if (P(f'(idx1, ..., idxN))) {
     g(idx1, ...., idxN);
  } else {
     rel_prior(idx1, ...);
  }
}
```

The tool processes relations in two basic phases. First, a set of all dereferences is built up and an initial scalarized function of all visible values is assembled. The second phase is to process assignments. The assignment processing is very order-dependent so it must be done carefully (e.g., `op1() ; op2()` will have a different relative order for assignments done in `op1` than for `op2() ; op1()`). In the case of conflict predicate and lock correctness, this means that two separate expressions will be generated for the two different final states of the relation, where the main difference will be the relative order of any assignment-related conditional expressions.

9.1.3 Reductions

Reductions are derived by mapping a predicate over a relation. The predicate maps relation values into elements of an Abelian group, and then aggregates that set of values into a single element of that group. At the moment, the tool supports only one kind of reduction (`count`), which operates over the group of non-negative integers.

There are two things the tool needs to do for reductions. One is to compute a correct initial value for the reduction. After all, if the SAT solver generates a legal state where a relation is populated by 3 values, the sum reduction had better be outputting 3. Since ACCLAM restricts states with invariants (rather than requiring the tool to produce entire histories starting with object construction), the tool needs to be able to process relation expressions and derive an expression for the initial state of the reduction that is correct. In practice, the tool uses the set of dereferenced indices from relation processing and maps the reduction value mapping over that set.

The remaining responsibility of the tool is to make sure that reductions stay in sync with their associated relation. This is actually fairly simple. Each normal assignment is wrapped in conditional adjustments to the reduction. Given a membership relation `in` and a reduction over it like:

```
boolean in[T];
int ct = count(T x ; in[x] ; 1 ; 0);
```

An assignment `in[ex1] = ex2;` would be expanded to something like:

```
if (in[ex1]) {
   ct = ct − 1;
}
in[ex1] = ex2;
if (in[ex1]) {
   ct = ct + 1;
}
```

Now the expression for `ct` will automatically be converted to an expression conditioned on the value of `in`.

Reductions over relations that are modified by a `forall` expression are very difficult to reason about. Except in cases where the predicate is completely pre-computable at tool processing time (a boolean constant, say), the tool can't figure out how many relation slots were modified by a `forall` update. Of course, if two relation expressions are equal for all possible index values, their reductions will also be equal, so the reduction delta for a `forall` update can also be assumed to be equal and set to the group's identity value. However, that isn't possible if the relation expressions differ from one another. Therefore the ACCLAM tool can't fully reason about reductions whose relations are updated in `forall` statements.

9.1.4 Invariants

ACCLAM allows modelers to specify invariants that define the legal states for the data type they are modeling. This is very powerful because it means the system can leverage SAT solvers to generate all legal states without having to evaluate all possible histories from some starting point. Furthermore, producing a set of histories is obviously computationally expensive, and it's not clear that it could possibly fully model an abstract data type, in general. ACCLAM supports both kinds of quantification, and so invariants can be specified using both existential (`exists`) and universal (`forall`) quantification.

Existentially quantified expressions are very straightforward in a SAT circuit. Each quantified variable becomes a top-level 'floating' variable composed of SAT literals that the SAT solver will be free to set. The challenge is that invariants tend to be expressed in terms of universal quantification, which a SAT solver would ordinarily struggle with. The tool solves this problem by relying on the fact that ACCLAM models are finite. For a given problem and model, the set of all variables of all types is completely known and guaranteed to be finite. This means that all the state, and therefore all the inputs to any expression being used, can be fully enumerated. Therefore, it is possible to expand a universally quantified statement into a conjunction of assertions about every element in that finite set of variables. The expansion-based approach requires that quantifiers not be alternated. If an `exists` were nested within a `forall`, that would require that variables be instantiated and that the SAT solver be forced to explore an essentially unbounded number of states, and that would make the model not provable by our techniques (except perhaps for data types with a very small number of possible values).

9.1.4.1 Forall Expansion

The ACCLAM tool performs `forall` invariant expansion by instantiating the body of the invariant expression for each possible variable that the model is using. This information is needed to build the CNF clauses, so it's readily available. The verification tool can filter down this set of variables by type, and further by performing a limited form of slicing [70] (to ensure that 'helper variables' like the relation function indexing variables aren't quantified over). The end result is a set of expressions that can be AND-ed together to yield a finite instantiation of a `forall` invariant.

Consider the example of a MultiSet. This is a set that permits elements to be included multiple times. Let's say the modeler chooses to model a multiset by having two relations. One relation maps elements to a boolean value and the other relation maps elements to an integer counter. The model would look something like:

```
model MultiSet<T> {
  boolean in[T];
  int counter[T];
  ...
  void add(T addObj) {
     in[addObj] = true;
     counter[addObj] = counter[addObj] + 1;
  }
}
```

We can start to reason about the invariants that this model should have. If an element is in the set, the counter should be greater than 0. That would be expressed like:

```
invariant forall(T x ; in[x] → counter[x] > 0);
```

In the state after `add(x) ; add(y)`, we can instantiate this invariant over all instances of type T, which results in an expression like:

$$in[x] \rightarrow counter[x] > 0 \wedge in[y] \rightarrow counter[y] > 0$$

In practice, the invariant would also be instantiated over any initial state that was also generated by the ACCLAM verification tool.

9.1.5 Predicate construction

The verification tool processes the model, converting the methods into expressions that can be chained together. Then, these are used by the tool to assemble what is called 'The

Predicate', which is the top-level expression that expresses the proof condition we'd like the SAT solver to tackle. ACCLAM permits the modeler to specify conflict expressions that involve states from any point in the execution of a pair of methods. The verifier will record the expressions for each member of the state before and after method invocations. This means that the tool can expand any expression in the predicate in terms of the state before or after any method invocation. Because the mutations specified in the model have been turned into expressions, it is possible to model complex predicates that require access to intermediate states and return values even though it may be inefficient or impossible to implement such state sampling in a practical system. In this way ACCLAM can model both practical and currently impractical systems.

As an example, consider the predicate that would have to be assembled for conflict predicate verification. Given two methods: op_1, op_2, the full predicate would be: $(\sigma_{12} \neq \sigma_{21} \vee return^1_{12} \neq return^1_{21} \vee return^2_{12} \neq return^2_{21}) \rightarrow conflict$. To assemble this expression, the verifier builds up state expressions for each state member of the model for both method orders. The return expressions are chained off of the initial state or prior method's final state. Then this expression is joined by implication with the modeler-supplied conflict predicate, which itself can depend on state variables sampled from the initial, intermediate, and final states.

9.2 The Verifier's Back End

The products of the verifier front end are: a set of state variables, a state-sampling predicate assembled on top of those variables, and a collection of invariant expressions defined in terms of those variables. The resulting expressions are composed of:

1. Variables

2. Basic boolean operations (AND, OR, XOR, bit-inversion)

3. Basic logic operations (NOT, ==, $\rightarrow$)

4. Basic arithmetic operations (integer addition, subtraction, etc.) and shifting operations (left, right, arith. and logical)

5. If-then-else expressions

The back end's main job is to convert these expressions into a circuit form that can be directly converted into a conjunctive-normal-form that a SAT solver can consume. Because there tends to be a lot of redundancy and recomputation in these expressions, a great deal of benefit can be gained by attempting to simplify and share sub-expressions.

9.2.1 Expression Conversion

ACCLAM is a strongly-typed language based on, and attempting to model, Java-like languages. Therefore, each variable in ACCLAM is well-typed, and because of this Java heritage the exact bit-width of every type is known. Variables are converted to tuples of bits (each one of which will eventually become a SAT literal).

Boolean operations are handled by building circuits to manipulate the bits directly. Logical operations either operate on boolean (single-bit values) already, or take two variables and compare their constituent bits to produce a single-bit output.

Arithmetic operations are handled by building conventional combinatorial circuits. For example, addition is modeled by building a ripple-carry adder circuit of the appropriate width. Shifting operations are handled similarly.

Conditional expressions (if-then-else) are modeled as multiplexers. The then and else clauses must produce the same type, therefore they can be treated as the two inputs to a 2-way multiplexer. Because the condition is necessarily a boolean expression, its output is connected to the single-bit selection input to the 2-way multiplexer.

9.2.2 Circuit Simplification

The verifier's front end assembles expression fragments and instantiates them and wires them together in different orders to build up compound expressions that fully model a state through multiple method invocations. This reuse induces a lot of redundancy within the expressions. The front end also doesn't perform a lot of optimizations, so there are also a lot of expressions that have simpler, more compact forms (e.g., $x \wedge x \Rightarrow x$). Therefore, the back end first takes all the expressions produced by the front end and does some basic constant propagation and circuit simplification.

The expressions are all pure, so the back end can just do a static flow analysis to propagate values through the expressions. Although it is referred to as a constant propagator in the code, the propagator also propagates facts about conditional expressions as it visits each arm of an if-then-else. That is, as the then clause is visited, the conditional expression will evaluate to `true` (`false` for the else clause). The front end tends to produce nested conditional expressions, so the constant propagator handles nested expressions (using a stack). As each expression is visited by the constant propagator, another component examines the resulting expression and applies boolean peephole simplification. This peephole optimizer has a set of basic boolean simplification rules (e.g., $x \vee true \Rightarrow true$) that it tries to apply to the expression to reduce its size.

9.2.3 Circuit Memoization

Even after a circuit is simplified at the expression level, two different expressions may share one or more sub-expressions. This is very likely given the way the front end stitches together expressions (although there is no explicit labeling or plumbing within the tool to provide this information to the back end). The back end detects shared subexpressions dynamically via a process I call 'circuit memoization'. As each expression is fed into the code that converts it to SAT form, the expression is hashed and used to look up prior circuits. If there's a hit, and the original expression matches the current expression, the circuit builder will just use the old circuit rather than construct a new one. This is totally safe because these circuits are pure (no register-like state). This memoization can greatly reduce the size of the resulting SAT problem, and although there is no guarantee that a smaller SAT problem will be easier to solve, reducing the number of clauses tends to speed up SAT solving.

Because the back end always runs circuit memoization, the circuit simplification pass attempts to order expressions so that they'll tend to be recognized as duplicates. For example, `a | b` is the same expression as `b | a`, but a lossy process such as hashing their string representations may not catch this fact. Therefore, the circuit simplifier will rewrite the second expression instance so that it will tend to hash to the same as the first. This has the added benefit of making the peephole rule sets smaller (because constant values will always be in a fixed position relative to non-constant values).

To illustrate the importance of simplification and memoization, I ran the large model `ParamOrderedSet` through the verification tool with simplification and memoization disabled. The `remove` method with memoization and simplification enabled, produces a SAT problem with over 72 thousand variables and almost 4 million clauses. Without simplification or memoization, there are over 224 thousand variables and almost 6.25 million clauses. The results across all methods for conflict predicate verification are summarized in Table 9.1. In that table, 'diff' refers to the difference between the non-simplified and simplified versions, and 'ratio' is the ratio of the non-simplified to simplified versions (to help illustrate how much larger the non-simplified versions can be). The numbers in the table speak for themselves, and illustrate that the problem size is greatly reduced by these optimizations. Additionally, I had to increase the memory allowed to the SAT solver to 1.5GB (the default was 512MB) in order to load and run the larger SAT problems. Interestingly, the larger problems are also harder for the SAT solver to simplify and solve quickly. The simplified and memoized problem versions had 31 out of 64 solved via a contradiction that was detected as the problem was being loaded into the SAT solver (shows up as 0 time spent). However, the non-simplified non-memoized problems had only 11 out of 64 that were similarly solvable. Therefore, to tackle any reasonably complex modeling problem, it is important to be able to optimize the expressions and circuits before handing them over to a SAT solver.

	avg diff	max diff	min diff	avg ratio	max ratio	min ratio
SAT vars	52650	161248	12969	6.38	15.59	3.00
SAT clauses	186200	2254131	8286	1.23	1.69	1.04
time (ms)	4161	81746	0	568	16255	0

Table 9.1. Summary of Non-simplified Problem Results for ParamOrderedSet

9.3 Verification via SAT

The back end consumes pure expressions and produces a set of boolean variables and a collection of CNF clauses. These are then handed to a third-party SAT solving library (sat4j) that consumes the CNF form and runs a SAT solver on it. sat4j includes several modern optimized SAT solvers (such as zchaff [48]) as well as a clause pre-processor that can detect contradictions while processing the CNF clauses. What this means is that the SAT solver tends to complete in a few seconds, but can sometimes detect UNSAT without doing any variable exploration at all.

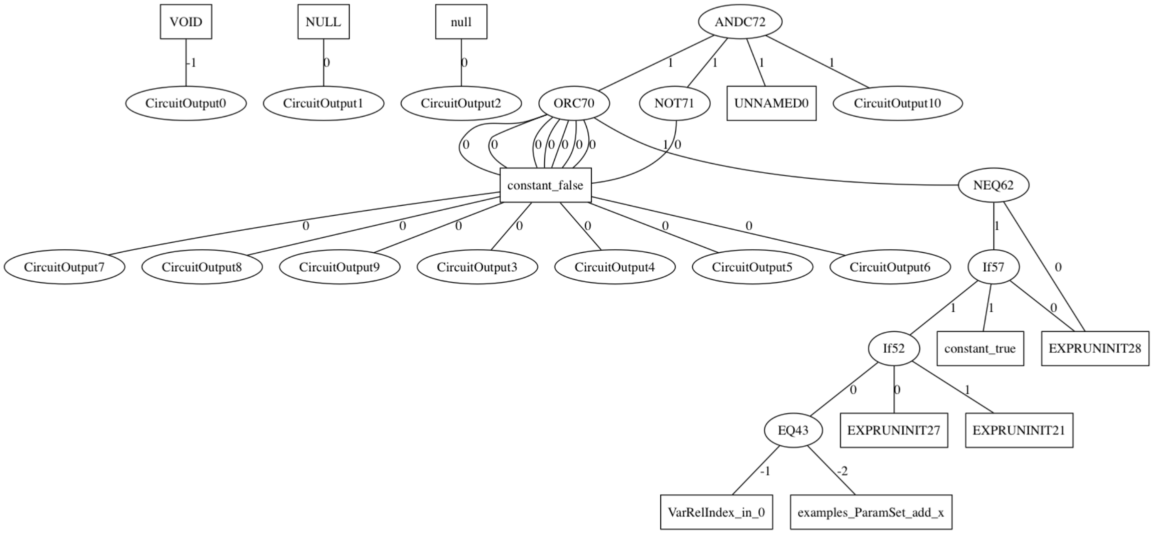
For the safety proof conditions we're interested in, the predicates are structured so that UNSAT will mean that the condition held in all legal states, and SAT will mean that the condition did not hold. Moreover, because SAT is being used as a failure case, we can take the assignment produced by the SAT solver and use it as a counter-example to display to the modeler.

9.3.1 Using a counter-example

The verification tool can take a counter-example and map the values onto inputs and outputs for the various circuit elements that were used to build the CNF problem. This can then be used to produce a DOT file which can be processed to produce a graphical form of the expression circuit. The circuit operations (AND, etc.) are the vertices and the edges are the variables flowing between the operations (which are labeled with the assignments the SAT solver gave them). The operations can be mapped back onto the original expressions by examining the tool output. In the future, it would be more convenient to just plumb these tags through to the DOT-file generation.

In the example output in Figure 9.2, we can see the circuit structure for the `ParamSet` model doing conflict checking for `add(x) * find(y)`. At the top of the graph is the big conjunction that conjoins the state comparison subcircuit (the ORC), the conflict predicate subcircuit (the NOT), and the invariant subcircuit (which in this case was simplified to a constant 1). The state comparison subcircuit ORs a whole bunch of expressions that had been simplified to false. This is because the return result of one of the methods is `void` and `void != void` is always false. Other trivially true state comparisons were simplified to false as well. The only state comparison that wasn't simplified to a constant was the expression for the return value comparison for `find`. The subcircuit compares the result of reading the initial state of the relation `in` with the state after `add` has mutated it. The structure of the relation expression is a chain of if-then-else expressions. The incorrect conflict predicate for this case is `false`. When the `in` relation differs, the return value comparison is true, which forces the state comparison OR also to be true.

Figure 9.2. An Example Output Graph of an Assignment



9.4 Limits

ACCLAM can prove useful things about models of real data structures. And the problems discussed so far have all been very tractable and within the ability of a zChaff-derived SAT solver to solve on laptop hardware. However, the verification tool does produce SAT problems, and there is a limit to how large those problems can be before the tool has difficulty producing results in a reasonable amount of time, or at all. The largest contributor by far to problem size are the invariants. Because universally quantified invariants are expanded into an exhaustive set of conjoined covering expressions, it is easy to see how they can grow to overwhelm the SAT solver.

As an example, I added the following invariant to the `ParamOrderedSet` model:

