Satellite Workshop at CONCUR 2007

Verification and Analysis of Multi-threaded Java-like Programs

VAMP'07

Lisbon, Portugal

September 3, 2007

Contents

Preface

In the past seven years or so, a considerable number of analysis tools for Java and C# have emerged with the aim of putting formal techniques to work for popular, modern programming languages. Such tools include software model checkers (e.g., the Java Path Finder), generic extended static checking tools based on automatic theorem proving (e.g., ESC/Java and Spec#), program verification tools (e.g., the KeY tool), specialized analysis tools to uncover particular flaws (e.g., race condition checkers and type systems for race freeness) and runtime assertion checkers.

While some Java analysis tools such as model checkers and race condition checkers specifically address multithreaded programs, others, particularly program verification tools, are more adept at dealing with sequential programs where additional difficulties must be tackled to handle concurrency. These difficulties include explosion of the verification complexity (both the number of program states and the size and complexity of verification conditions) and the meaning of method contracts in a concurrent setting. Some particularly difficult challenges arise because the Java and C# memory models permit certain sequentially inconsistent executions in order to prevent unrealistic constraints on language implementations.

The goal of VAMP is to provide a forum for researchers interested in verification and analysis of multi-threaded Java-like programs. It aims to bring together practically minded tool builders and theoretically minded concurrency researchers who are interested in verification and analysis techniques for Java-like languages. Topics of interest include program logics, automatic verification and static analysis techniques, type-based verification, software model checking, specification techniques, formal semantics, formalizations of the Java memory model, race condition and deadlock detection, and static analysis for bug discovery.

The workshop programme consists of five contributed papers and two invited talks. The contributed papers were reviewed and selected by the programme committee consisting of:

- Stephen Freund, Williams College, USA
- Christian Haack, Radboud Universiteit Nijmegen, The Netherlands
- Marieke Huisman, INRIA Sophia Antipolis, France
- Bart Jacobs, Katholieke Universiteit Leuven, Belgium
- Joe Kiniry, University College Dublin, Ireland
- Alexander Knapp, Ludwig-Maximilians-Universität München, Germany
- Erik Poll, Radboud Universiteit Nijmegen, The Netherlands
- Robby, Kansas State University, USA

We thank Radu Grigore and Edwin Rodríguez for helping to review the papers.

VAMP has been organized as satellite event to CONCUR and we warmly thank the CONCUR organizers for this opportunity and for taking care of the entire local organization (including the printing of the workshop proceedings). This made it possible to keep our organizational workload extremely low. Great thanks to the CONCUR workshop organizers Francisco Martins and António Ravara!

Nijmegen, August 8, 2007

Christian Haack
Marieke Huisman
Joe Kiniry
Erik Poll

VAMP 2007

Separation Logic, Concurrency and Java

Matthew Parkinson

University of Cambridge, United Kingdom

Subsystems: Reconciling Locks and Unchecked Exceptions

Bart Jacobs

Katholieke Universiteit Leuven, Belgium

Abstract

In Java-like languages, if an exception occurs in a thread while the thread holds an object's lock, by default the thread releases the lock and other threads that acquire the lock may see the object in an inconsistent state. Clearly, this default behavior is unsafe. We propose a language extension, called subsystems, that makes it easy for programmers to ensure that once a lock is released due to an unchecked exception, no other thread may successfully acquire the lock. This is achieved by propagating the exception to other threads as appropriate. The language extension is intended to simplify writing verifiably safe multithreaded object-oriented programs.

A Dynamic Logic for Deductive Verification of Concurrent Java Programs With Condition Variables

Bernhard Beckert and Vladimir Klebanov

Abstract

In this paper, we present an approach aiming at full functional deductive verification of concurrent Java programs, based on symbolic execution. We define a Dynamic Logic and a deductive verification calculus for a restricted fragment of Java with native concurrency primitives. Even though we cannot yet deal with non-atomic loops, employing the technique of symmetry reduction allows us to verify unbounded systems. The calculus has been implemented within the KeY system. In contrast to previous work, the version presented here includes the rules for handling condition variables.

1 Introduction

1.1 Motivation and Goals

In this paper, we present a Dynamic Logic and a deductive verification calculus for a fragment of the Java language, which includes concurrency. Our aim has been to design a logic that (1) reflects the properties of Java concurrency in an intuitive manner (2) has a sound and (relatively) complete calculus (3) requires no intrinsic abstraction, no bounds on the state space or thread number (4) allows reasoning about properties of the scheduler within the logic, but does not require such reasoning for program verification.

To achieve our goal, we currently have to make three important restrictions. (1) We do not consider thread identities in programs, (2) we do not handle dynamic thread creation (but systems with an unbounded number of threads), (3) we require that all loops are executed atomically. These restrictions allow us to employ very efficient symmetry reductions and thus symbolically execute programs in the presence of unbounded concurrency. We will discuss their significance in the next section.

Our calculus has been implemented in the KeY system [2,3], which has been successfully used for verification of non-concurrent Java programs. We benefit from the KeY system's 100% Java Card coverage, which includes full support for dynamic object creation (including static initialization), efficient aliasing treatment, full handling of exceptions and method calls, Java-faithful arithmetics, etc.

This paper extends [4] with rules to verify programs with condition variables. Conversely, the former paper includes an additional invariant rule, and a description of the application of our method to verify a piece of code from the Java standard library.

1.2 Achieved Java Coverage

On the sequential side, we benefit from the KeY system's 100% Java Card coverage, which includes full support for dynamic object creation (with static initialization), efficient aliasing treatment, full handling of exceptions and method calls, Java-faithful arithmetics, etc. All of these features can be used in concurrent programs. On the concurrent side, we have to restrict the program fragment as stated. Also, like all Java verification systems known to us, we assume an intuitive, sequentially consistent memory model, where updates to shared state are immediately visible to all threads. In reality, the Java Memory Model provides much weaker guarantees. We believe that our calculus could be extended to reflect these. Apart from this, our calculus faithfully models Java's concurrency.

One concurrency limitation concerns the use of explicit thread identities in programs. These are usually manifested by invocations of methods from the class `Thread`, the most important being `t.interrupt()` and `t.join()`. Since our calculus is strongly based on symmetry reduction such programs are not allowed. We believe, though, that this limitation precludes us from verifying only a small fraction of interesting code. In particular, it does not forbid the use of synchronized blocks or condition variables with `wait()/notify()`.

The only thread creation mechanism we currently provide is the possibility for the programmer to specify the initial thread configuration of a program (together with the initial local variable assignment). Note that the configuration values can be symbolic ("k threads"). While this limitation is indeed unfortunate, it does not impair the usefulness of the calculus much. It is in the nature of concurrent Java applications that most objects are passive entities. They are unaware of thread creation and can (and indeed have to) be verified for an arbitrary number of threads accessing them. The most prominent expression of this fact is library code, which has to be thread-safe for any number of client threads.

Finally, we require all loops to be atomic. The programmer has to ensure that no (significant) interleavings occur while the loop runs. This property can be checked by our method as described later on. We are working on overcoming this limitation by developing a more elaborated algebraic model of the scheduler.

1.3 Related Work

Several deductive calculi for (different fragments of) sequential Java exist, while not much work has been done to extend these calculi to cover concurrency. A notable exception is the Verger tool [1], a deductive verification system based on Hoare Logic. The system requires the programs to be augmented with auxiliary variables and annotated with Hoare-style assertions. From these, verification conditions are generated, which have to be discharged in PVS. The system has a good concurrent language coverage, including dynamic thread creation. It does, however, not serve our goal of focusing on symbolic execution of concurrent programs.

A huge body of work is available on verifying temporal properties of concurrent software. This includes model checkers and even deductive proof systems (e.g., by Manna and Pnueli [10]). In contrast to using temporal logic though, a proof system for dynamic logic allows functional verification, i.e., full reasoning about data. This way verification tasks can be tackled where not only safety or liveness but the input-output relation of a concurrent program is of interest.

The only dynamic logic for a programming language incorporating concurrency is—to our knowledge—the Concurrent Dynamic Logic (CDL) described by David Peleg in [12]. He notes, however, that this particular logic "suffers from the absence of any communication mechanisms; processes of CDL are totally independent and mutually ignorant". In [11], Peleg gives two extensions of CDL with interprocess communication: one with channels and one with shared variables. In both works cited, the focus is on studying concerns of expressivity and decidability of the logics (communication renders the logic highly undecidable, in short). The issue of a calculus or program verification in general is not touched.

A comprehensive control flow model of Java concurrency is given in [5]. The authors use a variant of Petri nets to model the concurrent "skeletons" of programs with an extension to treat the "partially non-blocking rendez-vous" nature of Java's `wait()/notify()` mechanism. As far as the basic representation formalism is concerned, this is closely related to our work, although we use full programs. The cited work describes a model checker, which verifies program models for safety properties expressed in terms of control flow. The framework does not cover functional verification.

Another class of verification tools for concurrent programs are static verifiers. A prominent example is the SPEC# system, which incorporates a static verifier for a concurrent object-oriented language [8]. Static verifiers are very good at detecting race conditions but are not geared towards input-output reasoning.

It is known that the efficiency of a verification system is bounded to a great degree by the compositionality of reasoning it offers. This aspect is currently not the target of our work though. Suggestions for modularizing reasoning about concurrent Java programs have been made in [6,14]. This research indicates that programmers use dedicated "serializability techniques" (mostly locking protocols and reference confinement) to ensure correctness of programs. We believe that the proposed specifications developed for model checking resp. static analysis can be put to efficient use in a deductive framework. We have already shown how certain serializability properties can be verified deductively in [9].

2 A Logic for Concurrent Java

The logic we present in this paper is an instance of Dynamic Logic (DL) [7], and the proof system is a sequent-style symbolic execution calculus, which ensures good understandability.

DL can be seen as a modal logic with a modality $\langle p \rangle$ for every program p, which refers to the successor states that are reachable by running p. The formula

$\langle p\rangle\phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\psi \rightarrow \langle p\rangle\phi$ is valid if for every state s satisfying pre-condition ψ a run of the program p starting in s terminates, and in the terminating state the post-condition ϕ holds. In standard DL there can be several such states because the programs can be non-deterministic; we have equipped our programs with a deterministic semantics via an underspecified scheduler function. This allows much stronger control over granularity of reasoning.

Concurrent Programs
The programs we consider are Java programs with the inherent restrictions posed in the introduction.

Several threads can execute a program concurrently. Thus, a program is a passive template "without life" unless a thread configuration is added, i.e., a description of which threads are executing the program. Threads are given a number, conventionally called *thread id* (tid); they are in fact identified with this number.

We present the theoretical foundations for programs with a single code template or thread class. The straightforward extension to several thread classes will be given with the example later.

Positions
We number all state-changing statements in a program (i.e., assignments; later also locking primitives and native method calls) from left to right, starting with one. We call these numbers the *positions* of the program. Their intuitive meaning is that if a thread is at a certain position, it is about to execute the corresponding statement when it is next scheduled to run. In addition, we consider the end of a program to be a position, which is reached when a thread has completed the execution of the program.

Configurations
A thread *configuration* specifies the threads waiting to execute at every position of a given program. A configuration (of size n) is an n-tuple of pairwise disjoint sets of tids. For example, $(\{3, 17, 5\}, \{\}, \{2\})$ is a configuration. A configuration of size n is compatible with programs that have n positions, i.e., that have $n - 1$ statements.

We write (compatible) pairs $c|p$ of thread configurations and programs by inlining the components of the configuration within the program. For example, the program

```
v=(x<10); if (v) {a=10; x=a+1}
```

together with the configuration $(\{5\}, \{3, 4\}, \{1\}, \{2\})$, where four threads are active and one has already terminated, is written as

$$^{\{5\}}\texttt{v=(x<10); if (v) \{}^{\{3,4\}}\texttt{a=x;}^{\{1\}}\texttt{x=a+1;\}}^{\{2\}}$$

A position *pos* is *enabled* in a configuration c iff its tid set is not empty and it is not the last position, which is reserved for threads that have run to completion. We define $\mathit{enabled}(c, \mathit{pos}) \equiv (c(\mathit{pos}) \neq \emptyset) \wedge (\mathit{pos} < \mathit{size}(c))$, where $\mathit{size}(c)$ is the length of the configuration tuple.

The Scheduler

The scheduler is (modeled by) the rigid function *sched*. That is, different models may interpret this function differently and, thus, have different schedulers. But within a model the scheduler is rigid, i.e., it does not depend on the program state. Intuitively, we assume the scheduling to be data-independent; it is not affected by the current values of variables and object attributes.

To model the fact that a scheduler may not always run the same thread for a given thread configuration, we make it dependent on a *seed*: $sched(r, c)$ is the id of the thread scheduled to run next in configuration c given the seed r. If no position is enabled in c, $sched(r, c) = 0$. Fairness or other scheduler properties are not built into our model. Our scheduler may select an arbitrary thread id provided it occurs in the configuration c and is not already at the last position. Properties such as fairness can, however, be specified by adding axioms restricting the function *sched*. It should be noted that Java itself is only “statistically fair”.

Signatures and Variables

The formulas of our logic are built over a set V of logical (quantifiable) variables and a signature Σ of function and predicate symbols. Function symbols are either *rigid* or *non-rigid*. Rigid function symbols have a fixed interpretation for all states (e.g., addition on integers). In contrast, the interpretation of non-rigid function symbols may differ from state to state.

Logical variables are rigid in the sense that if a logical variable has a value, it is the same for all states. They cannot be assigned to in programs. Everything that is subject to assignment during program execution (variables, object attributes, arrays) is modeled by non-rigid functions. We will call these functions *program variables*. In particular, arrays and object attributes give rise to functions with arity $n > 0$.

We now further sub-divide the bulk of program variables into local and shared. Every thread has its own copy of each local variable (allocated on the thread's stack), so that assignments to these are not visible in other threads. To distinguish local variables in different threads, we use combinations of variable name and thread id within the logic. Formally: we give non-rigid functions used to model thread-local variables another argument, which is the thread id. For example, $l(k)$ denotes the copy of variable l used by the thread with id k. This distinction, though, is unavailable *within programs*, as one thread is unaware of other threads' copy of the same local variable. As a consequence, every thread-local variable (which is, again, a non-rigid function) of arity n appears with $n - 1$ arguments in the concurrent program.

Shared state manipulation can arise when these local variables are dereferenced. Whether `o`(13)`.a` refers to the same memory location as `o`(17)`.a` depends on the values of `o` in the threads 13 and 17. This is a standard aliasing question, which is resolved just like in the sequential KeY calculus. On the other hand, our logic also has explicit *shared variables*, which are used to model static fields. Shared variables exist only once and assignments changing their value are immediately visible to all threads.

Formulas

The set of formulas is defined as common in first-order dynamic logic. That is, they are built using the connectives $\wedge, \vee, \rightarrow, \neg$ and the quantifiers $\forall, \exists$ (first-order part). If p is a program, c is a configuration, r is a scheduling seed, and ϕ a formula, then $\langle r|c|p\rangle\phi$ (the "diamond" modality) and $[r|c|p]\phi$ (the "box" modality, which is a shorthand for $\neg\langle r|c|p\rangle\neg\phi$) are formulas. In the examples, we omit the scheduling seed r where it is not relevant.

Intuitively, a diamond formula $\langle r|c|p\rangle\phi$ means that all threads from the configuration c for a program p and random seed r must terminate normally (run to completion) and afterwards ϕ has to hold. The meaning of a box formula is the same, but termination is not required, i.e., ϕ must only hold *if* the program terminates.

Furthermore, $\{\mathit{lhs}{:=}\mathit{rhs}\}\phi$ is a formula. The expression $\{\mathit{lhs}{:=}\mathit{rhs}\}$ is called a state update. Note that, unlike assignments, state updates can refer to the local copies of local variables. They cannot be used within programs and, as opposed to programs, their evaluation does not require a thread configuration or a scheduling seed. State updates (together with an update simplification calculus, which is a standard part of KeY) are used to handle assignments, resolve aliasing, and also relate logical and program variables.

Semantics of Terms, Programs, and Formulas

The semantic domains used to interpret DL formulas are Kripke structures $\mathcal{K} = (S, \rho)$, where S is the set of program states and ρ is the transition relation interpreting programs with a given thread configuration and a given scheduling seed. Since we use deterministic programs and the scheduling is deterministic for a given configuration and a given seed, ρ is a (partial) function, i.e., for every program p, configuration c, and seed r, $\rho(r, c, p) : S \rightarrow S$.

The states $s \in S$ are first-order structures for the signature Σ, providing interpretations of non-rigid functions (which include program variables). In fact, we assume that the set S of states of any Kripke structure consists of *all* first-order structures with signature Σ over some universe and for some interpretation of the rigid symbols. Rigid function symbols have a fixed interpretation for all states, while the interpretation of non-rigid function symbols may differ from state to state. We also work under the constant domain assumption, i.e., for any two states $s_1, s_2 \in S$ the universes of s_1 and s_2 are the same set U. We refer to U as *the* universe of $\mathcal{K}$.

Since the transition relation ρ (by definition) corresponds to the fixed semantics of our programming language, the only things that can change from one model (Kripke structure) to the other are: the signature, the universe, and the interpretation of the rigid symbols (including that of the scheduler function *sched*).

The valuation $val_{s,\beta}$ of terms w.r.t. a given state s and a given logical variable assignment β is as usual in first-order logic. The semantics $\rho_\beta(r, c, p)$ of a program p reflects the behavior of the corresponding Java program. Algebraically it is a relation between initial and final states, which is parameterized by a scheduling seed r and a thread configuration c. The semantics of modal formulas is as usual for first-order modal logic, i.e., $val_{s,\beta}(\langle r, c, p\rangle\phi) = \mathit{true}$ iff $(s, s') \in \rho(r, c, p)$ for some state s' with $val_{s',\beta}(\phi) = \mathit{true}$. For formulas with updates, $val_{s,\beta}(\{\mathit{lhs}{:=}\mathit{rhs}\}\phi) = \mathit{true}$ iff

$val_{s',\beta}(\phi) = true$ for some state s', which is identical to s except that the value of *lhs* is changed to $val_{s,\beta}(rhs)$.

A Kripke structure is a *model* of a formula ϕ iff ϕ is true in all states of that structure. A formula ϕ is *valid* if all Kripke structures are a model of ϕ.

A Deductive Calculus

We employ a sequent calculus that consists of the rules for symbolically executing concurrent programs presented in the following, together with standard structural first-order rules, rules for integers and other datatypes (which include induction) and rules for update simplification. All the latter rules are inherited from the standard KeY calculus and are not shown here.

A *sequent* is of the form $\Gamma \Longrightarrow \Delta$, where Γ and Δ are sets of formulas. Its informal semantics is the same as that of the formula $\bigwedge_{\phi \in \Gamma} \phi \quad \rightarrow \quad \bigvee_{\psi \in \Delta} \psi$. As common in sequent calculus, the direction of entailment in the rules is from premisses (sequents above the bar) to the conclusion (sequent below), while reasoning in practice happens the other way round: by matching the conclusion to the goal.

From all rules presented we have omitted the usual context Γ and Δ, as well as a sequence of updates $\mathcal{U}$, which can preceed the formulas involved. The modality $\langle\!\![\cdot]\!\!\rangle$ can mean both a diamond and a box, as long as this choice is consistent within a rule.

3 Symbolic Execution of Concurrent Programs

3.1 Extending Symmetry Reduction

Symmetry reduction is a well-known idea that different threads with the same properties (which boil down to local data and program counter) need not be distinguished. Most model checking frameworks use some sort of symmetry reduction to prune the state space. This is described prominently in [13] (the Bogor tool) and [15] (on-the-fly model-checking with TVLA).

Due to their nature, these approaches only detect symmetry between threads with exactly the same concrete local data. In a deductive verification system we can give this idea a new twist. We know that proofs about a program have significantly fewer cases than the program possible inputs. In other words, even threads with different local data will exhibit the same behavior in terms of their execution path through the code. Furthermore, there is only a finite and relatively small number of different paths; this number is dictated by the shape of the program. Since we are executing programs symbolically (and have already paid a price for that in form of case distinctions), we can reap higher benefits and, as a start, identify threads with different local data as long as they follow the same path.

Furthermore, we can achieve even stronger symmetry reduction by separating thread scheduling and control flow. We obtain symmetry between threads with different paths through the program, by forcing each thread to linearly traverse the program: There is no jumping back (except within an atomic loop), and each thread visits each position exactly once. This means, however, that threads can end up in “wrong” parts of if-then-else code. To preserve the original semantics of the program, we assume that the state is not changed by the program while its control flow is in the wrong place. For this small additional price, all thread traces are now completely symmetric.

Thus, we have completely eliminated the necessity to consider different orderings of threads that have reached the same position within the program. Together with exploiting atomic and independent code, this makes deductive verification of real concurrent systems feasible.

3.2 Expressing Unbounded Concurrency

As mentioned above, we force each thread to visit each program position exactly once. Assuming threads with tids $1, \ldots, n$, it is clear that for every position *pos*, there is a permutation $p_{pos} : \{1 \ldots n\} \to \{1 \ldots n\}$, which describes the order in which the threads are scheduled at this position.

Given these permutations, it is sufficient to know *how many* threads are at each position. This fixes the exact configuration as well and allows configurations with r positions of the form $(p_1 : k_1, \ldots, p_r : k_r)$, where $p_1, \ldots, p_r$ are terms representing the permutations and $k_1, \ldots, k_r$ are terms representing the number of threads. Using this notation, the next thread scheduled at position *pos* is the $(\mathit{Post}(\mathit{pos}) + 1)$th thread, which has the tid $p_{pos}(\mathit{Post}(\mathit{pos}) + 1)$ where $\mathit{Post}(\mathit{pos})$ is the number of threads already beyond *pos* in the implied current configuration: $\mathit{Post}(\mathit{pos}) = k_{pos+1} + \cdots + k_r$.

Consider a configuration of size 4 with 2, 3, 5 and 7 threads waiting at each position respectively. With the permutation functions $p_1, \ldots, p_4$ from above, we can write this configuration as $(p_1 : 2, p_2 : 3, p_3 : 5, p_4 : 7)$. If we now concentrate on position 2, we can see that $\mathit{Post}(2) = 5 + 7 = 12$ threads have already passed this position and the next one to execute will be the 13th in count. But exactly which one? Here the permutation functions come into play. The exact tid of the thread scheduled to run next at position 2 is given by $p_2(\mathit{Post}(2) + 1) = p_2(13)$. This way we can talk concisely about thread orderings even if we don't know them exactly.

The same way we can write configurations where the number of threads is not a concrete number but a variable. This very expressive form of writing allows us to formulate rules that do not take the scheduling order into account, as it is hidden inside the permutation functions. What we need for a complete calculus are then the usual algebraic properties of permutations and axioms of their interplay.

Altogether, our calculus works by reducing assertions about programs to assertions about integers and permutations, which encapsulate the scheduler decisions. In the desirable case that the program is scheduling-independent the permutations can be removed from the correctness assertions by application of standard algebraic lemmas. Scheduling independence means that the relevant part of a program's final result is always the same, in spite of possibly different intermediate states that it can assume in different runs. Scheduling independence is an important part of program correctness. When also the remaining assertions (now without permutations) can be discharged, then the program is fully correct w.r.t. its functional specification.

3.3 Program Unfolding

The rules of our calculus that symbolically execute programs (i.e., treat state changes and concurrency; they are explained in the following section), assume a certain normal form of the program. That is, complex sequential program parts must first be completely "unfolded".

This process results in a program that is trace-equivalent to the original, but each occurring expression is now simple and each assignment atomic. The program has more of each now in exchange. A version of this transformation is already a part of the sequential KeY calculus (see [3]), and we have in fact reused the bulk of the corresponding rules.

The only constructs in the resulting unfolded programs are assignments, conditionals and loops. We will extend these to locking primitives and certain native method calls later. Everything else, including object creation, exceptions, etc., is reduced to these ingredients. Moreover, the programs get normalized such that (a) the evaluation of assignment expressions cannot have side-effects, (b) the conditions of if-statements and loops are fresh local variables. The latter property eliminates technical difficulties when specifying execution path conditions.

During the unfolding process, the KeY calculus introduces fresh local variables. For instance, we unfold `o.a=u.a++;` into `v=u.a; u.a=v+1; o.a=v;` (where `v` is a fresh local variable). The Java program `if (o.a>1) {`α`} else {`β`}` unfolds to `v=o.a>1; if (v) {`α'`} else {`β'`}`, and, a little more involved, the Java program `while (o.a>1) {`α`}` expands to `v=o.a>1; while (v) {`α' `v=o.a>1;}`.

Method calls are handled by inlining method implementations and possibly adding conditionals for simulating dynamic binding. Remember, modular verification is not the goal of our current effort.

3.4 Concurrency-Related Rules

3.4.1 Configuration Skolemization

The following rule replaces concrete thread configurations by a compact permutation-based representation, while implying no particular knowledge of the introduced permutations as they are represented by new (Skolem) constants.

$$\mathsf{conf}\ \frac{\Rightarrow \langle\!| r|c_p|p |\!\rangle \phi}{\Rightarrow \langle\!| r|c|p |\!\rangle \phi}$$

where c is a thread configuration of the form $(\{i^1_1,\ldots,i^1_{l_1}\},\ldots,\{i^r_1,\ldots,i^r_{l_r}\})$; and c_p is a configuration of the form $(p_1 : l_1,\ldots,p_r : l_r)$, where $p_1,\ldots,p_r$ are fresh unary permutation functions.

3.4.2 Position Choice

Symbolic execution starts with the choice of an enabled position in the given configuration. For this we employ the function P, which is a projection of the scheduling function. For a configuration c and a seed r, $P(r,c)$ returns the position from which the next thread will be scheduled—or 0 if no enabled positions remain. Again, $\mathit{enabled}(c,\mathit{pos}) = (c(\mathit{pos}) > 0) \wedge (\mathit{pos} < \mathit{size}(c))$.

$$\textsf{step}\ \frac{\begin{array}{rl} & \Longrightarrow P(r,c) = pos \\ path(pos,p) & \Longrightarrow \{lhs^{*(pos)}{:=}rhs^{*(pos)}\}\langle\!|r|\ \pi^{\{p_{pos}:n-1\}}\, lhs{=}rhs^{\{p_{pos+1}:k+1\}}\ \omega|\!\rangle\phi \\ \neg path(pos,p) & \Longrightarrow \langle\!|r|\ \pi^{\{p_{pos}:n-1\}}\, lhs{=}rhs^{\{p_{pos+1}:k+1\}}\ \omega|\!\rangle\phi \end{array}}{\Longrightarrow \langle\!|r|\ \pi^{\{p_{pos}:n\}}\ \underbrace{lhs = rhs}_{\text{at position } pos \text{ in } p}{}^{\{p_{pos+1}:k\}}\ \omega|\!\rangle\phi}$$

Fig. 1. The concurrent symbolic execution rule

It is a rule of the calculus that the following axioms describing properties of P may at any time be added to the left side (the antecedent) of a sequent:

- The axiom $0 \leq P(r,c) < size(c)$ effectively amounts to a disjunction over the positions of c, which during the proof gives rise to a case distinction.
- The values of P are of course restricted to the positions enabled in a given configuration: $P(r,c) \neq 0 \rightarrow enabled(c, P(r,c))$.
- P may only return 0 if no position is enabled, which is expressed by the following axiom:

$$P(r,c) = 0 \rightarrow \\ \forall pos.(1 \leq pos < size(c) \rightarrow \neg enabled(c,pos))$$

3.4.3 The Rule for Concurrent Execution

Figure 1 shows the concurrent symbolic execution rule of our calculus. In the rule, π and ω denote unchanged program parts, and pos is the position of the executed assignment $lhs{=}rhs$ in the program p. The condition $path(pos,p)$ is the path condition of this assignment (which is at position pos) in the program p. It is a conjunction of all `if`-conditions on the path from the beginning of the program to the assignment. Each `if`-condition appears as given if the path goes through the then-part, and negated if the path goes through the else-part. For example, the path condition of the statement `v=t;` in the program `if (a) {if (b) {} else {v=t;}} else {}` is $\mathtt{b} = \mathit{FALSE} \wedge \mathtt{a} = \mathit{TRUE}$.

Furthermore, $\{lhs^{*(pos)}{:=}rhs^{*(pos)}\}$ is a state update built by replacing every occurrence of a local variable v in lhs and rhs, by $v(p_{pos}(Post(pos)+1))$ using the configuration of p (cf. definition of $Post(\cdot)$ in 3.2). This way, the update represents a "sequential instantiation" of the concurrent assignment, i.e., it makes explicit which thread-copy of the variable is involved.

For example, if we consider the assignment `v=o.a;` at position 1 in some program, and the configuration before execution is $(p_1 : 2,\ p_2 : 5,\ p_3 : 7)$, then the generated update is $\{\mathtt{v}(p_1(13)){:=}\mathtt{o}(p_1(13))\mathtt{.a}\}$. The update will be tackled by the update simplification rules, after the program has been completely executed. This will happen at some point, since the rule reduces the general measure of enabledness in the system.

$$\mathsf{lock}\ \frac{\begin{array}{rl} & \Longrightarrow P(r,c)=pos \\ path(pos,p) & \Longrightarrow \{o^{*(pos)}.\texttt{<lockcount>}:=o^{*(pos)}.\texttt{<lockcount>}+1\} \\ & \quad \{o^{*(pos)}.\texttt{<lockedby>}:=Post(pos)+1\} \\ & \quad \langle\!|r|\,\pi\ ^{\{p_{pos}:n-1\}}\,o.\texttt{<lock>()}^{\{p_{pos+1}:k+1\}}\ \omega|\!\rangle\phi \\ \neg path(pos,p) & \Longrightarrow \langle\!|r|\,\pi\ ^{\{p_{pos}:n-1\}}\,o.\texttt{<lock>()}^{\{p_{pos+1}:k+1\}}\ \omega|\!\rangle\phi \end{array}}{\Longrightarrow \langle\!|r|\,\pi\ ^{\{p_{pos}:n\}}\ \underbrace{o.\texttt{<lock>()}}_{\text{at position } pos \text{ in } p}\ ^{\{p_{pos+1}:k\}}\ \omega|\!\rangle\phi}$$

Fig. 2. The rule for lock acquisition

3.4.4 The Rule for Empty Programs

In case no position is enabled in a configuration, the program does nothing and the modality can be removed altogether. The following rule applies:

$$\mathsf{empty-program}\ \frac{\Longrightarrow P(r,c)=0 \qquad \Longrightarrow \phi}{\Longrightarrow \langle\!|r|c|p|\!\rangle\phi}$$

3.4.5 Reasoning About Permutations

For the calculus to be complete, we need to add standard axioms that characterize permutations. We do not present these axioms here. It is a rule of the calculus that axioms can be added to the left side of any sequent at any time.

