Rigorous Concurrency Analysis of Multithreaded Programs

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom*
School of Computing, University of Utah
{yyang | ganesh | gary}@cs.utah.edu

ABSTRACT

This paper explores the practicality of conducting program analysis for multithreaded software using constraint solving. By precisely defining the underlying memory consistency rules in addition to the intra-thread program semantics, our approach offers a unique advantage for program verification — it provides an *accurate* and *exhaustive* coverage of all thread interleavings for any given memory model. We demonstrate how this can be achieved by formalizing sequential consistency for a source language that supports control branches and a monitor-style mutual exclusion mechanism. We then discuss how to formulate programmer expectations as constraints and propose three concrete applications of this approach: execution validation, race detection, and atomicity analysis. Finally, we describe the implementation of a formal analysis tool using constraint logic programming, with promising initial results for reasoning about small but non-trivial concurrent programs.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Languages, Verification, Reliability

Keywords

Multithreaded programming, constraint solving, data races, race conditions, atomicity, memory consistency models, static checking

*This work was supported in part by Research Grant No. CCR-0081406 (ITR Program) of NSF and SRC Task 1031.001.

.

1. INTRODUCTION

Unlike a sequential program, which simply requires that each read observes the latest write on the same variable according to program order, a multithreaded program has to rely on the *thread semantics* (also known as the *memory model*) to define its legal outcome in a shared memory environment. The most commonly known memory model is *sequential consistency* (SC) [1]. As a natural extension to the sequential model, sequential consistency requires that (i) operations of all threads can exhibit a total order, (ii) operations of each individual thread appear in this total order following program order, and (iii) a read observes the latest write on the same variable according to this total order. Many weaker shared memory systems (see [2] for a survey) have also been developed to enhance performance.

Java is the first widely deployed programming language that provides built-in threading support at the language level. Unfortunately, developing a rigorous and intuitive Java Memory Model (JMM) has turned out to be very difficult. The existing JMM is flawed [3] due to the lack of rigor. It is currently under an official revision process and a new JMM draft [4] is proposed for community review.

Although multithreading provides a powerful programming paradigm for developing well structured and high performance software, it is also notoriously hard to get right. Programmers are torn on the horns of a dilemma regarding the use of synchronization: too much may impact performance and risk deadlock, too little may lead to race conditions and application inconsistency. Therefore, a formal analysis about thread behaviors is often needed to make a program more reliable. However, this can become a daunting task with a traditional pencil-and-paper approach.

For example, one common analysis is *race detection*. Consider program 1 in Figure 1 (taken from [4]), where each thread issues a read and a conditional write. Does this program contain data races? At the first glance, it may appear that the answer is "yes" since it seems to fit the conventional intuition about a race condition — two operations from dif-

Thread 1	Thread 2
$r1 = x;$	$r2 = y;$
if$(r1 > 0)$	if$(r2 > 0)$
$y = 1;$	$x = 1;$

(a) Program 1

Thread 1	Thread 2
$r1 = x;$	$r2 = y;$
if$(r1 > 0)$	if$(r2 >= 0)$
$y = 1;$	$x = 1;$

(b) Program 2

Figure 1: Initially, x = y = 0. Are these programs race-free?

Thread 1 (Deposit)	Thread 2 (Withdraw)
Lock $l1$;	Lock $l1$;
$r1$ = balance;	$r3$ = balance;
Unlock $l1$;	Unlock $l1$;
$r2 = r1 + 1$;	$r4 = r3 - 1$;
Lock $l1$;	Lock $l1$;
balance = $r2$;	balance = $r4$;
Unlock $l1$;	Unlock $l1$;

Figure 2: The transactions are not atomic even though the program is race-free.

ferent threads (e.g., $r1 = x$ in thread 1 and $x = 1$ in thread 2) attempt to access the same variable without explicit synchronization, and at least one of them is a write. However, a more careful analysis reveals that the control flow of the program, which must be consistent with the read values allowed by the memory model, needs to be considered to determine whether certain operations will ever happen. Therefore, before answering the question, one must clarify what memory model is assumed. With sequentially consistent executions, for instance, the branch conditions in program 1 will never be satisfied. Consequently, the writes can never be executed and the code is race-free. Now consider program 2 in Figure 1, where the only difference is that the branch condition in Thread 2 is changed to $r2 >= 0$. Albeit subtle, this change would result in data races.

Another useful analysis is *execution validation*, which is for verifying whether a certain outcome is permitted. This can help a programmer understand the memory ordering rules and aid them in code selection. Similar to a race analysis, data/control flow must be tracked for execution validation. For multithreaded programs, data/control flow is interwoven with shared memory consistency requirements. This makes it extremely hard and error-prone to hand-prove thread behaviors, even for small programs.

The *atomicity* requirement is also frequently needed to ensure that a set of operations appears to be executed atomically. As pointed out in previous publications (e.g., [5] [4]), the absence of race conditions does not guarantee the absence of atomicity violations. For example, consider the program in Figure 2 (adapted from [5]). Thread 1 and 2 respectively implement the deposit and withdraw transactions for a bank account. This program is race-free because all accesses to the global variable *balance* are protected by the same lock. If these two threads are issued concurrently when *balance* is initially 1, *balance* should remain the same after the program completes if implemented properly. But due to the use of local variables, one thread can interleave with another while in a "transient" state. Consequently, the final balance can be 0, 1, or 2 depending on the scheduling, which is clearly not what has been intended. Hence, it would be highly desirable to have a systematic approach to specifying and verifying such programmer expectations.

From these examples, several interesting conclusions can be drawn.

- The precise thread semantics, in addition to the intra-thread program semantics, must be taken into account to enable a rigorous semantics analysis of multithreaded software, because a property that is satisfied under one memory model can be easily broken under another.
- Program properties such as race conditions and atomicity need to be formalized because informal intuitions often lead to inaccurate results.
- An automatic verification tool with exhaustive coverage is extremely valuable for general software development purposes because thread behaviors are often confusing.