```
invariant forall(T x, T y, T z;
      (in[x] && in[z] && x = prev[z] &&
             order.lt(x,y) && order.lt(y,z)) →
                (next[y] = z && prev[y] = x && !in[y]));
```

I didn't modify any of the state or method definitions. Since this invariant instantiates three variables, it will be the largest invariant in the model. Running the verifier to check conflict predicates resulted in a number of Java OutOfMemory exceptions (by default, I would run with 512MB). Only by raising the memory limit to 1.5GB could I get the SAT solver to terminate on the larger problems. Table 9.2 summarizes the results when running the model with this large invariant. Both memory and time requirements increase; however, the average increase in time was larger than that in memory. This is not surprising, and one would expect that each additional clause or variable added would tend to increase the running time at a greater rate than the memory used.

9.5 Results

We ran our suite of models through the verifier tool to get a range of results for inverse, conflict predicate, and lock protocol verification. Additionally, I prototyped a conflict pred-

	increase			ratio		
	avg	max	min	avg	max	min
SAT vars	393	2303	34	1.03	1.08	1.01
SAT clauses	724553	9331390	16702	1.93	3.34	1.35
time (ms)	1053	30079	0	2.16	13	0

Table 9.2. Summary of SAT Results for ParamOrderedSet with Large Invariant

icate precision and lock protocol precision verifier and ran a less-precise version of the `Set` model through it. The tables below summarize the timing and SAT statistics with average, minimum and maximum values. The full results appear in Appendix A. SAT statistics are a rough metric that the SAT community uses to gauge problem size, and they consist of a variable count and a clause count. Of course, this is only an approximate way of measuring problem hardness, but it is a convenient way of comparing the complexity of different models.

The verification tool is implemented in Java, and is single threaded. The SAT verification was done with sat4j 2.0.5. The results were generated on a Macbook Pro laptop (2.2 GHz Intel Core i7 CPU, 8GB 1333 MHz DDR3 RAM). The JVM was limited to 512 MB of main heap size.

9.5.1 Data types Modeled

- *IntCell* An atomic integer container
- *Map* A type-parameterized Java-style map
- *MultiMap* A map that supports multiple values for a key
- *Set* A type-parameterized set of values
- *MultiSet* A set that permits multiple identical elements
- *Equality-parameterized Set* A set where equality is defined by a type-parameterized relation

- *Equivalence-class based Set* A set where equivalence is defined by an external model
- *Canonical element based Set* A set where equivalence is defined by an equivalence class represented by a canonical element
- *Set with Iterators* A parameterized set with an incremental and snapshot iterator
- *Total Order* A model of a total order
- *Partial Order* A model of a partial order
- *Ordering-parameterized Ordered Set* An ordered set, parameterized by an instance of a partial order
- *Queue* A type-parameterized LIFO data structure
- *Stack* A type-parameterized FIFO data structure

9.5.2 Inverse Results

For space reasons the timing and SAT statistics are presented in two separate tables (9.3 and 9.4). The timing results are interesting because no inverse verification took over a second. The SAT statistics show that for most models, the variable and clause count is fairly small. `ParamOrderedSet` is definitely the outlier here. Not only is it the most complex model, with the most abstract state, it also has the most invariants. And, in fact, the invariants are responsible for most of the clauses.

Model	Avg Time (ms)	Max time (ms)
Stack	77	154
Equivalence	0	0
PartialOrder	0	0
Map	35.8	107
ParamOrderedSet	24.88	105
ParamSetIterators	4	16
IntCell	0	0
MultiMap	69.6	244
Queue	101	218
MultiSet	226.9	810
Set	0.29	1
ParamSet (canon)	4.67	7
ParamSet (Equiv)	0.33	1
ParamSet (equ)	1.67	3

Table 9.3. Timing Summary for Inverse Verification

	Vars			Clauses		
Model	Avg	Min	Max	Avg	Min	Max
Stack	6598	3123	8352	28003	9311	38955
Equivalence	2022	2022	2022	5633	5633	5633
PartialOrder	1294	1294	1294	3785	3785	3785
Map	2411.2	1138	4337	8158	2558	16590
ParamOrderedSet	12432	2916	37030	379586	48751	1287310
ParamSetIterators	4423	2791	7132	11927	7498	18424
IntCell	103	103	103	107	107	107
MultiMap	5734	1710	11972	44192	3861	144402
Queue	17030	9821	21639	120560	46409	176419
MultiSet	8456	1420	13778	70980	3158	160161
Set	783	297	1147	1972	557	3033
ParamSet (canon)	2533	1235	3183	56895	4625	85957
ParamSet (Equiv)	542	104	761	1693	109	2487
ParamSet (equ)	3048	2791	3177	8766	7498	9401

Table 9.4. SAT Statistics for Inverse Verification

9.5.3 Conflict Predicate Results

The conflict predicate timing results (Table 9.5) show a bit more spread than the inverse results. The average time is still under a second, but the maximum times are starting to creep up into tens of seconds. The reason for this large increase is illustrated in the SAT statistics (Table 9.6), where we see that the problem sizes are roughly twice as large as in the inverse case. Twice as many clauses tends to translate into much more than twice as much time running the SAT solver, so the time increases make sense.

Model	Avg time (ms)	Max time (ms)
Stack	4.5	50
Equivalence	0	0
TotalOrder	0	0
PartialOrder	0	0
Map	40.1	398
ParamOrderedSet	650	23300
ParamSetIterators	3.20	146
IntCell	0.428	5
MultiMap	51.20	1240
Queue	70.80	649
MultiSet	272	5300
Set	3.410	34
ParamSet (canon)	31.4	43
ParamSet (Equiv)	0.35	1
ParamSet (equ rel)	13.7	50

Table 9.5. Timing Results Summary for Conflict Predicate Correctness

9.5.4 Abstract Lock Results

The abstract lock correctness timing results (Table 9.7), show better performance than the conflict predicate results. Of course, `ParamOrderedSet` remains the high-end outlier, but it is the only model to take over 2 seconds for any particular verification step. The average time is also smaller. Interestingly enough, looking at the SAT statistics (Table 9.8), they aren't radically smaller than the conflict predicate problems (`ParamOrderedSet`'s statistics are actually worse). Of course, a SAT problem with more clauses is not inher-

	Vars			Clauses		
Model	Avg	Min	Max	Avg	Min	Max
Stack	7389	2561	12540	29980	7765	63490
Equivalence	1013	380	1646	149400	13122	285666
TotalOrder	1426	381	1664	373900	14970	515130
PartialOrder	1407	378	1643	327000	12198	450784
Map	2030	506	6693	8121	999	36175
ParamOrderedSet	13760	2514	72443	511100	47653	3992170
ParamSetIterators	2061	514	20669	11640	1017	252835
IntCell	677.7	115	1110	1532	126	2501
MultiMap	4963	537	19143	30270	1031	199364
Queue	20690	8201	33121	201100	42076	386965
MultiSet	9556	1241	27079	69740	2733	277447
Set	2214	504	4602	8288	996	21081
ParamSet (canon)	9165	4182	10033	462700	76766	608690
ParamSet (Equiv)	468	117	1197	1359	129	4449
ParamSet (equ rel)	3564	389	10034	161600	2100	607675

Table 9.6. SAT Statistics Summary for Conflict Predicate Correctness

ently a harder problem, so we are perhaps being misled by what is, at best, an approximate metric. The expressions generated for the lock correctness proof involve only the state available at method entry, and in most cases involve only the method arguments. The overlap predicates are also simple. So the lock predicate in most cases won't involve evaluation of relation or reduction state. These simpler expressions end up generating sets of clauses that tend to be more tractable.

9.5.5 Conflict Predicate Precision

I extended the conflict predicate correctness verifier to wire up the final circuit to test predicate precision. I ran the `Set` model through with a conflict predicate specification where all the mutating operations had their conflict predicate as `true` (correct, but very imprecise).

Model	Avg time (ms)	Max time (ms)
Stack	0	0
Map	0	0
ParamOrderedSet	1707	21140
ParamSetIterators	4	180
IntCell	0	0
MultiMap	104	1470
Queue	0	0
MultiSet	300	1939
Set	6	55
ParamSet (canon)	14	66
ParamSet (Equiv)	1	2
ParamSet (equ)	0	0

Table 9.7. Timing Results Summary for Abstract Lock Correctness

9.5.6 Lock Tightness

I extended the lock correctness verifier to wire up the predicate for lock tightness and tested it on `Set`. I replaced all the point-wise locks with `Everything` locks (basically a global mutex) to test the verifier. Interestingly, the tightness verifier also detected that the point-wise locks were imprecise. This is true for many of the mutating operations if the initial state and final state would be the same. For example, `add * add` when the value is already in the set will commute. Therefore pessimistically locking is technically imprecise.

9.5.7 Invariant Maintenance

I created a prototype for proving invariant maintenance and tested it on the model with the most invariants (`ParamOrderedSet`). Most of the problems were quickly solved by the SAT solver. The results are summarized in Table 9.13.

	Vars			Clauses		
Model	Avg	Min	Max	Avg	Min	Max
Stack	7390	2561	12535	29980	7765	63490
Map	2030	506	6693	8013	999	35725
ParamOrderedSet	23090	9023	84794	760500	151674	5554487
ParamSetIterators	2061	514	20669	11700	1017	252835
IntCell	291	114	529	548	125	1084
MultiMap	5827	1544	20579	33680	3300	207350
Queue	20690	8203	33125	200900	39558	387781
MultiSet	9548	1241	27079	69820	2733	277447
Set	2214	504	4601	8361	996	21081
ParamSet (canon)	9169	4182	10044	464500	76766	615305
ParamSet (Equiv)	511	117	760	1507	129	3468
ParamSet (equ)	763	390	992	12350	2101	23781

Table 9.8. SAT Statistics Summary for Abstract Locking Correctness

Model	Avg time (ms)	Max time (ms)
Set	2.5	17

Table 9.9. Timing results for conflict predicate precision of the Set model

	Vars			Clauses		
Model	Avg	Min	Max	Avg	Min	Max
Set	2168	504	4602	7947	996	20631

Table 9.10. SAT statistics for conflict predicate precision of the Set model

Model	Avg time (ms)	Max time (ms)
Set	2.7	15

Table 9.11. Timing results for lock tightness of the Set model

	Vars			Clauses		
Model	Avg	Min	Max	Avg	Min	Max
Set	2215	505	4603	8179	998	20633

Table 9.12. SAT statistics for lock tightness of the Set model

		Vars		Clauses	
Model	Avg time (ms)	Avg	Max	Avg	Max
ParamOrderedSet	2	6937	23380	243800	1202662

Table 9.13. SAT statistics and timing for invariant maintanence

CHAPTER 10

CONCLUSION AND FUTURE WORK

10.1 Conclusion

We began with an overview of Transactional Memory as a promising implementation technique for concurrent systems. We also described the limitations inherent in conventional closed-nesting TM systems, as well as several proposals for overcoming those limitations (open nesting and boosting). All such techniques require that the TM run-time be given enough information to correctly perform concurrency control for the extended system. This has traditionally meant that a programmer that wants to use open nesting or boosting must annotate their code with concurrency and commutativity properties. The correctness of these properties is paramount, but they are non-trivial to reason about. We proposed here that a solution to this problem would be a machine-verifiable form for specifying the properties of the open nested or boosted data structure. We proposed a modeling language, ACCLAM, that allows implementers to prove properties about the abstract state of their data structures. ACCLAM provides structures that allow the programmer to specify abstract state and methods on that state, and we presented several examples to demonstrate that even though ACCLAM has no looping or recursion, it can be used to model practical, real-world data structures.

We described formally the structure and semantics of ACCLAM, proved the soundness of the language, and described a pure ‘circuit semantics’ for the language. This description is general enough that it can be used to produce an implementation of a language processing tool for ACCLAM. We also described the design and implementation of such a tool as well as results of feeding a suite of example models through the tool to verify correctness properties of conflict predicates, operation inverses, and locking protocols. The results illustrated that after some expression simplification and circuit memoization techniques, the resulting SAT problems were all tractable, many being solved in a matter of milliseconds. Formulating correct commutativity conditions is one of the great hurdles to adopting an STM extension such as open nesting. However, we demonstrated that a modeling language such as ACCLAM can be used to describe the abstract state of many practical data structures, and that those descriptions can be turned into a form that is fully machine verifiable. ACCLAM increases correctness of transactional code and reduces the overhead of adopting useful techniques like Open Nesting and Boosting.

10.2 Future Work

Broadly, there are several main directions for future work. One is expanding the scope of the verification tool to propose predicates and inverses rather than just verifying them, another is optimizing the way SAT problems are produced (for faster verification and verification of larger models), extensions to the language itself to enable more convenient or more precise description of abstract state, and finally ways of connecting the ACCLAM-specified abstract state with actual concrete implementations.

10.2.1 Generating Protocols and Inverses

The verification tool can be used to answer both correctness and precision questions. If there were a system that could propose locking protocols, perhaps by iteratively refining them, it could first check that the proposed protocol is correct, and then check the tightness of the new protocol.

10.2.1.1 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. This will require a verification tool that can process both lock correctness questions as well as lock precision questions. The precision predicate would have to be extended in order to compare two imprecise protocols. For example, for a map both a mutex and a read-write lock are imprecise, but the read-write lock is more precise (or less imprecise) than the mutex. The SAT formulation presented in Chapter 8 wouldn't provide any insight into relative imprecision. One would either need to formulate a SAT-friendly comparison predicate, perhaps employ a SAT variant (like MaxSAT), or something that isn't SAT like at all.

10.2.1.2 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). One has a similar difficulty in that it is necessary to compare incomplete inverses and somehow be able to determine that one is doing a better (although incomplete) job of inverting the state changes.

10.2.2 Optimization

The current tool employs a fairly basic set of expression simplifications, so there would be gains from adopting more powerful expression analysis. However, one of the largest gains may come from exploiting the fact that the total state space is finite and well-known. Currently, the tool models all types exactly as they are specified in the Java language specification. For example, an object is converted into a tuple of 64 SAT variables (to simulate a 64-bit pointer). However, if there are only ever 4 objects in the system, there is no benefit to modeling the entire 64 bit value space. Logically, for 4 objects and the `null` value, one may only need 3 bits. This can greatly reduce the SAT problem size.

10.2.3 Language Extensions

While implementing the tool concurrently with the example suite, many ideas for language features were proposed. Not all of the potentially useful ones could be included in the scope of this work. One idea was for a kind of assertion statement. It would be a modeler-specified proof request. The tool would generate expressions to verify that the assertion is true at that program point in all legal states. Additionally, the assertion expression could be assumed to be true for statements that happen after the assertion, which may lead to expression simplification.

Another extension that would be useful would be to improve the tool to allow it to understand lock acquisition in the middle of a method (rather than at method entry, as it currently is). Several of the example models could have had a more precise locking protocol if they could have grabbed locks conditionally within the body of the method itself.

10.3 Future Uses

This thesis presented ideas that have several potential uses. The main use is to model and prove correct abstract concurrency control properties for transactional data structures. We have demonstrated that it is possible to do this, and that the machine verifiable problems produced are tractable for many practical use cases. Therefore, ACCLAM or an ACCLAM-like system would be potentially useful to implementors of high-performance transactional code. Any programmer that needs to employ open nesting to interact with non-transactional code or improve performance is already in a position of having to reason about abstract concurrency control. It seems reasonable that any tool that simplifies that process would be welcome. Describing library code has the added advantage that as long as the interfaces don't change, the same model can prove things about different concrete implementations. Of course, this is the ideal and some implementation decisions may require model changes.

ACCLAM can prove that properties hold with respect to a given model. However, ACCLAM can't prove that a given implementation actually correctly implements a model.

This is often a desirable thing to demonstrate, so therefore another potential use for the ideas presented in this thesis would be to adapt something like ACCLAM to work with a system for proving properties about concrete implementations. This could allow expert implementors to co-evolve the model and implementation more seamlessly.

BIBLIOGRAPHY

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* North-Holland Publishing Co., Amsterdam, 1954.

[2] 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.

[3] 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.

[4] Egidio Astesiano. *Algebraic Foundations of Systems Specification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[5] 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.

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[7] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, April 2003.

[8] Jacob Burnim, George Necula, and Koushik Sen. Specifying and checking semantic atomicity for multithreaded programs. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 79–90, New York, NY, USA, 2011. ACM.

[9] 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.

[10] 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.

[11] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.

[12] 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.

[13] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[14] 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.

[15] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[17] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. Lock free data structures using STM in Haskell. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS'06, pages 65–80, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] 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.

[19] Guy Eddon and Maurice Herlihy. Language Support and Compiler Optimizations for STM and Transactional Boosting. In *Proceedings of the 4th International Conference on Distributed Computing and Internet Technology*, ICDCIT'07, pages 209–224, 2007.

[20] 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.

[21] Alan Fekete, Nancy Lynch, Michael Merritt, and William Weihl. Commutativity-based locking for nested transactions. *J. Comput. Syst. Sci.*, 41(1):65–156, August 1990.

[22] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs (summary). In *18th International Parallel and Distributed Processing Symposium*, IPDPS. IEEE Computer Society, 2004.

[23] 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.

[24] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In J. Gregory Morrisett and Manuel Fähndrich, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI, pages 338–349. ACM, 2003.

[25] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.

[26] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[27] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[28] Tim Harris and Keir Fraser. Language support for lightweight transactions. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 18th Annual ACM*

SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, OOPSLA, pages 388–402. ACM, 2003.

[29] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 14–25. ACM, 2006.

[30] Joseph M. Hellerstein and Michael Stonebraker. Anatomy of a database system. In *Readings in Database Systems*, pages 42–95. The MIT Press, 2005.

[31] Maurice Herlihy. *SXM: C# Software Transactional Memory*, 2005 (accessed August 20, 2014). http://research.microsoft.com/research/downloads/Details/6cfc842d-1c16-4739-afaf-edb35f544384/Details.aspx.

[32] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA, pages 253–262, 2006.

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

[34] Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2004.

[35] 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.

[36] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[37] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[38] Cliff B. Jones. Specification and design of (parallel) programs. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, IFIP Congress, pages 321–332, 1983.