Together with the following permutation interplay axiom

$$p_{i+1}(Post(i+1)+1) \in \{p_i(1)\dots p_i(Post(i))\} \setminus \{p_{i+1}(1)\dots p_{i+1}(Post(i+1))\}$$

the calculus is sound and complete. This axiom constrains the threads that can be scheduled in a given configuration at position $i+1$. These are exactly the threads that have already passed the position i, but are not yet past position $i+1$.

4 Treating Concurrency Primitives

4.1 Treating Locking Primitives

At this point we add rules for reasoning about synchronized methods and blocks. Synchronized code offers a way to ensure mutual exclusion of threads by block-structured acquisition and release of locks associated with objects. To make this process explicit, we extend the `Object` class with a pair of “ghost” methods `<lock>()` and `<unlock>()`. Code marked as synchronized is automatically surrounded by invocations of these methods during the unfolding stage. The locking methods manipulate the ghost integer fields `<lockedby>` (identity of the thread holding the lock) and `<lockcount>` (locking depth), which are also introduced into every object.

The lock acquisition method is symbolically executed by applying the rule shown in Figure 2. The structure of this rule is similar to the STEP rule for handling unlock

$$
\frac{
\begin{array}{r@{\;}l}
& \Longrightarrow P(r,c) = \mathit{pos} \\[1em]
\mathit{path}(\mathit{pos},p) & \Longrightarrow \{o^{*(\mathit{pos})}\texttt{.<lockcount>:=}o^{*(\mathit{pos})}\texttt{.<lockcount>-1}\} \\
& \qquad \langle\!\![r|\,\pi\ ^{\{p_{\mathit{pos}}:n-1\}}o\texttt{.<unlock>()}^{\{p_{\mathit{pos}+1}:k+1\}}\ \omega]\!\!\rangle\phi \\[1em]
\neg\mathit{path}(\mathit{pos},p) & \Longrightarrow \langle\!\![r|\,\pi\ ^{\{p_{\mathit{pos}}:n-1\}}o\texttt{.<unlock>()}^{\{p_{\mathit{pos}+1}:k+1\}}\ \omega]\!\!\rangle\phi
\end{array}
}{
\Longrightarrow \langle\!\![r|\,\pi\ ^{\{p_{\mathit{pos}}:n\}}\ \underbrace{o\texttt{.<unlock>()}}_{\text{at position } \mathit{pos} \text{ in } p}\ ^{\{p_{\mathit{pos}+1}:k\}}\ \omega]\!\!\rangle\phi
}
$$

Fig. 3. The rule for lock release

normal assignments. Execution is successful if the path condition is satisfied and the statement is enabled (remember, $P(r,c) = \mathit{pos}$ implies $\mathit{enabled}(c,\mathit{pos})$).

In addition, we also amend the enabledness predicate in order to capture the mutual exclusion semantics of locking. The new definition is (for o`.<lock>()` at pos):

$$
\begin{gathered}
\mathit{enabled}(c,\mathit{pos}) \equiv (c(\mathit{pos}) > 0) \wedge \\
(o\texttt{.<lockcount>} = 0 \vee o\texttt{.<lockedby>} = \mathit{Post}(\mathit{pos}) + 1)
\end{gathered}
$$

The added second line means that either the lock has to be available or it has been previously acquired by the thread requesting it (reentrant locking). A similar rule exists for the `<unlock>()` method, which decreases the lock count and clears the locked by status when the count reaches zero. For simplicity we do not clear the `<lockedby>` flag, since it does not prevent the acquisition of the lock once `<lockcount>` reaches zero.

The presence of locking opens a possibility for deadlock. Just as the sequential KeY calculus maps abrupt termination onto non-termination, we have decided to model deadlock logically as termination. It is still easy to discern a deadlocked state from normal termination by considering the final program configuration. Besides, the desired postcondition would still hold, even if the program becomes prematurely disabled.

4.2 Treating Condition Variables

An important feature of Java's concurrency mechanism is condition variables. It allows threads to suspend execution until an external signal is received. The signaling does not involve thread identities, but works via a shared reference to an arbitrary object.

The waiting thread must acquire the object's lock first. Calling `wait()` on the object releases the lock and suspends thread execution. When a wake-up signal is received, the thread leaves the suspended state but does not yet continue execution. It must compete now for the acquisition of the lock with other threads. When this succeeds, the state of the lock is restored as before the wait.

The notifying thread must possess the object lock as well. Sending a wake-up signal to one (randomly chosen) suspended thread requires calling `notify()` on the corresponding object. Waking up all threads waiting is possible by calling `notifyAll()`. Again, the waiting threads will be able to proceed *in the earliest* when the notifying thread has released the lock.

Since other threads can intervene and destroy the condition between the wake-up signal and lock re-acquisition (a phenomenon known as "barging"), it is in most cases compulsory to re-test the condition and return to the suspended state if it is not satisfied. This practice is advocated by all programming guidelines and followed by most of the programs. Unfortunately, it constitutes a non-atomic loop, which we cannot (yet) treat in our framework.

On the other hand, for conditions that are uniform and atomic (as outlined below), we can consider the whole wait-in-loop idiom as one atomic statement. Most programs in practice satisfy these requirements. Such programs can be verified with the calculus presented in the following.

4.2.1 Additional Means of Expression

We package the common implementation of a condition variable in a special ghost method `void <waitUntil>(boolean b, int depth)`, which we add to the `Object` class. The intuitive meaning of this method is to stall all thread movement at this point until the given condition is satisfied. The method also provides every passing thread with a lock on the object (which must be free for the method to execute), thus capturing the absence of barging.

The actual Java implementation to be verified is replaced by this method during the unfolding stage of the verification process. The method has two parameters: a boolean condition, which must evaluate to true for a thread to proceed (it is the negated condition of the condition-testing while loop in the original program), and an integer indicating the previous locking depth. The lock given to the proceeding thread will be set to this locking depth.

The appropriate locking depth is returned by another ghost method we introduce: `int <unlockFull>()`. It is placed by the unfolding process before every `<waitUntil>()`. The method decreases the locking depth to zero, effectively releasing the lock; also, the locking depth before the call is returned to the caller. The unfolding also adds a check for the appropriate lock state. An example of the unfolding is given in the Figures 4 and 5.

Finally, we need some means to differentiate between threads that are ready to enter the section guarded by `<waitUntil>()` and threads that have suspended their execution until a notification arrives. We employ the ghost field `<waiting>` present in every object to keep track of the number of suspended threads.

4.2.2 Restrictions Posed on Programs

In order to verify programs with condition variables with our calculus we have to pose several restrictions on programs.

The condition of the `<waitUntil>()` may not have any side effects. This can be expressed by an assignable clause and checked by a number of methods including deductive verification with KeY. On the other hand, this requirement can be relaxed to include arbitrary code as long as it is independent of the system under verification. This would allow, for instance, allocation of iterators.

Since our framework does not support thread identities, programs are not al-

```
private LinkedList list = new LinkedList();

public synchronized void put(Object o) {
    list.add(o);
    this.notifyAll();
}

public synchronized Object get() {
    try{
        while (list.isEmpty()) this.wait();
    } catch(InterruptedException e) {
        // If we get here, we were not actually notified.
        // Returning null doesn't indicate that the
        // queue is empty, only that the waiting was abandoned.
        return null;

    }
    return list.removeFirst();
}
```

Fig. 4. Blocking queue source code

```
private LinkedList list = new LinkedList();

public void put(Object o) {
    this.<lock>();
    list.add(o);
    boolean b = !Thread.holdsLock(this);
    if (b) throw new IllegalMonitorStateException();
    this.notifyAll();
    this.<unlock>();
}

public Object get() {
    this.<lock>();                                  //*
    boolean b = !Thread.holdsLock(this);            //*
    if (b) throw new IllegalMonitorStateException(); //*
    int d = this.<unlockFull>();                    //*
    this.<waitUntil>(!list.isEmpty(), d);
    return list.removeFirst();
    this.<unlock>();
}
```

Fig. 5. Blocking queue source code (unfolded)

lowed to call `interrupt()` on a thread. Thus, `<waitUntil>()` also never throws an `InterruptedException`.

Unsurprisingly, we also don't allow the use of the `wait(long timeout)` method, since our framework has no notion of real time.

4.2.3 An Example Application

A blocking queue allows producer and consumer threads to exchange data elements. For this purpose the queue offers the operations `put()` and `get()`. Calling `get()` on an empty queue results in the consumer being blocked until a new element from a producer arrives.

A typical specification of the queue could demand that connecting n producers and n consumers via the queue will result in all threads running to completion, and the items retrieved will be exactly the items deposited (in some order induced by notifyAll

$$\frac{\begin{array}{r l} & \Longrightarrow P(r,c) = \mathit{pos} \\ \mathit{path}(\mathit{pos},p) & \Longrightarrow \{o^{*(\mathit{pos})}.\texttt{<waiting>}{:=}0\} \\ & \quad \langle\!\!\{ r | \; \pi \;{}^{\{p_{\mathit{pos}}:n-1\}} o\texttt{.notifyAll()}^{\{p_{\mathit{pos}+1}:k+1\}} \;\; \omega \}\!\!\rangle \phi \\ \neg\mathit{path}(\mathit{pos},p) & \Longrightarrow \langle\!\!\{ r | \; \pi \;{}^{\{p_{\mathit{pos}}:n-1\}} o\texttt{.notifyAll()}^{\{p_{\mathit{pos}+1}:k+1\}} \;\; \omega \}\!\!\rangle \phi \end{array}}{\Longrightarrow \langle\!\!\{ r | \; \pi \;{}^{\{p_{\mathit{pos}}:n\}} \underbrace{o\texttt{.notifyAll()}}_{\text{at position } \mathit{pos} \text{ in } p} {}^{\{p_{\mathit{pos}+1}:k\}} \;\; \omega \}\!\!\rangle \phi}$$

Fig. 6. The rule for notification

the scheduler). This can be written as:

$$\begin{array}{l} \texttt{q.<lockcount>} = 0 \land \neg\texttt{q} = \texttt{null} \land \texttt{q.list.size} = 0 \rightarrow \\ \qquad \forall n.\; n > 0 \rightarrow \langle^{\{p_1:n\}}\texttt{q.put(in);}^{\{0\}}\texttt{||}^{\{p_m:n\}}\texttt{out=q.get();}^{\{0\}}\rangle \\ \qquad\qquad\qquad\qquad \forall k.\; 1 \leq k \leq n \rightarrow \texttt{out}(p_r(k)) = \texttt{in}(p_a(k)) \end{array}$$

where r and a are positions of list removal and addition operations respectively.

The diamond formula above includes two thread classes separated by ||. More thread classes can be added in similar manner. We number the positions in the program continuously from left to right, but now every thread class has its own extra "end-of-thread" position. We also amend the definition of $\mathit{Post}(\mathit{pos})$ to include only positions in the same thread class as pos. Everything else can remain the same.

A typical implementation for such a queue is shown in Figure 4, while Figure 5 shows the result of a partial unfolding (list operations and exceptions are not unfolded). A simplification is possible by leaving out the marked part of the code in the `get()` method, since it serves little purpose in this particular setting. A fixed value of 1 can be used for `d` in this case. Currently, we are working on a mechanized inductive correctness proof of this example.

4.2.4 The Rules for Symbolic Execution

We start with a rule for `notifyAll()`, which is shown in Figure 6. If this statement is enabled (first premiss) and the path condition is satisfied the rule wakes up all suspended threads by setting the `<waiting>` counter to zero (second premiss). If the path condition is not satisfied the statement is a no-op. A similar rule can be given for `notify()`, which decrements the `<waiting>` counter by one. If the program has more than one `wait()` on (potentially) the same object then position-indexed `<waiting>` fields have to be used. This extension is straightforward, and we leave it out here.

Now we look at the rule for symbolic execution of `<waitUntil>()` given in Figure 7. The first premiss requires that the position in question is enabled: $\mathit{enabled}(c, \mathit{pos})$. We have given a general definition of enabledness in Section 3.4.2, and updated it for locking operations in Section 4.1. For the o.`<waitUntil>()`

$$\mathsf{waitUntil}\quad \dfrac{\begin{array}{rl}
& \Longrightarrow P(r,c) = pos \\[1ex]
& \Longrightarrow \Phi \leftrightarrow \langle \texttt{boolean x = } b^{*(pos)}\texttt{;}\rangle \texttt{x} = \mathit{TRUE} \\[1ex]
\mathit{path}(pos,p), & \Phi \Longrightarrow \{o^{*(pos)}\texttt{.<lockcount>}{:=}\mathit{depth}\} \\
& \{o^{*(pos)}\texttt{.<lockedby>}{:=}\mathit{Post}(pos){+}1\} \\
& \langle\!\!\langle r|\, \pi\ ^{\{p_{pos}:n-1\}}\, o\texttt{.<waitUntil>(}b\texttt{, } \mathit{depth}\texttt{)}^{\{p_{pos+1}:k+1\}}\ \omega\rangle\!\!\rangle\phi \\[1ex]
\mathit{path}(pos,p), & \neg\Phi \Longrightarrow \{o^{*(pos)}\texttt{.<waiting>}{:=}o^{*(pos)}\texttt{.<waiting>}{+}1\} \\
& \langle\!\!\langle r|\, \pi\ ^{\{p_{pos}:n\}}\, o\texttt{.<waitUntil>(}b\texttt{, } \mathit{depth}\texttt{)}^{\{p_{pos+1}:k\}}\ \omega\rangle\!\!\rangle\phi \\[1ex]
\neg\mathit{path}(pos,p) & \Longrightarrow \langle\!\!\langle r|\, \pi\ ^{\{p_{pos}:n-1\}}\, o\texttt{.<waitUntil>(}b\texttt{, } \mathit{depth}\texttt{)}^{\{p_{pos+1}:k+1\}}\ \omega\rangle\!\!\rangle\phi
\end{array}}{\Longrightarrow \langle\!\!\langle r|\, \pi\ ^{\{p_{pos}:n\}}\ \underbrace{o\texttt{.<waitUntil>(}b\texttt{, } \mathit{depth}\texttt{)}}_{\text{at position } pos \text{ in } p}\ ^{\{p_{pos+1}:k\}}\ \omega\rangle\!\!\rangle\phi}$$

Fig. 7. The rule for `<waitUntil>()`

operation at position *pos*, we update the predicate again, to:

$$\mathit{enabled}(c,pos) \equiv (c(pos) - o\texttt{.<waiting>}) > 0 \land o\texttt{.<lockcount>} = 0$$

This means that at least one thread at *pos* has to be out of suspended state and the lock of o has to be available, since it will be acquired during the execution.

The second premiss captures the condition Φ of the condition variable. Φ can be $\langle\texttt{boolean x =}b^{*(pos)}\texttt{;}\rangle\texttt{x} = \mathit{TRUE}$ or its first-order equivalent. Note that the diamond formula is purely sequential and $b^{*(pos)}$ is the sequential instantiation of b for the next thread to run at *pos* (i.e., thread with id $p_{pos}(\mathit{Post}(pos) + 1)$). In the case of the blocking queue, Φ is simply $\texttt{q.list.size} > 0$. The rule is complete if the condition is uniform (i.e., if one thread satisfies it, then all do). This is the case when the condition is expressed in terms of a shared data structure. We have not yet fully investigated the completeness of the rule for non-uniform conditions.

The third premiss assumes that the condition Φ is satisfied. In this case one of the non-suspended threads can proceed past the `<waitUntil>()`. The proceeding thread will have acquired the object lock as the result of the execution of `<waitUntil>()`.

The fourth premiss assumes that the condition Φ does not hold. In this case there is no thread movement (the configuration does not change), but the number of suspended threads o`.<waiting>` increases by one.

The fifth premiss deals with the negative path condition. In this case, just as with other rules, the thread executes a no-op.

5 Conclusion

We have defined a Dynamic Logic for reasoning about input-output behavior of a subset of concurrent Java programs. The subset includes (common) programs utilizing condition variables. For this logic we have presented a deductive calculus that is based on efficient symbolic execution. This was made possible by a significant extension of the technique of symmetry reduction.

Currently, we are performing experiments with the mechanization of the calculus in the KeY system. Furthermore, we are working on extending the covered Java fragment—in particular to include non-atomic loops—by devising an algebraically more elaborated model of the scheduler.

References

[1] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theor. Comp. Sci.*, 331(2–3):251–290, 2005.

[2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[4] B. Beckert and V. Klebanov. A dynamic logic for deductive verification of concurrent programs. In M. Hinchey and T. Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK*. IEEE Press, 2007. To appear. Available from http://www.key-project.org.

[5] G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded Java programs. In J.-P. Katoen and P. Stevens, editors, *Proceedings, 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2280 of *LNCS*, pages 173–187. Springer, 2002.

[6] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 453–463, 2002.

[7] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[8] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Z. Liu and J. He, editors, *8th International Conference on Formal Engineering Methods, ICFEM, Macao, China, Proceedings*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006.

[9] V. Klebanov. A JMM-faithful non-interference calculus for Java. In *Scientific Engineering of Distributed Java Applications, 4th International Workshop, Proceedings, Luxembourg-Kirchberg*, volume 3409 of *LNCS*, pages 101–111. Springer, 2004.

[10] Z. Manna and A. Pnueli. Completing the temporal picture. In *Selected papers of the 16th international colloquium on automata, languages, and programming*, pages 97–130. Elsevier Science Publishers B. V., 1991.

[11] D. Peleg. Communication in concurrent dynamic logic. *J. Comput. Syst. Sci.*, 35(1):23–58, 1987.

[12] D. Peleg. Concurrent dynamic logic. *J. ACM*, 34(2):450–479, 1987.

[13] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. In *Proceedings SoftMC 2003, Workshop on Software Model Checking, ENTCS 89*, 2003.

[14] E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP*, LNCS 3586, pages 551–576. Springer, 2005.

[15] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–40. ACM Press, 2001.

Universes for Race Safety

D. Cunningham [a,1] S. Drossopoulou[a,2] S. Eisenbach[a,3]

[a] *Department of Computing*
Imperial College London
U.K.

Abstract

Race conditions occur when two incorrectly synchronised threads simultaneously access the same object. Static type systems have been suggested to prevent them. Typically, they use annotations to determine the relationship between an object and its "guard" (another object), and to guarantee that the guard has been locked before the object is accessed. The object-guard relationship thus forms a tree similar to an ownership type hierarchy.

Universe types are a simple form of ownership types. We explore the use of universe types for static identification of race conditions. We use a small, Java-like language with universe types and concurrency primitives. We give a type system that enforces synchronisation for all object accesses, and prove that race conditions cannot occur during execution of a type correct program.

We support references to objects whose ownership domain is unknown. Unlike previous work, we do so without compromising the synchronisation strategy used where the ownership domain of such objects is fully known. We develop a novel technique for dealing with non-final (i.e. mutable) paths to objects of unknown ownership domain using effects.

Keywords: ownership, synchronisation, concurrency, race condition, effects, atomicity

A race condition is an error that can occur in concurrent programs when two threads are not properly synchronised, and thus can simultaneously access the same object. This can then lead to corruption of data structures, and eventual software failure. To date, many well-known pieces of software have fallen foul of race conditions, often long after their initial development, sometimes leading to denial-of-service attacks or other security problems.

Programmers typically attempt to avoid race conditions through disciplined programming; by ensuring that the right lock is taken during all shared object accesses. They must choose which locks guard their objects and respect this relationship everywhere in their code [21].

Previous work [11,3] uses ownership type annotations [5,28] (or "guard" annotations that resemble ownership) to restrict variable bindings to objects in specific regions of the heap. Every region has an associated lock, so the type systems know which locks protect a block of code without precisely knowing which objects will be

[1] Email: dc04@doc.ic.ac.uk

[2] Email: scd@doc.ic.ac.uk

[3] Email: sue@doc.ic.ac.uk

accessed. This requires all types to be explicitly annotated by ownership parameters, allowing the expression of complex ownership structures, albeit at the expense of heavier annotation.

Universes [26,8] are a simple, yet powerful form of ownership types used in the JML tools [22]. Universes let programmers succinctly specify the topological relationship between objects using just a few keywords. As such, the owner of an object is implicitly understood by the type system, and implicitly stored by the run-time environment. Thus, programmers need not explicitly declare them.

We developed a type system for race safety using universes to partition the heap. As in [3,4,11,12,13,14] we treat objects in the same *ownership domain* (i.e. all objects sharing the same owner) as guarded by the same lock. At run-time we associate this lock with the objects' owner. All objects have an implicit reference to their owner.

Universes also allow references to objects whose owner is unknown through the annotation `any` [9] (in earlier work [26] called `readonly`, which is distinct from `final` because it describes the referenced object). This is not supported by [11,3].

The presence of `any` was a challenge for us, but turned out to increase the expressiveness of our language. The `any` annotation allows the expression of data structures that contain objects from various ownership domains. Use of such data structures does not require us to compromise the design of other data structures in our program. This improves upon [3,11], where in particular one sometimes has to alter the design of unrelated data structures so that they take the lock once for each access in an iteration. Iterating over the unrelated structures would be atomic in our system but could not be atomic in [3,11].

When the type does not indicate the owner of an object, we use paths as an alternative mechanism to guarantee correct synchronisation. We use an effect system where these paths are not final as would be required in [3,11].

Previous work [3,11] required entire ownership domains to be locked even if only a single object is accessed. It is straightforward to extend our system with single object locks.

The rest of this paper is organised as follows: In section 1 we explain universes and their suggested use for concurrency in an example program. In section 2 we give a formal model of universes. In section 3 we give (and prove) a simple type system that guarantees race safety. We discuss implementation in section 4 and related work in section 5. We discuss future directions in section 6 and conclude with section 7.

1 An example of Universes and Race Safety

The run-time state of an object-oriented program consists of a graph of objects linked by field references. In an ownership system, each object is owned by another object. The ownership relation describes a tree structure whose root is `null`. This tree structure represents the encapsulation inherent in the design of a program [5,28].

In a universe type system, types consist of a class and a keyword that indicates the topological relationship. We call the keyword an *ownership type qualifier*; it is one of `rep`, `peer`, and `any`.

```
class Stud { int mark ; boolean roomClean  }

class Dept {  // Closed list
    rep DeptStudNode first;
    void releaseMarks () { ... }
}

class DeptStudNode {
    peer Stud s;
    peer DeptStudNode next;
}

class Hall {  // Open list
    rep HallStudNode first;
    void cleanRooms () { ... }
}

class HallStudNode {
    any Stud s;
    peer HallStudNode next;
}
```

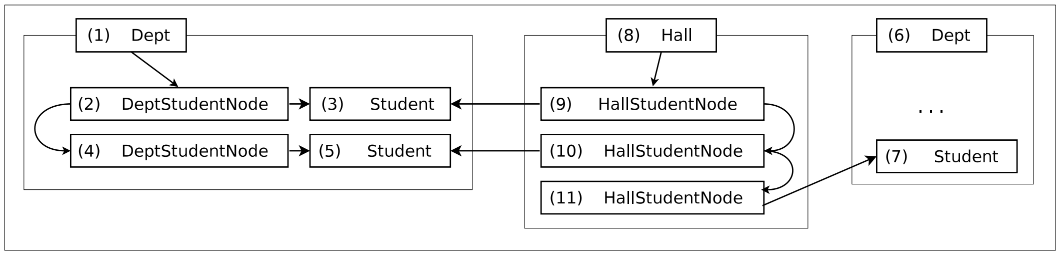


Fig. 1. Example program showing heap hierarchy structure

Types are relative to some *observer* object. When the observer has the same owner as another object, the second object is a `peer` of that observer. When the observer is the owner of another object, the second object is a `rep` of that observer. Any object can be `any`.

Consider, for example, the heap diagram from Fig. 1. Each object has an address, *e.g.,* (1), and a class name, *e.g.,* `Dept`. Owned objects are drawn in the domains of their owners; the tree is represented by the nesting of the boxes, (1) owns (2 – 5), and (8) owns (9 – 11). From observer (1), (3) has type `rep Stud` [4], but from observer (2), the (3) has type `peer Stud`. Thus, the type of (3) is relative to the observer. Also, from observer (8) the object (3) has type `any Stud`, and the object (1) has type `peer Dept` [5].

In Fig. 1 we show source code where types contain universe annotations, and which could give rise to the heap in the diagram. For example, class `Dept` has field `first` of type `rep DeptStudNode`, which, in the diagram corresponds to the reference from (1) to (2). On the other hand, `HallStudNode` has field `s` of type `any Stud`, which, in the diagram corresponds to the reference from (9) to (3).

Thus, through `any`, students (owned by their respective departments), can be accessed also from their halls of residence. We call the list inside `Dept` a *closed* list as the students are enclosed in the ownership domain of the list. In contrast the list inside `Hall` is *open* because its students can be in any department.

We now discuss the use of the tree-hierarchy imposed by the universe types to avoid races: We require that the run-time system records the owner of an object (which does not change). We associate a lock with each object, with objects guarded by their owner's lock rather than their own. Any accesses to a field of an object, for example $e'.f$ or $e'.f$ = ..., must be within a `sync` e block where e resolves to an object that is part of the same ownership domain as e'. This is in contrast to Java's `synchronized` which locks e and not the whole of its ownership domain.

[4] It trivially also has type `any Stud`.

[5] The reference from (8) to (3) is illegal in systems enforcing owners-as-dominators [5,28] but is legal in universe types, which, instead, enforce owners-as-modifiers.

```
18 void releaseMarks () {
19   sync (this) {
20     rep DeptStudNode i = this.first;
21     sync (i) {
22       while (i!=null) {
23         i.s.mark = ...;
24         i = i.next;
25 } } } }
```

```
26 void cleanRooms () {
27   sync (this) {
28     rep HallStudNode i = this.first;
29     sync (i) {
30       while (i!=null) {
31         sync (i.s)
32           { i.s.roomClean = true; }
33         i = i.next;
34 } } } }
```

Fig. 2. Method bodies for Fig. 1

One can also think of `sync` e as locking the object owning e. We propose that `synchronized` be no-longer used, in favour of `sync`. Nested ownership domains are disjoint; code that accesses both must take both locks. The ownership domain is statically verifiable when the ownership type qualifier of e' is not `any`.

Consider the code for `releaseMarks` from Fig. 2. Adhering to the rule set out above, the field access to `this.first` (line 20) is enclosed within `sync (this)` (line 19). More interesting is the body of the `while` loop, where a statically unknown number of field accesses through `i.next` (line 24) is correctly synchronised by acquiring a *single* lock, `sync (i)`, before the loop (line 21). Even though `i` will point to different objects at each iteration, the synchronisation is correct, because the field `next` is `peer` and thus all these objects will have the same owner. The same is true when we access the student (line 23).

A challenge we needed to tackle is, how to avoid races when the ownership domain of the accessed object is unknown, *i.e.,* when it has type `any C` for some class `C`. In such a case, any accesses of the form $p.f$ or $p.f = ...$, where p is a path [6] must be within a `sync` p block provided that the block does not assign to any of the fields appearing in p.

The difference between the body of `cleanRooms` in Fig. 2 and `releaseMarks` is that in the former, `HallStudNode` has an `any` pointer to `Stud`. Thus, the student is not necessarily a peer of the node `i`; therefore, when we access `i.s` (line 32) the `sync (i)` (line 29) is no longer sufficient. We must lock the owner of the student `i.s` and this is possible through the "fresh" `sync (i.s)` (line 31) even though `i.s` is `any`. We must be sure however that the body of the `sync (i.s)` block does not write to the field `s`, otherwise the type system would reject our program.

Note that in `releaseMarks`, students will receive their marks *atomically* (there is never a state visible where a subset of students have their marks) but this is not the case for the cleaning of rooms. A student may notice their room has been cleaned whereas another student's room has not. In general, we must lock individual elements when iterating through an open list. This is not necessary for a closed list.

In [11], an open list of students can be written if we design the student so that it has a final field that stores the owner. In other words, we create a class that can be referenced by a variable whose type does not specify an owner such as `s` of `HallStudNode`. However, this change is global to the program so every other reference (*e.g.,* the field `s` of `DeptStudNode`) must use the same type that does not specify an owner. This means that we cannot make a closed list of students, because the owner of the student is no longer indicated by its type. The only solution is to use open lists everywhere, which have the undesirable property that we cannot lock

[6] A path is a sequence of field accesses starting from a parameter or `this`.

```
// We have to design Stud like this:

class Stud {
   final Object owner;
   /* fields guardedby this.owner */
}

// Therefore, the HallStudNode looks like:

class HallStudNode<x> {
   Stud s guardedby x;
   HallStudNode<x> next guardedby x;
}

// Hall locks its students like so:

class Hall<x> {  // Open list
   HallStudNode<this> first guardedby x;
   void cleanRooms () {
      sync (x) {  //protects fields of this
         HallStudNode<this> i=this.first;
         sync (this) {  //protets nodes
            while (i) {
               final Stud s' = i.s;
               sync (s'.owner) {
                  s'.roomClean = true;
               }
               i = i.next;
}  }  }  }  }


// But a Stud is a Stud, so we must design
// DeptStudNode in the same manner:

class DeptStudNode<x> {
   Stud s guardedby x;
   DeptStudNode<x> next guardedby x;
}


// And thus we have to lock each student

class Dept<x> {  // must be open too!
   DeptStudNode<this> first guardedby x;
   void releaseMarks () {
      sync (x) {
         DeptStudNode<this> i=this.first;
         sync (this) {
            while (i) {
               final Stud s' = i.s;
               sync (s'.owner) {
                  s'.mark = ...;
               }
               i = i.next;
}  }  }  }  }
```

```
// We have to design student like this:

class Stud<x> {
   /* fields here */
}

// Therefore, HallStudNode looks like:

class HallStudNode<x> {
   Stud<self> s;
   HallStudNode<x> next;
}

// Hall locks its students like so:

class Hall<x> {  // Open list
   HallStudNode<this> first;
   void cleanRooms () {
      sync (x) {
         HallStudNode<this> i=this.first;
         sync (this) {
            while (i) {
               final Stud<self> s'=i.s;
               sync (s') {
                  s'.roomClean = true;
               }
               i = i.next;
}  }  }  }  }


// Since the students we reference have type
// Stud<self>, this field must be the same:

class DeptStudNode<x> {
   Stud<self> s;
   DeptStudNode<x> next;
}

// But we do not own the student, so we must
// lock each student individually.

class Dept<x> {  // must be open too!
   DeptStudNode<this> first guardedby x;
   void releaseMarks () {
      sync (x) {
         DeptStudNode<this> i=this.first;
         sync (this) {
            while (i) {
               final Stud<self> s'=i.s;
               sync (s') {
                  s'.mark = ...;
               }
               i = i.next;
}  }  }  }  }
```

Fig. 3. Example code in the systems of [11] (left) and [4] (right)

all the elements of the list at once, we have to acquire the same lock once for each student. Another implication is that iterating through the list cannot be atomic (as in `releaseMarks`).

