A Programming Model for Concurrent Object-Oriented Programs

BART JACOBS, FRANK PIESSENS, and JAN SMANS
Katholieke Universiteit Leuven
K. RUSTAN M. LEINO and WOLFRAM SCHULTE
Microsoft Research

Reasoning about multithreaded object-oriented programs is difficult, due to the non-local nature of object aliasing and data races. We propose a programming regime (or *programming model*) that rules out data races, and enables local reasoning in the presence of object aliasing and concurrency. Our programming model builds on the multithreading and synchronization primitives as they are present in current mainstream programming languages. Java or C# programs developed according to our model can be annotated by means of stylized comments to make the use of the model explicit. We show that such annotated programs can be formally verified to comply with the programming model. If the annotated program verifies, the underlying Java or C# program is guaranteed to be free from data races, and it is sound to reason locally about program behavior. Verification is modular: a program is valid if all methods are valid, and validity of a method does not depend on program elements that are not visible to the method. We have implemented a verifier for programs developed according to our model in a custom build of the Spec# programming system, and we have validated our approach on a case study.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.4 [**Software Engineering**]: Software/Program Verification—*Class invariants*; *Correctness proofs*; *Formal methods*; *Programming by contract*; F.3.1 [**Logics and Meanings of programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions*; *Invariants*; *Logics of programs*; *Mechanical verification*; *Pre- and post-conditions*; *Specification techniques*

General Terms: Verification
Additional Key Words and Phrases: Aliasing, data races, local reasoning, modular reasoning, ownership, verification condition generation

1. INTRODUCTION

Writing correct multithreaded software in mainstream languages such as Java or C# is notoriously difficult. The non-local nature of object aliasing and data races makes it hard to reason about the correctness of such programs. Moreover, many assumptions made by developers about concurrency are left implicit. For instance, in Java, many objects are not intended to be used by multiple threads, and hence it is not necessary to perform synchronization before accessing their fields. Other objects are intended to be shared with other threads and accesses should be synchronized, typically using locks. However, the program text does not make explicit

http://doi.acm.org/10.1145/1452044.1452045

if an object is intended to be shared, and as a consequence it is practically impossible for the compiler or other static analysis tools to verify if locking is performed correctly.

We propose a programming regime (or *programming model*) for concurrent programming in Java-like languages, and the design of a set of program annotations that make the use of the programming model explicit. For instance, a developer can annotate their code to make explicit whether an object is intended to be shared with other threads or not. These annotations provide sufficient information to static analysis tools to verify if locking is performed correctly: shared objects must be locked before use, unshared objects can only be accessed by the creating thread. Moreover, the verification can be done modularly, hence verification scales to large programs. We describe a particular modular verification approach, based on generating verification conditions suitable for discharge by an automatic theorem prover.

Several other approaches exist to verify race- and deadlock-freedom for multi-threaded code. They range from generating verification conditions [Detlefs et al. 1998; Flanagan et al. 2002; Freund and Qadeer 2004; Flanagan et al. 2005; Qadeer et al. 2004; Ábrahám-Mumm et al. 2002; Rodríguez et al. 2005], to type systems [Boyapati et al. 2002; Flanagan and Qadeer 2003]. (See Section 7 for an overview of related work.) Our approach is unique in proposing a way to reason about concurrent code that enables abstraction through invariants and ownership and that enables advanced forms of ownership transfer, such as appending two linked lists.

The contributions of this paper are as follows:

—We present a programming model and a set of annotations for safe concurrent programming in Java-like languages.

—Following our programming model ensures absence of data races.

—The generated verification conditions allow sound local reasoning about program behavior.

—We have prototyped a verifier as a custom build of the Spec# programming system [Barnett et al. 2004; Barnett et al. 2006], and in particular its program verifier for sequential programs.

—Through a case study we show the model supports useful, non-trivial programs and we assess the annotation overhead.

The present approach evolved from [Jacobs et al. 2005a], [Jacobs et al. 2006], and [Jacobs 2007]. It improves upon this prior work by adding a formalization of the approach with invariants and ownership. (A soundness proof [Jacobs et al. 2005b] accompanies [Jacobs et al. 2005a] but it does not formalize verification condition generation, and it does not formalize or prove the method effect framing approach.) As did the prior work, the present approach builds on and extends the Spec# programming methodology [Barnett et al. 2004] that enables sound reasoning about object invariants in sequential programs. For brevity, we omit the description of the deadlock prevention approach and the approach for verification of immutable objects, static fields, and lazy class initialization [Jacobs 2007] from this article.

The rest of the paper is structured as follows. We introduce the methodology in two steps. The model of Section 2 prevents low-level data races on individual fields. In this first model, a shared object's lock protects only its own fields. Section 3

adds prevention of races on data structures consisting of multiple objects through an object invariants and ownership system. Here, a shared object's lock protects its own fields as well as those of the objects it transitively owns. Section 4 deals shortly with lock re-entry. The remaining sections discuss experience (Section 5), limitations (Section 6), and related work (Section 7), and offer a conclusion (Section 8). Table II at the end of the article summarizes the notations used.

2. PREVENTING DATA RACES

In this section, we present our programming model and associated modular static verification approach for verification of the absence of data races in Java-like programs.

A data race occurs when multiple threads simultaneously access the same variable, and at least one of these accesses is a write access. Data races are usually programming errors, since they are a symptom of a lack of synchronization between concurrent operations on a data structure. Developers can protect data structures accessed concurrently by multiple threads by associating a mutual exclusion lock with each data structure and ensuring that a thread accesses the data structure only when it holds the associated lock. However, mainstream programming languages such as Java and C# do not force threads to acquire any locks before accessing data structures, and they do not enforce that locks are associated with data structures consistently.