[39] Deokhwan Kim and Martin C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. Technical Report MIT-CSAIL-TR-2010-056, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, December 2010.

[40] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *Commun. ACM*, 52(9):89–97, September 2009.

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

[42] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[43] Doug Lea. Overview of the collections package, April 1997. http://gee.cs.oswego.edu/dl/classes/collections/index.html.

[44] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[45] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Not.*, 41(1):346–358, 2006.

[46] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, New York, NY, USA, 2008. ACM.

[47] 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, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 359–370. ACM, 2006.

[48] 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.

[49] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, MA, 1985.

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

[51] 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 *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP, pages 68–78, 2007.

[52] Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 297–302. ACM, 2007.

[53] 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.

[54] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, November 1997.

[55] Michael F. Ringenburg 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.

[56] 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 *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP, pages 187–197, 2006.

[57] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[58] 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.

[59] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, 1984.

[60] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[61] 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.

[62] 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.

[63] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In Indranil Gupta and Roger Wattenhofer, editors, *PODC*, pages 338–339. ACM, 2007.

[64] 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.

[65] 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.

[66] 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.

[67] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–282, 1989.

[68] 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.

[69] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control.* Morgan Kaufmann Publishers, San Mateo, CA, 2001.

[70] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[71] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.

[72] Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In Chris Okasaki and Kathleen Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP, pages 175–188. ACM, 2004.

[73] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *Proceedings of the 2009 ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI ’09, pages 338–351, New York, NY, USA, 2009. ACM.

APPENDIX A

FULL RESULTS TABLES

A.1 Conflict Predicate Correctness with Good Predicates

This section summarizes in the following tables the SAT statistics for running our suite of example models through the language processing tool. These results are for conflict predicate correctness verification with known good predicates (i.e., the predicates are correct).

For the following results tables, the entry ‘C’ in the time column means that the SAT solver was able to detect a conflict while the clauses were being loaded. Therefore, the SAT solver didn’t technically run. ‘C’ is used rather than 0 to distinguish cases where the conflict was detected during loading as opposed to cases where the SAT solver did run, but finished in under a millisecond (modulo the precision of the JVM’s timer).

Table A.1. Results for model: Stack

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Stack.push * push	C	11062	59700
Stack.push * pop	C	12535	63490
Stack.push * size	C	5928	16445
Stack.pop * push	C	12535	63490
Stack.pop * pop	C	3522	9623
Stack.pop * size	C	6218	16445
Stack.size * push	C	5928	16445
Stack.size * pop	C	6218	16445
Stack.size * size	C	2561	7765

Table A.2. Results for model: Equivalence

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Equivalence.eq * eq	C	380	13122
Equivalence.eq * ne	C	1646	285666
Equivalence.ne * eq	C	1646	285666
Equivalence.ne * ne	C	380	13122

Table A.3. Results for model: TotalOrder

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
TotalOrder.eq * eq	C	383	14970
TotalOrder.eq * le	C	1619	412314
TotalOrder.eq * lt	C	1664	515130
TotalOrder.eq * ge	C	1619	412314
TotalOrder.eq * gt	C	1664	515130
TotalOrder.eq * ne	C	1664	515130
TotalOrder.le * eq	C	1619	412314
TotalOrder.le * le	C	381	14970
TotalOrder.le * lt	C	1619	412314
TotalOrder.le * ge	C	1576	296674
TotalOrder.le * gt	C	1619	412314
TotalOrder.le * ne	C	1619	412314
TotalOrder.lt * eq	C	1664	515130
TotalOrder.lt * le	C	1619	412314
TotalOrder.lt * lt	C	383	14970
TotalOrder.lt * ge	C	1619	412314
TotalOrder.lt * gt	C	1664	515130
TotalOrder.lt * ne	C	1664	515130
TotalOrder.ge * eq	C	1619	412314
TotalOrder.ge * le	C	1576	296674
TotalOrder.ge * lt	C	1619	412314
TotalOrder.ge * ge	C	382	14970
TotalOrder.ge * gt	C	1619	412314
TotalOrder.ge * ne	C	1619	412314
TotalOrder.gt * eq	C	1664	515130
TotalOrder.gt * le	C	1619	412314
TotalOrder.gt * lt	C	1664	515130
TotalOrder.gt * ge	C	1619	412314
TotalOrder.gt * gt	C	383	14970
TotalOrder.gt * ne	C	1664	515130
TotalOrder.ne * eq	C	1664	515130
TotalOrder.ne * le	C	1619	412314
TotalOrder.ne * lt	C	1664	515130
TotalOrder.ne * ge	C	1619	412314
TotalOrder.ne * gt	C	1664	515130
TotalOrder.ne * ne	C	383	14970

Table A.4. Results for model: PartialOrder

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
PartialOrder.eq * eq	C	379	12198
PartialOrder.eq * le	C	1598	360824
PartialOrder.eq * lt	C	1643	450784
PartialOrder.eq * ge	C	1598	360824
PartialOrder.eq * gt	C	1643	450784
PartialOrder.eq * ne	C	1643	450784
PartialOrder.le * eq	C	1598	360824
PartialOrder.le * le	C	378	12198
PartialOrder.le * lt	C	1598	360824
PartialOrder.le * ge	C	1555	259872
PartialOrder.le * gt	C	1598	360824
PartialOrder.le * ne	C	1598	360824
PartialOrder.lt * eq	C	1643	450784
PartialOrder.lt * le	C	1598	360824
PartialOrder.lt * lt	C	379	12198
PartialOrder.lt * ge	C	1598	360824
PartialOrder.lt * gt	C	1643	450784
PartialOrder.lt * ne	C	1643	450784
PartialOrder.ge * eq	C	1598	360824
PartialOrder.ge * le	C	1555	259872
PartialOrder.ge * lt	C	1598	360824
PartialOrder.ge * ge	C	378	12198
PartialOrder.ge * gt	C	1598	360824
PartialOrder.ge * ne	C	1598	360824
PartialOrder.gt * eq	C	1643	450784
PartialOrder.gt * le	C	1598	360824
PartialOrder.gt * lt	C	1643	450784
PartialOrder.gt * ge	C	1598	360824
PartialOrder.gt * gt	C	379	12198
PartialOrder.gt * ne	C	1643	450784
PartialOrder.ne * eq	C	1643	450784
PartialOrder.ne * le	C	1598	360824
PartialOrder.ne * lt	C	1643	450784
PartialOrder.ne * ge	C	1598	360824
PartialOrder.ne * gt	C	1643	450784
PartialOrder.ne * ne	C	379	12198

Table A.5. Results for model: Map

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Map.put * put	398	6368	34247
Map.put * size	0	1246	2958
Map.put * get	6	1860	6310
Map.put * remove	337	6530	36175
Map.put * containsKey	1	1358	3930
Map.size * put	C	1246	2957
Map.size * size	C	537	1031
Map.size * get	C	737	1461
Map.size * remove	C	1343	3086
Map.size * containsKey	C	506	999
Map.get * put	5	1700	4704
Map.get * size	C	737	1461
Map.get * get	C	737	1461
Map.get * remove	5	1668	4704
Map.get * containsKey	C	1346	2998
Map.remove * put	308	6530	36175
Map.remove * size	0	1342	3085
Map.remove * get	5	1828	6310
Map.remove * remove	297	6692	34891
Map.remove * containsKey	1	1358	3930
Map.containsKey * put	0	1358	3930
Map.containsKey * size	C	506	999
Map.containsKey * get	C	1346	2998
Map.containsKey * remove	1	1358	3930
Map.containsKey * containsKey	C	506	999

Table A.6. Results for model: ParamOrderedSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamOrderedSet.find * find	C	3617	93791
ParamOrderedSet.find * size	C	3617	93791
ParamOrderedSet.find * higher	C	9617	227028
ParamOrderedSet.find * insert	21	5313	195279
ParamOrderedSet.find * add	1	5315	195282
ParamOrderedSet.find * isEmpty	C	3617	93791
ParamOrderedSet.find * remove	1	7134	525362
ParamOrderedSet.find * delete	0	5313	195279
ParamOrderedSet.size * find	C	3617	93791
ParamOrderedSet.size * size	C	2514	47653
ParamOrderedSet.size * higher	C	7356	136027
ParamOrderedSet.size * insert	1	4352	95744
ParamOrderedSet.size * add	1	4354	95747
ParamOrderedSet.size * isEmpty	C	2514	47653
ParamOrderedSet.size * remove	C	6222	323578
ParamOrderedSet.size * delete	C	4448	95871
ParamOrderedSet.higher * find	C	19019	499907
ParamOrderedSet.higher * size	C	14304	296057
ParamOrderedSet.higher * higher	C	14304	296057
ParamOrderedSet.higher * insert	C	19388	504037
ParamOrderedSet.higher * add	1527	19636	791113
ParamOrderedSet.higher * isEmpty	C	14304	296057
ParamOrderedSet.higher * remove	363	44749	2970382
ParamOrderedSet.higher * delete	C	19019	499907
ParamOrderedSet.insert * find	0	5313	196203
ParamOrderedSet.insert * size	1	4353	95743
ParamOrderedSet.insert * higher	C	9617	227028
ParamOrderedSet.insert * insert	12	8172	211197
ParamOrderedSet.insert * add	13	8180	212120
ParamOrderedSet.insert * isEmpty	1	4586	96275
ParamOrderedSet.insert * remove	19	10359	543446
ParamOrderedSet.insert * delete	14	8372	212439
ParamOrderedSet.add * find	1	14467	321569
ParamOrderedSet.add * size	C	9680	160811
ParamOrderedSet.add * higher	1363	19797	821168
ParamOrderedSet.add * insert	35	17332	338407
ParamOrderedSet.add * add	1152	23451	474332
ParamOrderedSet.add * isEmpty	C	9913	161343
ParamOrderedSet.add * remove	6407	39065	1496591
ParamOrderedSet.add * delete	63	17525	338725

Table A.7. More Results for model: ParamOrderedSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamOrderedSet.isEmpty * find	C	3617	93791
ParamOrderedSet.isEmpty * size	C	2514	47653
ParamOrderedSet.isEmpty * higher	C	7356	136027
ParamOrderedSet.isEmpty * insert	2	4585	96276
ParamOrderedSet.isEmpty * add	1	4587	96279
ParamOrderedSet.isEmpty * isEmpty	C	2514	47653
ParamOrderedSet.isEmpty * remove	C	6455	324110
ParamOrderedSet.isEmpty * delete	C	4681	96403
ParamOrderedSet.remove * find	2	34942	1302595
ParamOrderedSet.remove * size	C	27224	872804
ParamOrderedSet.remove * higher	3339	44526	3100014
ParamOrderedSet.remove * insert	72	38167	1320679
ParamOrderedSet.remove * add	5379	42869	1698265
ParamOrderedSet.remove * isEmpty	C	27457	873336
ParamOrderedSet.remove * remove	11354	72442	3992170
ParamOrderedSet.remove * delete	75	38360	1320997
ParamOrderedSet.delete * find	0	5313	196203
ParamOrderedSet.delete * size	3	4450	95872
ParamOrderedSet.delete * higher	C	9617	227028
ParamOrderedSet.delete * insert	10	8372	212439
ParamOrderedSet.delete * add	11	8373	212438
ParamOrderedSet.delete * isEmpty	1	4683	96404
ParamOrderedSet.delete * remove	27	10552	543764
ParamOrderedSet.delete * delete	12	8559	211833

Table A.8. Results for model: ParamSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	C	551	3811
ParamSet.find * add	1	991	23780
ParamSet.find * remove	2	991	23780
ParamSet.add * find	2	991	23780
ParamSet.add * add	C	393	2109
ParamSet.add * remove	C	782	4018
ParamSet.remove * find	1	991	23780
ParamSet.remove * add	C	782	4018
ParamSet.remove * remove	C	389	2100

Table A.9. Results for model: ParamSetIterators

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSetIterators.find * find	C	551	3811
ParamSetIterators.find * incIterator	C	551	3811
ParamSetIterators.find * IncrementalIterator.next	C	551	3811
ParamSetIterators.find * SnapshotIterator.next	C	938	4678
ParamSetIterators.find * add	2	1795	26120
ParamSetIterators.find * snapIterator	C	938	4678
ParamSetIterators.find * remove	3	1795	26120
incIterator * find	C	940	4681
incIterator * incIterator	C	3224	8420
incIterator * IncrementalIterator.next	C	3224	8420
incIterator * SnapshotIterator.next	C	3224	8420
incIterator * add	C	516	1020
incIterator * snapIterator	C	3224	8420
incIterator * remove	C	516	1020

Table A.10. Results for model: IncrementalIterator

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
IncrementalIterator.next * find	C	1143	5149
IncrementalIterator.next * incIterator	C	3228	8426
IncrementalIterator.next * IncrementalIterator.next	2	3479	10291
IncrementalIterator.next * SnapshotIterator.next	3	3931	10121
IncrementalIterator.next * add	C	751	1488
IncrementalIterator.next * snapIterator	C	3228	8426
IncrementalIterator.next * remove	C	751	1488

Table A.11. Results for model: SnapshotIterator

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
SnapshotIterator.next * find	C	1141	5146
SnapshotIterator.next * incIterator	C	3226	8423
SnapshotIterator.next * IncrementalIterator.next	2	3929	10118
SnapshotIterator.next * SnapshotIterator.next	124	20669	252835
SnapshotIterator.next * add	C	749	1485
SnapshotIterator.next * snapIterator	C	3226	8423
SnapshotIterator.next * remove	C	749	1485

Table A.12. Results for model: ParamSetIterators

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSetIterators.add * find	2	1795	26120
ParamSetIterators.add * incIterator	C	514	1017
ParamSetIterators.add * IncrementalIterator.next	C	514	1017
ParamSetIterators.add * SnapshotIterator.next	C	514	1017
ParamSetIterators.add * add	C	1197	4449
ParamSetIterators.add * snapIterator	C	514	1017
ParamSetIterators.add * remove	C	1586	6358
snapIterator * find	C	1141	5146
snapIterator * incIterator	C	3226	8423
snapIterator * IncrementalIterator.next	C	3226	8423
snapIterator * SnapshotIterator.next	C	3226	8423
snapIterator * add	C	749	1485
snapIterator * snapIterator	C	3226	8423
snapIterator * remove	C	749	1485
ParamSetIterators.remove * find	2	1795	26120
ParamSetIterators.remove * incIterator	C	514	1017
ParamSetIterators.remove * IncrementalIterator.next	C	514	1017
ParamSetIterators.remove * SnapshotIterator.next	C	514	1017
ParamSetIterators.remove * add	C	1586	6358
ParamSetIterators.remove * snapIterator	C	514	1017
ParamSetIterators.remove * remove	C	1197	4449

Table A.13. Results for model: ParamSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	C	117	129
ParamSet.find * add	C	361	619
ParamSet.find * remove	C	361	619
ParamSet.add * find	1	360	618
ParamSet.add * add	0	371	1108
ParamSet.add * remove	0	760	3017
ParamSet.remove * find	1	360	618
ParamSet.remove * add	C	761	3018
ParamSet.remove * remove	C	372	1109

Table A.14. Results for model: IntCell

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
IntCell.set * set	5	692	2373
IntCell.set * putTwo	1	1110	2501
IntCell.set * put	5	692	2373
IntCell.set * get	C	337	1471
IntCell.set * setOne	0	1110	2501
IntCell.set * setTwo	0	1110	2501
IntCell.set * putOne	0	1110	2501
IntCell.putTwo * set	1	1110	2501
IntCell.putTwo * putTwo	0	725	1570
IntCell.putTwo * put	0	1110	2501
IntCell.putTwo * get	0	723	1567
IntCell.putTwo * setOne	C	532	1089
IntCell.putTwo * setTwo	0	725	1570
IntCell.putTwo * putOne	C	532	1089
IntCell.put * set	5	692	2373
IntCell.put * putTwo	0	1110	2501
IntCell.put * put	4	692	2373
IntCell.put * get	C	337	1471
IntCell.put * setOne	0	1110	2501
IntCell.put * setTwo	0	1110	2501
IntCell.put * putOne	0	1110	2501
IntCell.get * set	C	337	1471
IntCell.get * putTwo	0	723	1567
IntCell.get * put	C	337	1471
IntCell.get * get	C	115	126
IntCell.get * setOne	1	723	1567
IntCell.get * setTwo	0	723	1567
IntCell.get * putOne	0	723	1567

Table A.15. More Results for model: IntCell

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
IntCell.setOne * set	0	1110	2501
IntCell.setOne * putTwo	C	532	1089
IntCell.setOne * put	0	1110	2501
IntCell.setOne * get	0	723	1567
IntCell.setOne * setOne	0	725	1570
IntCell.setOne * setTwo	C	532	1089
IntCell.setOne * putOne	0	725	1570
IntCell.setTwo * set	0	1110	2501
IntCell.setTwo * putTwo	0	725	1570
IntCell.setTwo * put	0	1110	2501
IntCell.setTwo * get	0	723	1567
IntCell.setTwo * setOne	C	532	1089
IntCell.setTwo * setTwo	0	725	1570
IntCell.setTwo * putOne	C	532	1089
IntCell.putOne * set	0	1110	2501
IntCell.putOne * putTwo	C	532	1089
IntCell.putOne * put	0	1110	2501
IntCell.putOne * get	0	723	1567
IntCell.putOne * setOne	0	725	1570
IntCell.putOne * setTwo	C	532	1089
IntCell.putOne * putOne	0	725	1570

Table A.16. Results for model: MultiMap

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiMap.remove * remove	C	19143	199364
MultiMap.remove * size	C	3527	10043
MultiMap.remove * put	673	17459	140495
MultiMap.remove * getOne	C	4105	15377
MultiMap.remove * containsKey	1	2985	9423
MultiMap.size * remove	C	3527	10043
MultiMap.size * size	C	537	1031
MultiMap.size * put	C	3168	7727
MultiMap.size * getOne	4	2239	5263
MultiMap.size * containsKey	C	730	1480
MultiMap.put * remove	1236	17459	146919
MultiMap.put * size	C	3168	7727
MultiMap.put * put	C	16901	115605
MultiMap.put * getOne	7	4005	15628
MultiMap.put * containsKey	1	2886	9291
MultiMap.getOne * remove	C	4105	13771
MultiMap.getOne * size	4	2239	5263
MultiMap.getOne * put	7	4005	14022
MultiMap.getOne * getOne	C	730	1480
MultiMap.getOne * containsKey	C	1918	4301
MultiMap.containsKey * remove	1	2985	8620
MultiMap.containsKey * size	C	730	1480
MultiMap.containsKey * put	1	2886	8488
MultiMap.containsKey * getOne	C	1918	4301
MultiMap.containsKey * containsKey	C	730	1480

Table A.17. Results for model: Queue

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Queue.dequeue * dequeue	C	14728	311969
Queue.dequeue * size	C	15057	118071
Queue.dequeue * enqueue	C	32832	379537
Queue.size * dequeue	C	15057	118071
Queue.size * size	C	8201	42076
Queue.size * enqueue	C	18065	94090
Queue.enqueue * dequeue	C	33121	385965
Queue.enqueue * size	C	18065	94090
Queue.enqueue * enqueue	C	31063	265941

Table A.18. Results for model: MultiSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiSet.find * find	C	1241	2794
MultiSet.find * size	C	1241	2794
MultiSet.find * insert	1	3910	11922
MultiSet.find * add	2	3910	11922
MultiSet.find * isEmpty	C	1241	2794
MultiSet.find * remove	2	4201	13338
MultiSet.find * delete	1	4201	13338
MultiSet.size * find	C	1241	2794
MultiSet.size * size	C	1307	2733
MultiSet.size * insert	C	3679	9041
MultiSet.size * add	C	3679	9041
MultiSet.size * isEmpty	C	1307	2733
MultiSet.size * remove	3	5478	15455
MultiSet.size * delete	1	5478	15455
MultiSet.insert * find	1	3910	12725
MultiSet.insert * size	C	3679	9041
MultiSet.insert * insert	C	18487	123609
MultiSet.insert * add	880	17647	121128
MultiSet.insert * isEmpty	C	3912	9573
MultiSet.insert * remove	1072	21734	204362
MultiSet.insert * delete	1389	22124	208866
MultiSet.add * find	1	3910	12725
MultiSet.add * size	C	3679	9041
MultiSet.add * insert	594	17647	121931
MultiSet.add * add	C	17707	116207
MultiSet.add * isEmpty	C	3912	9573
MultiSet.add * remove	1071	21344	199342
MultiSet.add * delete	1793	21734	203846

Table A.19. Results for model: MultiSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiSet.isEmpty * find	C	1241	2794
MultiSet.isEmpty * size	C	1307	2733
MultiSet.isEmpty * insert	C	3912	9573
MultiSet.isEmpty * add	C	3912	9573
MultiSet.isEmpty * isEmpty	C	1307	2733
MultiSet.isEmpty * remove	C	5519	15507
MultiSet.isEmpty * delete	C	5519	15507
MultiSet.remove * find	2	4201	13338
MultiSet.remove * size	18	5479	15457
MultiSet.remove * insert	1128	21734	177060
MultiSet.remove * add	821	21344	172040
MultiSet.remove * isEmpty	C	5519	15507
MultiSet.remove * remove	C	26299	267407
MultiSet.remove * delete	2245	25821	273655
MultiSet.delete * find	2	4201	13338
MultiSet.delete * size	2	5478	15455
MultiSet.delete * insert	1016	22124	180761
MultiSet.delete * add	1227	21734	175741
MultiSet.delete * isEmpty	C	5519	15507
MultiSet.delete * remove	2539	25821	273655
MultiSet.delete * delete	C	27079	277447

Table A.20. Results for model: ParamSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	C	4182	76766
ParamSet.find * add	43	9542	414044
ParamSet.find * remove	29	9542	414044
ParamSet.add * find	29	9542	414044
ParamSet.add * add	34	10033	608590
ParamSet.add * remove	28	10033	608590
ParamSet.remove * find	41	9542	414044
ParamSet.remove * add	41	10033	608590
ParamSet.remove * remove	38	10032	606065

Table A.21. Results for model: Set

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Set.find * find	C	504	996
Set.find * size	C	504	996
Set.find * insert	1	1352	3921
Set.find * add	0	1352	3921
Set.find * isEmpty	C	504	996
Set.find * remove	1	1352	3921
Set.find * delete	1	1352	3921
Set.size * find	C	504	996
Set.size * size	C	535	1028
Set.size * insert	C	1240	2948
Set.size * add	C	1240	2948
Set.size * isEmpty	C	535	1028
Set.size * remove	C	1337	3077
Set.size * delete	C	1337	3077
Set.insert * find	0	1352	4845
Set.insert * size	C	1240	2948
Set.insert * insert	2	4213	19839
Set.insert * add	2	4215	19360
Set.insert * isEmpty	C	1473	3480
Set.insert * remove	34	4409	19682
Set.insert * delete	34	4413	21081
Set.add * find	1	1352	3921
Set.add * size	C	1240	2948
Set.add * insert	2	4215	19360
Set.add * add	2	4209	18881
Set.add * isEmpty	C	1473	3480
Set.add * remove	28	4403	19203
Set.add * delete	33	4407	19678

Table A.22. More Results for model: Set

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Set.isEmpty * find	C	504	996
Set.isEmpty * size	C	535	1028
Set.isEmpty * insert	C	1473	3480
Set.isEmpty * add	C	1473	3480
Set.isEmpty * isEmpty	C	535	1028
Set.isEmpty * remove	C	1570	3609
Set.isEmpty * delete	C	1570	3609
Set.remove * find	1	1352	3921
Set.remove * size	C	1337	3077