The type system of [4] is even more restrictive, as a closed list implementation can only contain self-owned objects. This means objects contained in an open list also can only exist in the root ownership domain. The code for both solutions is given in Fig. 3.

2 Formal Preliminaries

Universes are introduced in [26], and given a type theoretic presentation in [7]. We use ideas from [26] but with some differences: We decided that owners-as-modifiers, while useful for verification, are not needed for type soundness and race safety. Our type system allows field assignments through `any` objects as long as the heap

$$\mathcal{M} : (Id^c \times Id^m) \rightarrow TypeSig \qquad \mathcal{MBody} : (Id^c \times Id^m) \rightarrow SrcExpr$$

$$\mathcal{F} : (Id^c \times Id^f) \rightarrow Type \qquad c \in Id^c \qquad f \in Id^f$$

$$\begin{array}{rcl}
e \in SrcExpr & ::= & \texttt{this} \mid \texttt{x} \mid \texttt{null} \mid \texttt{new}\ t \mid e.f \mid e.f = e \mid e.m(e) \\
& \mid & (t)\ e \mid \texttt{spawn}\ e \mid \texttt{sync}\ e\ e \\
t \in Type & ::= & u\ c \\
u \in Universe & ::= & \texttt{rep} \mid \texttt{peer} \mid \texttt{any} \mid \texttt{self} \\
TypeSig & ::= & t\ m(t) \\
\Gamma \in Environment & = & Type \times Type \qquad \Gamma(\texttt{this}) = \Gamma{\downarrow_1},\ \Gamma(\texttt{x}) = \Gamma{\downarrow_2}
\end{array}$$

We define $(\leq_c)$, the reflexive transitive closure of the inheritance in the program.

We define $(\leq_u)$: $\texttt{self} \leq_u \texttt{peer} \leq_u \texttt{any} \quad \texttt{rep} \leq_u \texttt{any}$

Thus we define the subtype relation $(\leq)$: $u\ c \leq u'\ c' \iff u \leq_u u' \wedge c \leq_c c'$

Fig. 4. Source program definition

remains well-formed. Therefore our type system is more permissive and could be restricted to also require owners-as-modifiers.

For sequences, we use the notation $\overline{e}$ in the style of [19], and sometimes as $e_{1..n}$, both of which are distinct from the undecorated e. The ith element of $e_{1..n}$ is e_i and of $\overline{e}$ is $\overline{e}{\downarrow_i}$. We use similar notation when we access the ith element of a tuple: $(a, b, c){\downarrow_2} = b$. We use an underscore _ to represent a variable whose value can be arbitrary. To denote that a particular construct *e.g.,* `new` c occurs within an expression e, we sometimes write `new` $c \in e$. $\mathbb{P}$ is the powerset.

2.1 Syntax and Semantics

Programs are defined in Fig. 4, and consist of the three functions $\mathcal{M}$, $\mathcal{MBody}$, and $\mathcal{F}$, which define the method signatures, method bodies, and field types of each class in the program, together with $(\leq_c)$ which gives the inheritance relationship between classes. Note that all types t are annotated with an ownership type qualifier u which can be one of the three keywords `rep`, `any`, `peer`, or `self` which is the type of `this` and thus a specialisation of `peer`. It prevents the type system losing type information during local member access. We use `spawn` e to start a new thread to execute e, and `sync` e e' to acquire the lock that guards the object e while we execute the expression e'. We give the run-time syntax in Fig. 5 and use a small step semantics.

The program state consists of the heap h and a sequence of expressions $\overline{e}$. It is reduced with respect to a base stack frame σ according to the rules in Fig. 6. The (Interleave) rule uses the single-threaded semantics. The base stack frame contains the value of `this` and `x`. Single-threaded execution steps are decorated with actions, ranged over by β. If a step accesses an address a, then its action is a, otherwise its action is τ. At the multi-threaded level, each step may introduce at most one more thread, stopped threads are never eliminated from the system, and we wrap actions

$$
\begin{array}{rcll}
s \in \mathit{State} & : & \mathit{RunExpr} \times \mathit{Heap} & \\
h \in \mathit{Heap} & : & \mathit{Addr} \rightarrow \mathit{Object} & \\
\mathit{Object} & : & (\mathit{Val} \times \mathit{Id}^c \times (\mathit{Id}^f \rightarrow \mathit{Val})) & \texttt{//}\text{ owner, class, fields} \\
a \in \mathit{Addr} & : & \mathbb{N} & \\
v, w \in \mathit{Val} & ::= & a \mid \texttt{null} & \\
\sigma \in \mathit{Stack} & ::= & (a, v) & \sigma(\texttt{this}) = \sigma{\downarrow_1},\ \sigma(\texttt{x}) = \sigma{\downarrow_2} \\
\beta \in \mathit{Actions} & ::= & a \mid \tau & \\
e \in \mathit{RunExpr} & ::= & v \mid \texttt{this} \mid \texttt{x} \mid \texttt{new}\ t \mid e.f \mid e.f = e \mid e.m(e) \mid (t)\ e \mid \texttt{spawn}\ e & \\
 & \mid & \texttt{sync}_e\ e\ e \mid \texttt{synced}_e\ w\ e \mid \texttt{frame}\ \sigma\ e & \\
E[\cdot] & ::= & E[\cdot].f \mid E[\cdot].f = e \mid v.f = E[\cdot] \mid E[\cdot].m(e) \mid v.m(E[\cdot]) & \\
 & \mid & (t)\ E[\cdot] \mid \texttt{sync}_e\ E[\cdot]\ e \mid \texttt{synced}_e\ w\ E[\cdot] &
\end{array}
$$

Fig. 5. Run-time state and syntax

with the index of the thread that caused them.

Method calls are modelled by substituting the call construct with the method body in question and the stack which records the receiver and parameters. The `frame` σ e construct marks the boundaries between the different calling contexts in the run-time expression, and holds the new stack σ which is used to execute the method body e. One can see from the `sync` syntax of the run-time language, and the (Call) semantics rule, that `sync` expressions in the method body are translated slightly by S during method call. This does not affect the behaviour of the program, it was needed so that we could prove soundness.[7] The subtree e' is duplicated and recursively translated, to be held in the subscript of the new `sync` construct.

The semantics rules (Cast) and (New) use the universe type system to constrain their behaviour (*e.g.,* in (New) we set the owner field this way), this makes the proofs simple. The syntax allows for the expression `new self` c but this will always result in a stuck execution[8]. We also allow the expression `new any` c, where the owner of the new object is chosen arbitrarily.

For interleaved execution we use the context $C[\cdot]$, which extends the evaluation context syntax $E[\cdot]$ to add the stack frame construct:

$$C[\cdot] ::= \ldots \mid \texttt{frame}\ \sigma\ C[\cdot]$$

In rule (Lock), note that it is the owner of the object $w = h(a){\downarrow_1}$ that is actually locked. This is because we are locking the whole ownership domain, not just the

[7] We demonstrate the life-cycle of a `sync` construct with an example: The source expression `sync` e e'' is translated into the run-time expression $\texttt{sync}_e$ e' e''. Initially $e = e'$ but as the expression reduces, the e' will reduce until it reaches an object a, and then the lock that guards that object w is taken. The subscript persists on this $\texttt{synced}_e$ w e'' expression until e'' terminates and the lock released. The subscript expression e will always remain as a record of the initial locking expression, even after it has reduced to an object and the main body is executing.

[8] It would reduce to an unused address that is the same as $\sigma(\texttt{this})$, an existing address. In practice we could disallow the use of `self` and `any` when constructing new objects, in either the syntax or the static type system.

$$\frac{}{\sigma \vdash \texttt{this}, h \overset{\tau}{\rightsquigarrow} \sigma(\texttt{this}), h}\,(\textsc{This})$$

$$\begin{array}{rcl} S & : & \mathit{SrcExpr} \rightarrow \mathit{RunExpr} \\ S(\texttt{sync}\ e_1\ e_2) & = & \texttt{sync}_{S(e_1)}\ S(e_1)\ S(e_2) \\ S(\mathit{Con}(e_1 \ldots e_n)) & = & \mathit{Con}(S(e_1) \ldots S(e_n)) \\ & & \text{(for all other constructs } \mathit{Con} \in \mathit{SrcExpr}\text{)} \end{array}$$

$$\frac{}{\sigma \vdash \texttt{x}, h \overset{\tau}{\rightsquigarrow} \sigma(\texttt{x}), h}\,(\textsc{Var})$$

$$\frac{e = S(\mathcal{MBody}(h(a)\downarrow_2, m))}{\sigma \vdash a.m(v), h \overset{\tau}{\rightsquigarrow} \texttt{frame}\ (a, v)\ e, h}\,(\textsc{Call}) \qquad \frac{h, \sigma \vdash v : t}{\sigma \vdash (t)\ v, h \overset{\tau}{\rightsquigarrow} v, h}\,(\textsc{Cast})$$

$$\frac{h' = h[a\downarrow_3(f) \mapsto v]}{\sigma \vdash a.f = v, h \overset{a}{\rightsquigarrow} v, h'}\,(\textsc{Assign}) \qquad \frac{\sigma' \vdash e, h \overset{\beta}{\rightsquigarrow} e', h'}{\sigma \vdash \texttt{frame}\ \sigma'\ e, h \overset{\beta}{\rightsquigarrow} \texttt{frame}\ \sigma'\ e', h'}\,(\textsc{Frame1})$$

$$\frac{\sigma \vdash e_i, h \overset{\beta}{\rightsquigarrow} e_i', h'}{\sigma \vdash e_{1..n}, h \overset{(i,\beta)}{\rightsquigarrow} e_{1..i-1}\ e_i'\ e_{i+1..n}, h'}\,(\textsc{Interleave}) \qquad \frac{\begin{array}{l} h(a) \text{ undefined} \\ h' = h[a \mapsto (\_, c, \lambda f.\texttt{null})] \\ h', \sigma \vdash a : u\ \_ \end{array}}{\sigma \vdash \texttt{new}\ u\ c, h \overset{\tau}{\rightsquigarrow} a, h'}\,(\textsc{New})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{spawn}\ e'] \\ e_{n+1} = \texttt{frame}\ \mathit{Active}(\sigma, C[\cdot])\ e' \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[\texttt{null}]\ e_{i+1..n+1}, h}\,(\textsc{Spawn}) \qquad \frac{}{\sigma \vdash a.f, h \overset{a}{\rightsquigarrow} h(a)\downarrow_3(f), h}\,(\textsc{Field})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{sync}_{e'}\ a\ e] \qquad\qquad w = h(a)\downarrow_1 \\ \forall j \in \{1..n\}\ .\ \mathit{Locked}(e_j, w) \implies i = j \\ e'' = C[\texttt{synced}_{e'}\ w\ e] \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ e''\ e_{i+1..n}, h}\,(\textsc{Lock}) \qquad \frac{\sigma \vdash e, h \overset{\beta}{\rightsquigarrow} e', h'}{\sigma \vdash E[e], h \overset{\beta}{\rightsquigarrow} E[e'], h'}\,(\textsc{Ctx})$$

$$\frac{e_i = C[\texttt{synced}_{e'}\ w\ v]}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[v]\ e_{i+1..n}, h}\,(\textsc{UnLock}) \qquad \frac{}{\sigma \vdash \texttt{frame}\ \sigma'\ v, h \overset{\tau}{\rightsquigarrow} v, h}\,(\textsc{Frame2})$$

Fig. 6. Small step operational semantics

object specified by the sync block:

$$\texttt{sync}_{e'}\ a\ e \rightsquigarrow \texttt{synced}_{e'}\ w\ e$$

The predicate $\mathit{Locked}(e, w)$ determines whether the thread e has the lock on object w: It holds whenever the construct $\texttt{synced\_}\ w\ \_$ is a subexpression of e.

The function $\mathit{Active}(\sigma, e)$ provides the σ' from the innermost $\texttt{frame}\ \sigma'\ \_$ within the thread e according to the context rules for $C[\cdot]$. If there is no such $\texttt{frame}\ \sigma'\ \_$, it returns σ.

$$\frac{\Gamma \vdash e : t' \quad t' < t}{\Gamma \vdash e : t}\,(\textsc{Sub}) \qquad \frac{}{\Gamma \vdash \texttt{null} : t}\,(\textsc{Null}) \qquad \frac{}{\Gamma \vdash \texttt{x} : \Gamma(\texttt{x})}\,(\textsc{Var}) \qquad \frac{\Gamma \vdash e : t'}{\Gamma \vdash (t)\ e : t}\,(\textsc{Cast})$$

$$\frac{\Gamma \vdash e : u\ c \quad \mathcal{F}(c, f) = t}{\Gamma \vdash e.f : u \rhd t}\,(\textsc{Field}) \qquad \frac{u \neq \texttt{self}}{\Gamma \vdash \texttt{new}\ u\ c : u\ c}\,(\textsc{New}) \qquad \frac{}{\Gamma \vdash \texttt{this} : \Gamma(\texttt{this})}\,(\textsc{This})$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t'}{\Gamma \vdash \texttt{sync}\ e\ e' : t'}\,(\textsc{Sync}) \qquad \frac{\Gamma \vdash e : t'}{\Gamma \vdash \texttt{spawn}\ e : t}\,(\textsc{Spawn}) \qquad \frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{F}(c, f) = u \rhd t}{\Gamma \vdash e.f = e' : t}\,(\textsc{Assign})$$

$$\frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{M}(c, m) = t_r\ m(u \rhd t)}{\Gamma \vdash e.m(e') : u \rhd t_r}\,(\textsc{Call}) \qquad \frac{}{a, w \vdash a, w : \texttt{self}}\,(\textsc{Self})$$

$$\frac{}{\_, w \vdash \_, w : \texttt{peer}}\,(\textsc{Peer}) \qquad \frac{}{a, \_ \vdash \_, a : \texttt{rep}}\,(\textsc{Rep}) \qquad \frac{}{\_, \_ \vdash \_, \_ : \texttt{any}}\,(\textsc{Any})$$

$$\frac{\begin{array}{l}\forall c' \geq c\ .\ \mathcal{F}(c', f) = t \Longrightarrow \mathcal{F}(c, f) = t \\ \forall c' \geq c\ .\ \mathcal{M}(c', m) = t'_r\ m(t'_x) \Longrightarrow \mathcal{M}(c, m) = t_r\ m(t_x) \\ \qquad\qquad\qquad\qquad (\text{where } t_r \leq t'_r, t_x \geq t'_x) \\ \forall \mathcal{M}(c, m) = t_r\ m(t_x)\ .\ (\texttt{self}\ c, t_x) \vdash \mathit{MBody}(c, m) : t_r\end{array}}{\vdash c}\,(\textsc{WFClass})$$

A program is well-formed iff $\forall c. \vdash c$

Fig. 7. Static universe type system

2.2 Universe Type System

The universe type system is given in Fig. 7. The judgement $\Gamma \vdash e : t$ gives the type t of an expression e with respect to an environment Γ. As there is only one method parameter, this environment is simply a pair of the types of `this` and `x`.

Universe annotations have meaning only with respect to an observer as discussed in section 1. The type annotations in field and method signatures are meant with respect to the object that contains them. When we are typing method bodies, the annotations within are considered with respect to the object `this`, whatever value `this` might have at run-time. Thus the type returned by the type system is also

$$\frac{h, \sigma \vdash \sigma(\texttt{this}) : t}{h, \sigma \vdash \texttt{this} : t}(\textsc{This}) \qquad \frac{h, \sigma \vdash \sigma(\texttt{x}) : t}{h, \sigma \vdash \texttt{x} : t}(\textsc{Var}) \qquad \frac{}{h, \sigma \vdash \texttt{new}\ t : t}(\textsc{New})$$

$$\frac{}{h, \sigma \vdash \texttt{null} : t}(\textsc{Null}) \qquad \frac{h, \sigma \vdash e : t'}{h, \sigma \vdash (t)\ e : t}(\textsc{Cast}) \qquad \frac{h, \sigma \vdash e : t}{h, \sigma \vdash \texttt{synced}_{e''}\ a\ e : t}(\textsc{Synced})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ \mathcal{M}(c, m) = t_r\ m(u \rhd_{-} t) \\ h, \sigma \vdash e' : t \end{array}}{h, \sigma \vdash e.m(e') : u \rhd_{+} t_r}(\textsc{Call}) \qquad \frac{\begin{array}{l} h, \sigma' \vdash e : t \\ h, \sigma \vdash \sigma'(\texttt{this}) : u\ \_ \end{array}}{h, \sigma \vdash \texttt{frame}\ \sigma'\ e : u \rhd_{+} t}(\textsc{Frame})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : t' \\ t' < t \end{array}}{h, \sigma \vdash e : t}(\textsc{Sub}) \qquad \frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ \mathcal{F}(c, f) = t \end{array}}{h, \sigma \vdash e.f : u \rhd_{+} t}(\textsc{Field}) \qquad \frac{\begin{array}{l} h, \sigma \vdash e : u\ c \\ h, \sigma \vdash e' : t \\ \mathcal{F}(c, f) = u \rhd_{-} t \end{array}}{h, \sigma \vdash e.f = e' : t}(\textsc{Assign})$$

$$\frac{\begin{array}{l} h, \sigma \vdash e : t \\ h, \sigma \vdash e' : t' \end{array}}{h, \sigma \vdash \texttt{sync}_{e''}\ e'\ e : t}(\textsc{Sync}) \qquad \frac{\begin{array}{l} h(a)\downarrow_2 = c \\ \sigma(\texttt{this}), h(\sigma(\texttt{this}))\downarrow_1 \vdash a, h(a)\downarrow_1 : u \end{array}}{h, \sigma \vdash a : u\ c}(\textsc{Addr})$$

$$\frac{\forall i \in \{1..n\}\ .\ h, \sigma \vdash e_i : t_i}{h, \sigma \vdash e_{1..n} : t_{1..n}}(\textsc{Threads}) \qquad \frac{h, \sigma \vdash e : t'}{h, \sigma \vdash \texttt{spawn}\ e : t}(\textsc{Spawn})$$

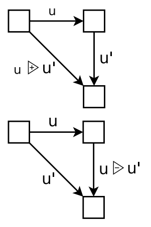
$$\frac{\begin{array}{l} h(a) = (\_, c, flds) \\ \forall \mathcal{F}(c, f) = t\ \ .\ \ h, (a, \_) \vdash flds(f) : t \end{array}}{h \vdash a}(\textsc{WFAddr})$$

Heap well-formedness:
$\vdash h \iff \forall a \in dom(h). h \vdash a$

Fig. 8. Run-time universe type system

	$u \rhd_{+} u'$	self	peer	rep	any
			u'		
u	self	self	peer	rep	any
	peer	peer	peer	any	any
	rep	rep	rep	any	any
	any	any	any	any	any

	$u \rhd_{-} u'$	self	peer	rep	any
			u'		
u	self	self	peer	rep	any
	peer	peer	peer	any	any
	rep	any	any	peer	any
	any	any	any	any	any



We extend to types by defining: $u \rhd_{+} (u'\ c) = (u \rhd_{+} u')\ c$ and $u \rhd_{-} (u'\ c) = (u \rhd_{-} u')\ c$

Fig. 9. Universe composition and decomposition

meant in respect to `this`. The ownership type qualifier `self` (a specialisation of `peer`) is used for the parameter `this`, e.g.

$$(\texttt{self Dept}, \_) \vdash \texttt{this.first.s} : \texttt{rep Student}$$
$$(\texttt{self Hall}, \_) \vdash \texttt{this.first.s} : \texttt{any Student}$$

The ownership type qualifier `self` has a special purpose – when calling local methods and accessing local fields, we want the type of such accesses to be exactly the annotation u given in the class. Note that $\texttt{self} \rhd u = u$. If we were to use `peer` instead of `self` as the type of `this`, then we would lose information in the case where $u = \texttt{rep}$ and the type system would be unnecessarily restrictive.

There is one aspect of this type system that deserves detailed discussion because we use it later. The purpose of $u \rhd u'$ is to determine the type that best describes an object that is "twice removed" from the observer by references of type u and u'. In other words, we have two subsequent references and two respective types, and we want to know the type of the most distant object from the observer. $(\rhd)$ is defined in Fig. 9. The object (1) in Fig. 1 observes the object (8) to be `peer`. (8) considers (9) to be `rep`, so (1) considers (9) to be $\texttt{peer} \rhd \texttt{rep} = \texttt{any}$.

We use $(\rhd)$ to 'translate' a type u' from one observer to another, where the old observer is u with respect to the new observer. This is useful for class member lookups where the type of the member is from the perspective of the object that contains it, but we want a type from the caller's perspective.

The complement of $(\rhd)$ is $(\triangleright)$, which we use to translate a type to another observer. The object (1) observes the objects $(2, 3)$ to be `rep`, however object (2) considers (3) to be $\texttt{rep} \triangleright \texttt{rep} = \texttt{peer}$.

Finally, we require classes to be well-formed. The types of method bodies must agree with their signatures. Note that when typing a method body, we use `self` in the type of `this`. This is consistent with our notion of observer for method bodies as described above. We also require consistency between field and method signatures in subclasses.

To prove soundness of this system, we need a type system for run-time expressions. This type system is capable of typing addresses using the owner stored in the heap, and typing variables `x` and `this` using the stack σ. The judgement is $h, \sigma \vdash e : t$, it is given in Fig. 8. In Appendix A we give several lemmas, including a substitution lemma mapping the static to the run-time type system. We finally give the soundness theorem for single threads and the multithreaded system, the proofs of which are in [7].

3 Race Safety

3.1 Static Types for Race Safety

In Fig. 10 we give a type system that requires correct synchronisation and thus guarantees race safety. The judgement $\mathbb{L}, \Gamma \vdash e : F$ denotes that the expression e is race free if all locks l in the *synchronisation set* $\mathbb{L}$ have been acquired for the duration of its execution. A lock l is either an ownership type qualifier $u \neq \texttt{any}$ or a path. The set F is the *effect* of e, *i.e.,* the set of fields that e may write to as it executes.

$$\frac{}{\emptyset, \Gamma \vdash \mathtt{x} : \emptyset}\,(\textsc{Var}) \qquad \frac{}{\emptyset, \Gamma \vdash \mathtt{this} : \emptyset}\,(\textsc{This}) \qquad \frac{}{\emptyset, \Gamma \vdash \mathtt{new}\ t : \emptyset}\,(\textsc{New})$$

$$\frac{}{\emptyset, \Gamma \vdash \mathtt{null} : \emptyset}\,(\textsc{Null}) \qquad \frac{\mathbb{L}, \Gamma \vdash e : F}{\mathbb{L}, \Gamma \vdash (t)\ e : F}\,(\textsc{Cast}) \qquad \frac{\emptyset, \Gamma \vdash e : \_}{\emptyset, \Gamma \vdash \mathtt{spawn}\ e : \emptyset}\,(\textsc{Spawn})$$

$$\frac{\begin{array}{l} \mathbb{L}', \Gamma \vdash e : F' \\ \mathbb{L}' \subseteq \mathbb{L} \qquad F' \subseteq F \\ \forall p \in \mathbb{L}. \mathbb{L}, \Gamma \vdash p : \_ \\ \mathbb{L} \mathrel{\#} F \end{array}}{\mathbb{L}, \Gamma \vdash e : F}\,(\textsc{Sub}) \qquad \frac{\begin{array}{l} \mathbb{L}, \Gamma \vdash e : F \\ \Gamma \vdash_{gb} e : l \\ \mathbb{L} \cup \{l\}, \Gamma \vdash e' : F \end{array}}{\mathbb{L}, \Gamma \vdash \mathtt{sync}\ e\ e' : F}\,(\textsc{Sync}) \qquad \frac{\begin{array}{l} \mathbb{L}, \Gamma \vdash e : F \\ \Gamma \vdash_{gb} e : l \\ l \in \mathbb{L} \end{array}}{\mathbb{L}, \Gamma \vdash e.f : F}\,(\textsc{Field})$$

$$\frac{\begin{array}{ll} \mathbb{L}, \Gamma \vdash e : F & \Gamma \vdash_{gb} e : l \\ \mathbb{L}, \Gamma \vdash e' : F & l \in \mathbb{L} \quad f \in F \end{array}}{\mathbb{L}, \Gamma \vdash e.f = e' : F}\,(\textsc{Assign}) \qquad \frac{\begin{array}{ll} \mathbb{L}, \Gamma \vdash e : F & \Gamma \vdash e : u\ c \\ \mathbb{L}, \Gamma \vdash e' : F & \mathcal{E}\mathit{ff}(c, m){\downarrow_2} \subseteq F \\ \mathbb{L}' \in \mathcal{E}\mathit{ff}(c, m){\downarrow_1} & (u, e, e') \rhd \mathbb{L}' \subseteq \mathbb{L} \end{array}}{\mathbb{L}, \Gamma \vdash e.m(e') : F}\,(\textsc{Call})$$

$$\frac{\begin{array}{l} \forall c' \geq c\ .\ F' = \mathcal{E}\mathit{ff}(c', m){\downarrow_2} \Longrightarrow \mathcal{E}\mathit{ff}(c, m){\downarrow_2} \subseteq F', \\ \qquad \mathbb{L}' \in \mathcal{E}\mathit{ff}(c', m){\downarrow_1} \Longrightarrow \exists \mathbb{L} \in \mathcal{E}\mathit{ff}(c, m){\downarrow_1}\ .\ \mathbb{L} \subseteq \mathbb{L}' \\ \forall \mathcal{M}(c, m) = t_r\ m(t_x), \mathbb{L} \in \mathcal{E}\mathit{ff}(c, m){\downarrow_1}\ . \\ \qquad \mathbb{L}, (\mathtt{self}\ c, t_x) \vdash \mathcal{MB}\mathit{ody}(c, m) : \mathcal{E}\mathit{ff}(c, m){\downarrow_2} \end{array}}{\vdash c}\,(\textsc{WFClass})$$

where ($\#$) and ($\rhd$) are defined below:
$\mathbb{L} \mathrel{\#} F \Longleftrightarrow \forall f \in F, p \in \mathbb{L}\ .\ f \notin p$

$$\frac{\begin{array}{l} \Gamma \vdash e : u\ \_ \\ u \neq \mathtt{any} \end{array}}{\Gamma \vdash_{gb} e : u}\,(\textsc{Univ}) \qquad \frac{}{\Gamma \vdash_{gb} p : p}\,(\textsc{Path})$$

$$\begin{array}{lll} (u, \_, \_) \rhd u' = u \rhd u' & \text{if } u \rhd u' \neq \mathtt{any} & (\text{undefined if } u \rhd u' = \mathtt{any}) \\ (\_, p, \_) \rhd p' = p'[p/\mathtt{this}] & \text{if } p' = \mathtt{this} \ldots & \\ (\_, \_, p) \rhd p' = p'[p/\mathtt{x}] & \text{if } p' = \mathtt{x} \ldots & \end{array}$$

$$(u, e, e') \rhd \mathbb{L} = \{\ (u, e, e') \rhd l \mid l \in \mathbb{L}\ \}$$

$$(\text{undefined if } (u, e, e') \rhd l \text{ is undefined for any } l \in \mathbb{L})$$

The function $\mathcal{E}\mathit{ff}$ returns pairs of sets of synchronisation sets and sets of fields, and paths are defined below (static paths do not contain addresses a):

$\mathcal{E}\mathit{ff} : (Id^c \times Id^m) \rightarrow (\mathbb{P}(\mathbb{P}(\mathbb{L})) \times \mathbb{P}(Id^f))$ $\qquad p ::= \mathtt{this} \mid \mathtt{x} \mid a \mid p.f$

Fig. 10. Static race safety type system

Well-typed expressions do not overwrite fields appearing in their synchronisation set, and their locks are prefix complete (e.g. $x.f \in \mathbb{L} \Rightarrow x \in \mathbb{L}$):

Lemma 3.1 *The effects of well-typed expressions do not undermine their locks.*

$$\mathbb{L}, \Gamma \vdash e : F \Longrightarrow \mathbb{L} \;\#\; F, \qquad \forall p \in \mathbb{L}. \mathbb{L}, \Gamma \vdash p : \_$$

Proof: induction on the derivation of $\mathbb{L}, \Gamma \vdash e : F$.

We say that an expression is *internally synchronised* if it can be typed with an empty synchronisation set, otherwise it is *externally synchronised*.

We now discuss the type system in greater detail. Suppose we have $\mathbb{L}, \Gamma \vdash e : \_$. The synchronisation set and effect of variables and constants are empty, *c.f.,* (Null), (Var), (This). This also holds for (New), as object creation does not interact with other threads. A cast does not require more locks or produce more effects than its sub-term, *c.f.,* (Cast). Spawning requires the new thread to be internally synchronised, and therefore requires its sub-term to have an empty synchronisation set. Since the sub-term is executed in a new thread, its effect is of no interest to the current thread, therefore the whole expression has empty effect, *c.f.,* (Spawn).