Based on these observations, we propose a systematic approach for reasoning about multithreaded software. Our key insight is that by capturing thread semantics and correctness properties as constraints, we can reduce a verification problem to a constraint satisfaction problem or an equivalent boolean satisfiability problem, thus allowing us to employ an efficient constraint/SAT solver to automate the analysis. Using a verification tool harnessed with these techniques, we can configure the underlying memory model, select a program property of interest, take a test program as input, and verify the result automatically under all executions. Existing analysis tools tend to concentrate on efficiency and scalability. While these are highly worthy goals, there is also a clear need for supporting exhaustive analysis. To the best of our knowledge, there has not been a generic constraint-based tool that enforces precise shared memory consistency requirements.

This paper offers the following contributions. (i) We develop a formal executable specification of sequential consistency for a non-trivial source language that supports the use of local variables, computations, control branches[1], and a monitor-like mechanism for mutual exclusion. One key result of this paper is to show that it is feasible and beneficial to precisely capture both program semantics (including local data dependence and local control dependence) and memory model semantics in the same setting. As far as we know, no one has provided such a formal executable specification. (ii) We propose a method to formulate program properties as constraints and automatically verify them using constraint solving. In particular, we formalize the conditions of three critical safety properties: execution legality, race conditions, and atomicity requirements. (iii) We build a tool using constraint logic programming (CLP) and report the experiences gained during the implementation.

The rest of the paper proceeds as follows. In Section 2, we provide an overview of our approach. Section 3 describes the source language used as the basis of our presentation. In Section 4, we apply our approach for execution validation. Race conditions and atomicity requirements are formalized in Section 5 and 6, respectively. We discuss the implementation of our prototype tool in Section 7. Related work is reviewed in Section 8. We conclude and explore future work in Section 9. The detailed formal specification is presented in the Appendix.

2. OVERVIEW

The approach is based on our previously developed specification framework called Nemos (Non-operational yet Executable Memory Ordering Specifications) [6] [7]. Nemos

[1]Currently our formal executable specification does not directly support loops.

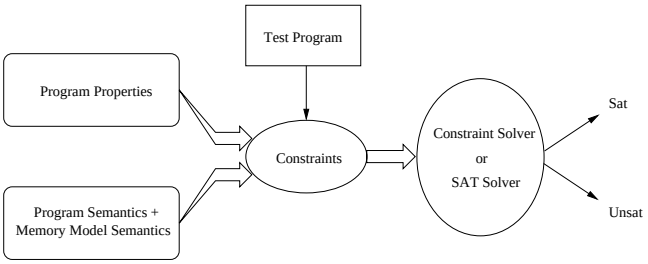

Figure 3: The processing flow of our approach.

defines a memory model in a declarative style using a collection of ordering rules. The processing flow of our verification methodology is shown in Figure 3, which comprises the following steps: (i) capturing the semantics of the source language, including the thread semantics, as constraints, (ii) formalizing program properties as additional constraints, (iii) applying these constraints to a given test program and reducing the verification problem to a constraint satisfaction problem, and (iv) employing a suitable tool to solve the constraint problem automatically.

2.1 Specifying the Constraints

We use predicate logic to specify the ordering constraints imposed on an ordering relation *order*. To make our specifications compositional and executable, our notation differs from previous formalisms in two ways. First, we employ a modest extension of predicate logic to higher order logic, i.e. *order* can be used as a parameter in a constraint definition, so that new refinements to the ordering requirement can be conveniently added. This allows us to construct a complex model using simpler components. Second, our specifications are fully explicit about all ordering properties, including previously implicit requirements such as totality, transitivity, and circuit-freedom. Without explicating such "hidden" requirements, a specification is not complete for execution.

2.2 Solving the Constraints

After the language semantics and program properties are specified as constraints, they can be applied to a finite execution trace. This process converts the system requirements from higher order logic to propositional logic.

The Algorithm: Given a test program $\mathcal{P}$, we first derive its execution *ops* from the program text in a preprocessing phase. The initial execution is fully *symbolic*, that is, *ops* may contain free variables, e.g., for data values and ordering relations. Suppose *ops* has n operations, there are n^2 ordering pairs among these operations. We construct a $n \times n$ adjacency matrix $\mathcal{M}$, where the element $\mathcal{M}_{ij}$ indicates whether operations i and j should be ordered. We then go through each requirement in the specification and impose the corresponding propositional constraints with respect to the elements of $\mathcal{M}$. The goal is to find a binding of the free variables in *ops* such that it satisfies the conjunction of all requirements or to conclude that no such binding exists. This is automated using a constraint solver.

3. THE SOURCE LANGUAGE

This section develops the formal semantics of sequential consistency for a source language that supports many common programming language constructs. The choice of using sequential consistency as the basis of our formal development is motivated by two factors. (i) SC is often the implicitly assumed model during software development, i.e., many algorithms and compilation techniques are developed under the assumption of SC. (ii) Many weak memory models, including the new JMM draft, define pivotal properties such as race-freedom using SC executions. Providing such an executable definition of race conditions will provide a solid foundation based on which a full-featured Java race-detector can be built.

Although this paper formalizes only SC, our framework is generic and allows an arbitrary memory model to be "plugged-into" the system for a formal comparison analysis. In our previous work, we have already applied Nemos to build a large variety of memory model specifications, including the Intel Itanium Memory Model [7] and a collection of classical memory models [6], such as sequential consistency, coherence, PRAM, causal consistency, and processor consistency.

Our previous specification of sequential consistency in [6] only deals with normal read and write operations. While this is sufficient for most processor level memory systems, it is not enough for describing language level thread activities. In order to handle more realistic programs, this paper extends the previous model by supporting a language that allows the use of local variables, computation operations, control branches, and synchronization operations.

3.1 Terminology

Variables: Variables are categorized as *global variables*, *local variables*, *control variables*, and *synchronization variables*. Global and synchronization variables are visible to all threads. Local and control variables are thread local. Control variables do not exist in the original source program — they are introduced by our system as auxiliary variables for control operations. Synchronization variables correspond to the locks employed for mutual exclusion. In this paper, we follow a convention that uses x, y for global variables, $r1$, $r2$ for local variables, $c1$, $c2$ for control variables, $l1$, $l2$ for synchronization variables, and 0, 1 for primitive data values.