Set.remove * insert	13	4409	19682
Set.remove * add	26	4403	19203
Set.remove * isEmpty	C	1570	3609
Set.remove * remove	3	4597	19525
Set.remove * delete	3	4601	20000
Set.delete * find	0	1352	4845
Set.delete * size	C	1337	3077
Set.delete * insert	16	4413	21081
Set.delete * add	13	4407	19678
Set.delete * isEmpty	C	1570	3609
Set.delete * remove	3	4601	20000
Set.delete * delete	3	4599	20475

A.2 Conflict Predicate Correctness Results for Bad Predicates

The following results are for running the suite of example models through the language processing tool but with a set of bad conflict predicates. Note that some read-only operations have no conflicts at all and therefore have no bad conflict predicates. Similarly to good predicates, a ‘C’ means that the SAT solver detected a conflict before SAT solving began.

Table A.23. Results for model: Stack

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Stack.push * push	8	11062	59700
Stack.push * pop	9	12535	63490
Stack.push * size	3	5928	16445
Stack.pop * push	50	12535	63490
Stack.pop * pop	1	3522	9623
Stack.pop * size	3	6218	16445
Stack.size * push	2	5928	16445
Stack.size * pop	5	6218	16445
Stack.size * size	0	2561	7765

Table A.24. Results for model: Equivalence

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Equivalence.eq * eq	C	380	13122
Equivalence.eq * ne	C	1646	285666
Equivalence.ne * eq	C	1646	285666
Equivalence.ne * ne	C	380	13122

Table A.25. Results for model: Map

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Map.put * put	153	6369	33797
Map.put * size	1	1246	2957
Map.put * get	4	1861	5860
Map.put * remove	295	6531	35725
Map.put * containsKey	0	1359	3480
Map.size * put	1	1246	2957
Map.size * size	0	537	1031
Map.size * get	0	737	1461
Map.size * remove	0	1343	3086
Map.size * containsKey	0	506	999
Map.get * put	3	1701	4254
Map.get * size	0	737	1461
Map.get * get	0	737	1461
Map.get * remove	4	1669	4254
Map.get * containsKey	0	1346	2998
Map.remove * put	173	6531	35725
Map.remove * size	0	1343	3086
Map.remove * get	4	1829	5860
Map.remove * remove	3	6693	34441
Map.remove * containsKey	0	1359	3480
Map.containsKey * put	1	1359	3480
Map.containsKey * size	0	506	999
Map.containsKey * get	0	1346	2998
Map.containsKey * remove	1	1359	3480
Map.containsKey * containsKey	0	506	999

Table A.26. Results for model: ParamOrderedSet

Problem		SAT	
Conflict Pred. Correctness	Time(ms)	Vars	Clauses
ParamOrderedSet.find * find	0	3617	93791
ParamOrderedSet.find * higher	0	9617	227028
ParamOrderedSet.find * size	0	3617	93791
ParamOrderedSet.find * insert	1	5313	195279
ParamOrderedSet.find * add	2	5316	194832
ParamOrderedSet.find * isEmpty	0	3617	93791
ParamOrderedSet.find * delete	1	5313	195279
ParamOrderedSet.find * remove	21	7135	524912
ParamOrderedSet.higher * find	0	19019	499907
ParamOrderedSet.higher * higher	0	14304	296057
ParamOrderedSet.higher * add	2963	19633	790647
ParamOrderedSet.higher * remove	3325	44746	2969921
ParamOrderedSet.size * find	0	3617	93791
ParamOrderedSet.size * size	0	2514	47653
ParamOrderedSet.size * insert	0	4353	95743
ParamOrderedSet.size * add	0	4355	95746
ParamOrderedSet.size * isEmpty	0	2514	47653
ParamOrderedSet.size * delete	0	4450	95872
ParamOrderedSet.size * remove	0	6224	323579
ParamOrderedSet.insert * find	1	5313	196203
ParamOrderedSet.insert * size	0	4353	95743
ParamOrderedSet.insert * insert	11	8172	211197
ParamOrderedSet.insert * add	7	8180	212120
ParamOrderedSet.insert * isEmpty	0	4586	96275
ParamOrderedSet.insert * delete	12	8372	212439
ParamOrderedSet.insert * remove	25	10359	543446
ParamOrderedSet.add * find	0	14468	321119
ParamOrderedSet.add * higher	1291	19795	821161
ParamOrderedSet.add * size	0	9680	160811
ParamOrderedSet.add * insert	43	17332	338407
ParamOrderedSet.add * add	1616	23452	473882
ParamOrderedSet.add * isEmpty	0	9913	161343
ParamOrderedSet.add * delete	32	17525	338725
ParamOrderedSet.add * remove	5748	39066	1496141

Table A.27. More Results for model: ParamOrderedSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamOrderedSet.isEmpty * find	0	3617	93791
ParamOrderedSet.isEmpty * size	0	2514	47653
ParamOrderedSet.isEmpty * insert	0	4586	96275
ParamOrderedSet.isEmpty * add	0	4588	96278
ParamOrderedSet.isEmpty * isEmpty	0	2514	47653
ParamOrderedSet.isEmpty * delete	0	4683	96404
ParamOrderedSet.isEmpty * remove	0	6457	324111
ParamOrderedSet.delete * find	1	5313	196203
ParamOrderedSet.delete * size	0	4450	95872
ParamOrderedSet.delete * insert	11	8372	212439
ParamOrderedSet.delete * add	29	8373	212438
ParamOrderedSet.delete * isEmpty	0	4683	96404
ParamOrderedSet.delete * delete	12	8559	211833
ParamOrderedSet.delete * remove	26	10552	543764
ParamOrderedSet.remove * find	205	34943	1302145
ParamOrderedSet.remove * higher	4483	44524	3100007
ParamOrderedSet.remove * size	0	27224	872804
ParamOrderedSet.remove * insert	147	38167	1320679
ParamOrderedSet.remove * add	3412	42870	1697815
ParamOrderedSet.remove * isEmpty	0	27457	873336
ParamOrderedSet.remove * delete	84	38360	1320997
ParamOrderedSet.remove * remove	23270	72443	3991720

Table A.28. Results for model: ParamSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	0	551	3811
ParamSet.find * add	1	992	23781
ParamSet.find * remove	3	992	23781
ParamSet.add * find	2	992	23781
ParamSet.add * add	1	394	2110
ParamSet.add * remove	8	783	4019
ParamSet.remove * find	3	992	23781
ParamSet.remove * add	6	783	4019
ParamSet.remove * remove	0	390	2101

Table A.29. Results for model: ParamSetIterators

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
find * find	0	551	3811
find * incIterator	0	551	3811
find * IncrementalIterator.next	0	551	3811
find * SnapshotIterator.next	0	938	4678
find * add	3	1796	26121
find * snapIterator	0	938	4678
find * remove	3	1796	26121
incIterator * find	0	940	4681
incIterator * incIterator	0	3224	8420
incIterator * IncrementalIterator.next	0	3224	8420
incIterator * SnapshotIterator.next	0	3224	8420
incIterator * add	0	516	1020
incIterator * snapIterator	0	3224	8420
incIterator * remove	0	516	1020

Table A.30. Results for model: IncrementalIterator

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
IncrementalIterator.next * find	0	1143	5149
IncrementalIterator.next * incIterator	0	3228	8426
IncrementalIterator.next * IncrementalIterator.next	1	3479	10291
IncrementalIterator.next * SnapshotIterator.next	1	3931	10121
IncrementalIterator.next * add	0	751	1488
IncrementalIterator.next * snapIterator	0	3228	8426
IncrementalIterator.next * remove	0	751	1488

Table A.31. Results for model: SnapshotIterator

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
SnapshotIterator.next * find	0	1141	5146
SnapshotIterator.next * incIterator	0	3226	8423
SnapshotIterator.next * IncrementalIterator.next	1	3929	10118
SnapshotIterator.next * SnapshotIterator.next	146	20669	252835
SnapshotIterator.next * add	0	749	1485
SnapshotIterator.next * snapIterator	0	3226	8423
SnapshotIterator.next * remove	0	749	1485

Table A.32. More Results for model: ParamSetIterators

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
add * find	3	1796	26121
add * incIterator	0	514	1017
add * IncrementalIterator.next	0	514	1017
add * SnapshotIterator.next	0	514	1017
add * add	1	1194	4441
add * snapIterator	0	514	1017
add * remove	7	1587	6359
snapIterator * find	0	1141	5146
snapIterator * incIterator	0	3226	8423
snapIterator * IncrementalIterator.next	0	3226	8423
snapIterator * SnapshotIterator.next	0	3226	8423
snapIterator * add	0	749	1485
snapIterator * snapIterator	0	3226	8423
snapIterator * remove	0	749	1485
remove * find	2	1796	26121
remove * incIterator	0	514	1017
remove * IncrementalIterator.next	0	514	1017
remove * SnapshotIterator.next	0	514	1017
remove * add	6	1587	6359
remove * snapIterator	0	514	1017
remove * remove	0	1198	4450

Table A.33. Results for model: ParamSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	0	117	129
ParamSet.find * add	1	361	619
ParamSet.find * remove	0	361	619
ParamSet.add * find	0	361	619
ParamSet.add * add	0	372	1109
ParamSet.add * remove	1	761	3018
ParamSet.remove * find	1	361	619
ParamSet.remove * add	1	761	3018
ParamSet.remove * remove	1	372	1109

Table A.34. Results for model: IntCell

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
IntCell.set * set	1	692	1473
IntCell.set * putTwo	1	724	1537
IntCell.set * get	0	338	1023
IntCell.set * put	0	692	1473
IntCell.set * setOne	1	724	1537
IntCell.set * setTwo	0	724	1537
IntCell.set * putOne	1	724	1537
IntCell.putTwo * set	1	724	1537
IntCell.putTwo * putTwo	1	533	1090
IntCell.putTwo * get	0	531	1087
IntCell.putTwo * put	1	724	1537
IntCell.putTwo * setOne	0	532	1089
IntCell.putTwo * setTwo	0	533	1090
IntCell.putTwo * putOne	0	532	1089
IntCell.get * set	0	338	1023
IntCell.get * putTwo	0	531	1087
IntCell.get * get	0	115	126
IntCell.get * put	0	338	1023
IntCell.get * setOne	0	531	1087
IntCell.get * setTwo	0	531	1087
IntCell.get * putOne	0	531	1087
IntCell.put * set	0	692	1473
IntCell.put * putTwo	1	724	1537
IntCell.put * get	0	338	1023
IntCell.put * put	0	692	1473
IntCell.put * setOne	1	724	1537
IntCell.put * setTwo	0	724	1537
IntCell.put * putOne	0	724	1537

Table A.35. Results for model: IntCell

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
IntCell.setOne * set	0	724	1537
IntCell.setOne * putTwo	0	532	1089
IntCell.setOne * get	0	531	1087
IntCell.setOne * put	1	724	1537
IntCell.setOne * setOne	0	533	1090
IntCell.setOne * setTwo	1	532	1089
IntCell.setOne * putOne	0	533	1090
IntCell.setTwo * set	0	724	1537
IntCell.setTwo * putTwo	1	533	1090
IntCell.setTwo * get	1	531	1087
IntCell.setTwo * put	0	724	1537
IntCell.setTwo * setOne	1	532	1089
IntCell.setTwo * setTwo	0	533	1090
IntCell.setTwo * putOne	1	532	1089
IntCell.putOne * set	1	724	1537
IntCell.putOne * putTwo	1	532	1089
IntCell.putOne * get	1	531	1087
IntCell.putOne * put	1	724	1537
IntCell.putOne * setOne	0	533	1090
IntCell.putOne * setTwo	0	532	1089
IntCell.putOne * putOne	1	533	1090

Table A.36. Results for model: MultiMap

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiMap.remove * remove	252	19143	199364
MultiMap.remove * size	1	3527	10043
MultiMap.remove * put	261	17460	140045
MultiMap.remove * getOne	3	4105	15377
MultiMap.remove * containsKey	1	2986	8973
MultiMap.size * remove	2	3527	10043
MultiMap.size * size	0	537	1031
MultiMap.size * put	1	3168	7727
MultiMap.size * getOne	4	2239	5263
MultiMap.size * containsKey	0	730	1480
MultiMap.put * remove	19	17460	146469
MultiMap.put * size	1	3168	7727
MultiMap.put * put	15	16901	115605
MultiMap.put * getOne	24	4006	15178
MultiMap.put * containsKey	7	2887	8841
MultiMap.getOne * remove	2	4105	13771
MultiMap.getOne * size	5	2239	5263
MultiMap.getOne * put	17	4006	13572
MultiMap.getOne * getOne	0	730	1480
MultiMap.getOne * containsKey	0	1918	4301
MultiMap.containsKey * remove	2	2986	8170
MultiMap.containsKey * size	0	730	1480
MultiMap.containsKey * put	6	2887	8038
MultiMap.containsKey * getOne	0	1918	4301
MultiMap.containsKey * containsKey	0	730	1480

Table A.37. Results for model: Queue

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Queue.dequeue * dequeue	302	14728	311969
Queue.dequeue * size	181	15057	118071
Queue.dequeue * enqueue	40	32832	379537
Queue.size * dequeue	15	15057	118071
Queue.size * size	0	8201	42076
Queue.size * enqueue	11	18065	94090
Queue.enqueue * dequeue	37	33121	385965
Queue.enqueue * size	40	18065	94090
Queue.enqueue * enqueue	649	31063	265941

Table A.38. Results for model: MultiSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiSet.find * find	0	1241	2794
MultiSet.find * size	0	1241	2794
MultiSet.find * insert	2	3911	11472
MultiSet.find * add	4	3911	11472
MultiSet.find * isEmpty	0	1241	2794
MultiSet.find * delete	3	4202	12888
MultiSet.find * remove	3	4202	12888
MultiSet.size * find	0	1241	2794
MultiSet.size * size	0	1307	2733
MultiSet.size * insert	1	3679	9041
MultiSet.size * add	6	3679	9041
MultiSet.size * isEmpty	0	1307	2733
MultiSet.size * delete	4	5286	14975
MultiSet.size * remove	5	5286	14975
MultiSet.insert * find	2	3911	12275
MultiSet.insert * size	1	3679	9041
MultiSet.insert * insert	403	18487	123609
MultiSet.insert * add	712	17648	120678
MultiSet.insert * isEmpty	1	3912	9573
MultiSet.insert * delete	43	22125	208416
MultiSet.insert * remove	18	21735	203912
MultiSet.add * find	5	3911	12275
MultiSet.add * size	2	3679	9041
MultiSet.add * insert	681	17648	121481
MultiSet.add * add	315	17707	116207
MultiSet.add * isEmpty	1	3912	9573
MultiSet.add * delete	49	21735	203396
MultiSet.add * remove	824	21345	198892

Table A.39. More Results for model: MultiSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiSet.isEmpty * find	0	1241	2794
MultiSet.isEmpty * size	0	1307	2733
MultiSet.isEmpty * insert	1	3912	9573
MultiSet.isEmpty * add	2	3912	9573
MultiSet.isEmpty * isEmpty	0	1307	2733
MultiSet.isEmpty * delete	2	5519	15507
MultiSet.isEmpty * remove	2	5519	15507
MultiSet.delete * find	4	4202	12888
MultiSet.delete * size	11	5286	14975
MultiSet.delete * insert	265	22125	180311
MultiSet.delete * add	20	21735	175291
MultiSet.delete * isEmpty	10	5519	15507
MultiSet.delete * delete	92	27079	277447
MultiSet.delete * remove	1982	25822	273205
MultiSet.remove * find	4	4202	12888
MultiSet.remove * size	14	5286	14975
MultiSet.remove * insert	26	21735	176610
MultiSet.remove * add	29	21345	171590
MultiSet.remove * isEmpty	13	5519	15507
MultiSet.remove * delete	5303	25822	273205
MultiSet.remove * remove	39	26299	267407

Table A.40. Results for model: ParamSet

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	0	4182	76766
ParamSet.find * add	42	9543	413129
ParamSet.find * remove	35	9542	408530
ParamSet.add * find	47	9543	413129
ParamSet.add * add	34	10034	607675
ParamSet.add * remove	47	10034	607675
ParamSet.remove * find	36	9542	408530
ParamSet.remove * add	50	10033	603076
ParamSet.remove * remove	49	10033	602738

Table A.41. Results for model: Set

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Set.find * find	0	504	996
Set.find * size	0	504	996
Set.find * insert	1	1353	3471
Set.find * add	1	1353	3471
Set.find * isEmpty	0	504	996
Set.find * delete	0	1353	3471
Set.find * remove	1	1353	3471
Set.size * find	0	504	996
Set.size * size	0	535	1028
Set.size * insert	1	1240	2948
Set.size * add	0	1240	2948
Set.size * isEmpty	0	535	1028
Set.size * delete	0	1337	3077
Set.size * remove	0	1337	3077
Set.insert * find	1	1353	4395
Set.insert * size	1	1240	2948
Set.insert * insert	3	4214	19389
Set.insert * add	3	4216	18910
Set.insert * isEmpty	1	1473	3480
Set.insert * delete	4	4414	20631
Set.insert * remove	4	4410	19232
Set.add * find	1	1353	3471
Set.add * size	1	1240	2948
Set.add * insert	3	4216	18910
Set.add * add	2	4210	18431
Set.add * isEmpty	1	1473	3480
Set.add * delete	4	4408	19228
Set.add * remove	11	4404	18753

Table A.42. More Results for model: Set

Problem (Conflict Pred. Correctness)	Time(ms)	SAT Vars	SAT Clauses
Set.isEmpty * find	0	504	996
Set.isEmpty * size	0	535	1028
Set.isEmpty * insert	0	1473	3480
Set.isEmpty * add	0	1473	3480
Set.isEmpty * isEmpty	0	535	1028
Set.isEmpty * delete	1	1570	3609
Set.isEmpty * remove	0	1570	3609
Set.delete * find	1	1353	4395
Set.delete * size	1	1337	3077
Set.delete * insert	17	4414	20631
Set.delete * add	14	4408	19228
Set.delete * isEmpty	0	1570	3609
Set.delete * delete	3	4600	20025
Set.delete * remove	3	4602	19550
Set.remove * find	1	1353	3471
Set.remove * size	0	1337	3077
Set.remove * insert	14	4410	19232
Set.remove * add	5	4404	18753
Set.remove * isEmpty	1	1570	3609
Set.remove * delete	2	4602	19550
Set.remove * remove	5	4598	19075

A.3 Inverse Results

This section summarizes the SAT statistics for the suite of example models run through the tool to verify that the specified inverses are correct. Because the inverse of a method can be verified without having to consider other methods, there will only be one entry per method. Methods without a specified inverse are assumed to have the default 'no-op' inverse that does nothing. This is the case for the read-only methods.

Table A.43. Results for model: Stack

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: Stack.push	154	8352	35743
inverse of method: Stack.pop	77	8320	38955
inverse of method: Stack.size	C	3123	9311

Table A.44. Results for model: Equivalence

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: Equivalence.eq	C	2022	5633
inverse of method: Equivalence.ne	C	2022	5633

Table A.45. Results for model: PartialOrder

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: PartialOrder.eq	C	1294	3785
inverse of method: PartialOrder.le	C	1294	3785
inverse of method: PartialOrder.lt	C	1294	3785
inverse of method: PartialOrder.ge	C	1294	3785
inverse of method: PartialOrder.ne	C	1294	3785
inverse of method: PartialOrder.gt	C	1294	3785

Table A.46. Results for model: Map

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: Map.put	72	4337	16526
inverse of method: Map.size	C	1138	2558
inverse of method: Map.get	C	1138	2558
inverse of method: Map.remove	107	4305	16590
inverse of method: Map.containsKey	C	1138	2558

Table A.47. Results for model: ParamOrderedSet

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: ParamOrderedSet.find	C	2916	48751
inverse of method: ParamOrderedSet.higher	C	12522	160765
inverse of method: ParamOrderedSet.size	C	2916	48751
inverse of method: ParamOrderedSet.insert	1	4551	104606
inverse of method: ParamOrderedSet.add	92	37030	1287310
inverse of method: ParamOrderedSet.isEmpty	C	2916	48751
inverse of method: ParamOrderedSet.remove	105	32057	1233146
inverse of method: ParamOrderedSet.delete	1	4551	104606

Table A.48. Results for model: ParamSet

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: ParamSet.find	C	2791	7498
inverse of method: ParamSet.add	3	3177	9401
inverse of method: ParamSet.remove	2	3176	9399

Table A.49. Results for model: ParamSetIterators

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: ParamSetIterators.find	C	2791	7498
inverse of method: ParamSetIterators.incIterator	C	2792	7500

Table A.50. Results for model: IncrementalIterator

Problem (Inverse)	Time(ms)	SAT Vars	SAT Clauses
ParamSetIterators.IncrementalIterator.next	16	7132	18424

Table A.51. Results for model: SnapshotIterator

Problem (Inverse)	Time(ms)	SAT Vars	SAT Clauses
ParamSetIterators.SnapshotIterator.next	8	6546	17322

Table A.52. Results for model: ParamSetIterators

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: ParamSetIterators.add	2	3177	9401
inverse of method: ParamSetIterators.snapIterator	C	5344	13947
inverse of method: ParamSetIterators.remove	2	3176	9399

Table A.53. Results for model: ParamSet

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: ParamSet.find	C	104	109
inverse of method: ParamSet.add	0	761	2487
inverse of method: ParamSet.remove	1	760	2483

Table A.54. Results for model: IntCell

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: IntCell.set	C	103	107
inverse of method: IntCell.putTwo	C	103	107
inverse of method: IntCell.get	C	103	107
inverse of method: IntCell.put	C	103	107
inverse of method: IntCell.setOne	C	103	107
inverse of method: IntCell.setTwo	C	103	107
inverse of method: IntCell.putOne	C	103	107

Table A.55. Results for model: MultiMap

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: MultiMap.remove	244	11567	144402
inverse of method: MultiMap.size	C	1710	3861
inverse of method: MultiMap.put	104	11972	64973
inverse of method: MultiMap.getOne	C	1710	3861
inverse of method: MultiMap.containsKey	C	1710	3861

Table A.56. Results for model: Queue

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: Queue.dequeue	218	19629	176419
inverse of method: Queue.size	C	9821	46409
inverse of method: Queue.enqueue	85	21639	138853

Table A.57. Results for model: MultiSet

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: MultiSet.find	C	1420	3158
inverse of method: MultiSet.size	C	1420	3158
inverse of method: MultiSet.insert	11	13687	83531
inverse of method: MultiSet.add	11	13687	83531
inverse of method: MultiSet.isEmpty	C	1420	3158
inverse of method: MultiSet.remove	756	13778	160161
inverse of method: MultiSet.delete	810	13778	160161

Table A.58. Results for model: ParamSet

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: ParamSet.find	C	1235	4625
inverse of method: ParamSet.add	7	3182	80103
inverse of method: ParamSet.remove	7	3183	85957

Table A.59. Results for model: Set

Problem	Time(ms)	SAT Vars	SAT Clauses
inverse of method: Set.find	C	297	557
inverse of method: Set.size	C	297	557
inverse of method: Set.insert	1	1147	3033
inverse of method: Set.add	1	1147	3033
inverse of method: Set.isEmpty	C	297	557
inverse of method: Set.delete	0	1147	3033
inverse of method: Set.remove	0	1147	3033

A.4 Abstract Lock Verification Results

This section summarizes the SAT statistics for verifying the abstract locks specified for our suite of example models.

Table A.60. Results for model: Stack

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
Stack.push * push	C	11062	59700
Stack.push * pop	C	12535	63490
Stack.push * size	C	5928	16445
Stack.pop * push	C	12535	63490
Stack.pop * pop	C	3522	9623
Stack.pop * size	C	6218	16445
Stack.size * push	C	5928	16445
Stack.size * pop	C	6218	16445
Stack.size * size	C	2561	7765

Table A.61. Results for model: Map

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
Map.put * put	C	6369	33797
Map.put * size	C	1246	2957
Map.put * get	C	1861	5860
Map.put * remove	C	6531	35725
Map.put * containsKey	C	1359	3480
Map.size * put	C	1246	2957
Map.size * size	C	537	1031
Map.size * get	C	737	1461
Map.size * remove	C	1343	3086
Map.size * containsKey	C	506	999
Map.get * put	C	1701	4254
Map.get * size	C	737	1461
Map.get * get	C	737	1461