A simple strategy to prevent data races is to lock every object before accessing it. Although this approach is safe, it is not used in practice since it incurs a major performance penalty, is verbose, and is prone to deadlocks. Instead, standard practice is to only lock the objects that are effectively shared between multiple threads. However, it is hard to distinguish shared objects (which should be locked) from unshared objects based on the program text. As a consequence, without additional annotations, a compiler cannot enforce a locking discipline where shared objects can only be accessed when locked.

An additional complication is that in order to decide whether a particular field access is correctly protected by a lock, it is not sufficient to inspect the method that contains the field access; indeed, the lock that protects the field might be acquired by the method's direct or transitive callers instead of by the method itself. Therefore, method contracts that specify which locks are held on entry are required for modular verification.

In this section, we describe a simple version of our approach that deals with data races on the fields of shared objects. The next section develops this approach further to deal with high-level races on multi-object data structures.

The approach is presented in two steps. First (Section 2.1), a *programming regime* (or *programming model*) that prevents data races is presented without regard to modular static verification. It is proven that if a program complies with the programming model, then it is data-race-free. In the second step (Section 2.2), the modular static verification approach is presented.

2.1 Programming model

This section presents the programming model, without regard to modular static verification. Section 2.1.1 describes the model informally. Section 2.1.2 formalizes

the syntax for a subset of Java augmented with the annotations required by the approach. Section 2.1.3 defines a small step semantics for execution of programs in this language, which tracks the extra state variables (specifically, access sets and shared sets) required by the programming model. The section also defines a set of *legal program states* that comply with the programming model. Section 2.1.4 proves that programs that reach only legal program states are data-race-free.

2.1.1 *Informal Description.* We describe our programming model in the context of Java, but it applies equally to C# and other similar languages. In the following sections, we formalize the approach with respect to a formally defined core subset of Java.

In our programming model, accesses to shared objects are synchronized using Java's **synchronized** statement. A thread may enter a **synchronized** (o) block only if no other thread is executing inside a **synchronized** (o) block; otherwise, the thread waits. In the remainder of the paper, we use the following terminology to refer to Java's built-in synchronization mechanism: when a thread enters a **synchronized** (o) block, we say it *acquires o's lock* or, as a shorthand, that it *locks o*; while it is inside the block, we say it *holds o's lock*; and when it exits the block, we say it *releases o's lock*, or, as a shorthand, that it *unlocks o*. Note that, contrary to what the terminology may suggest, when a thread locks an object, the Java language prevents other threads from locking the object but it does not prevent other threads from accessing the object's fields. This is the main problem addressed by the proposed methodology. While a thread holds an object's lock, we also say that the object *is locked* by the thread.

An important terminological point is the following: when a thread tid's program counter reaches a **synchronized** (o) block, we say the thread *attempts to lock o*. Some time may pass before the thread *locks o*, specifically if another thread holds o's lock. Indeed, if the other thread never unlocks o, tid never locks o. The distinction is important because our programming model imposes restrictions on attempting to lock an object.

Our programming model prevents data races by ensuring that no two threads have access to a given object at any one time. Specifically, it conceptually associates with each thread an *access set*, which is the set of objects whose fields the thread is allowed to read or write at a given point, and the model ensures that no two threads' access sets ever intersect. Access sets can grow and shrink when objects are created, objects are shared, threads are created, or when a thread enters or exits a **synchronized** block. Note that these access sets do not influence program behavior and therefore need not be tracked by a virtual machine implementation: we use them to explain the programming model, and to implement the static verification.

—**Object creation.** When a thread creates a new object, the object is added to the creating thread's access set. This means the constructor can initialize the object's fields without acquiring a lock first. This also means single-threaded programs just work: if there is only a single thread, it creates all objects, and it can access them without locking.

—**Object sharing.** In addition to access sets, our model introduces an additional global state variable called the *shared set*, which is a set of object identities. We

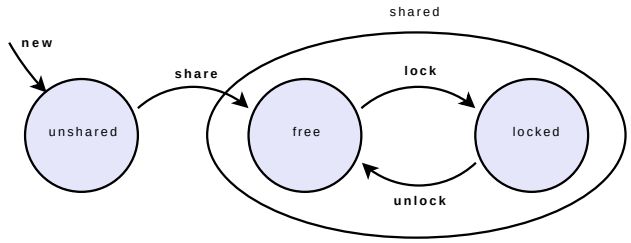

Fig. 1. The three states of an object.

call the objects in the shared set *shared* and objects in the complement *unshared*. The shared set, like the access sets, is conceptual: it is not present at run time, but used to explain the model and implement the verification.

A new object is initially unshared. Threads other than the creating thread are not allowed to access its fields. In addition, no thread is allowed to attempt to lock an unshared object: our programming model does not allow a **synchronized** (o) {...} operation unless o is shared. In our programming model, objects that are not intended to be shared are never locked.

If, at some point in the code, the developer wants to make the object available for concurrent access, they have to indicate this through an annotation (the **share** o; annotation). From that point on, the object o is shared, and threads can attempt to acquire the object's lock. At the point where an object is shared, the object is removed from the creating thread's access set and added to the shared set. If, subsequent to this transition, any thread, including the creating thread, wishes to access the object, it must acquire its lock first.

Once shared, an object can never revert to the unshared state.

—**Thread creation.** Starting a new thread transfers the accessibility of the receiver object of the thread's main method (i.e. the *Runnable* object in Java, or the *ThreadStart* delegate instance's target object in the .NET Framework) from the starting thread to the started thread. This is necessary since otherwise, the thread's main method would not be allowed to access its receiver.