Instruction: An *instruction* corresponds to a program statement from the program text. The source language has a syntax similar to Java, with *Lock* and *Unlock* inserted corresponding to the Java keyword *synchronized*. It supports the following instruction types:

Read:	e.g., $r1 = x$
Write:	e.g., $x = 1$, or $x = r1$
Computation:	e.g., $r1 = r2 + 1$
Control:	e.g. $if(r1 > 0)$
Lock:	e.g. $Lock\ l1$
Unlock:	e.g. $Unlock\ l1$

Execution: An *execution* consists of a set of symbolic execution instances generated by program instructions, each of which is called an *operation*. We assume that the expression involved in a computation or control operation only uses local variables. That is, if the original instruction performs a computation on global variables, it will be divided into read operations followed by computation operations.

Operation Tuple: An operation i is represented by a tuple. For brevity, we use a common data structure to represent

all operations, even though some of the fields are only used by certain operations. The tuple representation is as follows:

$\langle t, pc, op, var, data, local, localData, cmpExpr, ctrExpr, lock, matchID, id\rangle$, where

t $i = t$:	thread ID
pc $i = pc$:	program counter
op $i = op$:	operation type
var $i = var$:	global variable
data $i = data$:	data value
local $i = local$:	local variable
localData $i = localData$:	data value for local variable
cmpExpr $i = cmpExpr$:	computation expression
ctrExpr $i = ctrExpr$:	path predicate
lock $i = lock$:	lock
matchID $i = matchID$:	ID of the matching lock
id $i = id$:	global ID of the operation

For every global variable x, there is a default write operation for x, with the default value of x and a special thread ID t_{init}. We assume Lock and Unlock operations are properly nested. Each trailing Unlock stores the id of the matching Lock in its $matchID$ field.

3.2 Semantics

3.2.1 Control Flow

It is a major challenge to specify control flow in the context of nondeterministic thread interleavings. We solve this problem by (i) transforming control related operations to auxiliary reads and writes using control variables and (ii) imposing a set of consistency requirements on the "reads" and "writes" of control variables similar to that of normal reads and writes. The detailed steps are as follows:

- For each branch instruction i, say $if(p)$, add a unique auxiliary control variable c, and transform instruction i to an operation i' with the format of $c = p$. Operation i' is said to be a control operation (**op** $i' = Control$), and can be regarded as an *assignment* to the control variable c.
- Every operation i has a $ctrExpr$ field that stores its *path predicate*, which is a boolean expression on a set of control variables dictating the condition for i to be executed. An operation i can be regarded as a *usage* of the involved control variables in the path predicate. Without loops, the path predicate for every operation can be determined during the preprocessing phase. This can be achieved based on a thread local analysis since control variables are thread local.
- An operation i is *feasible* if its $ctrExpr$ field evaluates to $True$. We define a predicate **fb** to check the feasibility of an operation.
- In the memory ordering rules, feasibility of the involved operations is checked to make sure the consistency of control flow is satisfied.

By converting control blocks to assignments and usages of control variables, we can specify consistency rules for control flow in a fashion similar to data flow.

3.2.2 Loops

Loops are not directly supported in this specification. For the purpose of defining a memory model alone, nonetheless, our mechanism for handling control operations is sufficient for loops. This is because the task of a memory model specification can be regarded as answering the question of whether a given execution is allowed by the memory model. For any concrete terminated execution, loops have already been resolved to a finite number of iterations.

However, to enable a fully automatic and exhaustive program analysis involving loops, another level of constraints need to be developed so that the path predicate of an operation can conditionally grow. Another technique, as used by tools such as Extended Static Checker for Java (ESC/Java) [8], is to rely on the user to supply loop invariants — loops without invariants are handled in a manner that is unsound but still useful. This approach can be adopted by our system as well. As a future work, we plan to investigate effective approaches for handling loops.

3.2.3 Formal Specification

The semantics of the source language is defined as a collection of constraints. The detailed specification is presented in Appendix A. This section explains each of the rules.

As shown below, predicate `legalSC` is the overall constraint that defines the requirement of sequential consistency on an execution *ops* in which the operations follow an ordering relation *order*.

legalSC *ops order* $\equiv$
  requireProgramOrder *ops order* $\wedge$
  requireReadValue *ops order* $\wedge$
  requireComputation *ops order* $\wedge$
  requireMutualExclusion *ops order* $\wedge$
  requireWeakTotalOrder *ops order* $\wedge$
  requireTransitiveOrder *ops order* $\wedge$
  requireAsymmetricOrder *ops order*

Program Order Rule (Appendix A.1):
Constraint `requireProgramOrder` specifies that operations should respect program order, which is formalized by predicate `orderedByProgram`. In addition, the default writes are ordered before other operations.

Read Value Rules (Appendix A.2):
Constraint `requireReadValue` enforces the consistency of data flow across reads and writes. Informally, it requires that for each read k: (i) there must exist a suitable write i providing the data and (ii) there does not exist an overwriting write j between i and k. The assignments and usages of local variables (local data dependence) and control variables (local control dependence) follow the similar guideline to ensure consistent data transfer. Therefore, `requireReadValue` is decomposed into three subrules for global reads, local reads, and control reads, respectively. Because we apply *unique* control variables, `controlReadValue` does not need to check the second case listed above.

Computation Rule (Appendix A.3):
Constraint `requireComputation` enforces the program semantics. It is not directly related to the memory ordering, but is needed for analyzing realistic code. It requires that for every operation involving computations (i.e., when the op-

eration type is *Computation* or *Control*), the resultant data must be obtained by properly evaluating the expression in the operation. For brevity, the Appendix omits some details of the standard program semantics that is usually well understood. For example, we use a predicate `eval` to indicate that standard process should be followed to evaluation an expression. Similarly, `getLocals` and `getCtrs` are used to parse the *cmpExpr* and *ctrExpr* fields to obtain a set of ⟨variable, data⟩ entries involved in the expressions (these entries represent the local/control variables that the operation depends on and their associated data values), which can be subsequently processed by `getVar` and `getData`.