Map.get * remove	C	1669	4254
Map.get * containsKey	C	1346	2998
Map.remove * put	C	6531	35725
Map.remove * size	C	1343	3086
Map.remove * get	C	1829	5860
Map.remove * remove	C	6693	34441
Map.remove * containsKey	C	1359	3480
Map.containsKey * put	C	1359	3480
Map.containsKey * size	C	506	999
Map.containsKey * get	C	1346	2998
Map.containsKey * remove	C	1359	3480
Map.containsKey * containsKey	C	506	999

Table A.62. Results for model: ParamOrderedSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamOrderedSet.find * find	C	13456	305286
ParamOrderedSet.find * lower	C	19019	499907
ParamOrderedSet.find * higher	C	19019	499907
ParamOrderedSet.find * size	C	13456	305286
ParamOrderedSet.find * insert	2	20780	559870
ParamOrderedSet.find * add	1	17252	362137
ParamOrderedSet.find * isEmpty	C	13456	305286
ParamOrderedSet.find * remove	2	41775	1925064
ParamOrderedSet.find * delete	1	20780	559870
ParamOrderedSet.lower * find	C	19019	499907
ParamOrderedSet.lower * lower	C	14304	296057
ParamOrderedSet.lower * higher	C	20191	488632
ParamOrderedSet.lower * size	C	14304	296057
ParamOrderedSet.lower * insert	C	19661	501643
ParamOrderedSet.lower * add	2002	23712	851387
ParamOrderedSet.lower * isEmpty	C	14304	296057
ParamOrderedSet.lower * remove	7293	46715	2271556
ParamOrderedSet.lower * delete	C	19661	501643
ParamOrderedSet.higher * find	C	19019	499907
ParamOrderedSet.higher * lower	C	20191	488632
ParamOrderedSet.higher * higher	C	14304	296057
ParamOrderedSet.higher * size	C	14304	296057
ParamOrderedSet.higher * insert	C	19661	501643
ParamOrderedSet.higher * add	4484	23712	851387
ParamOrderedSet.higher * isEmpty	C	14304	296057
ParamOrderedSet.higher * remove	13311	46715	2271556
ParamOrderedSet.higher * delete	C	19661	501643

Table A.63. More Results for model: ParamOrderedSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamOrderedSet.size * find	C	13456	305286
ParamOrderedSet.size * lower	C	14304	296057
ParamOrderedSet.size * higher	C	14304	296057
ParamOrderedSet.size * size	C	9023	151674
ParamOrderedSet.size * insert	C	14192	307238
ParamOrderedSet.size * add	C	11553	188487
ParamOrderedSet.size * isEmpty	C	9023	151674
ParamOrderedSet.size * remove	C	33515	1342689
ParamOrderedSet.size * delete	C	14289	307367
ParamOrderedSet.insert * find	2	20780	560794
ParamOrderedSet.insert * lower	C	19661	501643
ParamOrderedSet.insert * higher	C	19661	501643
ParamOrderedSet.insert * size	C	14192	307238
ParamOrderedSet.insert * insert	C	23640	575338
ParamOrderedSet.insert * add	45	20283	379899
ParamOrderedSet.insert * isEmpty	C	14425	307770
ParamOrderedSet.insert * remove	7138	47585	1989776
ParamOrderedSet.insert * delete	3636	23839	577030
ParamOrderedSet.add * find	1	17252	362137
ParamOrderedSet.add * lower	1904	23746	882704
ParamOrderedSet.add * higher	1453	23746	882704
ParamOrderedSet.add * size	C	11553	188487
ParamOrderedSet.add * insert	53	20283	379899
ParamOrderedSet.add * add	C	27563	534646
ParamOrderedSet.add * isEmpty	C	11786	189019
ParamOrderedSet.add * remove	9773	41359	1543370
ParamOrderedSet.add * delete	65	20476	380217

Table A.64. More Results for model: ParamOrderedSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamOrderedSet.isEmpty * find	C	13456	305286
ParamOrderedSet.isEmpty * lower	C	14304	296057
ParamOrderedSet.isEmpty * higher	C	14304	296057
ParamOrderedSet.isEmpty * size	C	9023	151674
ParamOrderedSet.isEmpty * insert	C	14425	307770
ParamOrderedSet.isEmpty * add	C	11786	189019
ParamOrderedSet.isEmpty * isEmpty	C	9023	151674
ParamOrderedSet.isEmpty * remove	C	33748	1343221
ParamOrderedSet.isEmpty * delete	C	14522	307899
ParamOrderedSet.remove * find	3	43071	2001008
ParamOrderedSet.remove * lower	12503	47039	2378355
ParamOrderedSet.remove * higher	14235	47039	2378355
ParamOrderedSet.remove * size	C	33515	1342689
ParamOrderedSet.remove * insert	19496	48881	2052784
ParamOrderedSet.remove * add	21139	45163	1729336
ParamOrderedSet.remove * isEmpty	C	33748	1343221
ParamOrderedSet.remove * remove	C	84794	5554487
ParamOrderedSet.remove * delete	6721	49074	2053230
ParamOrderedSet.delete * find	1	20780	560794
ParamOrderedSet.delete * lower	C	19661	501643
ParamOrderedSet.delete * higher	C	19661	501643
ParamOrderedSet.delete * size	C	14289	307367
ParamOrderedSet.delete * insert	7042	23839	577030
ParamOrderedSet.delete * add	40	20476	380217
ParamOrderedSet.delete * isEmpty	C	14522	307899
ParamOrderedSet.delete * remove	5960	47778	1990222
ParamOrderedSet.delete * delete	C	24027	575974

Table A.65. Results for model: ParamSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	C	551	3811
ParamSet.find * add	C	992	23781
ParamSet.find * remove	C	992	23781
ParamSet.add * find	C	992	23781
ParamSet.add * add	C	394	2110
ParamSet.add * remove	C	783	4019
ParamSet.remove * find	C	992	23781
ParamSet.remove * add	C	783	4019
ParamSet.remove * remove	C	390	2101

Table A.66. Results for model: ParamSetIterators

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
find * find	C	551	3811
find * incIterator	C	551	3811
find * IncrementalIterator.next	C	551	3811
find * SnapshotIterator.next	C	938	4678
find * add	3	1795	26571
find * snapIterator	C	938	4678
find * remove	3	1795	26571
incIterator * find	C	940	4681
incIterator * incIterator	C	3224	8420
incIterator * IncrementalIterator.next	C	3224	8420
incIterator * SnapshotIterator.next	C	3224	8420
incIterator * add	C	516	1020
incIterator * snapIterator	C	3224	8420
incIterator * remove	C	516	1020

Table A.67. Results for model: IncrementalIterator

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
IncrementalIterator.next * find	C	1143	5149
IncrementalIterator.next * incIterator	C	3228	8426
IncrementalIterator.next * IncrementalIterator.next	1	3479	10291
IncrementalIterator.next * SnapshotIterator.next	3	3931	10121
IncrementalIterator.next * add	C	751	1488
IncrementalIterator.next * snapIterator	C	3228	8426
IncrementalIterator.next * remove	C	751	1488

Table A.68. Results for model: SnapshotIterator

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
SnapshotIterator.next * find	C	1141	5146
SnapshotIterator.next * incIterator	C	3226	8423
SnapshotIterator.next * IncrementalIterator.next	2	3929	10118
SnapshotIterator.next * SnapshotIterator.next	180	20669	252835
SnapshotIterator.next * add	C	749	1485
SnapshotIterator.next * snapIterator	C	3226	8423
SnapshotIterator.next * remove	C	749	1485

Table A.69. Results for model: ParamSetIterators

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
add * find	4	1795	26571
add * incIterator	C	514	1017
add * IncrementalIterator.next	C	514	1017
add * SnapshotIterator.next	C	514	1017
add * add	C	1194	4441
add * snapIterator	C	514	1017
add * remove	1	1586	6809
snapIterator * find	C	1141	5146
snapIterator * incIterator	C	3226	8423
snapIterator * IncrementalIterator.next	C	3226	8423
snapIterator * SnapshotIterator.next	C	3226	8423
snapIterator * add	C	749	1485
snapIterator * snapIterator	C	3226	8423
snapIterator * remove	C	749	1485
remove * find	2	1795	26571
remove * incIterator	C	514	1017
remove * IncrementalIterator.next	C	514	1017
remove * SnapshotIterator.next	C	514	1017
remove * add	1	1586	6809
remove * snapIterator	C	514	1017
remove * remove	C	1198	4450

Table A.70. Results for model: ParamSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	C	117	129
ParamSet.find * add	0	554	1069
ParamSet.find * remove	1	554	1069
ParamSet.add * find	0	554	1069
ParamSet.add * add	C	372	1109
ParamSet.add * remove	1	760	3468
ParamSet.remove * find	1	554	1069
ParamSet.remove * add	2	760	3468
ParamSet.remove * remove	C	372	1109

Table A.71. Results for model: IntCell

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
IntCell.set * set	C	338	573
IntCell.set * putTwo	C	338	605
IntCell.set * get	C	336	1020
IntCell.set * put	C	338	573
IntCell.set * setOne	C	338	605
IntCell.set * setTwo	C	338	605
IntCell.set * putOne	C	338	605
IntCell.putTwo * set	C	338	605
IntCell.putTwo * putTwo	C	115	126
IntCell.putTwo * get	C	529	1084
IntCell.putTwo * put	C	338	605
IntCell.putTwo * setOne	C	114	125
IntCell.putTwo * setTwo	C	115	126
IntCell.putTwo * putOne	C	114	125
IntCell.get * set	C	336	1020
IntCell.get * putTwo	C	529	1084
IntCell.get * get	C	115	126
IntCell.get * put	C	336	1020
IntCell.get * setOne	C	529	1084
IntCell.get * setTwo	C	529	1084
IntCell.get * putOne	C	529	1084
IntCell.put * set	C	338	573
IntCell.put * putTwo	C	338	605
IntCell.put * get	C	336	1020
IntCell.put * put	C	338	573
IntCell.put * setOne	C	338	605
IntCell.put * setTwo	C	338	605
IntCell.put * putOne	C	338	605

Table A.72. Results for model: IntCell

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
IntCell.setOne * set	C	338	605
IntCell.setOne * putTwo	C	114	125
IntCell.setOne * get	C	529	1084
IntCell.setOne * put	C	338	605
IntCell.setOne * setOne	C	115	126
IntCell.setOne * setTwo	C	114	125
IntCell.setOne * putOne	C	115	126
IntCell.setTwo * set	C	338	605
IntCell.setTwo * putTwo	C	115	126
IntCell.setTwo * get	C	529	1084
IntCell.setTwo * put	C	338	605
IntCell.setTwo * setOne	C	114	125
IntCell.setTwo * setTwo	C	115	126
IntCell.setTwo * putOne	C	114	125
IntCell.putOne * set	C	338	605
IntCell.putOne * putTwo	C	114	125
IntCell.putOne * get	C	529	1084
IntCell.putOne * put	C	338	605
IntCell.putOne * setOne	C	115	126
IntCell.putOne * setTwo	C	114	125
IntCell.putOne * putOne	C	115	126

Table A.73. Results for model: MultiMap

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiMap.remove * remove	C	20579	207350
MultiMap.remove * size	C	3882	10947
MultiMap.remove * put	1470	18895	148481
MultiMap.remove * getOne	11	4819	18900
MultiMap.remove * containsKey	1	3700	12496
MultiMap.size * remove	C	3882	10947
MultiMap.size * size	C	1737	3654
MultiMap.size * put	C	3524	8631
MultiMap.size * getOne	12	3053	7083
MultiMap.size * containsKey	C	1544	3300
MultiMap.put * remove	1069	18895	154905
MultiMap.put * size	C	3524	8631
MultiMap.put * put	C	18337	123591
MultiMap.put * getOne	10	4721	18701
MultiMap.put * containsKey	1	3602	12364
MultiMap.getOne * remove	10	4819	17294
MultiMap.getOne * size	12	3053	7083
MultiMap.getOne * put	9	4721	17095
MultiMap.getOne * getOne	C	1544	3300
MultiMap.getOne * containsKey	C	3222	8744
MultiMap.containsKey * remove	1	3700	11693
MultiMap.containsKey * size	C	1544	3300
MultiMap.containsKey * put	1	3602	11561
MultiMap.containsKey * getOne	C	3222	8744
MultiMap.containsKey * containsKey	C	1544	3300

Table A.74. Results for model: Queue

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
Queue.dequeue * dequeue	C	14732	313785
Queue.dequeue * size	C	15059	113142
Queue.dequeue * enqueue	C	32836	381353
Queue.size * dequeue	C	15059	113142
Queue.size * size	C	8203	39558
Queue.size * enqueue	C	18069	95906
Queue.enqueue * dequeue	C	33125	387781
Queue.enqueue * size	C	18069	95906
Queue.enqueue * enqueue	C	31067	267757

Table A.75. Results for model: MultiSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiSet.find * find	C	1241	2794
MultiSet.find * size	C	1241	2794
MultiSet.find * insert	2	3910	11922
MultiSet.find * add	1	3910	11922
MultiSet.find * isEmpty	C	1241	2794
MultiSet.find * delete	2	4201	13338
MultiSet.find * remove	1	4201	13338
MultiSet.size * find	C	1241	2794
MultiSet.size * size	C	1307	2733
MultiSet.size * insert	C	3679	9041
MultiSet.size * add	C	3679	9041
MultiSet.size * isEmpty	C	1307	2733
MultiSet.size * delete	C	5286	14975
MultiSet.size * remove	C	5286	14975
MultiSet.insert * find	2	3910	12725
MultiSet.insert * size	C	3679	9041
MultiSet.insert * insert	C	18487	123609
MultiSet.insert * add	766	17647	121128
MultiSet.insert * isEmpty	C	3912	9573
MultiSet.insert * delete	1254	22124	208866
MultiSet.insert * remove	1830	21734	204362
MultiSet.add * find	1	3910	12725
MultiSet.add * size	C	3679	9041
MultiSet.add * insert	642	17647	121931
MultiSet.add * add	C	17707	116207
MultiSet.add * isEmpty	C	3912	9573
MultiSet.add * delete	1466	21734	203846
MultiSet.add * remove	1303	21344	199342

Table A.76. More Results for model: MultiSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
MultiSet.isEmpty * find	C	1241	2794
MultiSet.isEmpty * size	C	1307	2733
MultiSet.isEmpty * insert	C	3912	9573
MultiSet.isEmpty * add	C	3912	9573
MultiSet.isEmpty * isEmpty	C	1307	2733
MultiSet.isEmpty * delete	C	5519	15507
MultiSet.isEmpty * remove	C	5519	15507
MultiSet.delete * find	1	4201	13338
MultiSet.delete * size	C	5286	14975
MultiSet.delete * insert	877	22124	180761
MultiSet.delete * add	1192	21734	175741
MultiSet.delete * isEmpty	C	5519	15507
MultiSet.delete * delete	C	27079	277447
MultiSet.delete * remove	1939	25821	273655
MultiSet.remove * find	1	4201	13338
MultiSet.remove * size	C	5286	14975
MultiSet.remove * insert	931	21734	177060
MultiSet.remove * add	1027	21344	172040
MultiSet.remove * isEmpty	C	5519	15507
MultiSet.remove * delete	1457	25821	273655
MultiSet.remove * remove	C	26299	267407

Table A.77. Results for model: ParamSet

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
ParamSet.find * find	C	4182	76766
ParamSet.find * add	C	9550	414399
ParamSet.find * remove	C	9543	413129
ParamSet.add * find	C	9550	415664
ParamSet.add * add	C	10044	615305
ParamSet.add * remove	56	10038	610208
ParamSet.remove * find	C	9543	413129
ParamSet.remove * add	66	10038	614003
ParamSet.remove * remove	C	10034	607675

Table A.78. Results for model: Set

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
Set.find * find	C	504	996
Set.find * size	C	504	996
Set.find * insert	0	1352	3921
Set.find * add	1	1352	3921
Set.find * isEmpty	C	504	996
Set.find * delete	0	1352	3921
Set.find * remove	0	1352	3921
Set.size * find	C	504	996
Set.size * size	C	535	1028
Set.size * insert	C	1240	2948
Set.size * add	C	1240	2948
Set.size * isEmpty	C	535	1028
Set.size * delete	C	1337	3077
Set.size * remove	C	1337	3077
Set.insert * find	0	1352	4845
Set.insert * size	C	1240	2948
Set.insert * insert	C	4214	19389
Set.insert * add	2	4215	19360
Set.insert * isEmpty	C	1473	3480
Set.insert * delete	55	4413	21081
Set.insert * remove	52	4409	19682
Set.add * find	1	1352	3921
Set.add * size	C	1240	2948
Set.add * insert	3	4215	19360
Set.add * add	C	4210	18431
Set.add * isEmpty	C	1473	3480
Set.add * delete	46	4407	19678
Set.add * remove	40	4403	19203

Table A.79. Results for model: Set

Problem (Lock Correctness)	Time(ms)	SAT Vars	SAT Clauses
Set.isEmpty * find	C	504	996
Set.isEmpty * size	C	535	1028
Set.isEmpty * insert	C	1473	3480
Set.isEmpty * add	C	1473	3480
Set.isEmpty * isEmpty	C	535	1028
Set.isEmpty * delete	C	1570	3609
Set.isEmpty * remove	C	1570	3609
Set.delete * find	1	1352	4845
Set.delete * size	C	1337	3077
Set.delete * insert	20	4413	21081
Set.delete * add	17	4407	19678
Set.delete * isEmpty	C	1570	3609
Set.delete * delete	C	4600	20025
Set.delete * remove	4	4601	20000
Set.remove * find	1	1352	3921
Set.remove * size	C	1337	3077
Set.remove * insert	18	4409	19682
Set.remove * add	42	4403	19203
Set.remove * isEmpty	C	1570	3609
Set.remove * delete	4	4601	20000
Set.remove * remove	C	4598	19075

APPENDIX B

ACCLAM GRAMMAR

This is a formal description of the grammar of ACCLAM.

An ACCLAM program is a collection of models. These models are organized into packages, which partition a global namespace. The description below lists keywords in bold, and assumes that there are lexical primitives for numeric and character literals and a lexical rule ID, that evaluates to a Java-legal symbol identifier [26].

Declarations:

$$\mathit{Program} \leftarrow \mathit{ModelDescription}^{*} \tag{B.1}$$

$$\mathit{ModelDescription} \leftarrow \mathit{PackageDec}\ \ \mathit{ModelDec} \tag{B.2}$$

$$\mathit{PackageDec} \leftarrow \textbf{package}\ \ \mathit{Name} \tag{B.3}$$

$$\mathit{Name} \leftarrow \mathit{ID}(\textbf{.}\mathit{ID})^{*} \tag{B.4}$$

$$\mathit{ModelDec} \leftarrow \textbf{model}\ \mathit{ID}\ \mathit{TypeParamDec}?\ \mathit{ModelBody} \tag{B.5}$$

$$\mathit{TypeParamDec} \leftarrow <(\mathit{ID}\ \mathit{TypeConstraints}^{+})^{+}> \tag{B.6}$$

$$\mathit{TypeConstraints} \leftarrow \textbf{extends}\ \mathit{TypeInstance} \mid \textbf{implements}\ \mathit{TypeInstance} \tag{B.7}$$

$$\mathit{TypeInstance} \leftarrow \mathit{Name}\ \mathit{TypeParamInstance}? \tag{B.8}$$

$$\mathit{TypeParamInstance} \leftarrow <\mathit{TypeInstance}^{+}> \tag{B.9}$$

$$\mathit{ModelBody} \leftarrow (\mathit{MemberDec} \mid \mathit{MethodDec} \mid \mathit{CtorDec} \mid \mathit{InvariantDec})^{*} \tag{B.10}$$

$$\mathit{MemberDec} \leftarrow (\mathit{JavaDec} \mid \mathit{RelationDec} \mid \mathit{ReductionDec})\textbf{;} \tag{B.11}$$

$$\mathit{JavaDec} \leftarrow \mathit{PrimDec} \mid \mathit{ObjDec} \tag{B.12}$$

$$\mathit{ObjDec} \leftarrow \mathit{NameID}\ (\textbf{=}\ \mathit{Expression})? \tag{B.13}$$

$$\mathit{PrimDec} \leftarrow \mathit{JavaPrimitiveType}\ \mathit{ID}\ (\textbf{=}\ \mathit{Expression})? \tag{B.14}$$

$$\mathit{RelationDec} \leftarrow \mathit{TypeInstance}\ \mathit{ID}\ \textbf{[}\ \mathit{TypeInstance}^{*}\ \textbf{]} \tag{B.15}$$

$$\mathit{ReductionDec} \leftarrow \mathit{PrimDec}\ \textbf{=}\ \mathit{ID}\textbf{(}\mathit{MethodArg}^{*}\textbf{;}\mathit{Expr}\textbf{;}\mathit{Expr}\textbf{;}\mathit{Expr}\textbf{)} \tag{B.16}$$

$$\mathit{MethodDec} \leftarrow \mathit{AbstractMethodDec}\ \mid \mathit{ConcreteMethodDec} \tag{B.17}$$

$$\mathit{AbstractMethodDec} \leftarrow \textbf{abstract}\ \mathit{MethodPrototype}\ \textbf{;} \tag{B.18}$$

$$\mathit{MethodPrototype} \leftarrow \mathit{ReturnType}\ (\mathit{TypeParamInstance})?\ \mathit{ID}\ \textbf{(}\ \mathit{MethodArg}^{*}\ \textbf{)} \tag{B.19}$$

$$\mathit{ConcreteMethodDec} \leftarrow \mathit{MethodPrototype}\ \mathit{Prevals}?\ \mathit{LockUse}^{*}\ \mathit{MethodBody} \tag{B.20}$$

$$\mathit{MethodArg} \leftarrow \mathit{TypeName}\ \mathit{ID} \tag{B.21}$$

$$\mathit{ReturnType} \leftarrow \textbf{void} \mid \mathit{TypeName} \tag{B.22}$$

$$\mathit{TypeName} \leftarrow \mathit{JavaPrimitiveType} \mid \mathit{RelationTypeName} \mid \mathit{TypeInstance} \tag{B.23}$$

$$\mathit{RelationTypeName} \leftarrow \mathit{TypeInstance}\ \textbf{[}\ \mathit{TypeInstance}^{*}\ \textbf{]} \tag{B.24}$$

$$\mathit{CtorDec} \leftarrow \mathit{ID}\ \textbf{(}\ \mathit{MethodArg}^{*}\ \textbf{)}\ \mathit{MethodBody} \tag{B.25}$$

$$\mathit{Prevals} \leftarrow \textbf{[}\ (\mathit{JavaDec} \mid \mathit{RelationDec})^{*}\ \textbf{]} \tag{B.26}$$

$$\mathit{LockUse} \leftarrow \textbf{@lock}\ \textbf{(}\ \mathit{ID}\ \mathit{TypeParamInstance}?\ \textbf{(}\ \mathit{ID}^{*}\ \textbf{)}\ \textbf{,}\ \mathit{ID}\ \textbf{)} \tag{B.27}$$

$$\mathit{MethodBody} \leftarrow \mathit{StmtBlock} \tag{B.28}$$

$$\mathit{InvariantDec} \leftarrow (\textbf{forall} \mid \textbf{exists})\textbf{(}\ \mathit{MethodArg}^{*}\ \textbf{;}\ \mathit{Expression}\textbf{)} \tag{B.29}$$

Expressions

$$\mathit{Expression} \leftarrow \mathit{TernaryExpression} \mid \mathit{BinExpression} \mid \mathit{UnaryExpression} \tag{B.30}$$

$$\mid \quad \mathit{ParenExpression} \mid \mathit{DerefExpression} \mid \mathit{CallExpression} \mid \mathit{NewExpr} \tag{B.31}$$

$$\mid \quad \mathit{Name} \mid \mathit{Constant} \tag{B.32}$$

$$\mathit{TernaryExpression} \leftarrow \mathit{Expression}\ \mathbf{?}\ \mathit{Expression}\ \mathbf{:}\ \mathit{Expression} \tag{B.33}$$

$$\mathit{BinExpression} \leftarrow \mathit{Expression}\ \mathit{BinOp}\ \mathit{Expression} \tag{B.34}$$

$$\mathit{BinOp} \leftarrow \mathbf{\&} \mid | \mid \wedge \mid - > \mid \mathbf{==} \mid \mathbf{!=} \mid < \mid <= \mid >= \mid > \mid + \mid - \mid * \mid \mathbf{/} \mid \mathbf{\%} \tag{B.35}$$

$$\mathit{UnaryExpression} \leftarrow (\mathbf{!} \mid + \mid -)\ \mathit{Expression} \tag{B.36}$$

$$\mathit{ParenExpression} \leftarrow \mathit{LPar}\ \mathit{Expression}\ \mathit{RPar} \tag{B.37}$$

$$\mathit{LPar} \leftarrow \mathbf{(} \tag{B.38}$$

$$\mathit{RPar} \leftarrow \mathbf{)} \tag{B.39}$$

$$\mathit{DerefExpression} \leftarrow \mathit{Expression}\ \mathbf{[}\ \mathit{Expression}^*\ \mathbf{]} \tag{B.40}$$

$$\mathit{CallExpression} \leftarrow \mathit{Name}\mathbf{(}\ \mathit{Expression}^*\ \mathbf{)} \tag{B.41}$$

$$\mathit{NewExpression} \leftarrow \mathbf{new}\ \mathit{Name}\ \mathbf{(}\ \mathit{Expression}^*\ \mathbf{)} \tag{B.42}$$

$$\mathit{Name} \leftarrow \mathit{ID} \tag{B.43}$$

$$\mathit{Constant} \leftarrow \mathit{IntegerLiteral} \mid \mathit{FloatLiteral} \mid \mathit{BooleanLiteral} \mid \mathit{CharLiteral} \tag{B.44}$$

$$\tag{B.45}$$

Statements

$$\mathit{Statement} \leftarrow \mathit{ExpressionStmt} \mid \mathit{BlockStmt} \mid \mathit{ConditionalStmt} \mid \mathit{ForallStmt} \tag{B.46}$$

$$\mid \quad \mathit{ChooseStmt} \mid \mathit{ReturnStmt} \mid \mathit{VarDecStmt} \tag{B.47}$$

$$\mathit{ExpressionStmt} \leftarrow \mathit{Expression}\ \textbf{;} \tag{B.48}$$

$$\mathit{BlockStmt} \leftarrow \{\ \mathit{Statement}^{*}\ \} \tag{B.49}$$

$$\mathit{ConditionalStmt} \leftarrow \textbf{if}\ \mathit{ParenExpression}\ \mathit{Statement}\ (\textbf{else}\ \mathit{Statement})? \tag{B.50}$$

$$\mathit{ForallStmt} \leftarrow \textbf{forall (}\ \mathit{MethodArg}^{*}\ \textbf{;}\ \mathit{Expression}\ \textbf{)}\ \mathit{Statement} \tag{B.51}$$

$$\mathit{ChooseStmt} \leftarrow \textbf{choose (}\ \mathit{MethodArg}^{*}\ \textbf{;}\ \mathit{Expression}\ \textbf{)}\ \mathit{Statement} \tag{B.52}$$

$$\mathit{ReturnStmt} \leftarrow \textbf{return}\ \mathit{Expression} \tag{B.53}$$

$$\mathit{VarDecStmt} \leftarrow \mathit{JavaDec} \mid \mathit{RelationDec} \mid \mathit{ReductionDec} \tag{B.54}$$

APPENDIX C

FULL MODELS

This appendix contains listings for all the models and conflict predicates used to generate the results in Appendix A.

C.1 IntCell