—**Acquiring and releasing locks.** At the point where an object becomes shared, it is removed from the creating thread's access set and added to the shared set. Since the object is now not part of any thread's access set, no thread is allowed to access it. To gain access to such a shared object, a thread must lock the object first. When a thread locks an object, the object is added to the thread's access set for the duration of the **synchronized** block.

As illustrated in Figure 1, an object can be in one of three states: *unshared*, *free* (not locked by any thread and shared) or *locked* (locked by some thread and shared). Initially, an object is unshared. Some objects eventually undergo a **share** operation (at a program point indicated by the developer). After this operation, the object is not part of any thread's access set and is said to be *free*. To access a free object, it must be locked first, changing its state to locked and adding the object to the locking thread's access set. Unlocking the object removes it from the access set and makes it free again.

Let's summarize. Threads are only allowed to access objects in their corresponding access set. A thread's access set consists of all objects it created or whose lock it acquired, plus the receiver of its main method (if any), minus the objects on which it subsequently performed a share operation or which it used as the target in a thread creation, or whose lock it released. Our programming model prevents data races by ensuring that access sets never intersect.

2.1.1.1 *Lock re-entry.* Java's **synchronized** blocks are re-entrant; that is, if a thread already holds an object o's lock, then another, nested attempt by the same thread to enter a **synchronized** (o) block succeeds immediately. For simplicity, we rule out lock re-entry in the programming model. However, in Section 4, we show how our approach can be extended to support lock re-entry.

2.1.2 *Programs.* To formalize the rules imposed by our programming model, we first define a small language consisting of a subset of Java (minus static typing) plus two kinds of annotations (indicated by the gray background): share statements and method contracts. Its syntax is shown in Figure 2. An example program in this language is shown in Figure 3. We discuss the language and define *well-formedness* of programs.

$\mathcal{C}$	class names	$\mathcal{M}$	method names	Φ	logical formulae
$\mathcal{I}$	interface names	$\mathcal{F}$	field names	$\underline{\mathcal{O}}$	object references
		$\mathcal{X}$	variable names	$\underline{\Theta}$	thread-relevant states

$$C \in \mathcal{C}, I \in \mathcal{I}, m \in \mathcal{M}, f \in \mathcal{F}, x \in \mathcal{X}, \varphi \in \Phi, \underline{o \in \mathcal{O}, \theta \in \Theta}$$

$$\begin{array}{rcl}
\pi & ::= & \langle \mathit{iface} \mid \mathit{class} \rangle^* \ s^* \\
\mathit{iface} & ::= & \textbf{interface}\ I\ \{\ mh^*\ \} \\
mh & ::= & m(x^*)\ \ \textbf{requires}\ \varphi;\ \textbf{ensures}\ \varphi; \\
v & ::= & \mathsf{null} \mid \underline{o} \\
g & ::= & \textbf{this} \mid x \mid v \\
\mathit{class} & ::= & \textbf{class}\ C\ \textbf{implements}\ I^*\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
\mathit{field} & ::= & f; \\
\mathit{meth} & ::= & mh\ \{\ s^*\ \} \\
\tau & ::= & C \mid I \\
s & ::= & \textbf{if}\ (g = g)\ \{\ s^*\ \}\ \textbf{else}\ \{\ s^*\ \} \mid \textbf{assert}\ g\ \textbf{instanceof}\ \tau; \\
 & & \mid x := g.f; \mid g.f := g; \mid x := \textbf{new}\ C; \mid x := g.m(g^*); \mid \textbf{start}\ g.m(); \\
 & & \mid\ \textbf{share}\ g; \mid \textbf{synchronized}\ (g)\ \{\ s^*\ \} \\
 & & \mid \textbf{return}\ g; \mid \underline{x := \textbf{receive}\ [\theta]\ o.m(v^*);} \mid \underline{\textbf{unlock}\ o;}
\end{array}$$

Fig. 2. Syntax of a small Java-like language without static typing, but with two kinds of annotations (indicated by the gray background): method contracts and share statements. The underlined elements appear only as part of continuations during program execution (see Section 2.1.3) and are not allowed to appear in well-formed programs. The syntax of the logical formulae used in method contracts is given in Section 2.2.

A program π consists of a number of classes and interfaces and a main routine. A class may declare zero or more fields, and both classes and interfaces may declare zero or more methods. Interface methods consist of a header only, whereas class methods consist of a header and a body. Each method header includes a

```
class Counter { count; }

class Session {
  counter;
  run()
    requires L_t = ∅ ∧ this ∈ A;
  {
    /* wait for event */;
    c := this.counter;
    assert c instanceof Counter;
    synchronized (c) { n := c.count; c.count := n + 1; }
    this.run();
  }
}

c := new Counter;
share c;
s := new Session; s.counter := c; start s.run();
s := new Session; s.counter := c; start s.run();
```

Fig. 3. An example program in the formal syntax of Figure 2. (Note: the example also uses integer values and integer operations; these are omitted from the formal development for simplicity.) Method *run*'s precondition states that the thread's lockset is empty and that the receiver is in the thread's access set. The syntax and semantics of method contracts is detailed in Section 2.2.

method contract, consisting of a *precondition* (**requires** clause) and a *postcondition* (**ensures** clause). The precondition and postcondition specify an assertion that must hold in the pre-state and the post-state of method calls, respectively. Since method contracts are used only for verifying modularly whether a given program complies with the programming model, and are not part of the programming model itself, it is safe to ignore them in this section.

In addition to method contracts, the syntax supports a second type of annotations, namely *share statements*. Using share statements, a programmer indicates when an object transitions from unshared to shared.