Mutual Exclusion Rule (Appendix A.4):
Constraint `requireMutualExclusion` enforces mutual exclusion for the block of operations enclosed by matched Lock and Unlock operations.

General Ordering Rules (Appendix A.5):
These constraints require *order* to be transitive, total, and circuit-free (asymmetric).

4. EXECUTION VALIDATION

A direct application of this formal specification is for execution validation. Studying thread behaviors with small code fragments (generally known as *litmus tests*) is very helpful for understanding the implications of a threading model. In fact, many memory model proposals rely on a collection of litmus tests to illustrate critical properties. In [9] [10], we have also demonstrated the effectiveness of abstracting a common programming pattern (such as the Double-Checked Locking algorithm or Peterson's algorithm) as a litmus test, thus facilitating a formal analysis.

While defining the legality of a thread behavior is the common goal for all memory model specifications, the ability to automate such analysis has been lacking in previous declarative specification methods. Our system supports such analysis by allowing a user to add annotations about the read values, and verifying those assertions automatically via constraint solving.

Constraint `validateExecution` verifies whether a given execution *ops* is legal under the formal model.

validateExecution *ops* $\equiv (\exists$ *order*. **legalSC** *ops order*$)$

5. RACE DETECTION

Race conditions are usually inadvertently introduced and may lead to unexpected behaviors that are hard to debug. Therefore, catching these potential defects is highly useful for developing reliable software. Furthermore, many relaxed memory systems guarantee that race-free programs behave in the same way as sequentially consistent programs, which allows programmers to resort to their intuitions about SC during software development. This also makes race-detection even more important in practice.

Our definition of a data race is according to [11], which has also been adopted by the new JMM draft [4]. In these proposals, a *happens-before* order (based on Lamport's *happened-before* order [12] for message passing systems) is used for formalizing *concurrent* memory accesses. Further, data-race-free programs (also referred to as *correctly synchronized programs*)is clarified as being free of conflicting and concurrent accesses under all sequentially consistent executions. The reason for using SC executions to define data races this to make it easier for a programmer to determine whether a program is correctly synchronized.

We define constraint `detectDataRace` to catch any potential data races. It does this by trying to find a total order *scOrder* and a *happens-before* order *hbOrder* such that there exists a pair of conflicting operations which are not ordered by *hbOrder*. This formalizes the notion of data races under sequentially consistent executions.

detectDataRace *ops* $\equiv \exists$ *scOrder*, *hbOrder*.
  legalSC *ops scOrder* $\wedge$
  requireHbOrder *ops hbOrder scOrder* $\wedge$
  mapConstraints *ops hbOrder scOrder* $\wedge$
  existDataRace *ops hbOrder*

Happens-before order is defined in `requireHbOrder`. Intuitively, it states that two operations are ordered by happens-before order if (i) they are program ordered, (ii) they are ordered by synchronization operations, or (iii) they are transitively ordered by a third operation.

requireHbOrder *ops hbOrder scOrder* $\equiv$
  requireProgramOrder *ops hbOrder* $\wedge$
  requireSyncOrder *ops hbOrder scOrder* $\wedge$
  requireTransitiveOrder *ops hbOrder*

Since sequential consistency requires a total order among all operations, the happens-before edges induced by synchronization operations must follow this total order. This is captured by `requireSyncOrder`. Similarly, `mapConstraints` is used to make sure *scOrder* is consistent with *hbOrder*.

requireSyncOrder *ops hbOrder scOrder* $\equiv \forall\ i\ j \in$ *ops*.
  (**fb** i $\wedge$ **fb** j $\wedge$ **isSync** i $\wedge$ **isSync** j $\wedge$ **scOrder** $i\ j$)
  $\Rightarrow$ **hbOrder** $i\ j$

mapConstraints *ops hbOrder scOrder* $\equiv \forall\ i\ j \in$ *ops*.
  (**fb** i $\wedge$ **fb** j $\wedge$ **hbOrder** $i\ j$) $\Rightarrow$ **scOrder** $i\ j$

With a precise definition of happens-before order, we can formalize a race condition in constraint `existDataRace`. A race is caused by two feasible operation that are (i) conflicting, i.e., they access the same variable from different threads (**t** $i \neq$ **t** j), and at least one of them is a write, and (ii) concurrent, i.e., they are not ordered by happens-before order.

existDataRace *ops hbOrder* $\equiv \exists\ i, j \in$ *ops*.
  fb $i \wedge$ **fb** j $\wedge$ **t** $i \neq$ **t** j $\wedge$ **var** $i =$ **var** j $\wedge$
  (**op** $i = Write$ $\wedge$ **op** $j = Write$ $\vee$
  op $i = Write$ $\wedge$ **op** $j = Read$ $\vee$
  op $i = Read$ $\wedge$ **op** $j = Write$) $\wedge$
  $\neg$(**hbOrder** $i\ j$) $\wedge$ $\neg$(**hbOrder** $j\ i$)

To support race analysis for the new JMM proposal, this race definition needs to be extended, e.g., by adding semantics for volatile variable operations — which should be a relatively straightforward process.

6. ATOMICITY VERIFICATION

Atomicity ensures certain atomic transactions. If atomicity can be verified, a compiler may ignore the fine-grained in-

terleavings and apply standard sequential compilation techniques when treating an atomic block. However, race-freedom is neither *necessary* nor *sufficient* to ensure atomicity. As shown by the example in Figure 2, a monitor-style mutual exclusion mechanism, if used improperly, cannot guarantee atomicity even if the code is race-free. Therefore, a different mechanism is needed to specify and verify atomicity.

For this purpose, we allow a programmer to annotate an atomic block by enclosing it with keywords *AtomicEnter* and *AtomicExit*. To simplify some implementation details, we assume that the annotations are properly inserted. For the operation tuple, we add three more fields: $abEnter$, $abExit$, and $matchAbID$.

abEnter $i = abEnter$:	if i is the start of an atomic block;
abExit $i = abExit$:	if i is the end of an atomic block;
matchAbID $i = matchAbID$:	ID of the matching start of the atomic block.