```
package examples;

model IntCell {
  int value;

  lockTable locks1D<T> cellLocks;

  IntCell (){ value = 0; }

  IntCell (int x) { value = x; }

  int get ()
  @lock(cellLocks, Everything<T>(), S)
  { return value; }

  int put (int x)
  @lock(cellLocks, Everything<T>(), X)
  [int oldVal = value; ]
  {
    value = x;
    return value;
  } onabort {
    value = oldVal;
  }

  void set (int x)
  @lock(cellLocks, Everything<T>(), X)
  [int oldVal = value; ]
  {
    value = x;
  } onabort {
    value = oldVal;
  }
  int putOne ()
  @lock(cellLocks, Everything<T>(), X)
  [int oldVal = value; ]
  {
    value = 1;
  return value;
  } onabort {
    value = oldVal;
  }

  void setOne ()
  @lock(cellLocks, Everything<T>(), X)
  [int oldVal = value; ]
  {
    value = 1;
  } onabort {
    value = oldVal;
  }

  int putTwo ()
  @lock(cellLocks, Everything<T>(), X)
  [int oldVal = value; ]
  {
    value = 2;
  return value;
  } onabort {
    value = oldVal;
  }

  void setTwo ()
  @lock(cellLocks, Everything<T>(), X)
  [int oldVal = value; ]
  {
    value = 2;
  } onabort {
    value = oldVal;
  }

}
```

C.1.1 Conflicts

```
package examples;

IntCell c;

m = c.get() ∗ n = c.get() { false }
m = c.get() ∗ n = c.put(i) { value ≠ i }
m = c.get() ∗ c.set(i) { value ≠ i }
m = c.get() ∗ n = c.putOne() { value ≠ 1 }
m = c.get() ∗ c.setOne() { value ≠ 1 }
m = c.get() ∗ n = c.putTwo() { value ≠ 2 }
m = c.get() ∗ c.setTwo() { value ≠ 2 }
```

m = c.put(i) ∗ n = c.get() { value ≠ i }
m = c.put(i) ∗ n = c.put(j) { (i ≠ j) —— (value ≠ i) }
m = c.put(i) ∗ c.set(j) { (i ≠ j) —— (value ≠ i) }
m = c.put(i) ∗ n = c.putOne() { (i ≠ 1) —— (value ≠ 1) }
m = c.put(i) ∗ c.setOne() { (i ≠ 1) —— (value ≠ 1) }
m = c.put(i) ∗ n = c.putTwo() { (i ≠ 2) —— (value ≠ 2) }
m = c.put(i) ∗ c.setTwo() { (i ≠ 2) —— (value ≠ 2) }

c.set(i) ∗ n = c.get() { i ≠ value }
c.set(i) ∗ n = c.put(j) { (i ≠ j) —— (value ≠ i) }
c.set(i) ∗ c.set(j) { (i ≠ j) —— (value ≠ i) }
c.set(i) ∗ n = c.putOne() { (i ≠ 1) —— (value ≠ 1) }
c.set(i) ∗ c.setOne() { (i ≠ 1) —— (value ≠ 1) }
c.set(i) ∗ n = c.putTwo() { (i ≠ 2) —— (value ≠ 2) }
c.set(i) ∗ c.setTwo() { (i ≠ 2) —— (value ≠ 2) }

m = c.putOne() ∗ n = c.get() { value ≠ 1 }
m = c.putOne() ∗ n = c.put(j) { (j ≠ 1) —— (value ≠ 1) }
m = c.putOne() ∗ c.set(j) { (j ≠ 1) —— (value ≠ 1) }
m = c.putOne() ∗ n = c.putOne() { value ≠ 1 }
m = c.putOne() ∗ c.setOne() { value ≠ 1 }
m = c.putOne() ∗ n = c.putTwo() { **true** }
m = c.putOne() ∗ c.setTwo() { **true** }

c.setOne() ∗ n = c.get() { value ≠ 1 }
c.setOne() ∗ n = c.put(j) { (j ≠ 1) —— (value ≠ 1) }
c.setOne() ∗ c.set(j) { (j ≠ 1) —— (value ≠ 1) }
c.setOne() ∗ n = c.putOne() { value ≠ 1 }
c.setOne() ∗ c.setOne() { value ≠ 1 }
c.setOne() ∗ n = c.putTwo() { **true** }
c.setOne() ∗ c.setTwo() { **true** }

m = c.putTwo() ∗ n = c.get() { value ≠ 2 }
m = c.putTwo() ∗ n = c.put(j) { (j ≠ 2) —— (value ≠ 2) }
m = c.putTwo() ∗ c.set(j) { (j ≠ 2) —— (value ≠ 2) }
m = c.putTwo() ∗ n = c.putOne() { **true** }
m = c.putTwo() ∗ c.setOne() { **true** }
m = c.putTwo() ∗ n = c.putTwo() { value ≠ 2 }
m = c.putTwo() ∗ c.setTwo() { value ≠ 2 }

c.setTwo() ∗ n = c.get() { value ≠ 2 }
c.setTwo() ∗ n = c.put(j) { (j ≠ 2) —— (value ≠ 2) }
c.setTwo() ∗ c.set(j) { (j ≠ 2) —— (value ≠ 2) }
c.setTwo() ∗ n = c.putOne() { **true** }
c.setTwo() ∗ c.setOne() { **true** }
c.setTwo() ∗ n = c.putTwo() { value ≠ 2 }
c.setTwo() ∗ c.setTwo() { value ≠ 2 }

C.2 Map

package examples;

//model for a map that allows null

```
model Map<K,V> {

  V mapping[K];
  boolean isMapped[K];
  int sz = count(K z ; isMapped[z] ; 1 ; 0);

  lockTable locks1D<K> mapLocks;

  Map() {
    forall (K key ; true) {
      mapping[key] = null;
    }

    forall (K key ; true) {
      isMapped[key] = false;
    }
  }

  V put(K key, V newVal)
  [V oldVal = mapping[key]; boolean wasMapped = isMapped[key]; ]
  @lock(mapLocks, Point1D<K>(key), X)
  @lock(mapLocks, Everything<K>(), S)
  {
    mapping[key] = newVal;
    isMapped[key] = true;

    return oldVal;
  } onabort {
    mapping[key] = oldVal;
    isMapped[key] = wasMapped;
  }

  V get(K key)
  @lock(mapLocks, Point1D<K>(key), S)
  @lock(mapLocks, Everything<K>(), S)
  {
    return mapping[key];
  }

  boolean containsKey(K key)
  @lock(mapLocks, Point1D<K>(key), S)
  @lock(mapLocks, Everything<K>(), S)
  {
    return isMapped[key];
  }

  V remove(K key)
  [V oldVal = mapping[key]; boolean wasMapped = isMapped[key]; ]
  @lock(mapLocks, Point1D<K>(key), X)
  @lock(mapLocks, Everything<K>(), S)
  {
    mapping[key] = null;
    isMapped[key] = false;
```

```
      return oldVal;
    } onabort {
      mapping[key] = oldVal;
      isMapped[key] = wasMapped;
    }

    int size()
    @lock(mapLocks, Everything<K>(), X)
    {
      return sz;
    }
  }
```

C.2.1 Conflicts

```
package examples;

a = put(k, v) ∗ b = put(x,y) { x = k }
a = put(k, v) ∗ b = get(x) { x = k }
a = put(k, v) ∗ b = containsKey(x) { k = x }
a = put(k, v) ∗ b = remove(x) { k = x }
a = put(k, v) ∗ b = size() @pre { !isMapped[k] }

a = get(k) ∗ b = put(x,y) { x = k }
a = get(k) ∗ b = get(x) { false }
a = get(k) ∗ b = containsKey(x) { false }
a = get(k) ∗ b = remove(x) { k = x }
a = get(k) ∗ b = size() { false }

a = containsKey(k) ∗ b = put(x,y) { x = k }
a = containsKey(k) ∗ b = get(x) { false }
a = containsKey(k) ∗ b = containsKey(x) { false }
a = containsKey(k) ∗ b = remove(x) { k = x }
a = containsKey(k) ∗ b = size() { false }

a = remove(k) ∗ b = put(x,y) { x = k }
a = remove(k) ∗ b = get(x) { x = k }
a = remove(k) ∗ b = containsKey(x) { k = x }
a = remove(k) ∗ b = remove(x) { k = x }
a = remove(k) ∗ b = size() @pre { isMapped[k] }

a = size() ∗ b = put(x,y) { true }
a = size() ∗ b = get(x) { false }
a = size() ∗ b = containsKey(x) { false }
a = size() ∗ b = remove(x) { true }
a = size() ∗ b = size() { false }
```

C.3 MultiMap

```
package examples;

model MultiMap<K,V> {

  int mapCount[K];
```

```
boolean mapping[K,V];
int sz = count(K z ; mapCount[z] ≠ 0 ; 1 ; 0);

lockTable locks1D<K> mapLocks;

MultiMap() {
   forall (K key ; true) {
      mapCount[key] = 0;
   }
}

boolean containsKey(K key)
@lock(mapLocks, Point1D<K>(key), S)
{
   return mapCount[key] ≠ 0;
}

void put(K key, V newVal)
[boolean wasMapped = mapping[key, newVal]; ]
@lock(mapLocks, Point1D<K>(key), X)
{
   mapCount[key] = mapCount[key] + 1;
   mapping[key, newVal] = true;
} onabort {
   mapCount[key] = mapCount[key] − 1;
   mapping[key, newVal] = wasMapped;
}

V getOne(K key)
@lock(mapLocks, Point1D<K>(key), S)
{
   if (containsKey(key)) {
      choose (V val ; mapping[key, val]) {
         return val;
      }
   } else {
      return null;
   }
}

boolean remove(K key, V removeVal)
[boolean wasMapped = mapping[key, removeVal]; ]
@lock(mapLocks, Point1D<K>(key), X)
{
   mapping[key, removeVal] = false;

   if (wasMapped) {
      mapCount[key] = mapCount[key] − 1;
   }

   return wasMapped;
} onabort {
   mapping[key, removeVal] = wasMapped;
```

```
    if (wasMapped) {
      mapCount[key] = mapCount[key] - 1;
    }
  }

  int size()
  @lock(mapLocks, Everything<K>(), S)
  {
    return sz;
  }
}
```

C.3.1 Conflicts

```
package examples;

put(x, y) * put(w, z) { true }
put(x, y) * r = getOne(w) { w = x }
put(x, y) * r = containsKey(w) { x = w }
put(x, y) * r = remove(w, z) { x = w }
put(x, y) * r = size() { true }

s = getOne(x) * put(w, z) { w = x }
s = getOne(x) * r = getOne(w) { false }
s = getOne(x) * r = containsKey(w) { false }
s = getOne(x) * r = remove(w, z) { true }
s = getOne(x) * r = size() { false }

s = containsKey(x) * put(w, z) { w = x }
s = containsKey(x) * r = getOne(w) { false }
s = containsKey(x) * r = containsKey(w) { false }
s = containsKey(x) * r = remove(w, z) { x = w }
s = containsKey(x) * r = size() { false }

s = remove(x) * put(w, z) { w = x }
s = remove(x) * r = getOne(w) { true }
s = remove(x) * r = containsKey(w) { x = w }
s = remove(x) * r = remove(w, z) { true }
s = remove(x) * r = size() { true }

s = size() * put(w, z) { true }
s = size() * r = getOne(w) { false }
s = size() * r = containsKey(w) { false }
s = size() * r = remove(w, z) { true }
s = size() * r = size() { true }
```

C.4 Set

```
package testmodels;

model Set<T>{
    boolean in[T];
    int sz = count(T obj ; in[obj];1;0);
```

```
lockTable locks1D<T> setLocks;

Set(){
  forall(T z ; true){
    in[z] = false;
  }
}

Set(T x){
  forall(T z ; true){
    in[z] = false;
  }
  in[x] = true;
}

void add(T obj)
@lock(setLocks, Point1D<T>(obj), X)
{
  in[obj] = true;
}

void remove(T obj)
@lock(setLocks, Point1D<T>(obj), X)
{
  in[obj] = false;
}

boolean insert(T obj)
@lock(setLocks, Point1D<T>(obj), X)
{
  boolean present = in[obj];
  in[obj] = true;
  return !present;
}

boolean delete(T obj)
@lock(setLocks, Point1D<T>(obj), X)
{
  boolean present = in[obj];
  in[obj] = false;
  return present;
}

boolean find(T obj)
@lock(setLocks, Point1D<T>(obj), S)
{
  return in[obj];
}

int size()
@lock(setLocks, Everything<T>(), S)
{
  return sz;
}
```

```
    boolean isEmpty()
    @lock(setLocks, Everything<T>(), S)
    {
      return (sz = 0);
    }
}
```

C.4.1 Conflicts

```
package testmodels;

add(x) ∗ add(y) { x = y }
add(x) ∗ remove(y) { x = y }
add(x) ∗ z = insert(y) { x = y }
add(x) ∗ z = delete(y) { x = y }
add(x) ∗ z = find(y) { x = y }
add(x) ∗ z = size() { !in[x] }
add(x) ∗ z = isEmpty() { true }

remove(x) ∗ remove(y) { x = y }
remove(x) ∗ add(y) { x = y }
remove(x) ∗ z = insert(y) { x = y }
remove(x) ∗ z = delete(y) { x = y }
remove(x) ∗ z = find(y) { x = y }
remove(x) ∗ z = size() { in[x] }
remove(x) ∗ z = isEmpty() { true }

w = insert(x) ∗ z = insert(y) { x = y }
w = insert(x) ∗ remove(y) { x = y }
w = insert(x) ∗ add(y) { x = y}
w = insert(x) ∗ z = delete(y) { x = y }
w = insert(x) ∗ z = find(y) { x = y }
w = insert(x) ∗ z = size() { !in[x] }
w = insert(x) ∗ z = isEmpty() { true }

w = delete(x) ∗ z = delete(y) { x = y }
w = delete(x) ∗ z = insert(y) { x = y }
w = delete(x) ∗ remove(y) { x = y }
w = delete(x) ∗ add(y) { x = y}
w = delete(x) ∗ z = find(y) { x = y }
w = delete(x) ∗ z = size() { in[x] }
w = delete(x) ∗ z = isEmpty() { true }

w = find(x) ∗ z = find(y) { false }
w = find(x) ∗ z = delete(y) { x = y }
w = find(x) ∗ z = insert(y) { x = y }
w = find(x) ∗ remove(y) { x = y }
w = find(x) ∗ add(y) { x = y}
w = find(x) ∗ z = size() { false }
w = find(x) ∗ z = isEmpty() { false }

w = size() ∗ z = find(y) { false }
w = size() ∗ z = delete(y) { in[y] }
```

```
w = size() * z = insert(y) { !in[y] }
w = size() * remove(y) { in[y] }
w = size() * add(y) { !in[y] }
w = size() * z = size() { false }
w = size() * z = isEmpty() { false }

w = isEmpty() * z = find(y) { false }
w = isEmpty() * z = delete(y) { true }
w = isEmpty() * z = insert(y) { true }
w = isEmpty() * remove(y) { true }
w = isEmpty() * add(y) { true }
w = isEmpty() * z = size() { false }
w = isEmpty() * z = isEmpty() { false }
```

C.5 Equivalence

```
package examples;

model Equivalence<T> {
  boolean equ[T,T];
  invariant forall (T x; equ[x, x]);
  invariant forall (T x, T y; equ[x,y] = equ[y,x]);
  invariant forall (T x, T y, T z;
                    (equ[x, y] && equ[y, z]) → equ[x, z]);

  boolean eq(T x, T y) { return equ[x,y]; }
  boolean ne(T x, T y) { return !eq(x, y); }
}
```

C.5.1 Conflicts

```
package testmodels;

Equivalence e1;

z = e1.eq(x,y) * c = e1.eq(a,b) { false }
z = e1.eq(x,y) * c = e1.ne(a,b) { false }

z = e1.ne(x,y) * c = e1.eq(a,b) { false }
z = e1.ne(x,y) * c = e1.ne(a,b) { false }
```

C.6 Parameterized Sets

C.6.1 Set parameterized by an equ relation

```
package examples;

model ParamSet<T> {
  boolean in[T];
  boolean equ[T,T];

  lockTable locks1D<T> setLocks;
  invariant forall(T x, T y; equ[x,y] → (in[x] = in[y]));
  invariant forall(T x; equ[x,x]);
  invariant forall(T x, T y; equ[x,y] = equ[y,x]);
  invariant forall(T x, T y, T z;
                    (equ[x,y] & equ[y,z]) → equ[x,z]);