The language is not statically typed. Rather, to avoid formalizing a type system, type mismatches are considered run-time errors. The static verification approach detailed in Section 2.2 guarantees the absence of programming model violations as well as run-time errors (i.e., null dereferences and type mismatches).

For simplicity, the language does not include subclassing (i.e., Java's **extends** keyword). However, it does include dynamic binding. A program with subclassing could be encoded using delegation. Since the language also does not include inheritance among interfaces, the subtype relation is very simple: a type τ_1 is a subtype of a type τ_2 if either $\tau_1 = \tau_2$ or τ_1 is a class and τ_2 is an interface and τ_1 mentions τ_2 in its **implements** clause.

Our static verification approach performs modular verification. This means that for each method, all executions of that method are considered, not just those that occur in executions of the program. For example, for method *run* of class *Session* in Figure 3, all executions that satisfy *run*'s precondition are considered, including those where the *count* field contains a null value, even though there is no execution

of the program of Figure 3 where the method is called in a state where *count* is null. It follows that in the absence of the **assert** statement, the method would be considered invalid since in an execution where *count* is null, the **synchronized** statement would cause a null dereference. This limits the usefulness of the approach, since users are not interested in errors that do not occur in program executions. To alleviate this limitation, the programmer may restrict the set of executions considered by the static verification approach, by inserting **assert** statements into the program. If an **assert** statement's condition evaluates to false, the statement is said to *fail*. If in a given execution an **assert** statement fails, the execution is *stuck* (in Java, an exception is thrown), but for purposes of static verification, the execution is considered to be valid (or in other words, the execution is not considered further), since the failure is assumed to mean that the method execution never appears in an execution of the whole program. Other verification approaches outside the scope of this article, such as code review, model checking, or testing, may be used to verify such assumptions. (Note that the **assert** statement of this article, like the **assert** statement introduced in Java 1.4 and the *assert* macro in C, is a run-time assertion, rather than a verification-time assertion. In this unfortunate clash between the syntax of Java and that of Spec# [Barnett et al. 2004], where run-time assertions are denoted using the **assume** keyword and verification-time assertions using the **assert** keyword, we adhere to Java syntax.)

Note: in this article, we allow only **assert** statements of the form

$$\textbf{assert}\ g\ \textbf{instanceof}\ \tau;$$

which assert that g is a non-null object reference of type τ. We do not allow arbitrary expressions because the provided form is functionally complete and avoids the need to formalize expression syntax and semantics.

We only consider *well-formed* programs. Well-formed programs have no name clashes. To simplify the formalisation, we require even fields and methods declared in different classes or interfaces to have different names, except when a class method implements an interface method. Also, local variables need not be declared before they are assigned (in fact, our language does not have local variable declarations) but they must be assigned before they are used. By requiring that both branches of a conditional statement assign to the same variables, we can define a notion of *free variables* at each program point by considering an assignment to *bind* the variable occurrences that occur after it in the control flow and that are not hidden by a later assignment. Further, a well-formed program must not contain any of the syntactic forms that are intended to appear only during program execution (i.e., the constructs underlined in Figure 2). Lastly, classes must implement all methods of their declared interfaces, and if a method is used to start a new thread, its precondition must be as prescribed below.

DEFINITION 1. *A program π is* well-formed *if all of the following hold:*

—*No two interfaces have the same name. No two classes have the same name. No interface has the same name as a class. No two parameters of a given method have the same name. No two fields have the same name (even if they appear in different classes). No two interface methods have the same name (even if they appear in different interfaces). If two class methods have the same name m, then*

the methods are in different classes and the program declares an interface I that declares a method with name m and both classes implement interface I.

—*If one branch of an* **if** *statement contains an assignment to a variable x, then so does the other branch.*

—*If a statement uses a variable x, then an assignment to x appears before the statement in the method body (ignoring the other branch of an enclosing* **if** *statement), or x is a parameter of the enclosing method or* **this**.

—*The last statement of a method body and of the main routine is a* **return** *statement and a* **return** *statement does not appear anywhere else.*

—*The program does not contain any* **receive** *or* **unlock** *statements or object references.*

—*If a class implements an interface I and this interface declares a method with name m then the class declares a method with name m and its header is identical to the header of the method named m declared by interface I.*

—*Each class name, interface name, field name, and method name that appears in the program is declared by the program.*

—*The number of arguments specified in a call equals the number of parameters declared by the corresponding method.*

—*If a method is mentioned in a* **start** *statement, then its requires clause is exactly* $\mathsf{L_t} = \emptyset \wedge \mathbf{this} \in \mathsf{A}$. *(Note: the semantics of method contracts is defined in Section 2.2.)*

Notice that the example program of Figure 3 is well-formed.

All concepts in the remainder of this paper are implicitly parameterized by a program π.

2.1.3 *Program executions.* We now formalize the semantics of our annotated Java subset. While remaining faithful to Java semantics, our semantics additionally tracks the extra state variables (specifically, access sets and shared sets) required by the programming model. We also define a set of *legal program states*. A program state is legal if all thread states are legal. A thread is in a legal state if it is not about to violate the programming model or cause a run-time error (in particular, a null dereference or a type mismatch). In Section 2.1.4, we show that programs that reach only legal states are data-race-free. Note: we define legality as a separate judgment rather than encoding it as absence of progress (i.e., thread execution getting stuck) since in concurrent executions, on the one hand stuckness of a thread might not be due to an error in that thread (for example, if the thread is waiting for a lock that is never released), and on the other hand a thread that is stuck due to an error might become un-stuck again as a result of actions of other threads (for example, if the thread is attempting to lock an unshared object and the object is subsequently shared by another thread).