The (Sub) rule is a form of subsumption as it increases the effect and synchronisation set, provided that none of the fields in the new effects F appear in any of the paths of the new synchronisation set $\mathbb{L}$, thus preserving lemma 3.1.

The (Field) and (Assign) rules are similar. They calculate the lock l that guards the object access in question, using the *guarded by* judgement $\Gamma \vdash_{gb} e : l$. This lock must be acquired before the execution of the access in order to guarantee race safety, therefore l is included in the synchronisation set $\mathbb{L}$. The judgement finds the owner via the ownership type qualifier u provided that $u \neq$ `any` (Univ), or uses the path p when e is such a path (Path). If both rules are applicable, then the locks obtained will be syntactically different (e.g. `self` and `this`), but will indicate the same owner. [9]

The (Sync) rule calculates the synchronisation set for the expression `sync` $e\ e'$ by removing the lock that guards the object accessed by e from the synchronisation set of e'.

Because methods in our system are not necessarily internally synchronised, we extend their signatures through $\mathcal{E}\!f\!f$, whose shape is defined in Fig. 10, which returns a set of synchronisation sets, and a set of fields to which the method body may assign. The (Call) rule thus requires that these locks and assignments are included in the resulting synchronisation set and effects F. [10] [11] Because the synchronisation sets expressed in $\mathcal{E}\!f\!f$ are given from the perspective of the target of the method call, they need to be translated into the perspective of the receiver before being used. This is done through the operaor $\trianglerighteq$, defined in Fig. 10.

Well-formed classes, *c.f.,* (WFClass), requires, in addition to the requirements imposed for universe type soundness, that: Firstly, if a method m existed in a

[9] Obviously, if neither rule is applicable the expression is type incorrect.

[10] The reason we use a *set* of synchronisation sets, rather than one single synchronisation set, is, that there may be more than one way to correctly synchronise a method call. E.g. a method with body `this.f.s = x` might have $\mathcal{E}\!f\!f$ as follows: ({{`this`, `this.f`}, {`this`, `rep`}, {`self`, `this.f`}, {`self`, `rep`}}, {`s`}).

[11] The synchronisation sets will grow with the number of accesses in a method body. Therefore, in practice we would need a better syntax that scales more favourably, e.g. `this`|`self`, `this.first`|`rep`. This is outside the scope of the current paper.

superclass, then the superclass's synchronisation sets and effects should be larger than those in the subclass. Secondly, each of synchronisation sets $\mathbb{L}$ in $\mathcal{E}\!f\!f(c,m)\downarrow_1$ should be sufficient for correct synchronisation of the body of m.

3.1.1 Examples

We now discuss the application of our type rules with some examples. We first consider the body of `releaseMarks` in Fig. 2. We have an environment Γ_1 where $\Gamma_1(\texttt{this}) = \texttt{self Dept}$. Because our tiny language does not include local variables, we will consider `i` as a field in class `Dept` of type `rep DeptStudentNode`, and mentally map each appearance of `i` in Fig. 2, onto `this.i`. Thus,

$$\begin{array}{l}\emptyset, \Gamma_1 \vdash \texttt{sync (this) \{ this.i} = \texttt{this.first;} \\ \qquad\qquad\qquad\qquad \texttt{sync (this.}i\texttt{) \{this.i.s.mark} = \ldots; \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{this.i} = \texttt{this.i.next\} \}} : \{\texttt{i}, \texttt{mark}\}\end{array}$$

On the other hand, in Fig. 2, line 31, we obtain the following with an environment Γ_2 where $\Gamma_2(\texttt{this}) = \texttt{self Hall}$:

$$\begin{array}{l}\{\texttt{this}, \texttt{this.i}\}, \Gamma_2 \vdash \texttt{sync (this.i.s) \{ this.i.s.roomClean = ...;} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{this.i = ... \}} : \{\texttt{roomClean}, \texttt{i}\}\end{array}$$

Because `this.i.s` has type `any Student`, the type system can only use the guard rule (PATH). The type system accepts the above synchronised block because we are not assigning to the fields `i` or `s` within the block. However, the expression `sync(this.i) {... this.i = this.i.next }` would be type incorrect in Γ_2, and thus we are forced to lock at every loop iteration.

The method `badCode()` in Fig. 11 accesses and synchronises `this.getFirst()` (line (9)). This is not a path, but has type `rep DeptStudentNode` so the type system can use (UNIV) rule to accept the code. The `sync (this.first2)` block (line (12)) fails type checking because the path being locked is `any` and also comprises a field `first2` which is assigned during the synchronised block. The final access (line (17)) is not a path and has type `any`, so no amount of synchronisation will persuade the type system to accept it.

Finally, we give examples of method calls. Assume a method `clean()` [12] in class `Student` such that:

$$\mathcal{E}\!f\!f(\texttt{Student}, \texttt{clean}) = (\{\{\texttt{self}\}\}, \{\texttt{cleanRoom}\})$$

Then, in class `Dept`, it holds that $\Gamma_1 \vdash \texttt{this.first.s} : \texttt{rep Student}$. Thus, by application of (CALL), we obtain:

$$\{\texttt{rep}\}, \Gamma_1 \vdash \texttt{this.first.s.clean()} : \{\texttt{cleanRoom}\}$$

In class `Hall`, $\Gamma_2 \vdash \texttt{this.first.s} : \texttt{any Student}$. Here, the call `this.first.s.clean()` causes a type error. On the other hand, with a method `makeBed()`, where:

$$\mathcal{E}\!f\!f(\texttt{Student}, \texttt{makeBed}) = (\{\{\texttt{self}\}, \{\texttt{this}\}\}, \{\texttt{cleanRoom}\})$$

[12] For simplicity, we ignore method parameters.

```
class BrokenDept extends Dept {
    any Student first2;
    any DeptStudentNode getFirst2() {
        sync (this) { return this.first2; }
    }
    rep DeptStudentNode getFirst() {
        sync (this) { return this.first; }
    }
    void badCode() {
        sync (this) {
            sync (this.getFirst()) {
                this.getFirst().s = NULL;
            }
            sync (this.first2) { // FAIL
                this.first2 = this.first2.next;
                this.first2.s = NULL;
            }
            sync (???) {
                this.getFirst2().s = NULL; //FAIL
}   }   }   }
```

Fig. 11. Example

We would obtain

$$\mathbb{L}, \Gamma_2 \vdash \texttt{this.first.s.makeBed()} : \{\texttt{cleanRoom}\}$$
$$(\text{where } \mathbb{L} = \{\texttt{this}, \texttt{this.first}, \texttt{this.first.s}\})$$

3.2 Run-time Type System

As is standard, we give a run-time type system in order to prove soundness and race safety (presented in Fig. 12). We type run-time expressions e according to a heap h and stack σ. The judgement has the shape $\mathbb{L}, h, \sigma \vdash e : F$. The meanings of $\mathbb{L}$ and F are unchanged. The function $h(\sigma, p)$ executes p in the given heap and stack to retrieve a value in a finite number of steps bounded by the size of p. If this process attempts to dereference `null`, we define it to return `null`. In Fig. 10 we used (Path) and (Univ) to derive locks from expressions. We needed to extend this functionality to derive locks from partially executed expressions so we added the rule (Val) and replaced (Path) by the rules (Var) and (Field). (Univ) was changed to use the run-time universe type system, which understands partially executed expresions. (Call) needed us to extend ($\triangleright$) to translate locks in the context of partially executed targets and arguments.

The type system is lifted to the sequence of threads that ultimately comprises our model of the run-time state by (Threads). We require all the threads to be internally synchronised and also that no two threads have the same lock. The shape of the judgement is $h, \sigma \vdash \overline{e}$.

We use the predicate $Virgin(e)$ to note that e has not yet been executed *i.e.,* contains no addresses, `synced` or `frame` constructs. $Reachable(e)$ (Fig. 13)

$$\frac{h(\sigma,p) = v}{h,\sigma \vdash_{gb} v : p}\,(\textsc{Val}) \qquad \frac{p \in \{\mathtt{x},\mathtt{this}\}}{h,\sigma \vdash_{gb} p : p}\,(\textsc{Var}) \qquad \frac{h,\sigma \vdash_{gb} p : p'}{h,\sigma \vdash_{gb} p.f : p'.f}\,(\textsc{Field})$$

$$\frac{\begin{array}{ll} \mathbb{L},h,\sigma \vdash e : F & \mathbb{L}' \in \mathit{Eff}(c,m)\downarrow_1 \\ \mathbb{L},h,\sigma \vdash e' : F & \mathit{Eff}(c,m)\downarrow_2 \subseteq F \\ h,\sigma \vdash e : u\ c & (h,\sigma,u,e,e') \trianglerighteq \mathbb{L}' \subseteq \mathbb{L} \end{array}}{\mathbb{L},h,\sigma \vdash e.m(e') : F}\,(\textsc{Call}) \qquad \frac{\begin{array}{l} \mathbb{L}',h,\sigma' \vdash e : F \\ \sigma' = (a,v) \quad h,\sigma \vdash a : u\ \_ \\ \mathbb{L} = (h,\sigma,u,a,v) \trianglerighteq \mathbb{L}' \end{array}}{\mathbb{L},h,\sigma \vdash \mathtt{frame}\ \sigma'\ e : F}\,(\textsc{Frame})$$

$$\frac{\begin{array}{ll} h,\sigma \vdash_{gb} e' : l & \mathbb{L},h,\sigma \vdash e' : F \\ h,\sigma \vdash_{gb} e'' : l & \mathbb{L},h,\sigma \vdash e'' : F \\ \mathbb{L} \cup \{\ l\ \},h,\sigma \vdash e : F & \end{array}}{\mathbb{L},h,\sigma \vdash \mathtt{sync}_{e''}\ e'\ e : F}\,(\textsc{Sync}) \qquad \frac{\begin{array}{ll} h,\sigma \vdash_{gb} a : l & h(a)\downarrow_1 = w \\ h,\sigma \vdash_{gb} e' : l & \mathbb{L},h,\sigma \vdash e' : F \\ \mathbb{L} \cup \{\ l\ \},h,\sigma \vdash e : F & \end{array}}{\mathbb{L},h,\sigma \vdash \mathtt{synced}_{e'}\ w\ e : F}\,(\textsc{Synced})$$

$$\frac{}{\emptyset,h,\sigma \vdash a : \emptyset}\,(\textsc{Addr})$$

(Var) (This) (Null) (Sub) (Cast) (New) (Field) (Assign) (Spawn) (Univ) are the same as in Fig. 10 but with Γ replaced by h,σ

$$\frac{\begin{array}{l} \forall i \in \{1..n\}\ .\ \emptyset,h,\sigma \vdash e_i : \_ \\ \forall i,j \in \{1..n\},w\ .\ \mathit{Locked}(e_i,w) \wedge \mathit{Locked}(e_j,w) \implies i = j \end{array}}{h,\sigma \vdash e_{1..n}}\,(\textsc{Threads})$$

$$\begin{array}{rcll} (\_,\_,u,\_,\_) \trianglerighteq u' & = & u \trianglerighteq u' & \text{if } u \trianglerighteq u' \neq \mathtt{any} (\text{undefined if } u \trianglerighteq u' = \mathtt{any}) \\ (h,\sigma,\_,e,\_) \trianglerighteq p' & = & p'[p/\mathtt{this}] & \text{where } h,\sigma \vdash_{gb} e : p, \quad p' = \mathtt{this}\ldots \\ (h,\sigma,\_,\_,e) \trianglerighteq p' & = & p'[p/\mathtt{x}] & \text{where } h,\sigma \vdash_{gb} e : p, \quad p' = \mathtt{x}\ldots \\ (h,\sigma,u,e,e') \trianglerighteq \mathbb{L} & = & \{\ (h,\sigma,u,e,e') \trianglerighteq l \mid l \in \mathbb{L}\ \} & \\ & & \multicolumn{2}{r}{(\text{undefined if } (h,\sigma,u,e,e') \trianglerighteq l \text{ is undefined for any } l \in \mathbb{L})} \end{array}$$

Fig. 12. Run-time race safety type system

denotes that the subterms of e have been executed in the right order, *e.g.,* $\mathit{Reachable}(a.f = y.f)$ but $\neg\mathit{Reachable}(y.f = a.f)$. We extend this to sequences of expressions $\mathit{Reachable}(\overline{e})$ if all the expressions are reachable.

Because of the instrumentation of `sync` with a subscript that records the initial lock expression, we need to use the same substitution as used in the semantics rule (Call) when defining the following substitution lemma:

$$\begin{array}{ll}
\mathit{Reachable}(e) & \Longleftarrow e \in \{\mathtt{x}, \mathtt{this}, v, \mathtt{new}\ t\} \\
\mathit{Reachable}(e.f) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}((t)e) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}(e_1.f = e_2) & \Longleftarrow \mathit{Reachable}(e_1) \wedge \mathit{Virgin}(e_2) \\
\mathit{Reachable}(v.f = e) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}(e_1.m(e_2)) & \Longleftarrow \mathit{Reachable}(e_1) \wedge \mathit{Virgin}(e_2) \\
\mathit{Reachable}(v.m(e)) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}(\mathtt{spawn}\ e) & \Longleftarrow \mathit{Virgin}(e) \\
\mathit{Reachable}(\mathtt{sync}_{e_1}\ e_2\ e_3) & \Longleftarrow \mathit{Virgin}(e_1) \wedge \mathit{Reachable}(e_2) \wedge \mathit{Virgin}(e_3) \\
\mathit{Reachable}(\mathtt{synced}_{e_1}\ w\ e_2) & \Longleftarrow \mathit{Virgin}(e_1) \wedge \mathit{Reachable}(e_2) \\
\mathit{Reachable}(\mathtt{frame}\ \sigma\ e) & \Longleftarrow \mathit{Reachable}(e)
\end{array}$$

Fig. 13. Definition of *Reachable*

Lemma 3.2 *Static race safety implies run-time race safety*

$$\left.\begin{array}{l}
\mathbb{L}, \Gamma \vdash e : F \\
h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \\
h, \sigma \vdash \mathtt{this} : \Gamma(\mathtt{this})
\end{array}\right\} \Longrightarrow \begin{array}{l}
\mathbb{L}, h, \sigma \vdash S(e) : F \\
\mathit{Virgin}(S(e))
\end{array}$$

Proof: Induction over derivation of $\mathbb{L}, \Gamma \vdash e : F$

Using the above lemma in the case of method calls, it is possible to prove the soundness of the race safety type system. Firstly we state soundness for single-threaded execution. Note that we require the heap to be well-formed. This is necessary so that field accesses yield objects of the correct owner. We give some lemmas that lead to this result in Appendix B.

Lemma 3.3 *The type of a thread is preserved over the execution of that thread.*

$$\left.\begin{array}{l}
\mathit{Reachable}(e) \\
\vdash h \\
h, \sigma \vdash e : t \\
\mathbb{L}, h, \sigma \vdash e : F \\
\sigma \vdash e, h \leadsto e', h'
\end{array}\right\} \Longrightarrow \begin{array}{l}
\mathbb{L}, h', \sigma \vdash e' : F \\
\mathit{Reachable}(e')
\end{array}$$

Proof: Induction over derivation of $\mathbb{L}, h, \sigma \vdash e : F$

The soundness of the complete type system can now be stated:

Theorem 3.4 *Well-typedness of the system is preserved over execution.*

$$\left.\begin{array}{l} \vdash h \\ h, \sigma \vdash \overline{e} : \overline{t} \\ h, \sigma \vdash \overline{e} \\ \sigma \vdash \overline{e}, h \rightsquigarrow \overline{e}', h' \\ \mathit{Reachable}(\overline{e}) \end{array}\right\} \Longrightarrow \begin{array}{l} h', \sigma \vdash \overline{e}' \\ \mathit{Reachable}(\overline{e}') \end{array}$$

Proof: Case analysis of $\sigma \vdash \overline{e}, h \rightsquigarrow \overline{e}', h'$

Now we work towards a theorem of race safety, i.e. that well-typed programs can exhibit no race conditions. Firstly the following lemma states that objects are only accessed if the appropriate lock has been acquired by the thread in question. Note the use of the action a to denote an access of address a by the execution step.

Lemma 3.5 *Objects are only accessed while their owners are locked.*

$$\left.\begin{array}{l} \mathbb{L}, h, \sigma \vdash e : \_ \\ \sigma \vdash e, h \overset{a}{\rightsquigarrow} \_, \_ \end{array}\right\} \Longrightarrow \begin{array}{l} (\exists l \in \mathbb{L}. h, \sigma \vdash_{gb} a : l) \vee \\ \mathit{Locked}(e, h(a)\downarrow_1) \end{array}$$

Proof: Induction over structure of $\mathbb{L}, h, \sigma \vdash e : \_$.

The multi-threaded case follows. We are dealing with entire threads here (as opposed to sub-terms), so we do not need the set $\mathbb{L}$ of locks taken in the current context.

Theorem 3.6 *Objects are only accessed while their owners are locked by the corresponding thread.*

$$\left.\begin{array}{l} h, \sigma \vdash e_{1..n} \\ \sigma \vdash e_{1..n}, h \overset{(i,a)}{\rightsquigarrow} \_, \_ \end{array}\right\} \Longrightarrow \mathit{Locked}(e_i, h(a)\downarrow_1)$$

Proof: Case analysis of $\sigma \vdash e_{1..n}, h \overset{(i,a)}{\rightsquigarrow} \_, \_$

For race conditions, we use the definition from [10], where the state of a multi-threaded system exhibits an *instantaneous race condition* if the semantics allows two possible execution steps of different threads, that both access the same object. The required non-determinism is provided by the (Interleave) rule.

We prove that no well-typed state can ever have an instantaneous race condition, and by theorem 3.4, all intermediate states of execution will be free from instantaneous race conditions. We show, for an arbitrary well-typed run-time state, that if two possible execution steps can access the same object, then those steps must be steps of the same thread:

Theorem 3.7 *Race safety*

$$\left.\begin{array}{l} h, \sigma \vdash \overline{e} \\ \sigma \vdash \overline{e}, h \overset{(i,a)}{\rightsquigarrow} \_, \_ \\ \sigma \vdash \overline{e}, h \overset{(j,a)}{\rightsquigarrow} \_, \_ \end{array}\right\} \Longrightarrow i = j$$

Proof: Application of Theorem 3.6 and (Threads).

4 Implementation Issues

We hope to implement our type system, and have considered ways to achieve good performance within our semantics.

The constraints on extending classes may seem severe, particularly the requirement that an overriding method cannot have more effects F than the original method. However, if we forsake separate compilation, we can infer the effect F of each method, and thus we do not need to restrict inheritance. We believe the constraints on the synchronisation set required for a call to be race safe are not so severe because most functions will be internally synchronised, whereas one cannot hide field assignments F within a function. In general separate compilation does not mix well with concurrency since concurrency is concerned with the effect of the rest of the program. Separate compliation requires specification at module boundaries, and a behaviour specification is quite hard to write and maintain. In our work such a specification is represented by the sets $\mathbb{L}$ and F.

In practice, we could use more accurate techniques of alias detection, *e.g.,* using a points-to analysis, to refine the type system's judgement about whether a field assignment affects a path p in $\mathbb{L}$. This would allow us to reject fewer correct programs. We could easily distinguish between identically named fields in different classes by prepending the class name to all fields.

It may be useful to use the method of [4] for preventing deadlock. We can require the programmer to write additional type annotations to firstly divide the heap into a statically bounded set of regions [24] and secondly specify a partial order over these regions. Types are therefore augmented with a region identifier, which specifies the region where the owner of the object lies (locks are associated with the owner). The type system would check that the specified order has no cycles, and that assignments do not let variables of a certain region reference objects of other regions.

To actually prevent deadlocks, the type system has to ensure that locks are taken (i.e. sync blocks are nested) in the order specified. Method signatures can be annotated with the lowest lock that they take, and thus each call can be checked to make sure its context has not already locked any higher than the method will lock.

Since we require an implicit owner field in all objects, for casts and synchronisation of `any` expressions, there may unnecessary memory overhead [13]. If this becomes an issue, we propose introducing new types to lay alongside `rep`, `peer`,

[13] In some cases, previous work required the programmer to use an explicit owner field, see Fig. 3

and `self` which would have identical semantics except that they may not be cast to `any`. Objects of these types would not require an owner field since the owner would always be statically known. We anticipate that programmers would use these types only when memory use was a problem. Static analysis could also be used to infer which objects are never cast to `any`, and optimise away the owner field.

Other type systems [2,12,14,3,4] have extra features such as thread local storage and final variables. We did not formalise them, as although they are useful in practice, they are well-understood and can be easily added to an implementation.

4.1 Distinguishing Reads and Writes

We consider two simultaneous accesses of the same object by different threads to be a race condition, but this need only be so if one of the accesses is a write. Distinguishing between reads and writes would allow more liberal synchronisation. We now show how this can be done with an extension of our formalism.

We need to distinguish between read and write accesses, i.e. distinguish between field lookups and field assignments. We also need read/write locks, i.e. we need a pair of constructs like $\texttt{sync}^R(...)$ and $\texttt{sync}^W(...)$ that attempt to acquire the read/write lock on an object, respectively. The semantics would allow multiple threads to have the read lock at any one time, so long as no other thread was writing. This would be reflected in the rule (Threads) which would ensure that if a thread is writing then no other thread can write. These locks have already been implemented in Java, and are easy to add to the type system.

4.2 Single Object Locking

While we need to lock the whole ownership domain when iterating through nodes as in Fig. 2 (line 21), we do not need to lock all the peers of an object if we do not use it for iteration. E.g. we do not need to lock the peers of `this` when we access `this.first`. We'd like to give the programmer the choice of when to lock the whole domain (as is currently available with `sync`) and when to lock only a single object. Then, it would be possible for two threads to execute in parallel the `releaseMarks` method of the two departments in the diagram of Fig. 1. We may extend the type system and semantics as shown in Fig. 14.

Programmers use `syncobj` when they only want to lock the object in question e.g. Fig. 2 (line 19), and the old `sync` construct if they want to lock that object and all of its peers. We do not let a thread lock a whole domain if any of the objects in that domain have been locked. Likewise, we do not let a thread lock a single object if another thread has locked that object's domain. We use the old predicate $Locked(e, w)$ but we also add $LockedObj(e, a)$ which holds when e contains `syncedobj_` a _.

We augment the lock syntax so the whole-domain locks are now denoted with $(*, u)$ or $(*, p)$, whereas the single object locks are denoted with $(1, p)$. We do not need $(1, u)$, as this would describe a whole domain. We updated the (Univ) and (Path) rules to wrap their result in $(*, \ldots)$ and give a new guard rule that returns $(1, p)$ if the expression is a path. The type rule for threads guarantees that no two threads have conflicting locks.

$$\frac{\mathbb{L}, \Gamma \vdash e : F \qquad \Gamma \vdash_{gb} e : l = (*, \_) \qquad \mathbb{L} \cup \{l\}, \Gamma \vdash e' : F}{\mathbb{L}, \Gamma \vdash \texttt{sync}\ e\ e' : F} \ (\text{Sync})$$

$$\frac{\mathbb{L}, \Gamma \vdash e : F \qquad \Gamma \vdash_{gb} e : l = (1, \_) \qquad \mathbb{L} \cup \{l\}, \Gamma \vdash e' : F}{\mathbb{L}, \Gamma \vdash \texttt{syncobj}\ e\ e' : F} \ (\text{SyncObj})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{sync}_{e'}\ a\ e] \qquad w = h(a)\downarrow_1 \\ \forall j \in \{1..n\}\ .\ \mathit{Locked}(e_j, w) \Longrightarrow i = j \\ \qquad \forall b\ .\ (h(b)\downarrow_1 = w \wedge \mathit{LockedObj}(e_j, b)) \Longrightarrow i = j \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[\texttt{synced}_{e'}\ w\ e]\ e_{i+1..n}, h} \ (\text{LockDomain})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{syncobj}_{e'}\ a\ e] \\ \forall j \in \{1..n\}\ .\ \mathit{Locked}(e_j, h(a)\downarrow_1) \Longrightarrow i = j \\ \qquad \mathit{LockedObj}(e_j, a) \Longrightarrow i = j \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[\texttt{syncedobj}_{e'}\ a\ e]\ e_{i+1..n}, h} \ (\text{LockObj})$$

Source syntax:	Runtime syntax:
$e ::= ...\ \mid \texttt{syncobj}\ e\ e$	$e ::= ...\ \mid \texttt{syncobj}_e\ e\ e \mid \texttt{syncedobj}_e\ a\ e$

$$\frac{\begin{array}{l} \forall i \in \{1..n\}\ .\ \emptyset, h, \sigma \vdash e_i : \_ \\ \forall i, j \in \{1..n\}, h(a)\downarrow_1 = w\ . \\ \qquad \mathit{Locked}(e_i, w) \wedge \mathit{Locked}(e_j, w) \Longrightarrow i = j \\ \qquad \mathit{Locked}(e_i, w) \wedge \mathit{LockedObj}(e_j, a) \Longrightarrow i = j \\ \qquad \mathit{LockedObj}(e_i, a) \wedge \mathit{LockedObj}(e_j, a) \Longrightarrow i = j \end{array}}{h, \sigma \vdash e_{1..n}} \ (\text{Threads})$$

Fig. 14. Single object locking

An efficient implementation might use a counter in the owner of an object to record how many of its child objects have been individually locked, rather than iterating through them all. Lock-free programming techniques can be used to ensure this has negligible performance cost compared to a standard mutex implementation. There has been a lot of research [1] into increasing the performance of lock operations in the most common cases. We have implemented a proof-of-concept prototype of a single object lock. It lacks features such as re-entrancy and readers/writers but early tests show that it performs only a few percent worse than a `java.util.concurrent.locks.ReentrantLock`. The performance can be further improved by refining our naive implementation to use ideas from previous work.

We believe single object locking (in the context of static race safety checkers) is a novel idea. None of the prior work supports it. It is a small extension of our formalism, and as such we did not incorporate it into our proofs; however, in practice it should allow more parallelism without much linear cost.

4.3 Atomicity

As described by Flanagan [15], there are program misbehaviours due to thread interactions that are not classed as race conditions by the standard definition [14]. Ideally, we would like to identify *atomicity violations* [13,14,15] in programs, which would include bugs such as stale value errors that we currently cannot detect. In fact, a program that does not type in our system, can be "corrected" by wrapping each access in a tiny synchronised block, thus converting all its race conditions to stale value errors.

Atomicity checking basically requires that non-redundant `sync` blocks are always nested (never composed) in an atomic block. As noted elsewhere [27], checking atomicity relies on checking race safety, so it is natural to provide this atomicity checking as an extension to our basic system.

We require the programmer to specify that certain blocks of code are intended to be atomic with the syntax `atomic` e. The type system will ensure that the execution of these blocks in the context of arbitrary race-free threads will be equivalent to a serialised execution with no interleaving of other threads. Our extension is a strengthening of our existing type system.

We propose the additional judgement $\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F$ which indicates what locks need to be taken to guarantee an expression is atomic. $\mathbb{A}$ and $\mathbb{B}$ are sets of locks just like $\mathbb{L}$, and are supplied for each method by the programmer (we leave inference as further work). F is the set of accessed fields we used previously. Object accesses generate locks in $\mathbb{A}$ and $\mathbb{B}$ like they have previously in $\mathbb{L}$. Locks are not eliminated from $\mathbb{B}$ by (Sync). $\mathbb{A}$ is the set of locks that need to be taken before e will be atomic. $\mathbb{B}$ is the set of locks required for an expression to be a *both mover* [15]. If e is atomic and e' is a both mover (or vice versa), then $e; e'$ is atomic, however $e; e'$ is only a both mover if both e and e' are both movers. The value chosen for $\mathbb{A}$ is illustrated by the rule for a sequential composition operator (;), if one was added to the model:

$$\frac{\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F \qquad \mathbb{A}', \mathbb{B}', \Gamma \vdash e' : F \qquad \mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}}{\mathbb{A}'', \mathbb{B} \cup \mathbb{B}', \Gamma \vdash e; e' : F} \ (\textsc{Seq})$$

For $e; e'$ to be atomic, we require enough locks so that e is atomic and e' is a both mover, or vice versa. For $e; e'$ to be a both mover we need to take all the locks generated by their accesses, regardless of the synchronisation present in e and e'. The idiom $\mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}$ is used where one sub-term is executed after another, and is used in (Assign), (Sync), and also in (Call) which needs to consider the body of the method and thus involves 3 sets. The full system is presented in Fig. 15. We include $\mathbb{A}$ and $\mathbb{B}$ in method type signatures, which we model with $\mathcal{E}\!f\!f(c, m)\downarrow_3$.