During the preprocessing phase, we setup the operation i that immediately *follows* an *AtomicEnter* with **abEnter** $i = True$. Similarly, we setup the operation j that immediately *precedes* the matching *AtomicExit* with **abExit** $j = True$. We also record the id of i into the $matchAbID$ field of j (**matchAbID** j = **id** i). Given an execution *ops* transformed from an annotated program, we can use constraint `verifyAtomicity` to catch atomicity violations.

verifyAtomicity *ops* $\equiv \exists$ *order*.
  legalSC *ops order* $\wedge$
  existAtomicityViolation *ops order*

existAtomicityViolation *ops order* $\equiv$
  $\exists\, i, j, k \in$ *ops*.
  (**fb** i $\wedge$ **fb** j $\wedge$ **fb** k $\wedge$
  abEnter i $\wedge$ **abExit** j $\wedge$
  id i = **matchAtID** j $\wedge$ **id** $i \neq$ **id** j $\wedge$
  isViolation k $\wedge$
  $\neg$(**order** k i) $\wedge$ $\neg$(**order** j k))

isViolation k $\equiv$ (**t** $k \neq$ **t** i)

The definition of `existAtomicityViolation` is generic, in that `isViolation` can be fine-tuned to capture other desired semantics. For illustration purposes, we only provide a very strong requirement here. It states that no operation from another thread can be interleaved between the atomic block. In practice, it is benign to interleave certain operations as long as the effect cannot be observed. For example, it might be desirable to define a "variable window" (a set of variables manipulated within an atomic block) and only detect an atomicity violation when the intruding operation "overlaps" the variable window.

7. IMPLEMENTATION

Constraint-based analyses can be quickly prototyped using a constraint logic programming language such as FD-Prolog[2]. We have built a tool named *DefectFinder* to test the proposed techniques. Our prototype is written in SICStus Prolog [13]. This section discusses the implementation of this tool and the experiences gained.

[2]FD-Prolog refers to Prolog with a finite domain (FD) constraint solver. For example, SICStus Prolog and GNU Prolog have this feature.

7.1 Constraint Solver

Two mechanisms from FD-Prolog can be applied for solving the constraints in our specification. One applies backtracking search for all constraints expressed by logical variables, and the other uses non-backtracking constraint solving techniques such as *arc consistency* [14] for finite domain variables, which is potentially more efficient and certainly more complete (especially under the presence of negation) than with logical variables. This works by adding constraints in a monotonically increasing manner to a constraint store, with the built-in constraint propagation rules of FD-Prolog helping refine the variable ranges when constraints are asserted to the constraint store.

In a sense, the built-in constraint solver from Prolog provides an effective means for bounded software model checking by explicitly exploring all program executions, but symbolically reasoning about the constraints imposed to free variables.

7.2 Constraint Generation

Translating the constraints specified in the Appendix to Prolog rules is straightforward. One caveat, however, is that most Prolog systems do not directly support quantifiers. While existential quantification can be realized via Prolog's backtracking mechanism, we need to implement universal quantification by enumerating the related finite domain. For instance, constraint `requireWeakTotalOrder` is originally specified as follows:

requireWeakTotalOrder *ops order* $\equiv \forall\, i, j \in$ *ops*.
  (**fb** i $\wedge$ **fb** j $\wedge$ **id** $i \neq$ **id** j) $\Rightarrow$ (**order** i j $\vee$ **order** j i)

In the Prolog code, predicate `forEachElem` is recursively defined to call the corresponding `elemProg` for every element in the adjacency matrix *Order* (variable names start with a capital letter in Prolog).

```
requireWeakTotalOrder(Ops,Order,FbList):-
    forEachElem(Ops,Order,FbList,doWeakTotalOrder).

elemProg(doWeakTotalOrder,Ops,Order,FbList,I,J):-
    const(feasible,Feasible),
    length(Ops,N),
    matrix_elem(Order,N,I,J,Oij),
    matrix_elem(Order,N,J,I,Oji),
    nth(I,FbList,Fi),
    nth(J,FbList,Fj),
    (Fi #= Feasible #/\ Fj #= Feasible #/\ I #\= J)
        #=> (Oij #\/ Oji).
```

Barring some implementation details, one technique shown by the above example is worth noting. That is, the adjacency matrix *Order* and the feasibility list *FbList* are passed in as finite domain variables. The domain of the elements in these lists (which is *boolean* in this case) is previously setup in the top level predicate. Providing such domain information significantly reduces the solving time, hence is critical for the performance of the tool.

The searching order among the constraints may also impact performance. In general, it is advantageous to let the

```
Tinit       Thread 1            Thread 2

(1)wr(x,0); (3)rd(x,r1,1);      (6)rd(y,r2,0);
(2)wr(y,0); (4)ctr(c1,[r1>0]); (7)ctr(c2,[r2>=0]);
            (5)wr(y,1,[c1]);    (8)wr(x,1,[c2]);
```

Figure 4: The execution derived from program 2 in Figure 1 with $r1 = 1$ and $r2 = 0$.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	0	X	1	1	1	1	1	1
2	X	0	1	1	1	1	1	1
3	0	0	0	1	1	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	1	1	1	0	1	1
7	0	0	1	1	1	0	0	1
8	0	0	1	1	1	0	0	0

Figure 5: The adjacency matrix for the execution shown in Figure 4 under sequential consistency.

solver satisfy the most restrictive goal first. For example, read value rules should precede the general ordering rules.

7.3 Concurrency Analysis

DefectFinder is developed in a modular fashion and is highly configurable. It supports all three applications described in this paper. It also enables *interactive* and *incremental* analyses, meaning it allows users to selectively enable or disable certain constraints to help them understand the underlying model piece by piece.