We use the following notation. $\mathcal{T}$ denotes the set of thread identifiers. Furthermore, v represents a value (i.e. $v \in \mathcal{O} \cup \{\mathsf{null}\}$) and o represents an object reference (i.e., a non-null value). $\mathsf{fields}(C)$ represents the set of fields declared by class C. We use $C \prec I$ to denote that class C implements interface I. $\mathsf{declaringClass}(f)$ denotes the name of the class that declares field f, and $\mathsf{declaringType}(m)$ denotes

the name of the interface that declares method m, or the name of the class that declares m if no interface declares a method m. Also, we assume the existence of a function $\mathsf{classof}$ that maps object references to class names. We assume that for each class name $C \in \mathcal{C}$, there are infinitely many object references $o \in \mathcal{O}$ such that $\mathsf{classof}(o) = C$. $\mathsf{mbody}(o.m(\overline{v}))$ denotes the body of method m declared in class $\mathsf{classof}(o)$, with o substituted for **this** and $\overline{v}$ substituted for the method's parameters. $\mathsf{objectRefs}(\phi)$ and $\mathsf{free}(\phi)$ denote the free object references and free variables, respectively, in syntactic entity ϕ. We use $f[x \mapsto y]$ to denote the update of the function f at argument x with value y. Specifically, $f[x \mapsto y](z)$ equals y if $z = x$ and $f(z)$ otherwise. Similarly, $f \setminus \{(x, y)\}$ removes the mapping of x to y from f, and thereby removes x from the domain of f. We use the notation $\overline{s}[v/x]$ to denote substitution of a value v for a program variable x in a thread continuation (i.e., sequence of statements) $\overline{s}$. This substitution replaces only the free occurrences of x, i.e., the ones that are not bound by an assignment inside $\overline{s}$. We denote the empty list as ϵ and a list with head h and tail t as $h \cdot t$. As in Java, we use juxtaposition to denote the concatenation of two statements or sequences of statements.

The dynamic semantics is defined as a small step relation on *program states*.

DEFINITION 2. *A* program state

$$\sigma = (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$$

consists of:

—*the heap* H*, a partial function that maps object references to object states. An object state is a partial function that maps field names to values.*

$$\mathsf{H} : \mathcal{O} \hookrightarrow (\mathcal{F} \hookrightarrow \mathcal{O} \cup \{\mathsf{null}\})$$

The domain of H *consists of all allocated objects. The domain of an object state* $\mathsf{H}(o)$ *consists of the declared fields of the class* $\mathsf{classof}(o)$ *of* o*.*

—*the lock map* L*, a partial function that maps a locked object to the identifier of the thread that holds the lock*

$$\mathsf{L} : \mathcal{O} \hookrightarrow \mathcal{T}$$

—*the shared set* S*, the set of shared objects*

—*the thread set* T*. Each thread state* $(\mathsf{tid}, \mathsf{A}, \mathsf{F}) \in \mathsf{T}$ *consists of a unique thread identifier* $\mathsf{tid} \in \mathcal{T}$*, an access set* $\mathsf{A} \subseteq \mathcal{O}$ *and a list of activation records* $\mathsf{F} \in (s^*)^*$*.*

We shall sometimes use uncurried syntax for the heap: $\mathsf{H}(o, f)$ is shorthand for $\mathsf{H}(o)(f)$, and $\mathsf{H}[(o, f) \mapsto v]$ is shorthand for $\mathsf{H}[o \mapsto \mathsf{H}(o)[f \mapsto v]]$.

Figure 4 shows the definition of *legality* $\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash t : \mathsf{legal}$ of a thread state t with respect to heap H, lock map L, and shared set S. Legality captures the rules of the programming model, as well as absence of run-time errors (i.e., null dereferences and type mismatches). Figure 5 shows the definition of the small step relation $\rightarrow$ on program states.

The rule IF is standard. An **assert** statement that fails (either because the operand is null or because it is not of the specified type) causes the thread to block forever (ASSERT). For reading (READ) or writing (WRITE) a field f, the target object must be non-null, part of the current thread's access set, and of the class

$$[\textsc{Legal-If}]\ \mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{if}\ (v_1 = v_2)\ \{\ \overline{s}_1\ \}\ \textbf{else}\ \{\ \overline{s}_2\ \}\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}$$

$$[\textsc{Legal-Assert}]\ \mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{assert}\ v\ \textbf{instanceof}\ \tau;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}$$

$$[\textsc{Legal-Read}]\ \frac{v \neq \mathsf{null} \qquad v \in \mathsf{A} \qquad \mathsf{classof}(v) = \mathsf{declaringClass}(f)}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(x := v.f;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Legal-Write}]\ \frac{v_1 \neq \mathsf{null} \qquad v_1 \in \mathsf{A} \qquad \mathsf{classof}(v_1) = \mathsf{declaringClass}(f)}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(v_1.f := v_2;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Legal-New}]\ \mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(x := \textbf{new}\ C;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}$$

$$[\textsc{Legal-Share}]\ \frac{v \neq \mathsf{null} \qquad v \in \mathsf{A} \qquad v \notin \mathsf{S}}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{share}\ v;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Legal-Synchronized}]\ \frac{v \neq \mathsf{null} \qquad v \notin \mathsf{L}^{-1}(\mathsf{tid}) \qquad v \in \mathsf{S}}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{synchronized}\ (v)\ \{\ \overline{s}'\ \}\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Legal-Unlock}]\ \frac{o \in \mathsf{A} \qquad (o,\mathsf{tid}) \in L}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{unlock}\ o;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Legal-Call}]\ \frac{v \neq \mathsf{null} \qquad \mathsf{classof}(v) \preceq \mathsf{declaringType}(m)}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(x := v.m(\overline{v});\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Legal-Return}]\ \mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{return}\ \mathsf{v};)\cdot\mathsf{F}) : \mathsf{legal}$$

$$[\textsc{Legal-NewThread}]\ \frac{v \neq \mathsf{null} \qquad v \in \mathsf{A} \qquad \mathsf{classof}(v) \preceq \mathsf{declaringType}(m)}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{start}\ v.m();\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