[14] Programmers often use "race condition" to describe these errors as well.

$$\frac{\_,\_,\Gamma \vdash e : \_}{\emptyset,\{\bot\},\Gamma \vdash \texttt{spawn}\ e : \emptyset}\ (\textsc{Spawn})$$

$$\frac{e \in \{\texttt{null},\texttt{x},\texttt{this},\texttt{new}\ t\}}{\emptyset,\emptyset,\Gamma \vdash e : \emptyset}\ (\textsc{Triv})$$

$$\frac{\begin{array}{ll}\mathbb{A},\mathbb{B},\Gamma \vdash e : F & \Gamma \vdash_{gb} e : l \\ \mathbb{A}' \cup \{l\},\mathbb{B}',\Gamma \vdash e' : F & l \in \mathbb{B}' \\ \mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\} & \end{array}}{\mathbb{A}'',\mathbb{B} \cup \mathbb{B}',\Gamma \vdash \texttt{sync}\ e\ e' : F}\ (\textsc{Sync})$$

$$\frac{\begin{array}{ll}\mathbb{A}',\mathbb{B}',\Gamma \vdash e : F' & \\ \mathbb{A}' \subseteq \mathbb{A}, & \mathbb{B}' \subseteq \mathbb{B}, \\ F' \subseteq F, & \mathbb{A},\mathbb{B} \mathrel{\#} F\end{array}}{\mathbb{A},\mathbb{B},\Gamma \vdash e : F}\ (\textsc{Sub})$$

$$\frac{\begin{array}{lll}\mathbb{A},\mathbb{B},\Gamma \vdash e : F & & \Gamma \vdash_{gb} e : l \\ \mathbb{A}',\mathbb{B}',\Gamma \vdash e' : F & l \in \mathbb{A}'',\mathbb{B}'' & f \in F \\ \mathbb{B}'' = \mathbb{B} \cup \mathbb{B}' & \multicolumn{2}{l}{\mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}}\end{array}}{\mathbb{A}'',\mathbb{B}'',\Gamma \vdash e.f = e' : F}\ (\textsc{Assign})$$

$$\frac{\mathbb{A},\mathbb{B},\Gamma \vdash e : F}{\mathbb{A},\mathbb{B},\Gamma \vdash (t)\ e : F}\ (\textsc{Cast})$$

$$\frac{\begin{array}{l}\mathbb{A},\mathbb{B},\Gamma \vdash e : F \\ \Gamma \vdash_{gb} e : l \\ l \in \mathbb{A},\mathbb{B}\end{array}}{\mathbb{A},\mathbb{B},\Gamma \vdash e.f : F}\ (\textsc{Field})$$

$$\frac{\begin{array}{ll}\mathbb{A}_1,\mathbb{B}_1,\Gamma \vdash e_1 : F & \mathbb{A}_2,\mathbb{B}_2,\Gamma \vdash e_2 : F \\ \Gamma \vdash e_1 : u\ c & (\mathbb{A}_3,\mathbb{B}_3) \in \mathit{Eff}(c,m){\downarrow_3} \\ \mathbb{B}_4 = (u,e_1,e_2) \triangleright \mathbb{B}_3 & \mathit{Eff}(c,m){\downarrow_2} \subseteq F \\ \mathbb{A}_4 = (u,e_1,e_2) \triangleright \mathbb{A}_3 & \mathbb{B}_4 \subseteq \mathbb{B}_1 \cup \mathbb{B}_2 \\ \multicolumn{2}{l}{\mathbb{A}_5 \in \{\mathbb{A}_1 \cup \mathbb{B}_2 \cup \mathbb{B}_4,\ \mathbb{B}_1 \cup \mathbb{A}_2 \cup \mathbb{B}_4,} \\ \multicolumn{2}{r}{\mathbb{B}_1 \cup \mathbb{B}_2 \cup \mathbb{A}_4\}}\end{array}}{\mathbb{A}_5,\mathbb{B}_1 \cup \mathbb{B}_2,\Gamma \vdash e_1.m(e_2) : F}\ (\textsc{Call})$$

Fig. 15. Static atomicity type system

If a new thread is spawned part-way through an atomic block e, it could take a lock not yet taken by e, and see state that should not have been visible until after e had completed. Therefore, the execution was not atomic according to the theory of reduction [23]. We prevent this by requiring the lock $\bot$ for `spawn` to be a both-mover. Such a lock is impossible to acquire, and thus `spawn` is never a both mover. This means there can be only one `spawn` in an atomic block, in the centre of the nesting of `sync` blocks. If we were to distinguish between the locks required for an expression to be a left mover and right mover, as in [15] (we currently treat such expressions as atomic), we could let spawn be a left mover. This would allow the spawning of any number of threads after the acquisition of locks in an atomic section.

5 Related Work

Previous race safety work can be divided into two categories. There are those that use ownership or guard annotations to specify the locking discipline as a relationship between objects, and there are those that use a finite set of programmer-supplied or inferred region names, and specify the locking discipline as a relationship between objects and regions.

The latter work [2,18,30] results in fewer annotations since regions are easier to infer than ownership types. However, they have the disadvantage that the set of locks is finite, and thus the program does not scale as well to many threads. This was noted in [25,18]. The more recent papers [18,30] have been able to infer the synchronisation completely. For us this would mean being able to infer the set $\mathbb{L}$, which is desirable, and we will attempt it as further work.

Of the former variety, the first work to exclude race conditions from object-oriented programs using a static type system was [10], using the concurrent object calculus. This idea was subsequently refined to more concrete models of object-oriented languages [11], which included parameterised classes and even some degree of inference [12]. These papers were variations on the same approach: The programmer supplied guard annotations in their classes; the guard annotations were a form of ownership types; this is clear through their use of final expressions and parameters, although individual fields were owned instead of entire objects. Similar results were also obtained using ownership types directly [3,4]. Our work complements these approaches by discussing a different kind of ownership type system (universes) and its application to race safety. Although it is interesting to see how static race safety can be achieved using universes, our major contributions are increased expressiveness and greater concurrency.

There is a substantial difference between our work and that just discussed; we allow paths of non-final field types (in fact our formalism does not have the `final` type qualifier) whereas [11,3] require paths to be constructed from only final field dereferences. The price we pay for forsaking this restriction is that we must find another way to ensure that the meaning of paths is not affected by the side-effects of the lock expression. For this we use a system of effects, and for this to work we require the effect of overriding methods to be restricted to that of the method they override. A combination of our approach and that of [11,3]would achieve the best of both worlds.

Effects are also used to prove preservation of properties of ownership type system in [29]. A concept similar to universes was studied in conjunction with synchronisation in [20]. This was mainly for the purpose of verifying object invariants rather than absence of race conditions. Objects can “change hands” over time, therefore their owners are not constant at run-time. Also, there is no concept of `peer`.

Locking an object in our system does not lock the whole tree as in [20,3,4]. We lock only the immediate ownership domain and further lock acquisitions are required if deeper objects are accessed. With more locks, we reduce contention and let more threads execute in parallel than [3,4].

6 Further Work

Universes may seem under-powered because they cannot express the relationship between objects when they are neither `rep` nor `peer`. However we expect that using generics [9], we will have a lot more power, e.g. when we parameterise a list to hold elements of a particular domain. We expect the extension to include generics to be straightforward.

We would also like to use *path dependent types*, which are currently being studied in the context of universes. These would allow us to use types like $\mathtt{x}.f.\mathtt{rep}$, which characterise the objects owned by $\mathtt{x}.f$, greatly increasing our expressiveness.

We soon hope to implement our semantics and type system and try it out by adapting existing concurrent programs.

6.1 Inferring Locks

Recent work [18,30] has investigated the complete inference of synchronisation, *i.e.,* inferring not only the fields F but the set $\mathbb{L}$ as well. This would lead to systems where the programmer does not write any locking code (only using `atomic`); such systems would have all the benefits of software transactional memory [17]. We doubt we will be able to infer ownership annotations, but perhaps we can drop back to a finite set of inferred regions if ownership annotations are not available. Thus, the programmer can choose to add ownership annotations if more parallelism is required.

6.2 STMBench7

We have considered the applicability of our system (compared to the related work), on a benchmark suite [16]. This is a complex datastructure, a tree with a graph at each leaf, with operations that scan, search, and access single elements. It is demanding to synchronise efficiently, correctly, and without introducing deadlocks. The language the authors used did not check their synchronisation. They presented two locking disciplines, the simpler of which uses a single lock for the whole datastructure. In our system this is achieved by making everything `peer`, although with single object locking, many operations that access single elements can run in parallel. Their more complex locking discipline relies on the number of levels in the tree being statically bounded (they used 6 levels). It associates a lock for each level, and a lock for all the leaf graphs.

We found that their example was hard to type-check in our system or the previous work. Firstly, one cannot parameterise the `Node` class to specify the next level:

```
class Node <ourLock,below> {
        Node <below,???> left, right;
}
```

If we do not use parameters, but use a final field to hold the lock (in our system this means the fields `left` and `right` are `any`), then one must lock each node individually. It would not be possible to lock the whole tree before a scan. Locking each node during the scan would cause deadlock unless all the scans were top down or bottom up. One can work around this by defining a class for each level; however, this does not permit iteration through the layers.

The problem is that using field annotations to specify the locking discipline does not work well when the latter is disjoint from the structure of the data. The natural locking discipline is to exploit the hierarchy of the tree, using `rep` for each `left` and `right` field in the node, but we cannot scan the tree from the bottom-up in this case, as it might deadlock with concurrent top-down scans. Even considering each leaf graph in its own ownership domain is problematic when iterating over them. We leave solving these problems as further work.

7 Conclusion

We wanted to use universes for race safety because we believe that they are simpler and thus a more programmer-friendly type system. We have given a language, semantics, and race safety type system. We have proved that our system prevents races; full proofs will soon be available from [6]. We took the approach of defining a minimal system and then giving a number of straightforward extensions that add features to the type system and allow more parallelism in programs. These extensions can a) distinguish between read/writes, b) prevent deadlocks, c) verify atomicity, and d) allow locks to be taken at the granularity of single objects.

Ownership types are important for good race safety type systems because they allow the type system to understand the extent of heap accesses in while loops and recursive functions. This is also the area where our work differs most. Our system has the qualifier `any`, which can be used to implement open data-structures without constraining the ownership hierarchy and thus the synchronisation of the rest of the program.

We found that our type system required fewer annotations than previous work [11,3] and that the cases of accessing objects from different domains, it could understand more programs. We also used a system of effects, that we do not believe has been tried before, which lets the programmer use non-final paths.

Another advantage of using universes is that they have already been implemented in JML[22]; we hope to extend JML with race safety features based on our type system.

In the future we would like to extend and refine our system to include generics, and to study atomicity in greater detail.

Acknowledgement

We would like to thank P. Müller and W. Dietl for comments and suggestions that were invaluable in the writing of this paper. We would also like to thank C. Flanagan and C. Boyapati for their elucidating correspondence, and the SLURP group, particularly T. Allwood for comments and suggestions.

References

[1] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Retrospective: Thin locks. In Kathryn S. McKinley, editor, *Twenty Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*, April 2004.

[2] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 382–400, Minneapolis, Minnesota, October 2000.

[3] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications*, Tampa Bay, Florida, October 2001.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18–22 1998. ACM Press.

[6] Dave Cunningham. Universes for Race Safety - proofs, 2007. available from http://www.doc.ic.ac.uk/~dc04/race_safety_proofs/.

[7] Dave Cunningham, Adrian Francalanza, Sophia Drossopou-lou, Werner Dietl, and Peter Mueller. UJ: Type Soundness for Universe Types, 2006. preliminary version available at http://www.doc.ic.ac.uk/~scd/uj.pdf.

[8] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[9] Werner Dietl, Sophia Drossopoulou, and Peter Mueller. GUJ: Generic Universe Types for Java-like languages. In *Proceedings of ECOOP'07*, 2007.

[10] Cormac Flanagan and Martin Abadi. Object types against races. In *International Conference on Concurrency Theory*, pages 288–303, 1999.

[11] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[12] Cormac Flanagan and Stephen N. Freund. Type inference against races. In *SAS*, pages 116–132, 2004.

[13] 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 Press.

[14] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[15] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[16] Rachid Guerraoui, Michał Kapałka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. Technical report, EPFL, 2006.

[17] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[18] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06).*, June 2006. (to appear).

[19] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[20] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. Jml reference manual, 2002.

[23] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[24] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 47–57, January 1988.

[25] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. *SIGPLAN Not.*, 41(1):346–358, 2006.

[26] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262. Springer-Verlag, 2002.

[27] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. *SIGPLAN Not.*, 42(1):327–338, 2007.

[28] James Noble, John Potter, David Holmes, and Jan Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, Brussels, Belgium, July 20 - 24, 1998.

[29] Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. In *ECOOP International Workshop on Aliasing, Confinement and Ownership (IWACO 2003)*, 2003.

[30] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *SIGPLAN Not.*, 41(1):334–345, 2006.

A Universe Lemmas

Lemma A.1 *The ($\trianglerighteq$) operator composes types:*

$$\left.\begin{array}{l} a, w \vdash a', w' : u \\ a', w' \vdash a'', w'' : u' \end{array}\right\} \Longrightarrow a, w \vdash a'', w'' : u \trianglerighteq u'$$

Proof: Case analysis of u and u'.

Lemma A.2 *The ($\triangleright$) operator decomposes types:*

$$\left.\begin{array}{l} a, w \vdash a', w' : u \\ a, w \vdash a'', w'' : u' \end{array}\right\} \Longrightarrow a', w' \vdash a'', w'' : u \triangleright u'$$

Proof: Case analysis.

Firstly we guarantee that at all times after an object is constructed, both its class and its owner remain constant. As a corollary, execution will not affect the universe type judgement of another expression. We present a "substitution" lemma (although there is no substitution here since we are using a stack to hold the arguments. We require h and σ to be consistent with Γ, but the expression e is the same on both sides. Finally we state soundness. In the soundness theorem, m is either n or $n+1$. New threads can initially have any type, but must maintain this type as they execute.

Lemma A.3 *Ownership and class membership are constant:*

$$\left.\begin{array}{l} h(a) = (v, c, \_) \\ \_ \vdash \_, h \rightsquigarrow \_, h' \end{array}\right\} \Longrightarrow h'(a) = (v, c, \_)$$

Proof: Induction over structure of reduction.

Lemma A.4 *The run-time types of expressions are preserved over the execution of other expressions.*

$$\left.\begin{array}{l} h, \sigma \vdash e : t \\ \_ \vdash \_, h \rightsquigarrow \_, h' \end{array}\right\} \Longrightarrow h', \sigma \vdash e : t$$

Proof: Induction over the structure of $h, \sigma \vdash e : t$.

Lemma A.5 *Static type safety implies run-time type safety with respect to a suitable stack.*

$$\left.\begin{array}{l}\Gamma \vdash e : t \\ h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \\ h, \sigma \vdash \mathtt{this} : \Gamma(\mathtt{this})\end{array}\right\} \Longrightarrow h, \sigma \vdash S(e) : t$$

Proof: Induction over the structure of $\Gamma \vdash e : t$.

Theorem A.6 *Run-time types and heap well-formedness are preserved over execution.*

$$\left.\begin{array}{l}\vdash h \\ h, \sigma \vdash e : t \\ \sigma \vdash e, h \rightsquigarrow e', h'\end{array}\right\} \Longrightarrow \begin{array}{l}\vdash h' \\ h', \sigma \vdash e' : t\end{array}$$

Proof: Induction over the structure of $h, \sigma \vdash e : t$.

Theorem A.7 *The well-typedness of all threads in a system is preserved over a step of multithreaded execution.*

$$\left.\begin{array}{l}\vdash h \\ h, \sigma \vdash e_{1..n} : t_{1..n} \\ \sigma \vdash e_{1..n}, h \rightsquigarrow e'_{1..m}, h'\end{array}\right\} \Longrightarrow \begin{array}{l}\vdash h' \\ h', \sigma \vdash e'_{1..m} : t_{1..m}\end{array}$$

Proof: Case analysis of $\sigma \vdash e_{1..n}, h \rightsquigarrow e'_{1..m}, h'$.

B Race Safety Lemmas

Lemma B.1 *Path resolution is preserved over execution.*

$$\left.\begin{array}{l}h(\sigma, p) = v \\ \sigma \vdash p, h \rightsquigarrow e, h'\end{array}\right\} \Longrightarrow \begin{array}{l}e = p', h = h' \\ h'(\sigma, p') = v \\ \forall f \notin p \ . \ f \notin p'\end{array}$$

Proof: induction over $\sigma \vdash p, h \rightsquigarrow e, h'$.

A given expression should be guarded by the same lock no matter what state of execution the expression has reached. We require heap well-formedness because we need the universe type judgements within the guard logic to be preserved.

Lemma B.2 *Guards are preserved over execution.*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ \vdash h \\ \sigma \vdash e, h \leadsto e', h' \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e' : l$$

Proof: Case analysis of $h, \sigma \vdash_{gb} e : l$.

If the heap has changed enough since the lock p was taken, so that p resolves to a different object, then p no longer guards the objects it used to. This lemma ensures heap changes are not sufficient, by ensuring the effect of the executing expression does not conflict with p.

Lemma B.3 *Path resolution is preserved over execution of other expressions.*

$$\left.\begin{array}{l} h(\sigma, p) = v \\ \sigma' \vdash e, h \leadsto e', h' \\ \_, h, \sigma' \vdash e : F \\ \{p\} \ \# \ F \end{array}\right\} \Longrightarrow h'(\sigma, p) = v$$

Proof: Induction over steps of resolution.

The next lemma helps to prove that the guard of an expression should be unaffected by the execution of other expressions. However, we must ensure the reducing expression will not interfere with any paths that the guard might be using.

Lemma B.4 *Guards are preserved over the execution of other expressions*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ \sigma' \vdash e', h \leadsto \_, h' \\ \_, h, \sigma \vdash e' : F' \\ \{l\} \ \# \ F' \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $h, \sigma \vdash_{gb} e : l$.

A similar result relies on the fact that if $Virgin(e)$ then only (Field) and (Var) are used in the derivation of $h, \sigma \vdash_{gb} e : p$ then and neither rule uses h or σ.

Lemma B.5 *Virgin guards are preserved over the execution of other expressions.*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ Virgin(e) \\ \_ \vdash \_, h \leadsto \_, h' \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $h, \sigma \vdash_{gb} e : l$.

The following lemma is used to show that the part of an expression that has not executed yet is not affected by the part of that expression that is executing.

Lemma B.6 *Types of virgin expressions are preserved over the execution of other expressions.*

$$\left.\begin{array}{l} \mathbb{L}, h, \sigma \vdash e : F \\ Virgin(e) \\ \_ \vdash \_, h \rightsquigarrow \_, h' \end{array}\right\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

This lemma requires that the locks taken by one thread are disjoint to the locks guarding a path. This means that the modifications made by the executing thread cannot affect the resolution of the path.

Lemma B.7 *Path resolution is preserved over the execution of other expressions when locks do not collide.*

$$\left.\begin{array}{l} h(\sigma, p) = v \\ \sigma' \vdash e', h \rightsquigarrow \_, h' \\ \emptyset, h, \sigma' \vdash e' : \_ \\ \mathbb{L}, h, \sigma \vdash p : \_ \\ \{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | Locked(e', w)\} = \emptyset \end{array}\right\} \Longrightarrow h'(\sigma, p) = v$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

The following lemma requires that the executing thread's locks are disjoint to the locks required to guard a lock, and can therefore show that the derivation of the lock is unaffected by the execution.

Lemma B.8 *Guards are preserved over the execution of other expressions when locks do not collide.*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ l \in Path \Longrightarrow \mathbb{L}, h, \sigma \vdash l : \_ \\ \sigma' \vdash e', h \rightsquigarrow \_, h' \\ \emptyset, h, \sigma' \vdash e' : \_ \\ \{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | Locked(e', w)\} = \emptyset \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

Finally, this lemma shows that the execution of one thread will not interfere with the typing of another thread.

Lemma B.9 *Types are preserved over the execution of other expressions when locks do not collide.*

$$\left.\begin{array}{l}
\mathbb{L}, h, \sigma \vdash e : F \\
\sigma' \vdash e', h \rightsquigarrow \_, h' \\
\emptyset, h, \sigma' \vdash e' : \_ \\
\mathit{Reachable}(e) \\
\forall w. \neg(\mathit{Locked}(e, w) \wedge \mathit{Locked}(e', w)) \\
\{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \ \cap \{w | \mathit{Locked}(e', w)\} = \emptyset
\end{array}\right\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

The Stability Problem for Verification of Concurrent Object-Oriented Programs

Marieke Huisman[1,2]

INRIA Sophia Antipolis
2004, route des Lucioles BP 93
06902 Sophia Antipolis, France

Clément Hurlin[1,3]

INRIA Sophia Antipolis
2004, route des Lucioles BP 93
06902 Sophia Antipolis, France

Abstract

Modular static verification of concurrent object-oriented programs remains a challenge. This paper discusses the impact of concurrency on the use and meaning of behavioural specifications, and in particular on method contracts and class invariants.
Atomicity of methods is often advocated as a solution to the problem of verification of multithreaded programs. However, in a design-by-contract framework atomicity in itself is not sufficient, because it does not consider specifications. Instead, we propose to use the notion of stability of method contracts to allow sound modular reasoning about method calls. A contract is stable if it cannot be broken by interferences from concurrent threads. We explain why stability of contracts cannot always be shown directly, and we speculate about different approaches to prove stability. Finally, we outline how a proof obligation generator for sequential programs can be extended to one for concurrent programs by using stability information.
This paper does not present a full technical solution to the problem, but instead shows how it can be decomposed into several smaller subproblems. For each subproblem, a solution is sketched, but the technical details still need to be worked out.

Keywords: Multithreading, Design by Contract, stability.

1 Introduction

With the high demands on performance of software, the use of concurrency has become compulsory. Unfortunately, often the gains in speed are cancelled out by the bugs due to the use of concurrency. Therefore, formal techniques to analyse and reason about concurrent programs are necessary. Model checking provides a partial solution, but because of the state space explosion, this often does not scale

[1] This work is partially funded by the IST programme of the EC, under the IST-FET-2005-015905 Mobius project, and the French national research organisation (ANR), under the ANR-06-SETIN-010 ParSec project.

[2] Email: marieke.huisman@sophia.inria.fr

[3] Email: clement.hurlin@sophia.inria.fr

up to complex programs and properties. Instead, we propose to use logic-based techniques for the verification of concurrent programs. We focus in particular on the verification of multithreaded Java programs, where threads communicate via a shared global memory. In the literature two classical approaches exist to verify such programs: the non-modular Owicki-Gries method [17], and Jones's modular rely-guarantee method [9]. However, both approaches require one to write very detailed specifications about the interactions between the different threads, and therefore they do not scale.

Instead, we take a different approach, and identify as many code fragments as possible that can be verified sequentially. In particular, we use method and class specifications, as advocated in the Design by Contract approach [14] (and its Java-instantiation JML [11]), and discuss the impact of multithreading on their use and verification. The basic idea behind Design by Contract is that *preconditions* impose conditions on clients, while *postconditions* provide guarantees. In addition, *class invariants* specify properties that hold throughout the execution.

The use of such specifications provides the basis for sound modular verification of sequential object-oriented programs [15]: when verifying a code fragment containing a method call, the method call can be abstracted by its specification. Consider for example the following code fragment, annotated with JML.

```
//@ requires P;
//@ ensures Q;
void call(){ ... }

void method(MyObject o){
  o.call();
  // (1) }
```

When reasoning sequentially, one can assume that `Q[o/this]` holds after `o.call()` (at `(1)`). However, in a multithreaded setting this need not be the case: any other thread simultaneously executing within the object pointed to by `o` can change the validity of `Q[o/this]`, and in particular, `Q[o/this]` might be invalidated between the end of `call` and the continuation of `method`. Therefore we propose to use the notion of *stable* contract (cf. [3]), to indicate that a contract cannot be invalidated by another thread. Contract stability is crucial for the verification of multithreaded programs: after a method call, a stable postcondition can be used as a precondition for the next statement, whereas an unstable postcondition must be discarded. Similarly, a method implementation can only rely on a stable precondition.

This paper discusses how contract stability can be used to reason about multithreaded programs, and it sketches several speculative ideas how it can be proven. Section 2 discusses in more detail the difficulties of reasoning with method specifications for multithreaded programs. Section 3 defines stability of contracts and discusses how locking, confinement, immutability, and semantics might be used to prove stability. Section 4 demonstrates how to extend a sequential verification condition generator for multithreading, while Section 5 concludes.

2 Reasoning with Specifications in Concurrent Programs

2.1 Method Specifications

Traditionally, *atomicity* [5] has been advocated as a means to decide whether a method could be verified sequentially. A method is said to be atomic if it contains at most one instruction that is sensitive to interference (in the special case where the method contains no such instruction, it is said to be *independent*). If a method is atomic, its method body can be verified without considering interleavings from other threads. However, existing atomicity analyses do not take method specifications into account, and in particular they do not consider whether a pre- or postcondition can be invalidated by another thread. Therefore, it is not possible to reason *about* calls to atomic methods in terms of method specifications. In particular, if an atomic method has an unstable contract, reasoning about calls to this method cannot be done in the standard modular way. However, contract stability is not strictly a stronger notion than atomicity: if the instructions that break the atomicity of the method do not influence validity of the stable method specification, it still can be verified sequentially if the method respects its contract.

Also, we would like to remark that atomicity (and independence) are often conditional properties, i.e., they only hold under particular conditions (see also [5]). Typical examples of such conditions are that a particular lock is held, or that a method parameter is local to a thread. However, when reasoning in terms of method specifications, we think it is sufficient to list these conditions as part of the method's precondition. An alternative approach would be to list such conditions separately, and to interpret the method specification differently, depending on whether this condition is satisfied or not. If the condition is satisfied, and the contract is stable, it is interpreted as is. However, if the condition is not satisfied, all unstable parts should be removed from the contract (cf. [3]). But this would mean that from the client's point of view, methods could have different behaviours. We think this is an undesirable and counter-intuitive situation; making these conditions part of the precondition ensures that the method only can be called in cases where the contract is stable, and thus where one can rely on the method behaving as specified.

Finally, notice that also the meaning of the `modifies` clause must be changed. This clause specifies which variables may be modified during a method call, and all others are implicitly unchanged. However, in a concurrent setting unstable variables can always be modified by another thread. Therefore, a `modifies` clause only implicitly specifies which *stable* variables remain unchanged.

2.2 Class specifications: Invariants and Constraints

The standard meaning of invariants also needs to be revised in a multithreaded context. JML defines a visible-state semantics for class invariants: *all* invariants of *all* allocated objects have to hold in *all* visible states, i.e., basically all states in which a method is called or finished (except for so-called "helper" methods) [12, §8.2]. This means in particular that invariants may be freely broken *inside* a method body.

But in a multithreaded program, when one thread is inside a method body, and thus has the right to break the invariants, another thread might just be entering a method, and thus require the invariants to hold. This is particularly relevant for *true concurrent objects* [19], where several threads may access simultaneously the same object. However, even if we would exclude such behaviour (which could decrease performance [1]), the problem remains, because JML requires *all* invariants to hold in a visible state. Thus, even if different parallel threads would never be allowed to access the same part of memory at the same time, the standard JML semantics would have to be adapted, by requiring for example that only the invariants that are related to that part of memory that can be accessed by the current thread have to hold. However, deciding statically which invariants are relevant to a particular program point is an open problem.

A partial solution is the introduction of so-called *strong invariants*, i.e., invariants that are never broken, not even temporarily. Another possibility is to specify explicitly the properties on which a thread can rely when it has certain locks. This could be expressed using an expression like `locking_rely(l1, ..., ln) P`, meaning that if a thread acquires the locks `l1, ..., ln`, it can assume the property `P`, and when it releases the locks, it has to establish `P`. Alternatively stated, the environment guarantees `P` when `l1, ..., ln` are held. One can also imagine further combinations of the different possibilities, where the thread holding the locks has to ensure that the predicate `P` holds in any of its visible states. A particular instance of this would be to restrict object invariants to specify properties about the state that is protected by the object's lock. Thus, having the object's lock means that one can rely on the object's invariants.

A related problem exists for constraints. A constraint describes a relation that is supposed to hold between every pair of consecutive visible states. But in a multithreaded setting these might belong to different threads. Thus, a naive approach to verify a constraint would be to consider all possible interleavings of the different threads, but this results in non-modular verification. Another possibility is to redefine the notion of constraint, so that it only relates visible states within the same thread. However, in that case, one needs to ensure that other threads cannot break the constraint in-between related states.

In the rest of this paper, we will focus on how we can show that method specifications are stable, and how this can exploited for verification. We assume that the contracts have been extended with conditions arising from class invariants and history constraints. However, from the above, it should be clear that simply desugaring *all* class invariants and constraints into method pre- and postconditions would make it virtually impossible to show that such an extended method contract is stable. Therefore, in addition we will need to develop the means to modularise class invariants and constraints over the program - so that the method contract is extended only with the relevant class specifications.

3 Stability of Contracts

The stability of a contract depends on how variables that have to be read to evaluate the contract can be modified by threads. We use the term *footprint* to denote an upper bound on the set of locations that are accessed during contract evaluation. It depends on the expressiveness of the specification language how precisely a contract's footprint can be computed. For a contract which only reads fields and contains no qualified expressions, the contract's footprint is the set of all fields appearing in the contract.

However, for more complex specification expressions, computing the footprint is more complicated. For example, to compute the footprint of a contract containing a quantifier, we need to be able to determine the quantifier's domain. Fortunately, often the domain of a quantifier is finite, e.g., a universal quantifier ranging over all elements of an array. As an example, the footprint of `oneify`'s contract below is the whole array `a`; thus if no elements of `a` can be written by other threads, `oneify`'s contract is stable.

```
//@ ensures (\forall int i; i >= 0 && i < a.length; a[i] == 1);
void oneify(int[] a){ for (int i = 0; i < a.length; i++) a[i] = 1; }
```

Another problem is when the footprint of a contract cannot be expressed as a list of accessed fields. For example, the footprint of `multiply`'s contract below depends on the way `Node` objects are linked. To show stability of `multiply`'s contract, one has to show that all objects that are recursively accessible via the `next` field are stable (i.e., cannot be modified by concurrent threads). We plan to investigate whether we can use JML's ownership type system [4], to show stability of contracts in such cases.

```
class Node{
  int x; Node next;
  //@ invariant x > 0 && ((next != null) ==> next.x < this.x);

  //@ measured_by(this.x);
  //@ ensures (next != null) ==> (\result == x * next.multiply());
  //@ ensures (next == null) ==> (\result == x);
  /*@ pure @*/ int multiply(){
   if(next == null)
     return x;
   else
     return x * next.multiply(); }}
```

To summarize, the footprint of a contract indicates which objects should be thread-local or lock-protected so that the contract is stable. In some simple cases, a contract's footprint can be computed directly, but in general this is not feasible. The next paragraph describes a permission system used to control concurrent access to objects. This permission system permits to show properties that can be used to prove contract stability. The remainder of this section then shows how stability is proven.

3.1 A Program in Need of Permissions

In order to show stability of contracts, we use a permission system whose general ideas are described elsewhere [7]. This system is a generalization of Boyland's fractional permissions system [2] to object-oriented programs with dynamic thread creation, joins and locks. Our permission system permits to verify properties that are sufficient to show stability. In the following we recall the most relevant features of our system.