To illustrate how the tool works, recall program 2 in Figure 1. Consider the problem of checking whether $r1 = 1$ and $r2 = 0$ is allowed by sequential consistency. Figure 4 displays the corresponding execution derived from the program text (it only shows the operation fields relevant to this example). When constraint `validateExecution` is imposed, DefectFinder immediately concludes that the execution is legal, with an output displaying the adjacency matrix shown in Figure 5. A matrix element $\mathcal{M}_{ij}$ can have a value of 0, 1, or X, where 0 indicates i is not ordered before j, 1 indicates i must precede j, and X means the ordering relation between i and j has not been instantiated based on the accumulated constraints. In general, there usually exists many X entries where alternative interleavings are allowed. If desired, a Prolog predicate *labeling* can be called to instantiate all variables. Our tool also outputs a possible interleaving *1 2 6 7 8 3 4 5* which is automatically derived from this matrix.

If the execution with $r1 = 0$ and $r2 = 1$ is checked, the tool would quickly determine that it is illegal. The user can also ask "what if" queries by selectively commenting out some ordering rules to identify the root cause of a certain program behavior.

Applying DefectFinder for a different application simply involves selecting the corresponding goal. For example, if the programs in Figure 1 are checked for race conditions, the tool would report that program 1 is race-free and program 2 is not, in which case the conflicting operations and an interleaving that leads to the race conditions are displayed.

Similarly, when the program in Figure 2 is verified for race conditions, our utility would report that it is race-free. However, an atomicity violation would be detected if the transaction is annotated by an atomic block. Having detected this defect, the user can subsequently modify the code and do the test again. For instance, if a transaction is protected by a single Lock/Unlock pair and both transactions use the same lock, the bug would be removed.

7.4 Performance

Precise semantic analysis such as race detection is NP-hard in general [15]. Nonetheless, constraint-based methods have become very successful in practice, thanks to the efficient solving techniques developed in recent years.

Our tool has been applied to analyze a large collection of litmus tests — each of them is designed to reveal a certain memory model property or to simulate a common programming pattern. Running on a Pentium 366 MHz PC with 128 MB of RAM using Windows 2000, most of the tests complete within a few seconds.

There is still a lot of room for improvement in performance, which offers an interesting direction for our future work. For instance, we can add a "constraint configuration" component that automatically filters out or reoders certain rules according to the input program, e.g., rules regarding control flow can be excluded if the program does not involve branch statements. Other solving techniques may also help make our approach more effective. We have shown in [7] that a slight variant of the Prolog code can let us benefit from a propositional SAT solver. We are also working on the integration of a *quantified boolean formulae* (QBF) solver that can directly accept quantifiers.

8. RELATED WORK

Constraint solving was historically applied in AI planning problems. In recent years, it has started to show a lot of potential for program analysis as well. For example, constraints are used in [16] to analyze programs written in a factory control language called Relay Ladder Logic. A constraint system is developed in [17] for inferring static types for Java bytecode. The work in [18] performs points-to analysis for Java by employing annotated inclusion constraints. CLP is proposed for performing bounded software model checking in [19]. Our experience indicates that a constraint-based approach is also viable for reasoning about correctness properties in multithreaded programs.

Extensive research has been done in model checking Java programs, e.g., [20, 21, 22, 23, 24]. These tools, however, do not address memory model issues. Therefore, they cannot analyze fine-grained thread interleavings. We can imagine our method being incorporated into these tools to make their analyses more accurate.

There is a large body of work on race analysis. These techniques can be classified as static or dynamic. The latter can be further categorized as on-the-fly or post-mortem, depending on how the execution information is collected. Each strategy typically involves a tradeoff among efficiency, accuracy, and coverage.

Netzer and Miller [25] [26] have categorized data races and proposed an algorithm using the post-mortem method. Adve and Hill [27] have proposed the *data-race-free* model and unified four weakly ordered memory systems according to a common property — on these systems, programs would appear sequentially consistent in the absence of data races. They further propose a formal definition of data races under

weak memory models [11]. Lamport's happened-before relation has been applied in dynamic analysis tools, e.g., [28] [15] [29]. Several on-the-fly methods have been proposed that collect information based on the underlying cache coherence protocol, e.g., [30] [31] [32]. The drawback of these dynamic techniques is that they can easily miss a data race, depending on how threads are scheduled.

Our approach is based on the definition given in [11]. It also employs the happened-before relation, which enables our system to handle many different synchronization styles. Unlike the dynamic approaches, we propose a static method that examines a symbolic execution and achieves an exhaustive coverage.

Several race detectors have been designed specifically for the lock-based synchronization model, e.g. [33] [34] [35]. Tools such as ESC/Java [8] and Warlock [36] rely on user-supplied annotations to statically detect potential data races. Type-based approaches have also been proposed for object-oriented programs, e.g., [37] [38] [39].

While effective in practice, these tools do not address the issue that we focus on, which is how to rigorously reason about multithreaded programs running in a complex shared memory environment. Our framework makes these analyses possible and can be particulary useful in helping people understand the underlying concurrency model and conduct verification for common programming patterns. The capability of studying program correctness under relaxed memory models is also essential in verifying critical programs such as JVMs and garbage collectors that run on weak memory systems.

Flanagan and Qadeer have developed a type system to enforce atomicity with an annotation keyword *atomic* [5]. It assumes that a race analysis is performed in advance. Therefore, the effectiveness of their approach depends on the accuracy of the race detector. Their work is based on Lipton's theory of right and left movers [40]. It will be interesting to investigate if the requirements of movers can be captured as constraints for type inference.

9. CONCLUSION

We have presented a novel approach that handles both program semantics and memory model semantics in a declarative constraint-based framework. With three concrete applications — execution validation, race detection, and atomicity verification — we have demonstrated the feasibility and effectiveness of applying such a "memory-model-aware" analysis tool for verifying multithreaded programs that, albeit small, can be extremely difficult to analyze by hand.

To summarize, our system offers the following benefits:

- It is rigorous. Based on formal definitions of program properties and memory model rules, our system enables a precise semantic analysis. Specifications developed in such a rigorous manner can also be sent to a theorem proving utility, such as the HOL theorem prover [41], for proving generic properties.