Fig. 4. Legal thread states

that declares the field. Note that field updates change the heap: the old value of the field is replaced with the new value. When creating a new object (New), an unused object reference is chosen from $\mathcal{O}$, is inserted into the heap and all its fields are initialized to the default value null. New objects are initially only accessible to their creator and therefore the reference is added to the creating thread's access set. A thread may share (Share) an unshared object in its access set. By doing so, it removes the object from its access set and adds it to the global shared set S. Shared objects may be locked (Synchronized) provided they are not locked yet. As noted in Section 2.1.1.1, we consider lock re-entry to be illegal. Our method effect framing approach, described in Section 2.2, relies on this. For the duration of the synchronized block, the lock map L marks the current thread as holder of the lock. The object is also added to the access set. When the end of the synchronized block is reached (Unlock), the object must still be in the thread's access set. At this time, the object is removed from the access set and its corresponding lock is released. Invoking a method m (Call) within a thread tid results in the addition of a new activation record to tid's call stack. The activation record contains the body of m where all free variables (**this** and parameters) are replaced with the corresponding argument values. In the caller's activation record,

$$[\textsc{If}]\ \frac{v_1 = v_2 \Rightarrow \overline{s}' = \overline{s}_1 \qquad v_1 \neq v_2 \Rightarrow \overline{s}' = \overline{s}_2}{(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\textbf{if } (v_1 = v_2) \ \{\ \overline{s}_1\ \} \textbf{ else } \{\ \overline{s}_2\ \}\ \overline{s}) \cdot \mathsf{F})) \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\overline{s}'\ \overline{s}) \cdot \mathsf{F}))}$$

$$[\textsc{Assert}]\ \frac{v \neq \mathsf{null} \qquad \mathsf{classof}(v) \preceq \tau}{(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\textbf{assert } v \textbf{ instanceof } \tau;\ \overline{s}) \cdot \mathsf{F})) \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\overline{s}) \cdot \mathsf{F}))}$$

$$[\textsc{Read}]\ (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (x := v.f;\ \overline{s}) \cdot \mathsf{F})) \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\overline{s}[\mathsf{H}(v, f)/x]) \cdot \mathsf{F}))$$

$$[\textsc{Write}]\ (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (v_1.f := v_2;\ \overline{s}) \cdot \mathsf{F})) \rightarrow (\mathsf{H}[(v_1, f) \mapsto v_2], \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\overline{s}) \cdot \mathsf{F}))$$

$$[\textsc{New}]\ \frac{o \notin \mathrm{dom}(\mathsf{H}) \qquad \mathsf{classof}(o) = C \qquad \mathsf{fields}(C) = \{f_1, ..., f_n\}}{\begin{array}{l}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (x := \textbf{new } C;\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}[o \mapsto \{(f_1, \mathsf{null}), \ldots, (f_n, \mathsf{null})\}], \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A} \cup \{o\}, (\overline{s}[o/x]) \cdot \mathsf{F}))\end{array}}$$

$$[\textsc{Share}]\ (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\textbf{share } v;\ \overline{s}) \cdot \mathsf{F})) \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S} \cup \{v\}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A} \setminus \{v\}, (\overline{s}) \cdot \mathsf{F}))$$

$$[\textsc{Synchronized}]\ \frac{v \notin \mathrm{dom}(\mathsf{L})}{\begin{array}{l}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\textbf{synchronized } (v)\ \{\ \overline{s}'\ \}\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}, \mathsf{L}[v \mapsto \mathsf{tid}], \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A} \cup \{v\}, (\overline{s}'\ \textbf{unlock } v;\ \overline{s}) \cdot \mathsf{F}))\end{array}}$$

$$[\textsc{Unlock}]\ (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\textbf{unlock } o;\ \overline{s}) \cdot \mathsf{F})) \rightarrow (\mathsf{H}, \mathsf{L} \setminus \{(o, \mathsf{tid})\}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A} \setminus \{o\}, (\overline{s}) \cdot \mathsf{F}))$$

$$[\textsc{Call}]\ \frac{\overline{s}' = \mathsf{mbody}(v.m(\overline{v}))}{\begin{array}{l}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (x := v.m(\overline{v});\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\overline{s}') \cdot (x := \textbf{receive } [(\mathsf{H}, \mathsf{L}^{-1}(\mathsf{tid}), \mathsf{S}, \mathsf{A})]\ v.m(\overline{v});\ \overline{s}) \cdot \mathsf{F}))\end{array}}$$

$$[\textsc{Return}]\ \begin{array}{c}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\textbf{return } v;) \cdot (x := \textbf{receive } \ldots;\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\overline{s}[v/x]) \cdot \mathsf{F}))\end{array}$$

$$[\textsc{NewThread}]\ \frac{\forall(\mathsf{tid}'', \_, \_) \in \mathsf{T} \bullet \mathsf{tid}' \neq \mathsf{tid}'' \qquad \mathsf{tid}' \neq \mathsf{tid}}{\begin{array}{l}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}, \mathsf{A}, (\textbf{start } v.m();\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \lhd (\mathsf{tid}', \{v\}, (\mathsf{mbody}(v.m())) \cdot \epsilon) \lhd (\mathsf{tid}, \mathsf{A} \setminus \{v\}, (\overline{s}) \cdot \mathsf{F}))\end{array}}$$