A class can be annotated with the keyword `permission` to indicate the permissions that exist for each instance of this class. Permissions can be base permissions, object permissions, or lock permissions. Base permissions have the form `f : \split(n)` where `f` is a field of the class considered and `n` is a natural number. Intuitively, this means there are at most 2^n threads accessing the field at the same time (since $2^0 = 1$, `\split(0)` means only one thread has access). Henceforth, permission `f : \split(0)` allows to write to and read to field `f`, permission `f : \split(n)` (with `n`$>$ 0) allows solely to read. A permission can be *split* into two smaller permissions using the equivalence `\split(n)` $\equiv$ `\split(n+1) && \split(n+1)`. To alleviate the annotation burden we use `W` as a shorthand for `\split(0)` and `R` as a shorthand for `\split(1)`.

Object permissions have the form `f : W (\split(p,n))` or `f : R (\split(p,n))` where `p` is a permission and `n` is a natural number. These permissions contain the base permission `f : W` (or `f : R`) and permission `\split(p,n)` on the object pointed to by `f`. As with base permissions, we have the equivalence `\split(p,n)` $\equiv$ `\split(p,n+1) && \split(p,n+1)`. When `n` is 0 we simply write `p`.

Lock permissions have the form `\split({l},n)` where `l` is a non-primitive final field of the class considered and `n` a natural number. For any `n`, permission `\split({l},n)` gives the right to lock `l`. When `n` is 0 we simply write `{l}`. A lock permission comes with a lock clause written `lock {l} = p, ..., q` to indicate that permissions `p, ..., q` are obtained when `l` is locked.

For every field `f` of an object, the permission system guarantees that *(i)* only one permission allows to write to `f`, and *(ii)* if a read permission to `f` exists, then there is no write permission to `f`. These conditions are crucial to show stability. Finally, our system is defined as an extension of JML [11], so permissions requirements and guarantees are expressed as part of method contracts.

The rest of this section speculates about different possibilities to prove contract stability by using our permission system.

3.2 Stability by Locking

Most concurrent programs rely on locking to achieve correct synchronization. Locking can be used to provide exclusive access to parts of the heap. Existing methodologies have ways to protect individual fields [18,7] or whole classes [8] by locks. The permission system described before allows to specify locking policies. Adherence to the specified locking policy of a class helps to show contract stability of the class's methods. Note that – although we do not detail it here – aliasing complicates stability checking: determining which lock protects which object can require aliasing information.

Class `Point` illustrates how locking policies help to show stability. Class `Point`'s `permissions` clause define that there exists a permission `p` per `Point` object. Permission p allows to synchronise on the Point object considered. Further, the lock clause indicates that synchronising on a Point object gives write permission to fields x and y of this object.

Precondition of method shiftX's requires callers to have permission {this} i.e., to synchronise on this before calling shiftX. This behavior is called *client-side locking* [8]. Note that because permission p means the right to lock this, it cannot be written as a precondition for shiftX whose correctness relies on this being locked before calling (which is the meaning of requires {this}).

```
class Point{
  int x, y;

  //@ permission p = {this};
  //@ lock {this} = x : W, y : W;

  //@ requires {this};
  //@ ensures  {this} && x == \old(x) + delta;
  void shiftX(int delta){ x += delta; }}
```

Client-side locking rules out interferences from other threads, thus it permits to show stability. Since shiftX's caller holds the lock on this, and since writes and reads to x are only allowed when this is held (because of the lock clause), throughout shiftX's execution concurrent accesses to x are not possible. Thus, all locations in shiftX's footprint ({\old(x),x}) cannot be written by other threads and the contract is stable.

3.3 Stability by Confinement

Another technique to ensure stability is to control concurrent access to objects. This allows one for example to show that an object can only be accessed by a single thread (it is said to be *thread local*). This technique is also particularly useful for lock-free algorithms (like the ones described in [13]), where accesses to objects are distributed by the algorithm. Our permission system supports such programs.

For example, class DoubleInt below specifies that there exists two permissions p and q on instances of class DoubleInt. Permission p allows to write to and read from field x (similarly for permission q and field y). Method incX of class DoubleInt requires permission p on this and returns the same permission when the call returns.

```
class DoubleInt{
  int x, y;

  //@ permission p = x : W;
  //@            q = y : W;

  //@ ensures p && q && x == 0 && y == 0;
  public DoubleInt(){ x = 0; y = 0; }
  //@ requires p && x == 0;
  //@ ensures  p && x == 1;
  public void incX(){ x++; }

  //@ requires q && y == 0;
  //@ ensures  q && y == 1;
  public void incY(){ y++; }}
```

The notation `r = di : R (q)` in class `IncMachine` below specifies an object permission: it contains a base permission `di : R` and also permission `q` on the `DoubleInt` object pointed to by `di`. This shows how permissions can be encapsulated into other permissions (following the object-oriented paradigm).

In method `main` below, the main thread first creates the `DoubleInt` object `di` and obtains `di`'s permissions `p` and `q`. Then, the main thread creates the `IncMachine` thread `t` and encapsulates `di`'s permission `q` in `t`'s permission `r`. When `t` is started, permission `r` is transferred from the main thread to `t`. Then, the `main` thread can call `di.incX()` (using `di`'s permission `p`), while the `IncMachine` thread can execute `di.incY()` (using its permission `r` which contains `di`'s permission `q`). This makes class `DoubleInt` truly concurrent, because two threads can execute simultaneously within an instance of this class. Also note that the main thread cannot call `di.incY()` once it has created `t`. Permissions are split when the `main` thread creates `t`: the main thread keeps `di`'s permission `p` but `di`'s permission `q` is encapsulated into `t`'s permission `r`, therefore becoming inaccessible to the main thread.

```
class IncMachine extends Thread{
  final DoubleInt di;

  //@ permission r = di : R (q);

  //@ requires di.q;
  //@ ensures  r;
  public IncMachine(DoubleInt di){ this.di = di; }

  //@ requires r;
  //@ ensures  r;
  public void run(){ di.incY(); }

    public static void main(){      // permissions owned by threads
    DoubleInt di = new DoubleInt(); // main has di.q and di.p
    Thread t = new IncMachine(di);  // main has di.p and t.r

    t.start();                      // main has di.p, t has r
    di.incX();
    // (1)
    }}
```

Because there is only one permission `p` to write to field `x` of any instance of class `DoubleInt`, concurrent writes to `di`'s field `x` are impossible and therefore `incX`'s postconditions is stable: `di.x == 1` can be assumed at point `(1)`. Thus, the permission system allows us to show stability of contracts of classes designed to be accessed by a single writer thread without locking: this is particularly useful for lock-free programs.

3.4 Stability by Immutability

Another way to prove stability of contracts is to use the notion of immutability [6]. An object is said to be *immutable* if it is never written after its initialization. Therefore, it is safe to access it without synchronization. With the annotation system described above, immutability can be expressed by `R` base permissions. Immutability can be used to show stability of contracts. For example, class `Fraction` below (adapted from Lea [10]) is an immutable class. Clause `permission` of class `Fraction` specifies a permission `p` which allows solely to read fields `n` and `d` of `Fraction` objects.

In order for multiple threads to simultaneously access `Fraction` objects, we need to distribute permission `p` among different threads. Our system supports this by splitting permissions. We use `\part(p)` to denote a *part* of permission `p`, that is `p` or `p` split any number of times (`\part(p)` is desugared into `(\exists n. split(p,n) && n >= 0)`).

Precondition of method `plus` requires callers to have a part of `p`. Stability of `plus`'s contract is trivial to prove, because its footprint only contains accesses to readonly fields (and the permission system ensures that there cannot be - interfering - write permissions to these fields in other parts of the program).

```
class Fraction{
  final int n; // numerator
  final int d; // denominator

  //@ permission p = n : R, d : R;

  //@ ensures p;
  public Fraction(int num, int den){ n = num; d = den; }

  //@ requires \part(p) && \part(f.p);
  //@ ensures  \part(p) && \part(f.p) && \result.p &&
  //@          \result.n == n * f.d + f.n * d && \result.d == d * f.d;
  public Fraction plus(Fraction f){
    return new Fraction(n * f.d + f.n * d, d * f.d); }}
```

3.5 Stability by Semantics

The techniques to show stability described above are syntactical techniques. In the following, we sketch an example to give the reader an intuitive notion of stability by semantics. Class `Account` below (adapted from [8]) uses a JML constraint (a simple temporal property) to express that items in the history are never deleted. Because the semantics of constraints have been influenced by rely-guarantee techniques, they can be used to show stability of contracts in a manner reminiscent of rely-guarantee reasoning (but applied to object-oriented programs).

```
class Account{
  //@ constraint (\forall Integer x;
  //@              \old(history).contains(x); history.contains(x));

  final Vector<Integer> history = new Vector<Integer>();
  int balance = 0;

  //@ permission p = {this};
  //@ lock {this} = balance : W, history : ...;

  //@ requires \part(p);
  //@ ensures  \part(p) &&
  //@          (\exists Integer x; history.contains(x); x == amount);
  synchronized void deposit(Integer amount){
    balance+=amount;
    history.add(amount); }}
```

In this example, `deposit`'s postcondition is sensible to interferences from other threads: between `deposit`'s returning and the caller resuming a thread may call `deposit`, thus adding an item to the history. However, the constraint and the fact that `amount` is put in the history during `deposit`'s executions entail stability of `deposit`'s postcondition. Generally, proving stability by semantics consists in using constraints to give additional assumptions in the presence of interferences from concurrent threads.

4 Lifting Sequential Program Verification to Concurrency

Above, we have shown how method's contracts can be shown to be stable, so that they can be used for reasoning about method calls. However, we also need a technique to discard properties that we no longer can rely upon. In particular, whenever the stability of objects changes, because of sharing or releasing locks, the unstable expressions need to be discarded from the intermediate assertions. This technique is inspired by the "havoc" approach introduced in the Boogie methodology [8]. However the stability information obtained is more fine-grained than the approach cited: less havoc statements are generated resulting in easier proof obligations. In particular, the Boogie approach forces programmers to protect whole classes by locks whereas we allow a per field protection. This permits us to havoc only certain fields of objects while the Boogie approach always havoc all fields.

We illustrate the discarding mechanism by an example, showing how the sequential verification of a `CommonWarehouse` is lifted to a concurrent one. The example uses a Floyd-Hoare-like proof outline, but a similar technique can be used to lift proof obligation generators based on, e.g., weakest precondition or strongest postcondition calculi from sequential to concurrent program verification.

Class `Quantity` in Figure 1 defines a permission `p` that gives write access to field

```
class Quantity{
  volatile int i; //@ invariant i >= 0;

  //@ permission p = i : W;

  //@ requires initial >= 0;
  //@ ensures p && i == initial;
  public Quantity(int initial){ i = initial; }

  //@ requires p && plus >= 0;
  //@ modifies i;
  //@ ensures p && i >= plus;
  public void add(int plus){ i += plus; }}

class CommonWarehouse{
  //@ permission q = ...;

  //@ requires \part(q) && o.p;
  //@ ensures  \part(q);
  void lend(Quantity o){ ... }

  //@ requires \part(q);
  //@ ensures  \part(q) && o.p;
  void takeBack(Quantity o){ ... }

  //@ requires \part(c.q) && j >= 0 && k >= 0;
  //@ ensures  \part(c.q) && \result.p && \result.i >= k;
  public static Quantity main(CommonWarehouse c, int j, int k){
    Quantity o = new Quantity(j);
      [o.p]    $\{o.i = j\}_s \leadsto \{o.i = j\}_c$
    c.lend(o);
      []       $\{o.i = j\}_s \leadsto \{\top\}_c$
    c.takeBack(o);
      [o.p]    $\{o.i = j\}_s \leadsto \{\top\}_c$
    o.add(k);
      [o.p]    $\{o.i \geq k\}_s \leadsto \{o.i \geq k\}_c$
   return o; }}
```

Fig. 1. Example: from sequential proof outline to concurrent proof outline

i. Note that class Quantity's contracts are stable by confinement. Notice further that its invariant is a strong invariant. The methods of class CommonWarehouse are annotated with pre- and postconditions indicating how permissions of parameters are updated. For example, lend's callers must have a part of permission q on the receiver and permission p on parameter o.

Method main is annotated with a proof outline (only showing relevant formulas). First, between square brackets, the evolution of permissions owned by the main thread is shown. Initially, the main thread has exclusive access to the new `Quantity` object, because it has permission `p` on this object (and as `Quantity`'s `permissions` clause shows, only one such permission may exist on a `Quantity` object). After calling `lend`, it gives away this permission (as indicated by `lend`'s contract), and the object pointed to by `o` becomes unstable, as it can be accessed by other threads. After calling `takeBack`, the main thread regains exclusive access to the `Quantity` object (because it gains back `o`'s permission `p`), and thus it knows `o.i` cannot be written by concurrent threads. The second column (enclosed with $\{\}_s$) are the *sequential formulas* obtained by a strongest postcondition calculation. Finally, the third column (enclosed with $\{\}_c$) shows the *concurrent formulas*, the result of weakening the intermediate assertions based on the stability information.

After constructing the new `Quantity` object and assigning it to `o`, sequential formula `o.i = j` holds, because of the constructor's postcondition. As this postcondition is stable, it is also a concurrent formula. Because of the implicit `modifies \nothing` clause, `lend` and `takeBack` do not change `o`'s contents, and the sequential formula remains `o.i = j`. However, the call to `lend` makes `o` unstable: another thread gains `o`'s permission `p` and this thread may write to `o.i`, therefore properties about `o` need to be discarded in the concurrent formula. After the call to `takeBack`, `main` has exclusive access to `o` again, but as we do not know what other threads did with the object in the mean time, we cannot assume anything about `o.i` here. Only after the call to `add`, we know something about `o.i`'s value. This information also holds in a concurrent setting, because `o` is stable: `o.i` cannot be written by other threads. However, notice that if the postcondition of `add` had contained the old value of `i`, the concurrent formula would have reduced to $\top$ again.

This shows how the stability information provided by the permission system can be used to compute intermediate assertions for concurrent programs. This approach is modular because *(i)* concurrency aspects are delegated to the permission system, *(ii)* interferences of concurrency with the intermediate assertions occur only at weakening points, and *(iii)* program behaviour is abstracted by method specifications. This improves for example over rely-guarantee techniques, where the stability information flows through the proof.

5 Conclusions and Future Work

This paper sketches a modular verification technique, based on method specifications for multithreaded Java programs. The main idea is that method specifications should be stable, i.e., their validity should not be affected by other threads. If method specifications are stable, modular sequential verification techniques can be adapted for a concurrent setting. We show how locking, controlling object access and immutability can be used to show stability of contracts. We also show how stability information allows to lift a sequential proof outline to a concurrent proof outline.

This paper does not describe finished work; instead it is a first step towards the development of a verification technique for multithreaded programs, without putting any major restrictions on the programming model used.

For this technique to be fully operational, the following topics need to be addressed: *(i)* class invariants and history constraints must be handled appropriately, probably requiring that certain locks are held before such specifications can be assumed, *(ii)* more precise techniques to compute contract's footprints have to be developed, *(iii)* techniques for stability checking need to be completed and implemented, and *(iv)* we need to extend a verification condition generator for sequential programs to concurrent programs, based on the weakening procedure described in Section 4.

Related Work

Jacobs et al. [8] recommend *client-side locking* to force contracts to rely on stable objects and require contracts to perform only legal access. They do not discuss how to check this for complex specifications. They use an ownership system to control object accesses, however in a more restrictive way, as they exclude for example, true concurrency. Contrary to Jacobs et al., we try not to impose a programming model and aim at verifying more varied patterns.

Rodriguez et al. [18] use the term *internal interference* to denote that a thread may change data observable by a method executed by another thread. They rule out internal interferences by using atomicity and independence i.e., by showing that an interleaved execution of a method can always be reduced to a sequential execution. They call *external interference* the problem of a thread making observable changes between a method call of another thread and the method's entry of this thread (or between the method's exit and the thread's resuming). As we pointed out in section 2, atomicity and independence are not sufficient to avoid this problem and the method presented in this work suffers from this defect: it is advocated that contracts should solely rely on *thread safe* objects, i.e., local or locked objects, but it is not described how this is enforced. Our technique handles both kind of interferences in a sound way.

Nienaltowski and Meyer [16] address the stability problem by having an alternative semantics for contracts. Preconditions are treated as wait-conditions: a non-satisfied precondition forces the client to wait until it becomes true. Postconditions are projected into the future: a postcondition on an object is required to be true only when this object is accessed. However, as the authors point out this solution raises liveness issues. Furthermore, it cannot be applied to Java where method calls and returns do not respect this alternative semantics.

Calcagno et al. [3] address the stability problem by *stabilizing* assertions: from unstable properties weaker stable properties are computed, using a fixpoint computation. This approach is not integrated into a Design by Contract framework. Notice that our technique does not require stabilization, we simply impose contracts to be stable. This can be seen as a worst case stabilization: we consider that shared objects can be affected in any way by other threads.

Acknowledgements

We thank our partners in the Mobius project, in particular Christian Haack, Erik Poll for their useful and constructive comments on the stability issue. We also thank Gustavo Petri for his insightful remarks.

References

[1] C. Baquero, R. Oliveira, and F. Moura. Integration of concurrency control in a language with subtyping and subclassing. In *USENIX Conference on Object-Oriented Technologies*, Monterey, California, 1995.

[2] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[3] C. Calcagno, M. Parkinson, and V. Vafeidis. Modular safety checking for fine-grained concurrency, 2007. Submitted.

[4] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[5] C. Flanagan and S. Qadeer. Types for atomicity. In *Types in Language Design and Implementation*. Association of Computing Machinery Press, 2003.

[6] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, editor, *European Symposium on Programming*, volume 4421 of *LNCS*, pages 347–362. Springer-Verlag, 2007.

[7] M. Huisman and C. Hurlin. Thread capability annotations for common multithreaded programming patterns, 2007. Manuscript, http://www-sop.inria.fr/everest/Clement.Hurlin/publis/annotations-patterns.pdf.

[8] B. Jacobs, K.R.M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods*, Koblenz, Germany, 2005.

[9] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[10] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns (Second Edition)*. Addison-Wesley Publishing Company, Boston, MA, USA, 1999.

[11] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06y, Iowa State University, 1998. (revised since then 2004).

[12] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

[13] C.E. Leiserson and H. Prokop. Minicourse on multithreaded programming, July 1998. http://supertech.csail.mit.edu/cilk/papers/index.html.

[14] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[15] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[16] P. Nienaltowski and B. Meyer. Contracts for concurrency. In *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages*, York, United Kingdom, July 2006.

[17] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.

[18] E. Rodríguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In A.P. Black, editor, *European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer-Verlag, July 2005.

[19] L. Thomas. Inheritance anomaly in true concurrent object oriented languages: A proposal. In *TENCON*, pages 541–545. IEEE Press, August 1994.

Java Memory Model Examples: Good, Bad and Ugly

David Aspinall [1,3] Jaroslav Ševčík [1,2,4]

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK

Abstract

We review a number of illustrative example programs for the Java Memory Model (JMM) [6,3], relating them to the original design goals and giving intuitive explanations (which can be made precise). We consider good, bad and ugly examples. The good examples are allowed behaviours in the JMM, showing possibilities for non sequentially consistent executions and reordering optimisations. The bad examples are prohibited behaviours, which are clearly ruled out by the JMM. The ugly examples are most interesting: these are tricky cases which illustrate some problem areas for the current formulation of the memory model, where the anticipated design goals are not met. For some of these we mention possible fixes, drawing on knowledge we gained while formalising the memory model in the theorem prover Isabelle [1].

1 Introduction

The Java Memory Model (JMM) [6,3] is a relaxed memory model which acts as a contract between Java programmers, compiler writers and JVM implementors. It explains possible and impossible behaviours for multi-threaded programs. The JMM is necessary to allow efficient execution and compilation of Java programs, which may result in optimisations that affect the order of memory operations. The case that is usually desirable is *sequential consistency* (SC) [4], which, roughly speaking, says that the outcome of executing a multi-threaded program should be equivalent to the outcome of executing some sequential ordering of its operations which agrees with the order that statements appear in the program. Sequential consistency has acted as a correctness criterion in the study of relaxed memory models and numerous variations have been explored; we discuss our own precise definition of SC for the JMM later.

[1] Supported by the EU project MOBIUS (IST-15905).

[2] Supported by *Resource Quantification in e-Science Technologies* project (ReQueST) funded by the UK EPSRC, grant EP/C537068.

[3] Email: `da@inf.ed.ac.uk`

[4] Email: `j.sevcik@sms.ed.ac.uk`

Sequential consistency helps make multi-threaded programming comprehensible to the programmer. But for parts of a program which are executing in unrelated threads, sequential consistency may not be required. Moreover, in pursuit of the best possible performances, sophisticated concurrent algorithms have been designed which work in the presence of data races. A *data race* in a program is a point where the program itself fails to specify an ordering on conflicting actions across threads, so sequential consistency is underdefined.

The JMM is one of the first explored memory models which connects a high-level programming language to low-level executions (most other relaxed memory models work at the hardware level). The JMM has been designed to make three guarantees:

(i) *A promise for programmers*: sequential consistency must be sacrificed to allow optimisations, but it will still hold for data race free programs. This is the data race free (DRF) guarantee.

(ii) *A promise for security*: even for programs with data races, values should not appear "out of thin air", preventing unintended information leakage.

(iii) *A promise for compilers*: common hardware and software optimisations should be allowed as far as possible without violating the first two requirements.

The key question one asks about a program is whether a certain outcome is possible through some execution. Unfortunately, with the present JMM, it can be quite difficult to tell the answer to this! In part, the memory model has been designed around a set of examples (e.g., the causality tests in [8]) which have helped shape the definitions; but gaps and informality in the present definitions mean that there are still unclear cases.

Points i and ii act to *prohibit* certain executions, whereas point iii acts to *require* certain executions. It seems that only point i provides a precise set of behaviours that are disallowed, i.e., the non-SC behaviours for data race free programs. Regarding point iii, exactly which optimisations must be allowed has been a source of some debate and is still in flux [8,9]. Regarding point ii, the "out of thin air" requirement has yet to be precisely characterised; we only know of forbidden examples which violate causality to allow behaviours that result in arbitrary values.

This paper discusses some illustrative examples for the Java Memory Model, relating them back to these goals and to the JMM definitions, and in particular, the definitions as we have formalised them [1]. Our contribution is to collect together some canonical examples (including tricky cases), and to explain how they are dealt with by our formal definitions, which represent an improvement and clarification of the official definitions. Despite the intricacies of the JMM definitions, we present the examples at an informal level as far as possible. We also give some opinions on future improvements of the JMM.

The rest of this paper is structured as follows. Section 2 explains intuitively how behaviours are justified in the memory model. Sections 3, 4 and 5 then present the examples: the good (allowed), the bad (prohibited) and the ugly (tricky cases where there is disparity between the JMM design aims and the actual definitions). Appendix A recalls some of the definitions of the JMM in a more precise format for reference; it should be studied by those who seek a complete understanding but can be ignored by a casual reader. Section 6 concludes.

initially `x = y = 0`	
`r1 := x`	`r2 := y`
`y := 1`	`x := 1`

A. (allowed)

initially `x = y = 0`	
`sync(m1)`	`sync(m2)`
`{r1:=x}`	`{r2:=y}`
`sync(m2)`	`sync(m1)`
`{y:=1}`	`{x:=1}`

B. (prohibited)

initially `x = y = 0`	
`r1 := x`	`r2 := y`
`y := r1`	`x := r2`

C. (prohibited)

Is it possible to get `r1` = `r2` = 1 at the end of an execution?

Fig. 1. Examples of legal and illegal executions.

2 A Bluffer's Guide to the JMM

Motivation.

Before we introduce the memory model, let us examine three canonical examples (from [6]), given in Fig. 1, which illustrate the requirements mentioned above. The programs show statements in parallel threads, operating on thread-local registers (`r1`, `r2`, . . .) and shared memory locations (`x`, `y`, . . .).
In an interleaved semantics, program A could not result in `r1` = `r2` = 1, because one of the statements `r1:=x`, `r2:=y` must be executed first, thus either `r1` or `r2` must be 0. However, current hardware can, and often does, execute instructions out of order. Imagine a scenario where the read `r1:=x` is too slow because of cache management. The processor can realise that the next statement `y:=1` is independent of the read, and instead of waiting for the read it performs the write. The second thread then might execute both of its instructions, seeing the write `y:=1` (so `r2` = 1). Finally, the postponed read of `x` can see the value 1 written by the second thread, resulting in `r1` = `r2` = 1. Similar non-intuitive behaviours could result from simple compiler optimisations, such as common subexpression elimination. The performance impact of disabling these optimisations on the current architectures would be huge; therefore, we need a memory model that allows these behaviours.

However, there are limits on the optimisations allowed—if the programmer synchronises properly, e.g., by guarding each access to a field by a synchronized section on a designated monitor, then the program should only have sequentially consistent behaviours. This is why the behaviour `r1` = `r2` = 1 must be prohibited in program B of Fig. 1.

Even if a program contains data races, there must be some security guarantees. Program C in Fig. 1 illustrates an unwanted "out-of-thin-air" behaviour—if a value does not occur anywhere in the program, it should not be read in any execution of the program. The out-of-thin-air behaviours could cause security leaks, because references to objects from possibly confidential parts of program could suddenly appear as a result of a self-justifying data race. This might let an applet on a web page to see possibly sensitive information, or, even worse, get a reference to an object that allows unprotected access to the host computer.

JMM framework.

Now we introduce the key concepts behind the JMM. Unlike interleaved semantics, the Java Memory Model has no explicit global ordering of all actions by time consistent with each thread's perception of time, and has no global store. Instead, executions are described in terms of memory related actions, partial orders on these actions, and a visibility function that assigns a write action to each read action.

An action is a tuple consisting of a *thread identifier*, an *action kind*, and a *unique identifier*. The action kind can be either a normal read from variable x, a normal write to x, a volatile read from v or write to v, a lock of monitor m, or an unlock of monitor m. The volatile read/write and lock/unlock actions are called *synchronisation actions*. An *execution* consists of a *set of actions*, a *program order*, a *synchronisation order*, a *write-seen* function, and a *value-written* function. The program order ($\leq_{po}$) is a total order on the actions of each thread, but it does not relate actions of different threads. All synchronisation actions are totally ordered by the synchronisation order ($\leq_{so}$). From these two orders we construct a happens-before order of the execution: action a happens-before action b ($a \leq_{hb} b$) if (1) a synchronises-with b, i.e., $a \leq_{so} b$, a is an unlock of m and b is a lock of m, or a is a volatile write to v and b is a volatile read from v, or (2) $a \leq_{po} b$, or (3) there is an action c such that $a \leq_{hb} c \leq_{hb} b$. The happens-before order is an upper bound on the visibility of writes—a read happening before a write should never see that write, and a read r should not see a write w if there is another write happening "in-between", i.e., if $w \leq_{hb} w' \leq_{hb} r$ and $w \neq w'$, then r cannot see w.[5]

We say that an execution is *sequentially consistent* if there is a total order consistent with the program order, such that each read sees the most recent write to the same variable in that order. A pair of memory accesses to the same variable is called a *data race* if at least one of the accesses is a write and they are not ordered by the happens-before order. A program is *correctly synchronised* (or *data-race-free*) if no sequentially consistent execution contains a data race.

A tricky issue is initialisation of variables. The JMM says

> The write of the default value (zero, false, or null) to each variable synchronises-with to the first action in every thread [7]

However, normal writes are not synchronisation actions and synchronises-with only relates synchronisation actions, so normal writes cannot synchronise-with any action. For this paper, we will assume that all default writes are executed in a special initialisation thread and the thread is finished before all other threads start. Even this interpretation has problems; we mention them in Sect. 8.

Committing semantics.

The basic building blocks are *well-behaved* executions, in which reads are only allowed to see writes that happen before them. In these executions, reads cannot see

[5] For details, see Defs A.2, A.4 and A.6 in App. A.

$$\begin{tikzcd}
 & W(x,0);W(y,0) & \\
R(x,0) \arrow[ur] \arrow[drr, dash, dotted] & & R(y,0) \arrow[ul] \arrow[dll, dash, dotted] \\
W(y,1) \arrow[u] & & W(x,1) \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & W(x,0);W(y,0) & \\
R(x,0) \arrow[ur] \arrow[drr, dash, dotted] & & {[R(y,1)]} \arrow[ul] \\
{[W(y,1)]} \arrow[u] & & W(x,1) \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & W(x,0);W(y,0) & \\
{[R(x,1)]} \arrow[ur] & & {[R(y,1)]} \arrow[ul] \\
{[W(y,1)]} \arrow[u] & & {[W(x,1)]} \arrow[u]
\end{tikzcd}$$

Fig. 2. Justifying executions of program A from Fig. 1.

writes through data races, and threads can only communicate through synchronisation. For example, programs A and C in Fig. 1 have just one such execution—the one, where `r1` = `r2` = 0. On the other hand, the behaviours of program B are exactly the behaviours that could be observed by the interleaved semantics, i.e. `r1` = `r2` = 0, or `r1` = 1 and `r2` = 0, or `r1` = 0 and `r2` = 1. In fact, if a program is correctly synchronised then its execution is well-behaved if and only if it is sequentially consistent. This does not hold for incorrectly synchronised programs, see Sect. 8.

The Java Memory Model starts from a well-behaved execution and *commits* one or more data races from the well-behaved execution. After committing the actions involved in the data races it "restarts" the execution, but this time it must execute the committed races. This means that each read in the execution must be either committed and see the value through the race, or it must see the write that happens-before it. The JMM can repeat the process, i.e., it may choose some non-committed reads involved in a data race, commit the writes involved in these data races if they are not committed already, commit the chosen reads, and restart the execution. The JMM requires all of the subsequent executions to preserve happens-before ordering of the committed actions.

This committing semantics imposes a causality order on races—the outcome of a race must be explained in terms of previously committed races. This prevents causality loops, where the outcome of a race depends on the outcome of the very same race, e.g., the outcome `r1` = 1 in program C in Fig. 1. The DRF guarantee is a simple consequence of this procedure. If there are no data races in the program, there is nothing to commit, and we can only generate well-behaved executions, which are sequentially consistent. In fact, the JMM actually commits all actions in an execution, but committing a read that sees a write that happens before it does not create any opportunities for committing new races, because reads must see writes that happen-before them in any well-behaved execution. Therefore the central issue is committing races, and we explain our examples using this observation.

3 Good executions

The interesting executions are those which are not sequentially consistent, but are legal under the JMM.