  ParamSet (){
    forall (T x; true) { in[x] = false; }
  }

  void add (T x)
  [boolean present = in[x]; ]
  //point locks don't work, because it should be locking the
  //entire equivalence class defined by equ. So lock everything
  @lock(setLocks, Everything<T>(), X)
  {
    forall(T z; equ[z,x]) { in[z] = true; }
  } onabort {
    if (!present){
      remove(x);
    }
  }

  void remove (T x)
  [boolean present = in[x]; ]
  @lock(setLocks, Everything<T>(), X)
  {
    forall(T z; equ[z,x]) { in[z] = false; }
  } onabort {
    if (present) {
      add(x);
    }
  }

  boolean find(T x)
  @lock(setLocks, Everything<T>(), S)
  {
    return in[x];
  }
}
```

C.6.2 Set parameterized by a canonical element mapping

```
// Parameterized Set done using integers to number
// the equivalence classes in in[T]
package examples;

model ParamSet<T> {
  boolean in[T];
  T canon[T];
  boolean equ[T,T];

  lockTable locks1D<T> setLocks;
  //invariants
  invariant forall(T x, T y; equ[x,y] = (x = y));
  invariant forall(T x, T y;
                    equ[x,y] = (canon[x] = canon[y]));
  invariant forall(T x; equ[x,x]);
  invariant forall(T x, T y; equ[x,y] = equ[y,x]);
  invariant forall(T x;
                    (x = null) = (canon[x] = null));

  ParamSet (){
    forall (T x; true) { in[x] = false; }
  }

  void add (T x)
  [boolean present = in[canon[x]]; ]
  @lock(setLocks, Point1D<T>(canon[x]), X)
  {
    in[canon[x]] = true;
  } onabort {
    if (!present){
      remove(x);
    }
  }

  void remove (T x)
  [boolean present = in[canon[x]]; ]
  @lock(setLocks, Point1D<T>(canon[x]), X)
  {
    in[canon[x]] = false;
  } onabort {
    if (present) {
      add(x);
    }
  }

  boolean find(T x)
  @lock(setLocks, Everything<T>(), S)
  {
    return in[canon[x]];
  }
}
```

C.6.3 Set parameterized by an `Equivalence`

```
package testmodels;

model ParamSet<T> {
  boolean in[T];
  Equivalence<T> e;

  lockTable locks1D<T> setLocks;

  ParamSet (){
    forall (T x; true) { in[x] = false; }
  }

  void add (T x)
  [boolean present = in[x]; ]
  @lock(setLocks, Point1D<T>(x), X)
  {
    forall(T z; e.eq(x,z)) { in[z] = true; }
  } onabort {
    if (!present){
      remove(x);
    }
  }

  void remove (T x)
  [boolean present = in[x]; ]
  @lock(setLocks, Point1D<T>(x), X)
  {
    forall(T z; e.eq(z,x)) { in[z] = false; }
  } onabort {
    if (present) {
      add(x);
    }
  }

  boolean find(T x)
  @lock(setLocks, Point1D<T>(x), S)
  {
    return in[x];
  }
}
```

C.6.4 Conflicts (shared by all implementations)

```
package examples;

add(x) * add(y) { equ[x,y] }
add(x) * remove(y) { equ[x,y] }
add(x) * z = find(y) { equ[x,y] }

remove(x) * remove(y) { equ[x,y] }
remove(x) * add(y) { equ[x,y] }
remove(x) * z = find(y) { equ[x,y] }

w = find(x) * z = find(y) { false }
w = find(x) * remove(y) { equ[x,y] }
w = find(x) * add(y) { equ[x,y] }
```

C.7 Parameterized Sets with Iterators

```
package examples;

model ParamSetIterators<T> {
  boolean in[T];
  boolean equ[T,T];
```

```
lockTable locks1D<T> setLocks;

//invariants
invariant forall(T x, T y; equ[x,y] → (in[x] = in[y]));
invariant forall(T x; equ[x,x]);
invariant forall(T x, T y; equ[x,y] = equ[y,x]);
invariant forall(T x, T y, T z;
                   (equ[x,y] & equ[y,z]) → equ[x,z]);

ParamSetIterators (){
  forall (T x; true) { in[x] = false; }
}

void add (T x)
[boolean present = in[x]; ]
@lock(setLocks, Everything<T>(x), X)
{
  forall(T z; equ[z,x]) { in[z] = true; }
} onabort {
  if (!present){
    remove(x);
  }
}

void remove (T x)
[boolean present = in[x]; ]
@lock(setLocks, Everything<T>(x), X)
{
  forall(T z; equ[z,x]) { in[z] = false; }
} onabort {
  if (present) {
    add(x);
  }
}

boolean find(T x)
@lock(setLocks, Everything<T>(x), S)
{
  return in[x];
}

examples.ParamSetIterators.SnapshotIterator snapIterator()
@lock(setLocks, Everything<T>(), S)
{
  return new SnapshotIterator();
}

IncrementalIterator incIterator()
@lock(setLocks, Everything<T>(), S)
{
  return new IncrementalIterator();
}
```

```
  model SnapshotIterator {
    boolean contents[T];
    int remaining = count (T z ; contents[z] ; 1 ; 0);

    SnapshotIterator() {
      forall (T idx ; true) {
        contents[idx] = in[idx];
      }
    }

    T next() {
      if (remaining > 0) {
        choose (T idx ; contents[idx]) {
          contents[idx] = false;
          return idx;
        }
      } else {
        throw new Exception();
      }
    }
  }

  model IncrementalIterator {
    boolean seenIt[T];

    IncrementalIterator() {
      forall (T idx ; true) {
        seenIt[idx] = false;
      }
    }

    T next()
    @lock(setLocks, Everything<T>(), S)
    {
      choose (T idx ; in[idx] && (!seenIt[idx])) {
        seenIt[idx] = true;
        return idx;
      }
    }
  }
}
```

C.7.1 Conflicts

```
package examples;

ParamSetIterators.SnapshotIterator snap;
ParamSetIterators.IncrementalIterator inc;

add(x) * add(y) { equ[x,y] }
add(x) * remove(y) { equ[x,y] }
add(x) * z = find(y) { equ[x,y] }
add(x) * z = snap.next() { !snap.contents[x] }
add(x) * z = IncrementalIterator.next() { true }
add(x) * z = SnapshotIterator.next() { false }
add(x) * incIterator() { false }
add(x) * snapIterator() { false }

remove(x) * remove(y) { equ[x,y] }
remove(x) * add(y) { equ[x,y] }
remove(x) * z = find(y) { equ[x,y] }
remove(x) * z = SnapshotIterator.next() { false }
remove(x) * z = IncrementalIterator.next() { true }
remove(x) * snapIterator() { false }
remove(x) * incIterator() { false }

w = find(x) * z = find(y) { false }
w = find(x) * remove(y) { equ[x,y] }
w = find(x) * add(y) { equ[x,y] }
w = find(x) * z = snap.next() { false }
w = find(x) * z = IncrementalIterator.next() { false }
w = find(x) * z = SnapshotIterator.next() { false }
w = find(x) * incIterator() { false }
w = find(x) * snapIterator() { false }

w = IncrementalIterator.next()
  * add(x) { !inc.seenIt[x] }
w = IncrementalIterator.next()
  * snapIterator() { false }
w = IncrementalIterator.next()
  * z = IncrementalIterator.next() { false }
w = IncrementalIterator.next()
  * z = SnapshotIterator.next() { false }
w = IncrementalIterator.next()
  * z = remove(y) { true }
w = IncrementalIterator.next()
  * z = find(y) { false }
w = IncrementalIterator.next()
  * incIterator() { false }

w = SnapshotIterator.next()
  * add(x) { false }
w = SnapshotIterator.next()
  * snapIterator() { false }
w = SnapshotIterator.next()
  * z = IncrementalIterator.next() { false }
w = SnapshotIterator.next()
  * z = SnapshotIterator.next() { false }
w = SnapshotIterator.next()
  * z = remove(y) { false }
w = SnapshotIterator.next()
  * z = find(y) { false }
w = SnapshotIterator.next()
  * incIterator() { false }

incIterator() * w = find(x) { false }
incIterator() * incIterator() { false }
incIterator() * snapIterator() { false }
incIterator() ∗ IncrementalIterator.next() { false }
incIterator() ∗ SnapshotIterator.next() { false }
incIterator() ∗ add(x) { false }
incIterator() ∗ remove(x) { false }

snapIterator() ∗ w = find(x) { false }
snapIterator() ∗ incIterator() { false }
snapIterator() ∗ snapIterator() { false }
snapIterator() ∗ IncrementalIterator.next() { false }
snapIterator() ∗ SnapshotIterator.next() { false }
snapIterator() ∗ add(x) { false }
snapIterator() ∗ remove(x) { false }
```

C.8 MultiSet

```
package examples;

model MultiSet<T>{
    int in[T];
    int sz = count(T obj ; (in[obj] > 0) ; 1 ; 0 );

    invariant forall(T elem ; in[elem] ≥ 0);

    MultiSet(){
      forall(T z ; true){
        in[z] = 0;
      }
    }

    MultiSet(T x){
      forall(T z ; true){
        in[z] = 0;
      }
      in[x] = 1;
    }

    void add(T obj)
    [boolean present = in[obj] ≠ 0; ]
    @lock(setLocks, Point1D<T>(obj), X)
    {
      in[obj] = in[obj] + 1;
    }onabort{
      if (!present) {
        remove(obj);
      }
    }

    void remove(T obj)
    [boolean present = in[obj] ≠ 0; ]
    @lock(setLocks, Point1D<T>(obj), X)
    {
      if (present) {
        in[obj] = in[obj] − 1;
      }
    }onabort{
      if (present) {
        add(obj);
      }
    }

    boolean insert(T obj)
    [boolean present = in[obj] ≠ 0; ]
    @lock(setLocks, Point1D<T>(obj), X)
    {
      in[obj] = in[obj] + 1;
      return present;
    }onabort{
      if(!present) {
        remove(obj);
      }
    }

    boolean delete(T obj)
    [boolean present = in[obj] ≠ 0; ]
    @lock(setLocks, Point1D<T>(obj), X)
    {
      if (present) {
        in[obj] = in[obj] − 1;
      }
      return present;
    }onabort{
      if(present){
        add(obj);
      }
    }

    boolean find(T obj)
    @lock(setLocks, Point1D<T>(obj), X)
    {
      return in[obj] ≠ 0;
    }

    int size()
    @lock(setLocks, Everything<T>(), S)
    {
      return sz;
    }

    boolean isEmpty()
    @lock(setLocks, Everything<T>(), S)
    {
      return (sz = 0);
    }
}
```

C.8.1 Conflicts

```
package examples;
```

add(x) ∗ add(y) { **true** }
add(x) ∗ remove(y) { x = y }
add(x) ∗ z = insert(y) { x = y }
add(x) ∗ z = delete(y) { x = y }
add(x) ∗ z = find(y) { x = y }
add(x) ∗ z = size() @pre { **true** }
add(x) ∗ z = isEmpty() @pre { in[x] = 0 }

remove(x) ∗ remove(y) { **true** }
remove(x) ∗ add(y) { x = y }
remove(x) ∗ z = insert(y) { x = y }
remove(x) ∗ z = delete(y) { x = y }
remove(x) ∗ z = find(y) { x = y }
remove(x) ∗ z = size() @pre { in[x] = 1 }
remove(x) ∗ z = isEmpty() @pre { **true** }

w = insert(x) ∗ z = insert(y) { **true** }
w = insert(x) ∗ remove(y) { x = y }
w = insert(x) ∗ add(y) { x = y}
w = insert(x) ∗ z = delete(y) { x = y }
w = insert(x) ∗ z = find(y) { x = y }
w = insert(x) ∗ z = size() @pre { **true** }
w = insert(x) ∗ z = isEmpty() @pre { in[x] = 0 }

w = delete(x) ∗ z = delete(y) { **true** }
w = delete(x) ∗ z = insert(y) { x = y }
w = delete(x) ∗ remove(y) { x = y }
w = delete(x) ∗ add(y) { x = y}
w = delete(x) ∗ z = find(y) { x = y }
w = delete(x) ∗ z = size() @pre { in[x] ≠ 0 }
w = delete(x) ∗ z = isEmpty() @pre { in[x] ≠ 0 }

w = find(x) ∗ z = find(y) { **false** }
w = find(x) ∗ z = delete(y) { x = y }
w = find(x) ∗ z = insert(y) { x = y }
w = find(x) ∗ remove(y) { x = y }
w = find(x) ∗ add(y) { x = y}
w = find(x) ∗ z = size() { **false** }
w = find(x) ∗ z = isEmpty() { **false** }

w = size() ∗ z = find(y) { **false** }
w = size() ∗ z = delete(y) { in[y] ≠ 0 }
w = size() ∗ z = insert(y) { **true** }
w = size() ∗ remove(y) { in[y] ≠ 0 }
w = size() ∗ add(y) { **true** }
w = size() ∗ z = size() { **false** }
w = size() ∗ z = isEmpty() { **false** }

w = isEmpty() ∗ z = find(y) { **false** }
w = isEmpty() ∗ z = delete(y) { **true** }
w = isEmpty() ∗ z = insert(y) @pre { in[y] = 0 }
w = isEmpty() ∗ remove(y) { **true** }
w = isEmpty() ∗ add(y) @pre { in[y] = 0 }

```
w = isEmpty() ∗ z = size() { false }
w = isEmpty() ∗ z = isEmpty() { false }
```

C.9 Ordering Definitions

C.9.1 Partial Order

```
package testmodels;

model PartialOrder<T> {
  boolean leq[T,T];
  invariant forall (T x; leq[x, x]);
  invariant forall (T x, T y, T z;
                    (!leq[x, y]) | (!leq[y, z]) | leq[x, z]);

  boolean le(T x, T y) { return leq[x, y]; }
  boolean ge(T x, T y) { return le(y, x); }
  boolean lt(T x, T y) { return le(x, y) & !le(y, x); }
  boolean gt(T x, T y) { return lt(y, x); }
  boolean eq(T x, T y) { return le(x, y) & le(y, x); }
  boolean ne(T x, T y) { return !eq(x, y); }
}
```

C.9.1.1 Conflicts

```
package testmodels;

PartialOrder p;

z = p.lt(x,y) ∗ c = p.lt(a,b) { false }
z = p.lt(x,y) ∗ c = p.le(a,b) { false }
z = p.lt(x,y) ∗ c = p.ge(a,b) { false }
z = p.lt(x,y) ∗ c = p.gt(a,b) { false }
z = p.lt(x,y) ∗ c = p.eq(a,b) { false }
z = p.lt(x,y) ∗ c = p.ne(a,b) { false }

z = p.le(x,y) ∗ c = p.lt(a,b) { false }
z = p.le(x,y) ∗ c = p.le(a,b) { false }
z = p.le(x,y) ∗ c = p.ge(a,b) { false }
z = p.le(x,y) ∗ c = p.gt(a,b) { false }
z = p.le(x,y) ∗ c = p.eq(a,b) { false }
z = p.le(x,y) ∗ c = p.ne(a,b) { false }

z = p.ge(x,y) ∗ c = p.lt(a,b) { false }
z = p.ge(x,y) ∗ c = p.le(a,b) { false }
z = p.ge(x,y) ∗ c = p.ge(a,b) { false }
z = p.ge(x,y) ∗ c = p.gt(a,b) { false }
z = p.ge(x,y) ∗ c = p.eq(a,b) { false }
z = p.ge(x,y) ∗ c = p.ne(a,b) { false }

z = p.gt(x,y) ∗ c = p.lt(a,b) { false }
z = p.gt(x,y) ∗ c = p.le(a,b) { false }
z = p.gt(x,y) ∗ c = p.ge(a,b) { false }
z = p.gt(x,y) ∗ c = p.gt(a,b) { false }
```

```
z = p.gt(x,y) ∗ c = p.eq(a,b) { false }
z = p.gt(x,y) ∗ c = p.ne(a,b) { false }

z = p.eq(x,y) ∗ c = p.lt(a,b) { false }
z = p.eq(x,y) ∗ c = p.le(a,b) { false }
z = p.eq(x,y) ∗ c = p.ge(a,b) { false }
z = p.eq(x,y) ∗ c = p.gt(a,b) { false }
z = p.eq(x,y) ∗ c = p.eq(a,b) { false }
z = p.eq(x,y) ∗ c = p.ne(a,b) { false }

z = p.ne(x,y) ∗ c = p.lt(a,b) { false }
z = p.ne(x,y) ∗ c = p.le(a,b) { false }
z = p.ne(x,y) ∗ c = p.ge(a,b) { false }
z = p.ne(x,y) ∗ c = p.gt(a,b) { false }
z = p.ne(x,y) ∗ c = p.eq(a,b) { false }
z = p.ne(x,y) ∗ c = p.ne(a,b) { false }
```

C.9.2 Total Order

```
package testmodels;

model TotalOrder<T> extends PartialOrder<T> {
  invariant forall(T x, T y; leq[x, y] | leq[y, x]);
}
```

C.9.2.1 Conflicts

```
package testmodels;

TotalOrder t;

z = t.lt(x,y) ∗ c = t.lt(a,b) { false }
```

C.10 Ordered Set

```
package testmodels;

model ParamOrderedSet<T> extends Set<T> {
  TotalOrder<T> order;
  T next[T]; // next higher T that is in the set
  T prev[T]; // next lower T that in in the set

  invariant forall(T x; x = null →
                              !in[x] && next[x] = null && prev[x] = null);
  invariant forall(T x, T y;
        (x ≠ null && y ≠ null && prev[y] = null && order.lt(x,y)) →
              (prev[x] = null && !in[x]));
  invariant forall(T y, T z;
        (y ≠ null && z ≠ null && next[y] = null && order.lt(y,z)) →
              (next[z] = null && !in[z]));
  invariant forall(T x, T y;
        (x ≠ null && in[y] && order.lt(x,y)) →
              (next[x] ≠ null && order.le(next[x],y)));
  invariant forall(T y, T z;
    (z ≠ null && in[y] && order.lt(y,z)) →
        (prev[z] ≠ null && order.ge(prev[z],y)));
invariant forall(T x;
        (x ≠ null && next[x] ≠ null)
            → (in[next[x]] && order.lt(x,next[x])));
invariant forall(T x;
        (x ≠ null && prev[x] ≠ null)
            → (in[prev[x]] && order.lt(prev[x],x)));
invariant forall(T x, T z;
    (in[x] && in[z] → (((z = next[x]) = (prev[z] = x)) &&
                        ((z = prev[x]) = (next[z] = z)))));

T higher (T x)
@lock(orderedLocks, Range1D<T>(x, next[x]), S)
{
   if (x = null)
     throw new Exception();
   return next[x];
}

T lower (T x)
@lock(orderedLocks, Range1D<T>(prev[x], x), S)
{
  if (x = null)
    throw new Exception();
  return prev[x];
}

void add (T x)
[boolean present = in[x]; ]
@lock(orderedLocks, Point1D<T>(x), X)
{
  if (x = null) {
    throw new Exception();
  }
  in[x] = true;

  forall (T z; order.lt(z,x) && (next[z] = next[x])) {
    next[z] = x;
  }
  forall (T z; order.gt(z,x) && (prev[z] = prev[x])) {
    prev[z] = x;
  }
}onabort{
  if(!present){
    remove(x);
  }
}

boolean insert(T obj)
[boolean present = in[obj]; ]
@lock(orderedLocks, Point1D<T>(obj), X)
{
  in[obj] = true;
  return !present;
} onabort {
  if (!present) {
    remove(x);
  }
}

void remove (T x)
[boolean present = in[x]; ]
@lock(orderedLocks, Point1D<T>(x), X)
{
  if (x = null) throw new Exception();

  in[x] = false;

  forall (T z; next[z] = x && order.lt(z,x) ){
    next[z] = next[x];
  }
  forall (T z; prev[z] = x && order.gt(z,x) ){
    prev[z] = prev[x];
  }
}onabort{
  if(present){
    add(x);
  }
}

boolean delete(T obj)
[boolean present = in[obj]; ]
@lock(orderedLocks, Point1D<T>(obj), X)
{
  in[obj] = false;
  return present;
}onabort{
  if(present){
    add(x);
  }
}

boolean find(T x)
@lock(orderedLocks, Point1D<T>(x), S)
{
  return in[x];
}

int size()
@lock(orderedLocks, Everything<T>(), S)
{
  return sz;
}

boolean isEmpty()
@lock(orderedLocks, Everything<T>(), S)
{
      return (sz = 0);
    }
  }
```

C.10.1 Conflicts