Fig. 5. Execution steps. ($\mathsf{T} \lhd t = \mathsf{T} \cup \{t\}$)

the method invocation is replaced with a **receive** statement, which keeps a record of the call's pre-state and arguments. The operands of the **receive** statement are not used by the dynamic semantics; they are used only in the soundness proof in Section 2.2.5. When a method call returns (RETURN), the top activation record is popped and the return value is substituted into the caller's activation record. A new thread (NEWTHREAD) is started by performing a **start** $o.m()$; operation. Accessibility of object o is transferred from the original thread to the new one. The new thread consists of a single activation record containing the body of method m where **this** is replaced by o.

Program execution starts in an *initial state*.

DEFINITION 3. *In a program's* initial state*, there is only a single thread. It has an empty access set and it executes the main routine. Moreover, the heap, the lock map, and the shared set are all empty.*

$$\mathsf{initial}((\emptyset, \emptyset, \emptyset, \{(\mathsf{tid}, \emptyset, \mathsf{program\_main})\}))$$

where program_main *denotes the program's main routine.*

A thread's execution is complete when the thread's call stack consists of a single activation record whose continuation is a **return** statement. Threads whose execution is complete are not garbage collected in our semantics, since this is irrelevant for the results of this paper.

The programs allowed by the programming model are the *legal programs*, which reach only *legal program states*.

DEFINITION 4. *A program state* $(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$ *is* legal *if all thread states are legal as per Figures 4:*

$$\mathsf{legal}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T}) \Leftrightarrow (\forall t \in \mathsf{T} \bullet \mathsf{H}, \mathsf{L}, \mathsf{S} \vdash t : \mathsf{legal})$$

DEFINITION 5. *A program is* legal *if it is well-formed and all program states reached by execution of the program are legal:*

$$\mathsf{program\_legal} \Leftrightarrow \mathsf{program\_wf} \wedge (\forall \sigma_0, \sigma \bullet \mathsf{initial}(\sigma_0) \wedge \sigma_0 \rightarrow^* \sigma \Rightarrow \mathsf{legal}(\sigma))$$

2.1.4 *Data-race-freedom.* The dynamic semantics defined above is an *interleaving* semantics, which implies a *sequentially consistent* memory model. This means that there is a total order on all field accesses such that each read of a field $o.f$ yields the value most recently written to $o.f$ in this total order. However, the Java Language Specification, Third Edition (JLS3) [Gosling et al. 2005] does not guarantee sequential consistency. In general, it allows threads to see writes performed by other threads *out of order*, which is necessary to allow efficient implementations involving optimizations at the level of the compiler, the memory hierarchy, and the processor. Still, JLS3 does guarantee sequential consistency for *correctly synchronized* programs. These are programs where all sequentially consistent executions are *data-race-free*.

In conclusion, in order for the dynamic semantics in this article to be sound with respect to Java and for the programs we care about, we need to prove that all executions of these programs under this semantics are data-race-free as per JLS3.

In this subsection, we show that legal programs are data-race-free, by proving that $\rightarrow$ maintains *well-formedness* of the program state. Specifically, each execution step maintains the property that the threads' access sets and the *free set* partition the heap.

DEFINITION 6. *The* free set *of a program state σ consists of all shared objects that are not locked.*

$$\mathrm{Free}(\sigma) = \mathsf{S} \setminus \mathrm{dom}(\mathsf{L})$$

DEFINITION 7. *A multiset of sets S* partitions *a set T ($S \ll T$) if all sets in S are disjoint and their union equals T.*

$$S \ll T \Leftrightarrow (\forall s_1 \in S, s_2 \in S - \{s_1\} \bullet s_1 \cap s_2 = \emptyset) \wedge \bigcup S = T$$

Execution steps maintain well-formedness of the heap, the shared set, and the lock map.

DEFINITION 8. *A heap is* well-formed *($\vdash \mathsf{H} : \mathsf{ok}$) if objects referenced from fields*

are allocated.

$$\forall o \in \mathrm{dom}(\mathsf{H}), f \in \mathrm{dom}(\mathsf{H}(o)) \bullet \mathsf{H}(o)(f) \in \mathrm{dom}(\mathsf{H}) \cup \{\mathsf{null}\}$$

DEFINITION 9. *A shared set is* well-formed *with respect to a heap (*$\mathsf{H} \vdash \mathsf{S} : \mathsf{ok}$*) if shared objects are allocated.*

$$\mathsf{S} \subseteq \mathrm{dom}(\mathsf{H})$$

DEFINITION 10. *A lock map is* well-formed *with respect to a heap and shared set (*$\mathsf{H}, \mathsf{S} \vdash \mathsf{L} : \mathsf{ok}$*) if locked objects are shared.*

$$\mathrm{dom}(\mathsf{L}) \subseteq \mathsf{S}$$

DEFINITION 11. *A program state*

$$\sigma = (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$$

is well-formed *(*$\mathsf{wf}(\sigma)$*) if the following conditions hold:*

—*The access sets and the free set partition the heap.*

$$(\biguplus_{(\_,\mathsf{A},\_)\in\mathsf{T}} \{\mathsf{A}\}) \uplus \{S \setminus \mathrm{dom}(\mathsf{L})\} \ll \mathrm{dom}(\mathsf{H})$$

—*The heap, the shared set, and the lock map are well-formed.*

$$\vdash \mathsf{H} : \mathsf{ok} \ \wedge \ \mathsf{H} \vdash \mathsf{S} : \mathsf{ok} \ \wedge \ \mathsf{H}, \mathsf{S} \vdash \mathsf{L} : \mathsf{ok}$$

—*The continuations in each thread's call stack contain only references to allocated objects and do not contain any free variables.*

$$\forall(\_, \_, \mathsf{F}) \in \mathsf{T} \bullet \forall \overline{s} \in \mathsf{F} \bullet$$
$$(\mathsf{objectRefs}(\overline{s}) \subseteq \mathrm{dom}(\mathsf{H}) \ \wedge \ \mathsf{free}(\overline{s}) = \emptyset)$$

Notice that well-formedness of a program state implies that access sets are disjoint and that accessible objects are allocated.