Simple reordering.

First, we demonstrate the committing semantics on program A in Fig. 1. In the well-behaved execution of this program, illustrated by the first diagram in Fig. 2, the reads of `x` and `y` can only see the default writes of 0, which results in `r1` = `r2` = 0.

There are two data races in this execution (depicted by the dotted lines, the solid lines represent the happens-before order)—one on `x` and one on `y`. We can commit either one of the races or both of them. Suppose we commit the race on `y`. In the second diagram we show the only execution that uses this data race; the committed actions are in brackets and the committed read sees the value of (the write in) the data race. The non-committed read sees the write that happens-before it, i.e., the default write. This execution gives the result `r1` = 0 and `r2` = 1. The JMM can again decide to commit a data race from the execution. There is only one such data race. Committing the data race on `x` gives the last diagram, and results in `r1` = `r2` = 1.

Complex optimisation.

The program

```
     initially x = y = 0
 ---------------------------------
 r1 := x          | r2 := y
 if (r1 != 0)     | x := r2
   y := r1        |               ,
 else             |
   y := 1         |
```

can result in `r1` = `r2` = 1 using a committing sequence that is very similar to the previous program. First we commit the race on `y` and then the race on `x` with value 1. Note that the actions are not tied to instructions in any way—the write to `y` is first committed using the `else` branch, but it is executed by the `then` branch in the final execution.

Although it might seem counter-intuitive, this behaviour could be the outcome of a compiler optimisation. The compiler may realise that the only values in the program are 0 and 1; that the first thread can never write 0 to `y`; thus it must write 1. As a result, it could replace the `if` statement with an assignment `y:=1`. Then it could reorder this with the statement `r1:=x`, giving the program:

```
 y := 1    | r2 := y
 r1 := x   | x := r2
```

After the reordering we can get `r1` = `r2` = 1 from a SC execution.

4 Bad executions

The JMM aims to prohibit out-of-thin-air behaviours, which are usually demonstrated by program C from Fig. 1.

It contains a causality loop, which potentially could explain any value being written and seen by `r1`. But the JMM prohibits the result `r1` = 1, for example, because we can only commit a race with the value 0. Even after restarting, all reads must see the value 0, because the non-committed ones can only see the default writes, and the committed ones were committed with value 0.

Another powerful tool for determining (in)validity of executions is the DRF guarantee. For example, this program (from [6]):

initially `x = y = 0`	
`r1 := x`	`r2 := y`
`if (r1>0) y := 42`	`if (r2>0) x := 42`

is data race free and thus it cannot be that `r1` = 42, because the writes of 42 do not occur in any sequentially consistent execution.

5 Ugly executions

Here we give examples that either show a bug in the JMM, i.e., behaviours that should be allowed but they are not, or behaviours that are surprising.

1. Reordering of independent statements.

The first example (from [2]) demonstrates that it is not the case that any independent statements can be reordered in the JMM [6] without changing the program's behaviour:

`x = y = z = 0`	
`r1:=z`	`r2:=x`
`if (r1==1) {x:=1; y:=1}`	`r3:=y`
`else {y:=1; x:=1}`	`if (r2==1 && r3==1)`
	`z:=1`

Can we get `r1` = `r2` = `r3` = 1? This requires that each register read sees a write of 1. The only way to commit a data race on `z` with value 1 is to commit the races on `x` and `y` with values 1, so that the write `z:=1` is executed. Note that the writes to `x` and `y` are committed in the `else` branch, and the write to `y` happens-before the write to `x`. However, once we commit the data race on `z`, the first thread must execute the `then` branch in the restarted execution, which violates the requirement on preservation of ordering of the committed actions. So this outcome is impossible.

If we swap the statements in either branch of the `if` statement so they are the same, we *can* justify the result `r1` = `r2` = `r3` = 1. This demonstrates that independent instruction reordering introduces a new behaviour of the program, and so is not a legal transformation in general. This falsifies Theorem 1 of [6].

[6] In [1], we suggest a weakening of the legality definition that fixes this problem while preserving the DRF guarantee.

2. Reordering with external actions.

Another counterexample to Theorem 1 of [6] shows that normal statements cannot be reordered with "external statements" (i.e. those which generate external actions such as I/O operations).

x = y = 0

`r1:=y`	`r2:=x`
`if (r1==1)`	`y:=r2`
`  x:=1`	
`else {print "!"; x:=1}`	

Here, the result $\texttt{r1} = \texttt{r2} = 1$ is not possible, because we must commit the printing before committing the data race on `x`[7]. However, after swapping the print with `x:=1` in the else branch, we can get the value 1 in both the registers by committing the race on `x` followed by committing the race on `y`. As a result, this reordering is not legal. A fix to this problem has not been proposed yet.[8]

3. Roach motel semantics.

A desirable property of the memory model is that adding synchronisation to a program could introduce deadlock, but not any other new behaviour. A special case is "roach motel semantics" ([7]), i.e., moving normal memory accesses inside synchronised regions. Another case is making a variable volatile. We now show examples falsifying the claim that the JMM ensures this property.

First, note that this program

initially `x = y = z = 0`

`lock m`	`lock m`	`r1 := x`	`r3 := y`
`x := 2`	`x := 1`	`lock m`	`z :   = r3`
`unlock m`	`unlock m`	`r2 := z`	
		`if (r1 == 2)`	
		`  y := 1`	
		`else`	
		`  y := r2`	
		`unlock m`	

cannot result in $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$, because the only way to get a write of 1 into `y` is to commit the data race on `x` with value 2. Once we commit the read of 2 from `x`, the read of `x` must always see value 2 and the register `r1` must be 2 in any subsequent execution, so we cannot have $\texttt{r1} = 1$ in the final execution.

However, if we move the assignment `r1:=x` inside the synchronised block, we can construct an execution where `x:=2` happens-before `r1:=x` using the synchronisation on `m`, and `r1:=x` sees value 2 (see execution A from Fig. 3). From this execution, we can commit the data race on `y` with value 1 without committing the read of `x`. Then we can restart and commit the data race on `z` with value 1, and after another restart of the execution, we let the read of `x` see value 1 (execution B from Fig. 3). As a result, we introduce a new behaviour $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$ after moving a normal access into a synchronised block.

Using the same reasoning, if `x` is volatile, we can also get $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$

[7] By rule 9 of legality, Def. A.7 in the Appendix.

[8] The apparent fix is to drop rule 9; the consequences of this are unknown, although DRF will be preserved.

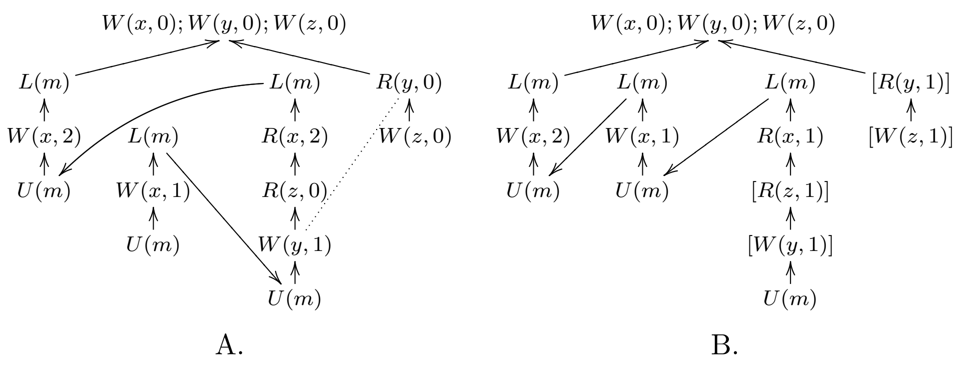


Fig. 3. Justifying and final executions for the roach motel semantics counterexample.

even if `r1:=x` is outside the synchronised block. This demonstrates that making a variable volatile can introduce new behaviours other than deadlock.

4. Reads affect behaviours.

In the JMM, reads can affect an execution in perhaps unexpected ways. Unlike an interleaved semantics, removing a redundant read from a program, such as replacing `r:=x;r:=1` by `r:=1` can decrease observable behaviours, because the read of `x` might have been previously committed and it must be used. As a result, introducing redundant reads to a program is not a legal program transformation.

To demonstrate this, consider the program

```
                x = y = z = 0
-----------------------------------------
r1 := z                      | x := 1
if (r1==0)                   | r2 := y
{ r3 := x                    | z := r2
  if (r3==1) y := 1 }        |
else {  r4 := x              |
        r4 := 1              |
        y := r1  }           |
```

and the outcome `r1` = `r2` = 1. This is a possible behaviour of the program—we can commit the race on `x` between `r3:=x` and `x:=1`, then the race on `y` between `y:=1` and `r2:=y`, and finally the race on `z` with value 1 to get the result. But if we remove the (redundant) read `r4:=x`, we cannot keep the committed race on `x` after we commit the race on `z` with value 1.

5. Causality tests.

In [8], the causality tests 17–20 are wrong—they are not allowed in the JMM contrary to the claim. We illustrate the problem on causality test 17 (all tests are an instance of one problem and we suggest a fix in [1]):

```
           x = y = 0
-------------------------------
r3 := x            | r2 := y
if (r3 != 42)      | x := r2
   x := 42         |
r1 := x            |
y := r1            |
```

The causality test cases state that $\texttt{r1} = \texttt{r2} = \texttt{r3} = 42$ should be allowed, because the compiler might realize that no matter what is the initial value of variable `x` in the first thread, `r1` will always be 42, and it can replace the assignment by `r1 := 42`. Then we can get the desired outcome by simple reordering of independent statements. However, there is a subtle bug in the memory model and it does not allow this behaviour (see [1] for more details). This is because rule 7 of legality (Def. A.7) is too strong—it requires the reads being committed to see already committed writes in the justifying execution.

One of the causality tests also answers an interesting question: can we always commit actions one-by-one, i.e., each commit set having one more element than the previous one? The answer is negative, as illustrated by the causality test 2:

```
           x = y = 0
-------------------------------
r1 := x            | r3 := y
r2 := x            | x := r3
if (r1 == r2)      |
   y := 1          |
```

To get $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$, we must commit both reads of `x` at the same time.

6. Sequential consistency and lock exclusivity.

Our next example execution is considered sequentially consistent by the JMM but does not reflect an interleaved semantics respecting mutual exclusion of locks. The JMM says that an execution is sequentially consistent if there is a total order consistent with the execution's program order such that each read sees the most recent write in that order. In [1], we show that this does not capture interleaved semantics with exclusive locks, demonstrated in this program:

```
         initially x = y = z = 0
-------------------------------------
r1 := y  | lock m     | lock m
x := r1  | r2 := x    | y := 1
         | z := 1     | r3 := z
         | unlock m   | unlock m
```

By the JMM, it is possible to get r1 = r2 = r3 = 1, using the total order `lock m`, `lock m`, `y:=1`, `r1:=y`, `x:=r1`, `r2:=x`, `z:=1`, `r3:=z`, `unlock m`, `unlock m`. It is not hard to show that this the only order of reads and writes that is consistent with the program order and the reads see only the most recent writes in that order. However, this order does not respect mutual exclusion of locks. We believe, therefore, that sequential consistency ought to require existence of a total order consistent with both the program order and the synchronisation order. In [1], we have proved that the DRF guarantee also holds for this definition of SC.

7. Default writes and infinite executions.

The writes of default values to variables introduce inconsistencies and surprising behaviours in the JMM. We noted earlier that the definition of the happens-before relation is flawed when default writes are considered carefully. Possible fixes, either by making default writes into synchronisation actions, or by forcing the initialisation thread to finish before all other threads start, conflict with infinite executions and observable behaviours in a subtle way.

For example, any infinite execution of the program (from [1]):

```
while(true) { new Object() { public volatile int f; }.f++; }
```

must initialize an infinite number of volatile variables before the start of the thread executing the while loop. Then the synchronisation order is not an omega order, which violates the first well-formedness requirement. So the program above cannot have any well-formed execution in Java.

A related problem arises with the notions of observable behaviour and "hung" program given in [7]: in the program that precedes the above loop with an external action, there can only be a finite observable behaviour but the criteria for being "hung" are not met. Because of this, we suggest to restrict the executions to finite ones, and exclude the default write actions from observable behaviours.

8. Well-behaved executions and SC.

For justifying legal executions, the JMM uses "well-behaved executions" [6, Section 1.2]. These are executions with certain constraints, in particular, that each read sees a most recent write in the happens-before order. It might be interesting to examine the relationship between the well-behaved executions and the sequentially consistent executions.

Lemma 2 of [6] says that for correctly synchronised programs, an execution is sequentially consistent if and only if it is well-behaved.

However, the following two examples show that these two notions are incomparable for general programs. First, consider the program

x = 0	
x := 1	r1 := x

and its execution, where r1 = 1. This is sequentially consistent, but not well-behaved.

On the other hand, the program

```
lock m1     | lock m2
y := 1      | y := 2
x := 1      | x := 2
unlock m1   | unlock m2
lock m2     | lock m1
r1 := x     | r2 := x
r3 := y     | r4 := y
unlock m2   | unlock m1
```

has a well-behaved execution, where `r1` $=$ `r4` $= 1$ and `r2` $=$ `r3` $= 2$. This result is not possible in any SC execution.

6 Conclusions

We have collected together and explained a set of typical examples for the Java Memory Model, including those good and bad programs used to motivate the definitions, and a set of ugly programs which cause problems for the present JMM definitions. Our aim was to summarise the status of the current JMM, as we understand it, and at the same time make some of the technicalities more accessible than they are in other accounts.

We have explained the examples at a slightly informal level, giving an intuitive description of the checking process one can use to explain why some example is possible (by explaining the commit sequence), or argue why some example is impossible (by explaining that no commit sequence with the desired outcome is possible). We explained the commit sequences in terms of what happens for data races, which is the essential part of the process; this fact is mentioned in Manson's thesis [5] but not easily gleaned from the text or definitions in other published accounts [6,3].

The ugly cases are the most interesting; clearly something must be done about them to gain a consistent account. The first hope is that the JMM definitions can be "tweaked" to fix everything. We have suggested several tweaks so far. But, because the definitions have some *ad hoc* aspects justified by examples on a case-by-case basis, it isn't clear that it is possible to revise the definitions to meet the case-by-case examples of allowed and prohibited examples, while at the same time satisfy the global property of enabling all easily detectable optimisations of certain kinds. Further examination is required, but it may be that beginning from more uniformly derived semantic explanations (such as the approach of Cenciarelli et al [2]) is a better approach for achieving a more tractable definition.

Although we (naturally!) believe that our own explanations of the examples are accurate, the definitions are quite intricate and other people have made mistakes — perhaps some of the corner cases (such as the causality tests) were valid in previous versions of the model but became broken as it evolved. Therefore it is an obvious desire to have a way to check examples formally and automatically against memory model definitions; in future work we plan to investigate model checking techniques for doing this.

Related work.

We have cited the source of examples and made some comparisons in the text; the papers in the bibliography also contain ample further pointers into the wider literature on memory models. The key starting point for understanding the present Java Memory Model is the draft journal paper [7]. During the development of the present memory model, many examples have been discussed on the mailing list, as well as in the test cases [8,9].

Acknowledgements. The authors enjoyed discussions on some of the examples in this paper with M. Huisman, G. Petri and P. Cenciarelli. The second author is supported by a PhD studentship awarded by the UK EPSRC, grant EP/C537068. Both authors also acknowledge the support of the EU project Mobius (IST-15905).

References

[1] David Aspinall and Jaroslav Sevcik. Formalizing Java's data race free guarantee. To appear in Theorem Proving in Higher-Order Logics, 2007 (TPHOLs 07). See the web page at http://groups.inf.ed.ac.uk/request/jmmtheory/., 2007.

[2] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *16th ESOP*, 2007.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*, chapter Memory Model, pages 557–573. Addison-Wesley Professional, July 2005.

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

[5] Jeremy Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, 2004.

[6] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[7] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. Draft journal paper, 2005.

[8] William Pugh and Jeremy Manson. Java memory model causality test cases, 2004. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html.

[9] William Pugh and Jeremy Manson. Java memory model mailing list, 2005. http://www.cs.umd.edu/~pugh/java/memoryModel/archive/.

A Precise JMM definitions

The following definitions correspond to those in [3,6], but are mildly reformulated to match the way we have studied them in [1]. We use $\mathcal{T}$ for the set of thread identifiers, ranged over by t; $\mathcal{M}$ for synchronisation monitor identifiers, ranged over by m; $\mathcal{L}$ for variables (i.e., memory locations), ranged over by v (in examples, `x`, `y`, etc.); and $\mathcal{V}$ for values. The starting point is the notion of *action*.

Definition A.1 [Action] An *action* is a memory-related operation; each action has the following properties: (1) Each action belongs to one thread, denoted $T(a)$. (2) An action has one of the following *action kinds*:

- *volatile read* of $v \in \mathcal{L}$,
- *volatile write* to $v \in \mathcal{L}$,
- *lock* on monitor $m \in \mathcal{M}$,
- *unlock* on monitor $m \in \mathcal{M}$,
- *normal read* from $v \in \mathcal{L}$,
- *normal write* to $v \in \mathcal{L}$,
- *external action*.

An action kind includes the associated variable or monitor. The volatile read, write, lock and unlock actions are called *synchronisation actions*.

Definition A.2 [Execution] An *execution* $E = \langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$, where:

- $A \subseteq \mathcal{A}$ is a set of actions,
- P is a program, which is represented as a function that decides validity of a given sequence of action kinds with associated values if the action kind is a read or a write,
- the partial order $\leq_{po} \subseteq A \times A$ is the program order,
- the partial order $\leq_{so} \subseteq A \times A$ is the synchronisation order,
- $W \in \mathcal{A} \Rightarrow \mathcal{A}$ is a *write-seen* function. It assigns a write to each read action from A, the $W(r)$ denotes the write seen by r, i.e. the value read by r is $V(W(r))$. The value of $W(a)$ for non-read actions a is unspecified,
- $V \in \mathcal{A} \Rightarrow \mathcal{V}$ is a *value-written* function that assigns a value to each write from A, $V(a)$ is unspecified for non-write actions a.

Definition A.3 [Synchronizes-with] In an execution with synchronisation order $\leq_{so}$, an action a *synchronises-with* an action b (written $a <_{sw} b$) if $a \leq_{so} b$ and a and b satisfy one of the following conditions:

- a is an unlock on monitor m and b is a lock on monitor m,
- a is a volatile write to v and b is a volatile read from v.

Definition A.4 [Happens-before] The *happens-before* order of an execution is the transitive closure of the composition of its synchronises-with order and its program order, i.e. $\leq_{hb} = (<_{sw} \cup \leq_{po})^+$.

Definition A.5 [Sequential validity] We say that a sequence s of action kind-value pairs is *sequentially valid* with respect to a program P if $P(s)$ holds.

Definition A.6 [Well-formed execution] We say that an *execution* $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ is *well-formed* if

(i) $\leq_{po}$ restricted on actions of one thread is a total order, $\leq_{po}$ does not relate actions of different threads.

(ii) $\leq_{so}$ is total on synchronisation actions of A.

(iii) $\leq_{so}$ is an omega order, i.e. $\{y \mid y \leq_{so} x\}$ is finite for all x.

(iv) $\leq_{so}$ is consistent with $\leq_{po}$, i.e. $a \leq_{so} b \wedge b \leq_{po} a \implies a = b$.

(v) W is properly typed: for every non-volatile read $r \in A$, $W(r)$ is a non-volatile write; for every volatile read $r \in A$, $W(r)$ is a volatile write.

(vi) Locking is proper: for all lock actions $l \in A$ on monitors m and all threads t different from the thread of l, the number of locks in t before l in $\leq_{so}$ is the same as the number of unlocks in t before l in $\leq_{so}$.

(vii) Program order is intra-thread consistent: for each thread t, the sequence of action kinds and values[9] of actions performed by t in the program order $\leq_{po}$ is sequentially valid with respect to P.

(viii) $\leq_{so}$ is consistent with W: for every volatile read r of a variable v we have $W(r) \leq_{so} r$ and for any volatile write w to v, either $w \leq_{so} W(r)$ or $r \leq_{so} w$.

[9] The *value* of an action a is $V(a)$ if a is a write, $V(W(a))$ if a is a read, or an arbitrary value otherwise.

(ix) $\leq_{hb}$ is consistent with W: for all reads r of v it holds that $r \not\leq_{hb} W(r)$ and there is no intervening write w to v, i.e. if $W(r) \leq_{hb} w \leq_{hb} r$ and w writes to v then[10] $W(r) = w$.

Definition A.7 [Legality] A well-formed execution $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ with happens before order $\leq_{hb}$ is *legal* if there is a "committing" sequence of sets of actions C_i and well-formed "justifying" executions $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i\rangle$ with happens-before $\leq_{hb_i}$ and synchronises-with $<_{sw_i}$, such that $C_0 = \emptyset$, $C_{i-1} \subseteq C_i$ for all $i > 0$, $\bigcup C_i = A$, and for each $i > 0$ the following rules are satisfied:

(i) $C_i \subseteq A_i$.

(ii) $\leq_{hb_i} |_{C_i} \;=\; \leq_{hb} |_{C_i}$.

(iii) $\leq_{so_i} |_{C_i} \;=\; \leq_{so} |_{C_i}$.

(iv) $V_i|_{C_i} = V|_{C_i}$.

(v) $W_i|_{C_{i-1}} = W|_{C_{i-1}}$.

(vi) For all reads $r \in A_i - C_{i-1}$ we have $W_i(r) \leq_{hb_i} r$.

(vii) For all reads $r \in C_i - C_{i-1}$ we have $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$.

(viii) Let's denote the edges in the transitive reduction of $\leq_{hb_i}$ without all edges in $\leq_{po_i}$ by $<_{ssw_i}$. We require that if $x <_{ssw_i} y \leq_{hb_i} z$ and $z \in C_i - C_{i-1}$, then $x <_{sw_j} y$ for all $j \geq i$.

(ix) If x is an external action, $x \leq_{hb_i} y$, and $y \in C_i$, then $x \in C_i$.

Note that although the definition of legality does not mention the term "well-behaved execution" directly, rule 6 of legality ensures that the first justifying execution E_1 is well-behaved.

Definition A.8 [Sequential consistency] An execution is *sequentially consistent* if there is a total order consistent with the execution's program order and synchronisation order such that every read in the execution sees the most recent write in the total order.

Definition A.9 [Conflict] An execution has a *conflicting* pair of actions a and b if both access the same variable and either a and b are writes, or one of them is a read and the other one is a write.

Definition A.10 [DRF] A program is *data race free* if in each sequentially consistent execution of the program, for each conflicting pair of actions a and b in the execution we have either $a \leq_{hb} b$ or $b \leq_{hb} a$.

[10] The Java Specification omits the part "$W(r) = w$", which is clearly wrong since happens-before is reflexive.

The Java Memory Model: a Formal Explanation [1]

M. Huisman[2] G. Petri[3]

INRIA Sophia Antipolis, France

Abstract

This paper discusses the new Java Memory Model (JMM), introduced for Java 1.5. The JMM specifies the allowed executions of multithreaded Java programs. The new JMM fixes some security problems of the previous memory model. In addition, it gives compiler builders the possibility to apply a wide range of singlethreaded compiler optimisations (something that was nearly impossible for the old memory model). For program developers, the JMM provides the following guarantee: if a program does not contain any data races, its allowed behaviours can be described with an interleaving semantics.
This paper motivates the definition of the JMM. It shows in particular the consequences of the wish to have the data race freeness guarantee and to forbid any *out of thin air* values to occur in an execution. The remainder of the paper then discusses a formalisation of the JMM in Coq. This formalisation has been used to prove the data race freeness guarantee. Given the complexity of the JMM definition, having a formalisation is necessary to investigate all aspects of the JMM.

Keywords: Java Memory Model, formalisation, Data-Race-Freeness Guarantee

1 Introduction

With the emergence of multiprocessor architectures, shared memory has shown to be a simple and comfortable communication model for parallel programming that is both intuitive for programmers, and close to the underlying machine model. However, the use of shared memory requires synchronisation mechanisms to keep the memory of the overall system up-to-date and coherent. Such synchronisation mechanisms have a big impact on performance; to avoid these, several relaxations of the consistency (or coherence) of the memory system have been proposed [2,15,13]. However, these relaxations might cause the program to have unexpected behaviours (from the programmer's point of view). In general, the more relaxed the memory system, the harder it is to reason about the programs executing on it.

[1] This work is partially funded by the IST programme of the EC, under the IST-FET-2005-015905 Mobius project, and the French national research organisation (ANR), under the ANR-06-SETIN-010 ParSec project.

[2] Email: `Marieke.Huisman@inria.fr`

[3] Email: `Gustavo.Petri@inria.fr`

A *memory model* defines all the possible outcomes of a multithreaded program running on a shared memory architecture that implements it. In essence, it is a specification of the possible values that read accesses on the memory are allowed to return [4], and thus specifies the multithreaded semantics of the platform.

Java is one of the few major programming languages with a precisely defined memory model [19]. Java's initial memory model allowed behaviours with security leaks [21], and in addition, it prevented almost all singlethreaded compiler optimisations. Therefore, since Java 1.5, a new memory model has been introduced, that fixes these defects. The Java Memory Model (JMM) has been designed with two goals in mind: *(i)* as many compiler optimisations as possible should be allowed, and *(ii)* the average programmer should not have to understand all the intricacies of the model. To achieve the second goal, the JMM provides the following Data Race Freeness (DRF) guarantee: if a program does not contain data races, its allowed behaviours can be described by an interleaving semantics.

To capture basic ordering and visibility requirements on memory operations, the JMM is based on the happens before order [16]. This order inspires the so-called *happens before model*, identifying the actions that necessarily have to happen before any other actions. In other words, this order specifies the updates on the memory that any read must see. Only executions that do not violate this order are allowed. As the guarantees that this model provides are very weak, it allows many singlethreaded compiler optimisations. However, the happens before model allows executions in which values are produced *out of thin air* [5], by using circular justifications of actions, as will be discussed in Section 2. To avoid such circular justifications, and to guarantee DRF, the current version of the JMM is much more complex than the happens before model.

Ample literature about the JMM and related models exists [19,18,22,6], but most descriptions are very dense. In particular, the motivation of the justification procedure that defines legal executions is not extensively explained, i.e., it is unclear why it is defined as it is. This paper tries to overcome this problem by presenting our understanding of the JMM.

In addition, it also presents a formalisation of the JMM in Coq [9]. Our formalisation proves that the DRF guarantee holds for the JMM (a hand-written proof for this is given in [19], which contains some (minor) mistakes). As future work we plan to make a link with the Bicolano formalisation of the Java Virtual Machine (JVM) [20]. This motivates the use of Coq. We also plan to investigate other properties of the JMM formally, in particular the out-of-thin-air (OoTA) guarantee (however, this requires first to state formally what it means to avoid OoTA values).

The rest of this paper is organised is follows. Section 2 explains and motivates the definition of the JMM. Next, Section 3 describes our formalisation. Finally Section 4 describes how we plan to use our JMM formalisation further.

[4] We will in general—in conformance with memory model terminology—simply talk about "the write that a read sees", instead of "the write that writes the value in the memory returned by the read", as in general we do not care about the value, but only about the write action itself.

[5] A value that cannot be deduced from the program.

2 The Java Memory Model

The JMM, as previously mentioned, has two main goals. The first goal is: to allow as many compiler optimisations as possible. This is limited by the second goal: to make the task of multithreaded programming easier for the programmer. It is a well known fact that multithreaded programming is a hard task, and weak memory models further complicate this, in the sense that they add non-determinism to programs (caused by unexpected interleaving of actions). To achieve programmability, the approach of the JMM is that of data race free memory models [3,10,11]; i.e., the DRF guarantee, which reads:

> *Data Race Freeness*: Correctly synchronised programs have sequentially consistent semantics.

The notion of sequential consistency was first defined by Lamport [17]:

> "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

This definition has several consequences. First, the sequence of operations determines that there exists a total order which is consistent with the program order (the order dictated by the sequence of instructions in each thread), and where each read of a memory location sees the last value written to that location. This implies that sequential consistent executions can be described by an *interleaving semantics.* Another important consequence is that the execution has to have a result as if it was executed in a total order, but the actual order of the execution does not need to be total. Thus, compiler optimisations and parallel execution of instructions are allowed, provided that they can be serialised. This makes sequential consistency a very attractive model for concurrency. While sequential consistency has a clear semantics, it is difficult for a compiler to determine statically whether it can or not rearrange intructions or allow parallel execution of operations preserving sequentially consistent semantics. Therefore, many (but not all) common compiler optimisations for sequential code are prevented by this semantics. For that reason, weaker semantics are proposed for programs that are not correctly synchronised (i.e., that contain data races).

A program is said to be correctly synchronised if its sequential consistent executions are free of data races. A data race occurs when two or more threads concurrently access a common memory location, where at least one of the accesses updates the memory. Usually, the presence of data races is considered a bug in the program. In general the result of such races cannot be predicted, thus the programmer must take care to avoid them. In addition, the architecture must provide guarantees to rule out data races. However, notice that sometimes data races are intended, so-called *benign data races*; even though programs with this kind of races are not common.

An important distinction between weak and strong models, is that the former distinguishes between normal memory operations and synchronisation operations, while the second does not. The basic idea is that synchronisation operations induce visibility and ordering restrictions on the other operations in the execution. Furthermore, synchronisation operations in Java are intended to have a sequentially consistent semantics [19], i.e., there exists a total order of synchronisation operations, which is consistent with the program order, and where each synchronisation operation is aware of all the previous synchronisation operations in that order. The presence of such a total order guarantees that all processors see synchronisation operations in the same order (which is not the case for normal memory accesses). This order is called the *synchronisation order* (`so`). Lock and unlock operations on monitors and read and writes of volatile variables, are Java's main synchronisation operations. The order of actions issued by single threads is called the *program order* (`po`). Thus, `po` only relates actions of the same thread, while `so` can relate synchronisation actions of different threads.