- It is automatic. Our approach allows one to take advantage of the tremendous advances in constraint/SAT solving techniques. The executable thread semantics can also be treated as a "black box" whereby the users are not necessarily required to understand all the details of the model to benefit from the tool.

- It is generic. Since our method is not limited to a specific synchronization mechanism, it can be applied to reason about various correctness properties for any threading model, all using the same framework.

In terms of scalability, there are two aspects involved. One is the complexity of the shared memory system that can be modelled. The other is the size of programs that can be analyzed. For the former aspect, our system scales well with its compositional specification style. As demonstrated in [7], it is capable of formalizing memory ordering rules for modern commercial processors. As for improving the latter aspect, divide-and-conquer style verification methods will be investigated. Techniques developed in other tools, such as predicate abstraction, branch refinement, and assume-guarantee, can be integrated into our system. We also plan to explore more efficient solving techniques. In particular, the structural information of the constraints may be applied for improving the solving algorithms. We hope this paper can help pave the way towards future studies in these exciting areas.

10. ACKNOWLEDGMENTS

We are grateful to all contributors to the Java memory model discussion group who have brought up many interesting issues. We especially thank Bill Pugh, Sarita Adve, and Victor Luchangco for their insightful comments on race conditions.

11. REFERENCES

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

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1–11, 2000.

[4] JSR133: Java memory model and thread specification. http://www.cs.umd.edu/~pugh/java/memoryModel.

[5] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI*, 2003.

[6] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. Technical Report UUCS-03-019, University of Utah, 2003.

[7] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03), LNCS 2860*, October 2003.

[8] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, 2002.

[9] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Analyzing the CRF Java Memory Model.

In *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, 2001.

[10] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. UMM: An operational memory model specification framework with integrated model checking capability. *Concurrency and Computation: Practice and Experience*, to appear.

[11] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 234–243, 1991.

[12] L. Lamport. Time, clocks and ordering of events in distributed systems. 21(7):558–565, July 1978.

[13] SICStus Prolog. http://www.sics.se/sicstus.

[14] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.

[15] Robert H. B. Netzer. Race condition detection for debugging shared-memory parallel programs. Technical Report CS-TR-1991-1039, 1991.

[16] Alexander Aiken, Manuel Fähndrich, and Zhendong Su. Detecting races in relay ladder logic programs. *Lecture Notes in Computer Science*, 1384:184–200, 1998.

[17] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[18] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of Object-Oriented Programming Systems, Lanuages, and Applications*, pages 43–55, 2001.

[19] Cormac Flanagan. Automatic software model checking using CLP. In *Proceedings of ESOP*, pages 189–203, 2003.

[20] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[21] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java Model Checker. In *Post-CAV Workshop on Advances in Verification, Chicago*, 2000.

[22] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.

[23] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[24] D. Park, U. Stern, and D. Dill. Java model checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, 2000.

[25] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, 26(7):133–144, 1991.

[26] R. H. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[27] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[28] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, 1991.

[29] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, 1996.

[30] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 235–244, 1991.

[31] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1998.

[32] Edmond Schonberg. On-the-fly detection of access anomalies. In *Proceedings of PLDI*, pages 285–297, 1989.

[33] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[34] C. von Praun and T. Gross. Object-race detection. In *Proceedings of Object-Oriented Programming Systems, Lanuages, and Applications*, pages 70–82, 2001.

[35] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of PLDI*, 2002.

[36] N. Sterling. Warlock - a static data race analysis tool. *USENIX Winter Technical Conference*, pages 97–106, 1993.

[37] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *Proceedings of PLDI*, pages 219–232, 2000.

[38] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *Proceedings of Object-Oriented Programming Systems, Lanuages, and Applications*, 2000.

[39] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[40] R. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[41] T. F. Melham M. J. C. Gordon. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

APPENDIX

A. SEQUENTIAL CONSISTENCY

legalSC *ops order* $\equiv$
  requireProgramOrder *ops order* $\wedge$
  requireReadValue *ops order* $\wedge$
  requireComputation *ops order* $\wedge$
  requireMutualExclusion *ops order* $\wedge$
  requireWeakTotalOrder *ops order* $\wedge$
  requireTransitiveOrder *ops order* $\wedge$
  requireAsymmetricOrder *ops order*

A.1 Program Order Rule

requireProgramOrder *ops order* $\equiv \forall\, i, j \in ops.$
  (**fb** i $\wedge$ **fb** j $\wedge$ (**orderedByProgram** $i\ j \vee$
  t $i = t_{init}$ $\wedge$ **t** $j \neq t_{init}$)) $\Rightarrow$ **order** $i\ j$

A.2 Read Value Rules

requireReadValue *ops order* $\equiv$
  globalReadValue *ops order* $\wedge$
  localReadValue *ops order* $\wedge$
  controlReadValue *ops order*

globalReadValue *ops order* $\equiv \forall\, k \in ops.$
  (**fb** k $\wedge$ **isRead** k) $\Rightarrow$
  ($\exists\, i \in ops.$ **fb** i $\wedge$ **op** $i = Write$ $\wedge$ **var** $i =$ **var** $k \wedge$
  data $i =$ **data** k $\wedge$ $\neg$(**order** $k\ i$) $\wedge$
  ($\neg\exists\, j \in ops.$ **fb** j $\wedge$ **op** $j = Write$ $\wedge$ **var** $j =$ **var** $k \wedge$
  order $i\ j$ $\wedge$ **order** $j\ k$))

localReadValue *ops order* $\equiv \forall\, k \in ops.$ **fb** $k \Rightarrow$
  ($\forall\, e \in$ (**getLocals** k).
  ($\exists\, i \in ops.$ (**fb** $i \wedge$ **isAssign** $i \wedge$ **local** $i =$ **getVar** $e \wedge$
  data $i =$ **getData** e $\wedge$ **orderedByProgram** $i\ k$) $\wedge$
  ($\neg\exists\, j \in ops.$ (**fb** $j \wedge$ **isAssign** $j \wedge$ **local** $j =$ **getVar** $e \wedge$
  orderedByProgram $i\ j \wedge$ **orderedByProgram** $j\ k$))))

controlReadValue *ops order* $\equiv \forall\, k \in ops.$
  ($\forall\, e \in$ (**getCtrs** k).
  ($\exists\, i \in ops.$ **op** $i = Control \wedge$ **var** $i =$ **getVar** $e \wedge$
  data $i =$ **getData** e $\wedge$ **orderedByProgram** $i\ k$))