```
  package testmodels;

  add(x) ∗ add(y) { x = y }
  add(x) ∗ z = insert(y) { x = y }
  add(x) ∗ remove(y) { x = y }
  add(x) ∗ z = find(y) { x = y }
  add(x) ∗ z = higher(y) { order.gt(x,y) && order.le(z,x) }
  add(x) ∗ z = size() { true }
  add(x) ∗ z = isEmpty() { true }
  add(x) ∗ z = lower(y) { true }
  add(x) ∗ z = delete(y) { x = y }
  add(x) ∗ z = isEmpty() { !in[x] }

  remove(x) ∗ remove(y) { x = y }
  remove(x) ∗ add(y) { x = y }
  remove(x) ∗ z = insert(y) { x = y }
  remove(x) ∗ z = find(y) { x = y }
  remove(x) ∗ z = higher(y) { order.gt(x,y) && order.le(z,x) }
  remove(x) ∗ z = size() { true }
  remove(x) ∗ z = isEmpty() { true }
  remove(x) ∗ z = lower(y) { true }
  remove(x) ∗ z = delete(y) { x = y }
  remove(x) ∗ z = isEmpty() { in[x] }

  w = find(x) ∗ z = find(y) { false }
  w = find(x) ∗ remove(y) { x = y }
  w = find(x) ∗ add(y) { x = y }
  w = find(x) ∗ z = insert(y) { x = y }
  w = find(x) ∗ z = higher(y) { false }
  w = find(x) ∗ z = size() { false }
  w = find(x) ∗ z = isEmpty() { false }
  w = find(x) ∗ z = lower(y) { false }
  w = find(x) ∗ z = delete(y) { x = y }
  w = find(x) ∗ z = isEmpty() { false }

  w = higher(x) ∗ add(y)
      { order.gt(y,x) && (w = null | order.gt(w,y)) }
  w = higher(x) ∗ z = insert(y)
      { order.gt(y,x) && (w = null | order.gt(w,y)) }
  w = higher(x) ∗ remove(y)
      { order.gt(y,x) && (w = null | order.ge(w,y)) }
  w = higher(x) ∗ z = find(y) { false }
  w = higher(x) ∗ z = higher(y) { false }
  w = higher(x) ∗ z = size() { false }
  w = higher(x) ∗ z = isEmpty() { false }
  w = higher(x) ∗ z = lower(y) { false }
  w = higher(x) ∗ z = delete(y) { true }
  w = higher(x) ∗ z = isEmpty() { false }
```

w = size() ∗ add(y) { !in[y] }
w = size() ∗ z = insert(y) { !in[y] }
w = size() ∗ remove(y) { in[y] }
w = size() ∗ z = find(y) { **false** }
w = size() ∗ z = higher(y) { **false** }
w = size() ∗ z = size() { **false** }
w = size() ∗ z = isEmpty() { **false** }
w = size() ∗ z = lower(y) { **false** }
w = size() ∗ z = delete(y) { in[y] }
w = size() ∗ z = isEmpty() { **false** }

w = lower(x) ∗ add(y)
{ order.gt(x,y) && (w = **null** | order.gt(y,w)) }
w = lower(x) ∗ z = insert(y)
{ order.gt(x,y) && (w = **null** | order.gt(y,w)) }
w = lower(x) ∗ remove(y)
{ order.gt(x,y) && (w = **null** | order.ge(y,w)) }
w = lower(x) ∗ z = find(y) { **false** }
w = lower(x) ∗ z = higher(y) { **false** }
w = lower(x) ∗ z = size() { **false** }
w = lower(x) ∗ z = isEmpty() { **false** }
w = lower(x) ∗ z = lower(y) { **false** }
w = lower(x) ∗ z = delete(y) { **true** }
w = lower(x) ∗ z = isEmpty() { **false** }

w = delete(x) ∗ add(y) { x = y }
w = delete(x) ∗ z = insert(y) { x = y }
w = delete(x) ∗ remove(y) { x = y }
w = delete(x) ∗ z = find(y) { x = y }
w = delete(x) ∗ z = higher(y) { **true** }
w = delete(x) ∗ z = size() { in[x] }
w = delete(x) ∗ z = isEmpty() { in[x] }
w = delete(x) ∗ z = lower(y) { **true** }
w = delete(x) ∗ z = delete(y) { x = y }
w = delete(x) ∗ z = isEmpty() { in[x] }

w = insert(x) ∗ add(y) { x = y }
w = insert(x) ∗ z = insert(y) { x = y }
w = insert(x) ∗ remove(y) { x = y }
w = insert(x) ∗ z = find(y) { x = y }
w = insert(x) ∗ z = higher(y) { **true** }
w = insert(x) ∗ z = size() { !in[x] }
w = insert(x) ∗ z = isEmpty() { !in[x] }
w = insert(x) ∗ z = lower(y) { **true** }
w = insert(x) ∗ z = delete(y) { x = y }
w = insert(x) ∗ z = isEmpty() { !in[x] }

w = isEmpty() ∗ add(y) { !in[y] }
w = isEmpty() ∗ z = insert(y) { !in[y] }
w = isEmpty() ∗ remove(y) { in[y] }
w = isEmpty() ∗ z = find(y) { **false** }
w = isEmpty() ∗ z = higher(y) { **false** }
w = isEmpty() ∗ z = size() { **false** }

```
w = isEmpty() ∗ z = isEmpty() { false }
w = isEmpty() ∗ z = lower(y) { false }
w = isEmpty() ∗ z = delete(y) { in[y] }
w = isEmpty() ∗ z = isEmpty() { false }
```

C.11 Queue

```
package examples;

model Queue<T> {

  T next[T];
  T prev[T];
  T head;
  T tail;
  int sz = count(T z ; next[z] ≠ null ; 1 ; 0);

  lockTable locks1D<T> queueLocks;

  invariant forall(T z ;
      (head = null —— head = tail) → next[z] = null);
  invariant forall(T z ;
      (tail = null —— head = tail) → prev[z] = null);
  invariant forall( ; tail = null → head = null);
  invariant forall( ; head = null → tail = null);

  Queue() {
    forall (T x ; true) {
      next[x] = null;
    }

    forall (T y ; true) {
      prev[y] = null;
    }

    head = null;
    tail = null;
  }

  void enqueue(T val)
  [T oldHead = head; T oldTail = tail; ]
  @lock(queueLocks, Point1D<T>(head), X)
  @lock(queueLocks, Point1D<T>(tail), S)
  {
    if (oldHead = null) {
      head = val;
      tail = val;
    } else {
      next[val] = head;
      prev[head] = val;
      head = val;
    }
  } onabort {
    next[val] = null;
    prev[oldHead] = null;
    head = oldHead;
    tail = oldTail;
  }

  T dequeue()
  [T oldTail = tail; T oldHead = head; ]
  @lock(queueLocks, Point1D<T>(tail), X)
  @lock(queueLocks, Point1D<T>(head), S)
  {
    if (oldHead = oldTail) {
      tail = null;
      prev[oldTail] = null;
    head = null;
    } else {
      tail = prev[oldTail];
      prev[oldTail] = null;
      next[tail] = null;
    }

    return oldTail;
  } onabort {
    next[tail] = oldTail;
    prev[oldTail] = tail;
    head = oldHead;
    tail = oldTail;
  }

  int size()
  @lock(queueLocks, Everything<T>(), S)
  {
    return sz + 1;
  }
}
```

C.11.1 Conflicts

```
package examples;

enqueue(x) ∗ enqueue(y) { true }
enqueue(x) ∗ a = dequeue() @pre { sz ≤ 1 }
enqueue(x) ∗ a = size() { true }

a = dequeue() ∗ enqueue(y) { sz ≤ 1 }
a = dequeue() ∗ b = dequeue() { true }
a = dequeue() ∗ b = size() { true }

a = size() ∗ enqueue(y) { true }
a = size() ∗ b = dequeue() { true }
a = size() ∗ b = size() { false }
```

C.12 Stack

```
package examples;
```

```
model Stack<T> {

  T top;
  T next[T];
  int sz = count(T z ; next[z] ≠ null ; 1 ; 0);

  lockTable locks1D<T> stackLocks;

  Stack() {
    top = null;

    forall (T x ; true) {
      next[x] = null;
    }
  }

  void push(T x)
  [T oldTop = top; ]
  @lock(stackLocks, Point1D<T>(top), X)
  {
    next[x] = top;
    top = x;
  } onabort {
    next[x] = null;
    top = oldTop;
  }

  T pop()
  [T oldTop = top; ]
  @lock(stackLocks, Point1D<T>(top), X)
  {
    top = next[oldTop];
  next[oldTop] = null;
    return oldTop;
  } onabort {
    next[oldTop] = top;
    top = oldTop;
  }

  int size()
  @lock(stackLocks, Everything<T>(), S)
  {
    return sz + 1;
  }
}
```

C.12.1 Conflicts

```
package examples;

push(x) ∗ push(y) { true }
push(x) ∗ b = pop() { true }
push(x) ∗ b = size() { true }
```

a = pop() $*$ push(y) { **true** }
a = pop() $*$ b = pop() { **true** }
a = pop() $*$ b = size() { **true** }

a = size() $*$ push(y) { **true** }
a = size() $*$ b = pop() { **true** }
a = size() $*$ b = size() { **false** }

APPENDIX D

DETAILED TOOL DESIGN

This appendix will describe and summarize the design and implementation of the ACCLAM processing research tool that was used as a proof of concept of the ideas in this thesis. It is important to understand that this code is for a prototype, meant to handle the cases spelled out in this thesis. As a consequence the repository contains dead code, and the tool itself isn't necessarily particularly user friendly. The code was also implemented by a single person (me), and was evolved incrementally while my understanding of the meaning and scope of ACCLAM were also evolving. Almost certainly, given the description of the language in this thesis, I would have designed the tool very differently.

D.1 The Code

The code is all in the ALI SVN repository under the *lock inference* repository. The tool is implemented in Java, and all the development was done with Eclipse (and ANTLRWorks for the parsing part). There is no particular integration with a build tool like ANT or maven, so initially Eclipse will have to be used.

D.1.1 Dependencies

The dependencies are those needed for ANTLR 3 support as well as a release of sat4j. They are stored in `jar` form in the `lib` directory off the root.

D.1.2 Source

The source code is located in `sat-builder/src`, and that's where all the back-end code as well as the grammar files and generated Java for the front-end live.

D.1.3 Tests

The tests are JUnit4-based and are located in `sat-builder/test`. These are all end-to-end tests, so they run example models and conflict predicates through the tool and compare the results against provided expected results.

D.2 The Tool

The tool itself consists of several parts. There is the front-end which parses and does static analysis of the model descriptions. There is the back-end which distills the output of the front-end down to simple expressions. There is the circuit builder which takes the simple expressions and converts them to SAT clauses. Then there are the verifiers that glues these parts together to solve a specific problem. And last, there's the driver that handles all the quotidian details like churning through command-line arguments and deciding which verifier to run on which files.

D.2.1 The Front End

The front end consists of a parser, classes for an AST (abstract syntax tree), a type-checker, and a name resolver/scope checker.

The parser is generated by ANTLR 3, and the grammar file is located in the package `sat_builder.parser`. I personally used ANTLRWorks to build the grammar, but that is not a requirement. In fact, the AST is not an ANTLR-produced one, so the parser can be swapped out if desired.

The AST classes are defined in `sat_builder.tree` and correspond to all the statement and expression types described in Chapter 5. The tool doesn't use particularly fancy intermediate representations, so the various phases will 'lower' parts of the AST into either annotated instances of the same types, or more expression-like analogues. The standard way that the AST is traversed is with a visitor pattern, the base class of which is `sat_builder.tree.Visitor`.

The name resolver is named, due to historical accident, `ScopeResolver`, and it is located in `sat_builder.types`. It started life as a scope checker/resolver, and grew into a more general-purpose name resolver.

The type checker is named `sat_builder.types.TypeResolver`, and it handles all the normal Java type checking. However, its support for sophisticated use of type parameters is not well-tested. Since fancy type parameterization was not the focus of this research, only the type parameter support necessary to handle the simple inheritance hierarchies in the examples was exercised.

D.2.2 The Back End

The back end of the ACCLAM tool takes a type- and name-decorated AST and then lowers it incrementally into something that is more expression-like. The package that contains all the back end logic is `sat_builder.circuit`. Unfortunately, most of the logic is all in a single class, `ExprExpander`. Over time it became a kind of kitchen sink object that I never had sufficient free time to decompose. This class also contains some of the oldest and most tortured code because it weathered many significant changes in the direction of the work as well as my understanding of the ACCLAM language and its processing. I will do my best to describe the way it's intended to work.

In general, an ExprExpander is a kind of context object. It contains maps of models and state variable names to data structures or expressions that describe them. In general, the desire is to be able to, for each method in each model produce an expression for each element of that model's state. For state elements where chaining together two methods isn't simply making the output of one expression the input to another, the state is summarized by a data structure that can be chained and then processed to produce the desired expression. For example, scalarization produces a series of nested if expressions. It was more convenient in the code to store the state of a relation as a pair of ordered lists (one of assignments and one of dereferences). Each element in that list stores the expressions for the indexes and the expression for the relation element (if an assignment). This list is processed to produce the chain of if-then-else's. Lists were more convenient because changing the order of two methods just becomes appending or prepending the lists for each method together.

A model is processed by an `ExprExpander` in several phases. The first phase is to sweep through the model and build up a context for the model's state. At this point, uninitialized values are created for non-relation variables and fields.

The next phase is to process each method. The goal of this phase is to produce, for each state variable, an expression (or expression-like thing) that summarizes the method's transformation of that state variable. Within these expressions, the free variables will be the state at method entry and the arguments to the method. This is accomplished by rewriting the expressions to read the various typed `Var` expressions. They are generated by this processing pass, and are basically place holders for reading state declared outside the method. When methods are invoked back-to-back, a visitor replaces the free variables with the appropriate expressions flowing from the prior method (technically, this also happens with the first method, except it just reads an initial state). Relations and reductions are processed specially here. As mentioned above, they are summarized in a list-like fashion. Within a method, any read of a relation or reduction variable is rewritten to be a read of an appropriately typed guaranteed-unique variable (these are the `VarRelTemp` and `VarRedTemp` types). These particular `Var` types contain additional information so that their relative position within the list-like state is known. Then, when the list-like state representations are concatenated, those relative positions are used to derive sub-lists that are used to produce intermediate versions of the relation or reduction's state. Each of the particular state expressions is conditioned by the control flow information in a fashion described in Chapter 7. Since ACCLAM doesn't allow recursion, method invocations just become inlinings of the called method's definition.

The next phase is to take the method's state expressions and then derive a return expression for each method by re-visiting the method. This could be done in conjunction with the prior phase, but at the time return support was wired into the tool, this approach was more convenient. By building up a control-flow expression as the method is visited, an expression for the return value of the method as an expression of the input state and arguments is derived. This expression will include the `Var` expressions as well, so the return expressions can be rewritten to reflect different method invocation orders. This phase is also where exception throwing is processed, as part of the return. The tool can't currently handle exception catching, so a thrown exception is considered a different kind of return value that will equal only the exact same exception if compared.

Finally, any method handlers are processed (e.g., `onAbort`). These are just like methods, except their input states will be the output states of the forward-going method. Their contexts are stored in a separate set of maps under the forward-going method's name.

In each of these phases, as variables are encountered, they are added to various sets organized by type. These are used later as inputs to expanding universally quantified invariants.

At this point, the `ExprExpander` for this particular model has a lot of context. For each state variable, there is an initial state, and for each method there is a mapping from state variables to expressions. Additionally, there are the return expressions for each method as well. The rest of the machinery in `ExprExpander` performs the Var substitutions and converts the list-like data structures for relations and reductions into expressions. All of this context is exactly what a verifier needs in order to produce a concrete expression to feed into a circuit builder.

D.2.3 The Verifiers

A verifier is a user of an `ExprExpander` that produces some expressions that are fully bound to state variables and are suitable for feeding into something that can convert expressions into SAT problems. The verifiers are located in `sat_builder.verify`.

There are also various utilities within that package for stitching together the expressions coming out of the `ExprExpander`. The classes are:

- `ConflictVerifier` the verifier for conflict predicates
- `NewLockVerifier` the verifier for lock predicate correctness
- `InverseVerifier` the verifier for inverse correctness
- `ForallSideVerifier` the verifier for `forall` side conditions
- `ChooseSideVerifier` the verifier for `choose` side conditions
- `ConflictPredicateTightnessVerifier` the verifier for conflict predicate precision testing
- `LockTightnessVerifier` the verifier for locking protocol precision testing

Additionally, the classes for stitching together relation state lists, and for scalarizing relation expressions are located in this package. In general, each verifier has a 'root' method that performs the verification on a given problem configuration. This method is invoked by the driver and it produces a `Circuit` which is a circuit expression amenable to SAT processing. Unfortunately, due to time constraints, I was unable to rationalize the root methods across all the verifiers and come up with a common, reasonable interface.

The general flow in the verifiers is:

1. build up state expressions (e.g., for each state variable v, is $v_{op1;op2} = v_{op2;op1}$?)
2. build up return value expressions
3. scalarize the generated expressions
4. build up the top-level question in terms of the scalarized state
5. build the invariant expressions in terms of the variables from the methods

6. convert all the expressions into a single circuit-like expression

Scalarization is just if-then-else style ackermannization, and most of the art is in visiting things in a way that doesn't produce cycles (an artifact of the implementation). Producing the invariant expressions requires accessing the sets of variables gathered by the `ExprExpander`, and using them to exhaustively instantiate invariant expressions for the given problem. This code isn't particularly sophisticated, and so it will instantiate more expressions than it may strictly have to (e.g., it will produce both `a == b` and `b == a`, even though they are symmetric).

D.2.4 The Circuit Builders

Once the verifier has built up a problem into an expression that is free of `Vars`, then it can be converted into a circuit. The basic circuit primitives are in the `circuitGen` package. They consist of `CircuitVals` and `CircuitGates`. A `CircuitVal` is a value, and is an ordered tuple of boolean variables. A constant is just a `CircuitVal` with an added constraint that forces each variable to be a particular value. A `CircuitGate` is an abstraction of a gate that accepts one or more input `CircuitVals` and produces a `CircuitVal`. A `Circuit` is a set of input `CircuitVals`, a set of `CircuitGates` and a set of output `CircuitVals`. The final outputs are often constrained to be a particular value.

The `CircuitBuilder` class itself is an expression visitor that converts expressions into circuit values and gates, and adds them incrementally to an internal circuit object. The `FoldingCircuitBuilder` is a circuit builder that does some constant propagation and expression simplification as it processes the input expressions. It also contains a `MemoizedCircuit` which is a child class of `Circuit` that does circuit memoization. Circuit memoization works by maintaining a cache of circuits. The reason for using a cache rather than an unbounded map is that when handling large circuits, a map could grow to fill up memory. Therefore, I bounded the map by size and turned it into a cache. On modern hardware, memory is reasonably plentiful, so the cache limit may be too small. The cache is keyed by a canonically sorted list of inputs and the gate type, and the value is the output circuit value of the resulting gate. What this means is that for circuits that can fit in the cache, a given gate for the same inputs will always just refer to the same output circuit value (rather than recomputing it with a new set of clauses). For circuits that don't fit in the cache, memoization is still very useful, it's just not going to be guaranteed to simplify circuits in all cases. All circuits can convert themselves to SAT form as well as a DOT graph form.

D.2.5 The Driver

The driver is located in the top-level package and is called `Verifier`. It's job is to parse command line arguments, collect the set of models and conflicts to be analyzed, feed them all through individual `ExprExpander`s and to then run the verifier code on the child model to be verified, take the circuit and convert it into SAT form and feed it to the sat4j library. All of this is fairly straightforward. The most idiosyncratic thing is that all the command-line arguments are processed by the `VerifierFlags` class, rather than some external tool.

D.3 Testing and Debugging

Now that you have a rough idea as to how the whole thing is designed, how do you test and debug it? The verifier has a main line, so it can be run directly from the command line, and there is an abandoned testing system in the `test` directory that does this via python. There is a unit test harness that runs against a set of model files located in `sat-builder/test/testmodels`. The unit test programmatically inspects the output SAT state produced by the verifier, but the verifier also produces text output for command-line inspection.

If you encounter a bug,[1] how do you go about debugging it? The two main tools I used were the Eclipse debugger and the problem circuit DOT files. The verifier allows the user to specify a particular set of methods to verify (with the `-method` command-line argument), so that can be used to narrow the bug down to a single problem. Once that is done, the `-dot` flag can be used to produce a graph version of the problematic circuit (rather than feed it the SAT solver). The DOT graph's nodes are tagged with unique identifiers (e.g., AND123), and the output of the tool can be used to determine for which expressions that graph node was produced. At this point, I switch over to the Eclipse debugger and examine the expression as it flows to the circuit builder. When run with the `-debug` flag, each AST object is tagged with an allocation site. This can be used to figure out which part of the system produced the problem expression. Then I work backwards to figure out the chain of decisions that resulted in the mal-formed expression.

D.4 Known Issues

There are several categories of known issues:

Inheritance: the test models don't exercise this to a great extent, and there have been problems with the incorrect implementations of methods being chosen. In the short term, these issues can be worked around by manually including parent model state. But for generality and utility, the bugs will need to be sorted out.

Instances: By instances I mean having multiple named instances of models. The system was implemented initially assuming an implicit single instance. The instance support was added later and is incomplete. One of the main issues is that instances of the same type can cause the tool to alias state incorrectly. All the instances of this I encountered, I

[1] I apologize, but there are most assuredly bugs lurking in this code!

fixed. However, I suspect that a better approach would be to rework the tool itself to do all its processing relative to an instance as well as a model.

Reduction Initial State: Reductions need to have a set of constraints built around them to make sure that any initial relation state will produce a correct reduction state. Historically, errors in these constraints have caused a disproportionate number of bugs. These constraints were added after the initial reduction code, and it may be more productive to rework the whole constraint building mechanism.