The `so` relates synchronisation actions on different locations (either variables or monitors). This is, in general, too restrictive to define which are the minimal visibility conditions for actions. For example, read actions on different volatile variables are necessarily related by `so`, since these are synchronisation actions and `so` is total, but these need not impose restrictions among the threads involved. Therefore a weaker order, derived from the `so`, is defined; namely the *synchronises-with order* (`sw`). This is a per location restriction of the `so`, i.e., it only relates actions on the same location. The intuition of `sw` pairs is that they impose synchronisation of the memory between the intervening threads, in the sense that actions that happen before a synchronised write need to be visible by any other thread that can see that write via a volatile read that variable. The same intuition applies for unlock and lock actions. Thus, only unlock and volatile write actions appear in the source of a `sw` link while only volatile reads and unlock appear in the sink of the link. More precisely, `sw` links relate every volatile write with every subsequent (w.r.t. `so`) volatile read on the same variable, and every unlock with every subsequent (w.r.t. `so`) lock on the same monitor.

The `sw` and the `po` orders allow us to define what constitutes a data race in the JMM, captured by the happens before (`hb`) order. The `hb` order is formally defined as the transitive closure of the union of the `po` and `sw` orders, i.e., it extends the dependency of the `po` between different threads through `sw` edges. We say that two normal actions are conflicting if they operate on the same memory location and at least one of them is an update. An important note is that `so` and `po` must be consistent in JMM executions (i.e., synchronisation operations performed by a single thread must appear in `so` in the same order as they appear in `po`). This guarantees that `hb` is a partial order. A more operational intuition of these orders is that `hb` represents minimal visibility conditions, where `sw` links impose memory synchronisations between threads, and the actions of a single thread are aware of all previous actions (in `po`) by that thread. A *data race* in the JMM occurs when there are two (or more) conflicting actions not related by the `hb` order.

Thus, we can restate the DRF guarantee as:

If every sequentially consistent execution of a program is free of data races, these are all the executions allowed for that program.

We will now see how the `hb` order serves to formally define the basics of the JMM. To start we describe a simpler model, the so-called *happens before memory* *model* (HBMM) [19]. This uses the `hb` order to define which writes a certain read is allowed to see. In the HBMM a normal read `r` can see a write `w` on the same variable provided that[6]: *(i)* $\neg$ (`r` $\xrightarrow{hb}$ `w`), and *(ii)* for all writes `w'` on the same (normal) variable as `r`, $\neg$ (`w` $\xrightarrow{hb}$ `w'` $\xrightarrow{hb}$ `r`); in other words, a normal read can see any write on the same variable that is not related to it by `hb` (thus, forming a data race), or the write on the same variable that immediately precedes it in `hb`.

As mentioned before, volatile variables are meant to have a sequential consistent semantics. Therefore, for volatile variables the HBMM demands each volatile read to see the most recent volatile write on the same variable in the `so`.

Interestingly, the HBMM does not guarantee the DRF property desired for the JMM, as shown in Figure 1 (from [1,19]). In particular, Figure 1(a) shows how a value can be produced out of thin air (an OoTA value). The authors of the JMM decided that such behaviours should be avoided for all programs, thus, also programs with data races should avoid OoTA, one example of this kind of programs is given in Figure 1(b). We do not discuss other examples that motivate the JMM requirements[7], but it should be clear that avoiding OoTA values is one of the most important contributions of the JMM, and it is also the source of much of the complexity of its formal definition.

We will focus on the example in Figure 1(a) to see how the behaviour depicted could happen in the HBMM. First, notice that this program is correctly synchronised, because in every sequentially consistent execution none of the guarded writes is executed, therefore only the default writes are allowed to be seen by both reads. Now we will see that through speculative reasoning a compiler could determine that under the HBMM a behaviour producing an OoTA value is legal. Imagine that a compiler speculates that one of the writes could happen, if afterwards can justify it to happen it could optimise the program to make it happen always. Hence, assuming that any of the writes could happen we can justify such behaviour as follows: *(i)* assume first (w.l.o.g.) that the write of `42` to `x` in Thread 2 could happen, *(ii)* we conclude that the read of `x` in Thread 1 could see a value of `42`. *(iii)* This validates the guard, and justifies the write of 42 to `y` in Thread 1. *(iv)* With a similar reasoning as before the read of `y` in Thread 2 could see that write (reading a `42`) which validates the guard in Thread 2, and justifies the first write of `x` to happen, closing the circular justification cycle (we can see that *(i)* justifies *(i)* at the end). A similar kind of reasoning is applied to Figure 1(b) (but in that example it is more clear that the value `42` appears out of nothing for a programmer). This is the kind of behaviour that the authors of the JMM have named OoTA reads (usually associated to the self-justification of actions) and the actual JMM is designed as a restriction of the HBMM that disallows this kind of circular justifications. To achieve this, the JMM defines a special committing procedure that we explain below.

2.1 Formal Definitions

This subsection serves as a summary where we present the core definitions of the JMM [19], and in addition we give our intuitive understanding of these definitions.

[6] `r` $\xrightarrow{hb}$ `w` stands for: action `r` is ordered before action `w` in the `hb` order.

[7] The motivating examples can be found in [19]

`x == y == 0`

Thread 1	Thread 2

`r1 := x;`
`if (r1 != 0)`
`  y := 42;`
	`r2 := y;`
	`if (r2 != 0)`
	`  x := 42;`

Disallowed: `r1 = r2 = 42`

(a) *Race Free*

`x == y == 0`

Thread 1	Thread 2

`r1 := x;`
`y := r2;`
	`r2 := y;`
	`x := r2;`

Disallowed: `r1 = r2 = 42`

(b) Not Race Free

Fig. 1. Out of Thin Air Examples

The building blocks of the formal definition of the JMM are actions, executions, and a committing procedure to validate actions, which in turn validates complete executions.

Actions Formally, a JMM action is defined as a tuple `<t, k, v, u>` which contains; `t`, the thread identifier of the thread issuing the action; `k` the kind of the action, which can be one of `Read`, `Write`, `Volatile Read`, `Volatile Write`, `Lock`, `Unlock`[8]; `v` the variable involved; and `u` a unique identifier.

Executions With the notion of actions, an execution is defined as a tuple `<P, A, po, so, W, V, sw, hb>` containing: a program `P`; a set of actions `A`; the program order `po`; the synchronisation order `so`; the write seen function, `W`, that for each read action `r` (either normal or volatile) in `A` returns the write action `w` (also in `A`) seen by that read; the value written function, `V`, that for each write `w` (either normal or volatile) in `A` returns the value written `v` by that write; the synchronises-with order, `sw`; and the happens before order, `hb`. Notice that `sw` and `hb` are derived from the `po` and `so` orders.

Well-formedness of Executions The *well-formedness* conditions express basic requirements for valid JMM executions, e.g., *(i)* reads see only writes on the same variable, and volatile reads and writes are issued on volatile variables; *(ii)* `po` and `so` are consistent orders (which guarantees that the `hb` order is a partial order); *(iii)* the restrictions of the HBMM for normal variables as well as for volatile variables apply; *(iv)* `so` respects mutual exclusion, i.e., there is a one at a time semantics for locks; *(v)* the `so` order is of type less or equal to Omega[9]; and *(vi)* executions obey intrathread consistency. This last requirement means that the actions that each thread generates should be those that the singlethreaded semantics of Java (described in the JLS, not taking the JMM into account) would generate, provided that each read `r` of a shared variable returns a value of `V(W(r))`. This is the only link between the singlethreaded semantics and the JMM.

Every Java execution should be well-formed and furthermore, in the commitment procedure only well-formed executions are used to justify actions.

[8] Other secondary actions like thread start are not here.

[9] This means that there are no infinitely decreasing sequences of elements.

Causality Requirements The justification of an execution procedes by committing sets of actions, until every action of the execution is committed. To commit an action we must find an execution that justifies it, obeying certain conditions discussed below. Once an action is committed, it remains committed for the rest of the justification procedure. Thus the justification of an execution is a sequence of pairs of executions and commitment sets, satisfying certain rules. An execution is allowed by the JMM only if such a sequence can be found.

Below we will present the causality rules as stated in [19] with an explanation of our interpretation for each. We will use the restriction notation, both for relations and for functions (domain restriction). Thus for a set S, a relation R, and a function F we define:

- $R|_S = \{(x, y)|x, y \in S \wedge (x, y) \in R\}$ and
- $F|_S(x) = F(x)$ if $x \in S$ and undefined otherwise.

Following [19], we will use C_i to denote the i^{th} commitment set in the sequence, and $E_i = < P, A_i, \overset{po_i}{\rightarrow}, \overset{so_i}{\rightarrow}, W_i, V_i, \overset{sw_i}{\rightarrow}, \overset{hb_i}{\rightarrow} >$ to denote the i^{th} execution. The first pair of the justification sequence contains an empty commitment set, and any execution where every read sees a write that happens before it. The commitment set of the last justification must contains all the actions of the justified execution. It is interesting to note that the last justifying executions needs not to be the same as the execution being justified (this can be seen in rule 5, that only applies to the previous commitment as we shall see). These requirements are captured by the following JMM rules.

- $C_0 = \varnothing$
- $C_i \subset C_{i+1}$
- $A = \bigcup(C_0, C_1, C_2, ...)$

The main idea to prevent the behaviours depicted in Figure 1 is to disallow circular justification of actions. The committing procedure guarantees this by disallowing reads not already committed to see writes that do not happen before them. Moreover, for a read `r` to be able to see a write `w` that is not `hb` related to it, `w` must be committed before `r`, thus, it must be able to happen regardless of whether `r` sees it.

After presenting the rules we show how they disallow the executions in Figure 1.

The first three rules are simple: they require all the committed actions to be present in the justifying execution and the `hb` and `so` orders of the justifying and justified execution to coincide on the committed actions.

1 $C_i \subseteq A_i$

2 $\overset{hb_i}{\rightarrow} |_{C_i} = \overset{hb}{\rightarrow} |_{C_i}$

3 $\overset{so_i}{\rightarrow} |_{C_i} = \overset{so}{\rightarrow} |_{C_i}$

The next rule only refers to write actions (the domain of the `V` function) that are committed; it simply says that whenever a write is committed, the value written cannot change in future justifying executions.

4 $V_i \, |_{C_i} = V|_{C_i}$

For the purpose of presentation we show rule 6 first, followed by rule 5. The main purpose of the causality requirements is to avoid the self justification of actions as in the example of Figure 1(b). A simple way to achieve this would be to require that all reads see only writes that happen before them, motivated by the fact that `hb` contains no cycles as it is a partial order. Requiring this would preclude circular justification of writes, but unfortunately it would also prevent any kind of data race, which is not the intention. To allow data races in the execution the JMM restricts them to the committed actions. For this purpose the next rule requires all uncommitted reads to see writes that happen before them. Furthermore, reads currently being committed must see writes that happen before them.

6 $\forall r \in \mathit{Reads}(A - C_{i-1}) : W_i(r) \overset{hb_i}{\to} r$

The following rule refers to reads committed in the previous justification (if there were any). It says that when a read is not yet committed, it can see a write different from the one it sees in the justified execution. Moreover, when a read is being committed it can read a different write, but in the following justification step it must read the write of the justified execution. Recall from the previous rule that reads being committed must see writes that happen before them. If rule 5 was given on the current commitment set C_i, instead of the previous one C_{i-1}, all reads in the final execution should see writes that happen before them, completely disallowing the presence of data races. To avoid this the rule is expressed over reads committed in the previous execution (which could be involved in data races, since they are already committed). In general, to commit data races in the JMM, the sequence starts by committing the writes involved in the data race. Clearly, for any read involved in a data race two writes must be committed as required by rule 7 in conjunction with rule 6; one that is not ordered by `hb` in the final execution (that forms the data race), and one that happens before (required by rule 6). We commit the read seeing the write that happens before it; and then, in the following commitment step we make it see the write that corresponds to the justified execution (i.e., the write involved in the data race) as required by the following rule.

5 $W_i\,|_{C_{i-1}} = W|_{C_{i-1}}$

The following rule states that whenever a read is committed, both the write that it reads in the execution being justified (E) and in the justifying execution (E_i) must be already committed. As mentioned when explaining rule 5, when committing a read action `r`, it can see a different value in the justifying execution than what it sees in the execution justified. But, to guarantee that the write that `r` actually sees does not depend on the return value of `r`, we must show that that write can be committed without assuming that `r` saw it. Therefore, part (b) of this rule requires the write that `r` finally sees to be already committed. Furthermore, part (a) requires the write that `r` sees while being committed (and which happens before it, by rule 6) to be also committed. This guarantees that the read happens independently from the write it eventually sees.

7 (a) $\forall r \in \mathit{Reads}(C_i - C_{i-1}) : W_i(r) \in C_{i-1}$
(b) $\forall r \in \mathit{Reads}(C_i - C_{i-1}) : W(r) \in C_{i-1}$

Rules 2, 5, 6, and 7 are the most important rules to dissallow circular justifications of actions. Rule 7 forces that whenever a read is committed, the writes it sees in the justifications and the justified execution must already be committed, and rule 6 mandates that non-committed reads can only see writes that happen before them. Thus, this prevents circular read-write sequences in justifying executions. Rule 2 restricts `hb` links that have been committed to remain committed (and to be coherent with the justified execution), avoiding possible circularities (since `hb` is a partial order). Finally rule 5 allows read-write data races to be justified, provided these are not circularly justified.

The next rule only constraints synchronisation actions. The basic idea for this rule is that once synchronisation links have been used to commit actions those links must (transitively) remain. The `ssw` links are defined to be the `sw` links present in the transitive reduction of the `hb` order that are not in `po`. The intuition behind these links is that they extend the `hb` relation across threads. Following rule 6, a read `r` is allowed to see a write `w` that happens before it without being committed. But if after committing some actions that depend on such a read, the `sw` link that extended the `hb` relation among them was not required to be present in following executions, the `hb` link could also disappear. Therefore the actions committed based on this assumption would not be correct anymore. A more operational intuition is that if an action is justified, assuming that a synchronisation of the memory was issued between two threads, then that synchronisation should be performed[10].

8 $\forall x, y \in A_i, z \in (C_i - C_{i-1}) : x \overset{ssw_i}{\rightarrow} y \overset{hb_i}{\rightarrow} z \Rightarrow (\forall j \geq i : x \overset{sw_j}{\rightarrow} y)$

The last rule requires *external* actions, which are actions observable outside of an execution, to be committed before any action that happens after them. Typical examples of external actions are printing values, or communications with other processes. External actions are used to define the observable behaviour of executions. In turn this is used to define the allowed optimisations: optimisations are only allowed if the observable behaviour of the program is the same.

9 $\forall x, y \in A_i : \mathit{External}(x) \wedge y \in C_i \wedge x \overset{hb_i}{\rightarrow} y \Rightarrow x \in C_i$

Let us see now how the rules prevent the executions discussed in Figure 1. For the first example (Figure 1(a)) in order to get both reads to see the writes of `42` in the other thread, we need to have both writes performed (i.e., committed). Moreover, by rule 7, they have to be committed before any of the reads is allowed to see them in a committing step. So we know that apart from the default writes to `x` and `y`, the first actions that have to be committed are the writes of `42`. In addition, following rule 6, the reads in both threads are only allowed to see default writes (the only writes that happen before them in the execution). Now, we argue that it is impossible to find a justifying execution where both reads see the default writes and any of the writes of `42` happen, as this would be a violation of the intrathread consistency requirement of well-formed executions. Therefore there is no execution that makes it possible to commit a write of `42`, and thus the result is prevented. For the second example (in Figure 1(b)) the reasoning is quite similar. The writes must be committed first, and there is no intra-thread consistent execution where

[10] Note that with this description we are trying to explain the motivation of the rule as we understand it. This does not mean that we think that the rule achieves it purpose as stated. In fact, we think that the statement of the rule is not correct, though the intuition behind it should be as described.

the reads see writes that happen before them, that allows a write of `42` to appear. Therefore, the result is forbidden.

2.2 Some Problems of the Java Memory Model

During our study and formalisation of the JMM we and others realised that there are several problems with its definition. This section summarises the most important issues.

First of all, the formal definition of the model is hard to understand and to use. This is partly due to the lack of motivation for the rules that validate the executions, and partly to the non-constructiveness of the committing procedure itself. The latter makes it hard to reason about the possible interactions between the rules. The lack of motivation for the rules is reflected by the fact that there is no formal definition of the out-of-thin-air guarantee, and no proofs are given to show that the rules actually enforce it.

On a more concrete side, some inconsistencies have been found; we will briefly mention some of these. First, Cenciarelli et al. [6] found a counter example to the claim in Theorem 1 in [19], which says that every two adjacent instructions that are independent can be reordered. In addition, Aspinall and Sevcik [4] have pointed out that the Omega type restriction to the `so` order prevents executions with infinitely many volatile instance variables, since the initialisation of all of these must come before the first synchronisation action of any thread. Thus infinitely many actions have to be ordered before the first synchronisation action of any thread, breaking the Omega type restriction. As we shall see, this has an impact on our formalisation.

With respect to the OoTA property, we still find it hard to argue whether some reads constitute an OoTA violation or not. Furthermore, Aspinall and Sevcik [4] have shown that many of the executions required to be accepted by the JMM [8] are not allowed. We think that the definition of OoTA needs to be clarified and revised.

3 Formalisation

The issues exposed above indicate that there is a real need for a formalisation of the JMM. In addition, another motivation for this formalisation emerges as a requirement for the Mobius project. The Mobius project aims at developing a proof carrying code (PCC) infrastructure for the specification and verification of mobile Java programs. For this purpose, program logics for Java source and bytecode are developed within the project. This bytecode logic is proven correct w.r.t. Bicolano, a Coq formalisation of the Java bytecode operational semantics [20]. We have extended the Bicolano semantics to an interleaving semantics, called BicolanoMT, augmented with multithreaded instructions. This semantics will be used to prove the correctness of the Mobius program logic for correctly synchronised multithreaded Java programs. To motivate that is sufficient to prove soundness w.r.t. an interleaving semantics we plan to prove that the BicolanoMT semantics is in a one-to-one correspondence with the executions allowed by the JMM for correctly synchronised programs. This motivates a formally mechanised specification of the JMM and in particular a proof of the DRF guarantee.

As expected, some of the problems mentioned above appeared while formalising the model and influenced our formalisation. We will present a general overview of the model and we will explain the most important differences between our formalisation and the official JMM definitions [19]. For our formalisation we used the Coq proof assistant, which will allow us to integrate our model with Bicolano.

3.1 Brief Description of the Formalisation

To be able to prove the DRF guarantee mechanically we formalised the JMM as close as possible to its definition [19]. With this we expected to avoid problems derived from misunderstandings or biased interpretations. We did not have to formalise the whole JMM specification to prove the DRF guarantee. Therefore, we limited our formalisation to the needed definitions, except for the causality rules, that are completely formalised. However, some of them are currently not used in any proof. The kind of actions we formalised only contain volatile and normal reads and writes, lock and unlock actions; currently we do not include thread start, thread termination and other actions that are not important to prove the DRF guarantee. However, we believe that these could be easily added without changing the current proofs, as we plan to do.

Further, we have defined programs, values, variables, thread identifiers and unique identifiers as abstract data types, and we have axiomatised their interactions (e.g., two different actions have different unique identifiers). The data types for actions and executions are axiomatised using the Coq module system, in a way that is very similar to their description in the JMM paper, except for little technical details (e.g., the `sw` order is not part of the execution data type, since it can be derived from the `po` and `so` orders, which are parameters of the execution).

An interesting challenge is the specification of the commitment procedure. For this we used an abstract function that, given a program, an execution and a proof that the execution belongs to the allowed executions for that program, returns a list containing the sequence of pairs of commitment set and commitment execution, that justify the execution.

The use of Coq lists to represent the commitment procedure limits it to be finite, but in the JMM definition justifications could be infinite. This choice was made only to simplify the first version of the proofs as we can use induction to prove facts about finite lists. To cover the full semantics of the JMM we plan to replace current lists for streams which allow for infinite sequences, and where the proofs given by induction must be replaced for proofs by coinduction. We expect that this will not have any significant impact on the current proofs. In the rest of the formalisation, we did not assume finiteness.

The (causality) rules are axiomatised using a justifying list, as discussed above. To give a feeling what the formalisation looks like, here is the formalisation rule 2 of the causality requirements in [19].

```
Definition req2 :=
   ∀ E_exec_P j x y, In (justification E_exec_P) j →
         eIn (comm j) x → eIn (comm j) y →
         hb E x y ↔ hb (exec j) x y.
```

In this definition the function `justification` is the abstract function that returns the list containing the pairs of commitment set and committing execution. The bound variable `j` stands for a committing pair, and the predicate `In (justification E_exec_P) j` states that `j` is in the committing sequence of the execution `E`. The functions `comm` and `exec` extract from a committing pair the corresponding set and execution, respectively. Thus, the rule says that if both actions `x` and `y` are committed in `j`, they are related by `hb` identically in the justifying execution of `j` and in the justified execution.

In the following we present the statement of the main theorems and the assumptions we needed for their proofs. For each assumption, we justify how we plan to prove them in future versions of the formalisation.

The DRF proof sketched in [19] is separated in a lemma and the main theorem. We have followed their reasoning as close as possible. For both proofs we assume we have a correctly synchronised program P and an execution `E` of the program `P`. That is stated in our Coq formalisation as follows:

```
Hypothesis p_well_synch:  correctly_synchronized P.
Hypothesis E_exec_P: Program_Executions P E.
```

The definition of the `correctly_synchronized` predicate states formally that all sequentially consistent executions of the program `P` are free of data races. The `Program_Executions` predicate states that `E` belongs to the set of allowed executions for `P`, which is axiomatically defined in the model.

The lemma (Lemma 2 in [19]) takes as hypothesis that read actions only see writes that happen before them. This hypothesis is later proved in the theorem, which concludes the DRF proof using the lemma. This hypothesis is stated in our formalisation by the following definition.

```
Definition reads_see_hb (E:Exec):
   ∀ r act_r r_read, hb E (W E r act_r r_read) r.
```

To understand the definition above it is important to note that the `W` function takes four parameters. The first parameter `E` is the execution, the second parameter `r` is the read action whose write we are trying to obtain; the other two parameters are related to our choice for modelling partial functions in Coq. Our approach was that of "preconditions for partial functions" as in [5], where a proof must be given showing that the argument belongs to the domain of the function to apply it. In this particular case, the parameters `act_r` and `r_read` are proofs that the action `r` is an action of the execution `E` and that `r` is a read action, respectively.

Now we focus on additional assumptions we had to make, as these where left implicit in the original proofs of [19]. We expect to be able to prove each of these assumptions we added, and we will explain how to do it in each case. Both proofs of the lemma and the main theorem go by absurdity. Assuming that there is some non sequentially consistent execution of the correctly synchronised program `P`, a sequentially consistent execution that contains a data race can be constructed, contradicting the correct synchronisation hypothesis of `P`. To show a sequentially consistent execution with a data race a topological sort of the `hb` order is used, which is described in our formalisation as:

```
Definition to := top_sort (hb E) (hb_partial_order E).
```

The `top_sort` predicate is the axiomatisation of the particular sort needed. With `to` in hand, the proof of the lemma proceeds by identifying the first occurrence of an intromission of a write `w` between a read `r` and the write that it sees (`W E r act_r r_read`) in that `to` order. To guarantee that such a read exists we need to assume that `to` is well founded, and then the proof goes by well founded induction. This is not proved in the JMM paper. This is in fact true if we prove that the `po` is well founded and we assume that `so` has type Omega or less (as we said before, default actions should be revised to do guarantee this for `so`). The proof of this is not currently formalised, but it should be proved using these hypothesis. We plan to formalise it in following versions of our model. A simpler approach could be that of Aspinall and Sevcik [4], where they assume that the executions are finite, where well foundedness of `to` is trivial (every finite order is well founded).

```
Hypothesis wf_to:  well_founded to.
```

Finally, one problem that we encountered formalising the JMM is that the papers proof assumes that for any sequentially consistent "prefix" of an execution, there must exist a sequentially consistent continuation for it. This fact is not entirely true, since new default writes of variables could be needed (or discarded) as we take a different branch of the program. As default writes synchronise with the first action of any thread, actions should be added at the beginning (before any thread start action) in the `to`. To overcome this problem some revision on the definition of default write actions should be proposed. There are several alternatives to fix this problem; e.g., developing a different mechanism for initialising variables, or a different formulation of the sequential continuation property (e.g., subsegment starting from the first thread start action). We postponed this decision for later revisions of the formalisation and assume it correct as currently stated in the JMM.

With this assumption we have to show that there is always a sequential consistent continuation. To exhibit such continuation an interleaving of the instructions of the threads from the program point where they last executed should complete a sequential consistent continuation of the execution. The difficulty appears when giving such continuations at the level of the JMM formalisation, where nothing is said about instructions and programs, but only about actions. For this reason we added that fact as a hypothesis and expect to prove it when we establish a proper link between programs and executions for the JMM.

```
Hypothesis seq_continuation:
   Program_Executions P E → first_w_r_intromission to r w →
   ∃ E', Program_Executions P E' ∧
         eq_exec_complete_upto lower_r E E' ∧
         (∀ acr_r', W E' r act_r' r_read = w) ∧
         sequential_consistent E'.
```

The intuition of the definition above is that; given an execution `E` and the first occurrence of a read write race, such that the write `w` is the last write in `to` to the same variable as the read `r` ordered before `r` in to, and such that `r` does not see that write; then, there should be another execution of `P`, `E'` where the read `r` sees the last write in the `to`, `w`, and that `E'` is equal to `E` when restricted to the actions that come before `r` in `to`. The details of the definition are not simple, but the intuition is; it only says that there is a sequential consistent continuation of the prefix.

We expect to be able to prove this very intuitive result, but a link between the program instructions and the actual actions should be developed first. Several authors [6,4,22] agree that there is no good link between the semantics of Java as defined in the JLS [12] (disregarding the JMM itself) and the JMM (as in JSR-133 [14]). Currently this link is given at a very high level by the requirement of intra-thread consistency of the well-formedness requirements. Different approaches have been taken, which include an approach towards operational semantics proposed by Cenciarelli et al. [6], and a function that verifies the validity the semantics of each thread, proposed by Aspinall and Sevcik [4]. We plan to make this link for correctly synchronised programs only, using the BicolanoMT interleaving semantics.

With all hypotheses mentioned in context, the Coq formalisation of the theorems from the JMM paper are simple.

```
Theorem Lemma2:  ∀ E, reads_see_hb E → sequential_consistent E.
Theorem DRF: sequential_consistent E.
```

Summing up, to finish our formalisation, a proof that the `to` has order type Omega or less should be developed; and a link between actions and instructions should be given for correctly synchronised programs. For these proofs the initialisation of actions need to be fixed. We plan to do this in a following version.

The current version of the model occupies 3000 lines in Coq, and has been of great use to help understanding the model, as well as to identify problems in the definitions and underspecified assumptions. It is in our plans to use the formalisation to further explore facts about the JMM, for example the OoTA property.

4 Conclusions & Future Work

In this paper we have presented a formalisation of the Java Memory Model, using the Coq theorem prover, and we have formally proved the DRF guarantee, as claimed by the authors of the JMM. Because of the different requirements for the JMM (allowing many compiler optimisations, avoiding security problems caused by out-of-thin-air values, and easy programmability), the formal definition is fairly complex, and needs a formalisation to reason about it. Also, as part of the formalisation process, we have gained a good insight in the motivation of the rules that define the validity of executions, and we have tried to convey our insights, hopefully being more intuitive than the description of the rules as presented in [19].

In the near future, we plan to exploit the DRF guarantee, by linking the JMM formalisation with BicolanoMT. We will define a mapping from BicolanoMT programs to the program model, as used for the JMM. Then we will prove that programs that are correctly synchronised in BicolanoMT semantics, are also correctly synchronised according to JMM semantics. Moreover, for correctly synchronised programs, BicolanoMT executions are in a one to one relationship with JMM executions.

We also plan to study the OoTA guarantee more precisely. The authors of the JMM claim that it does not allow any out-of-thin-air values to be produced, but they do not give a formal definition. We would like to give a more formal definition for this property, and then verify whether the JMM actually respects it.

In the more long term, we also would like to study modularity of the JMM: if a part of a program is data race free, can the behaviour of that part then described with an interleaving semantics. This property is essential to support compositional verification. We are also interested in studying whether there are so-called benign data races, i.e., data races that still result in sequentially consistent executions. Finally, we are also interested in proving formally that a compiler optimisation is allowed by the JMM, by showing that the program transformation does not change the set of legal executions.

Acknowledgement

We would like to thank Jaroslav Sevcik and David Aspinall whose discussions helped to improve our understanding of the model. We also thank Clement Hurlin and the reviewers that contributed to the presentation of this work.

References

[1] S. V. Adve. *Designing memory consistency models for shared-memory multiprocessors.* PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 1993.

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

[3] S.V. Adve and M.D. Hill. Weak ordering - a new definition. In *International Symposium on Computer Architecture*, pages 2–14, 1990.

[4] D. Aspinall and J. Ševčík. Formalising Java's data-race-free guarantee. Technical report, LFCS, School of Informatics, University of Edimburgh, 2007. To appear.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[6] P. Cenciarelli, A. Knapp, and E. Sibilo. The Java memory model: Operationally, denotationally, axiomatically. *European Symposium on Programming*, 2007.

[7] Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from http://mobius.inria.fr, 2006.

[8] The JSR 133 Consortium. The Java memory model causality test cases, 2004. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html.

[9] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, mars 2004. http://coq.inria.fr/doc/main.html.

[10] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, Stanford University, 1995.

[11] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories (extended abstract). In *ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, 1991.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, third edition.* The Java Series. Addison-Wesley Publishing Company, 2005.

[13] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, The University of Calgary, 1998.

[14] JSR-133: Java Memory Model and Thread Specification, 2004.

[15] J. Kawash. *Limitations and capabilities of weak memory consistency systems.* PhD thesis, The University of Calgary, 2000.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer*, 28(9):690–691, 1979.

[18] J. Manson. *The Java Memory Model*. PhD thesis, Faculty of the Graduate School of the University of Maryland, 2004.

[19] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Principles of Programming Languages*, pages 378–391, 2005.

[20] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inia.fr/bicolano`. Summary appears in [7], 2006.

[21] W. Pugh. The java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

[22] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–172, 2007.