A.3 Computation Rule

requireComputation *ops order* $\equiv \forall\, k \in ops.$
  ((**fb** k $\wedge$ **op** $k = Computation$) $\Rightarrow$
  (**data** $k =$ **eval** (**cmpExpr** k))) $\wedge$
  ((**fb** k $\wedge$ **op** $k = Control$) $\Rightarrow$
  (**data** $k =$ **eval** (**ctrExpr** k)))

A.4 Mutual Exclusion Rule

requireMutualExclusion *ops order* $\equiv \forall\, i, j \in ops.$
  (**fb** i $\wedge$ **fb** j $\wedge$ **matchLock** $i\ j$) $\Rightarrow$
  ($\neg\exists\, k \in ops.$ **fb** k $\wedge$ **isSync** $k \wedge$
  lock $k =$ **lock** i $\wedge$ **t** $k \neq$ **t** i $\wedge$ **order** $i\ k$ $\wedge$ **order** $k\ j$)

A.5 General Ordering Rules

requireWeakTotalOrder *ops order* $\equiv \forall\, i, j \in ops.$
  (**fb** $i \wedge$ **fb** $j \wedge$ **id** $i \neq$ **id** j) $\Rightarrow$ (**order** $i\ j \vee$ **order** $j\ i$)

requireTransitiveOrder *ops order* $\equiv \forall\, i, j, k \in ops.$
  (**fb** $i \wedge$ **fb** $j \wedge$ **fb** $k \wedge$ **order** $i\ j \wedge$ **order** $j\ k$) $\Rightarrow$ **order** $i\ k$

requireAsymmetricOrder *ops order* $\equiv \forall\, i, j \in ops.$
  (**fb** i $\wedge$ **fb** j $\wedge$ **order** $i\ j$) $\Rightarrow \neg$(**order** $j\ i$)

A.6 Auxiliary Definitions

fb $i \equiv$ (**eval** (**ctrExpr** i) $= True$)

orderedByProgram $i\ j \equiv$ (**t** $i =$ **t** j $\wedge$ **pc** $i <$ **pc** j)

isAssign $i \equiv$ (**op** $i = Computation \vee$ **op** $i = Read$)

isSync $i \equiv$ (**op** $i = Lock \vee$ **op** $i = Unlock$)

matchLock $i\ j$ $\equiv$
  op $i = Lock$ $\wedge$ **op** $j = Unlock$ $\wedge$ **matchID** $j =$ **id** i

Note: for brevity, the following predicates are not explicitly defined here since they are typically well understood.

eval *exp*:	evaluate *exp* with standard program semantics;
getLocals k:	parse k and get the set of local variables that k depends on, with their associated data values;
getCtrs k:	parse the path predicate of k and get the set of control variables that k depends on, with their associated data values;
getVar e:	get variable from a ⟨variable, data⟩ entry;
getData e:	get data from a ⟨variable, data⟩ entry.

B. EXECUTION VALIDATION

validateExecution *ops* $\equiv \exists$ *order.* **legalSC** *ops order*

C. RACE DETECTION

detectDataRace *ops* $\equiv \exists$ *scOrder, hbOrder.*
  legalSC *ops scOrder* $\wedge$
  requireHbOrder *ops hbOrder scOrder* $\wedge$
  mapConstraints *ops hbOrder scOrder* $\wedge$
  existDataRace *ops hbOrder*

requireHbOrder *ops hbOrder scOrder* $\equiv$
  requireProgramOrder *ops hbOrder* $\wedge$
  requireSyncOrder *ops hbOrder scOrder* $\wedge$
  requireTransitiveOrder *ops hbOrder*

requireSyncOrder *ops hbOrder scOrder* $\equiv \forall\, i\ j \in ops.$
  (**fb** i $\wedge$ **fb** j $\wedge$ **isSync** i $\wedge$ **isSync** j $\wedge$ **scOrder** $i\ j$)
  $\Rightarrow$ **hbOrder** $i\ j$

mapConstraints *ops hbOrder scOrder* $\equiv \forall\, i\ j \in ops.$
  (**fb** i $\wedge$ **fb** j $\wedge$ **hbOrder** $i\ j$) $\Rightarrow$ **scOrder** $i\ j$

existDataRace *ops hbOrder* $\equiv \exists\, i, j \in$ *ops.*
  fb $i \wedge$ **fb** j $\wedge$ **t** $i \neq$ **t** j $\wedge$ **var** $i =$ **var** $j \wedge$
  (**op** $i = Write$ $\wedge$ **op** $j = Write \vee$
  op $i = Write$ $\wedge$ **op** $j = Read \vee$
  op $i = Read$ $\wedge$ **op** $j = Write$) $\wedge$
  $\neg$(**hbOrder** $i\ j$) $\wedge$ $\neg$(**hbOrder** $j\ i$)

D. ATOMICITY VERIFICATION

verifyAtomicity *ops* $\equiv \exists$ *order.*
  legalSC *ops order* $\wedge$
  existAtomicityViolation *ops order*

existAtomicityViolation *ops order* $\equiv \exists\, i, j, k \in$ *ops.*
  (**fb** i $\wedge$ **fb** $j \wedge$ **fb** $k \wedge$
  abEnter i $\wedge$ **abExit** $j \wedge$
  id i $=$ **matchAtID** j $\wedge$ **id** $i \neq$ **id** $j \wedge$
  isViolation $k \wedge$
  $\neg$(**order** $k\ i$) $\wedge$ $\neg$(**order** $j\ k$))

isViolation k $\equiv$ (**t** $k \neq$ **t** i)