THEOREM 1. *In a legal program, the small step relation preserves well-formedness.*

$$\mathsf{program\_legal} \Rightarrow (\forall \sigma_1, \sigma_2 \bullet (\mathsf{wf}(\sigma_1) \wedge \sigma_1 \rightarrow \sigma_2) \Rightarrow \mathsf{wf}(\sigma_2))$$

PROOF. *By case analysis on the step from* σ_1 *to* σ_2*. We consider cases* SHARE*,* SYNCHRONIZED*, and* UNLOCK*.*

—***Case*** SHARE*. By legality, we have that the object being shared is in the access set but not in the shared set. The step adds it to the shared set (and therefore the free set since unshared objects are not locked) and removes it from the access set. It follows that the partition is maintained.*

—***Case*** SYNCHRONIZED*. Assume that the object being locked is* o*. In* σ_1*,* o *is shared and not locked, and therefore it is part of* σ_1*'s free set. Since in a well-formed state the free set and the access sets partition the heap,* o *is not in any thread's access set. Adding* o *to a single access set and removing it from the free set (by adding an entry for* o *to the lock map) maintains the proper partitioning of the heap. Because locking an object modifies neither the heap nor the shared set*

and both are well-formed in the pre-state, they are well-formed in the post-state. Adding an entry for o (a shared object) to σ_1's well-formed lock set, preserves the fact that the lock set only contains shared objects. Finally, the continuations in σ_2 contain neither free variables nor unallocated objects, because σ_1 does not contain any and because locking did not introduce any.

—***Case*** UNLOCK. *By legality, the object being unlocked is in the access set and is locked by the current thread. The step removes it from the lock map (thus adding it to the free set, since locked objects are shared) and from the thread's access set. It follows that the partition is maintained.*

□

THEOREM 2. *States reached by legal programs are well-formed.*

PROOF. *By induction on the number of execution steps: the initial state is well-formed, and the induction step holds, as per Theorem 1.* □

The notion of *data race* is defined in JLS3 in terms of the *happens-before* relation on *execution steps*. We consider only finite executions, but we do not require an execution's last state to be a "final state". In other words, we use the term *execution* as a shorthand for *execution prefix*.

DEFINITION 12 EXECUTION, EXECUTION STEP. *An* execution *of a program is a finite sequence of program states $\sigma_0, \ldots, \sigma_n$ where σ_0 is an initial state and consecutive program states are related by the step relation.*

$$(\sigma_0, \ldots, \sigma_n) \in \mathcal{E} \Leftrightarrow \mathsf{initial}(\sigma_0) \wedge (\forall i \in \{0, \ldots, n-1\} \bullet \sigma_i \rightarrow \sigma_{i+1})$$

The execution steps *of an execution $(\sigma_0, \ldots, \sigma_n)$ are the integers $1, \ldots, n$, where step i denotes the step from state σ_{i-1} to state σ_i. σ_{i-1} is called step i's* pre-state *and σ_i is called its* post-state.

We say a step is *performed by a thread* tid if tid's thread state in the step's pre-state differs from tid's thread state in the step's post-state. It is easy to see from the step rules that in all executions, in each program state there is at most one thread state for each thread and in each execution step, exactly one thread's thread state changes.

DEFINITION 13 HAPPENS-BEFORE. *Consider an execution of a legal program. The* happens-before relation *on the execution is a partial order among the steps of the execution. Specifically, it is the smallest transitive relation such that*

—*Each step performed by a thread* tid *happens-before each subsequent step performed by* tid.

—*Each* UNLOCK *step on an object o happens-before each subsequent* SYNCHRONIZED *step on o.*

—*Each* NEWTHREAD *step that creates a thread* tid *happens-before each operation performed by* tid.

DEFINITION 14 DATA RACE. *Consider an execution of a legal program. A pair of steps of the execution constitutes a* data race *if one is a* WRITE *of a field $o.f$*

and the other is a READ *or* WRITE *of $o.f$ and the steps are not ordered by the happens-before relation.*

The following lemma states that in legal programs, no ordering constraints exist on execution steps beyond those imposed by the synchronization constructs (i.e., thread creation and **synchronized** blocks).

LEMMA 1. *In an execution of a legal program π, if two consecutive steps are not ordered by happens-before, then swapping them results again in an execution of π.*

PROOF. *By case analysis on the steps. We detail a few cases.*

—*The steps are not accesses of the same field, since this would mean access sets are not disjoint, and by Theorem 2 we have that program states are well-formed.*

—*A* NEW *step can be moved to the right. The other step does not access the newly created object since the* NEW *step does not modify the thread state of the other thread and well-formedness of the latter implies it does not mention unallocated objects.*

□

We are now ready to prove the main theorem of this section.

THEOREM 3. *Executions of legal programs do not contain data races.*

PROOF. *By contradiction. Consider a legal program π such that at least one of its executions contains a data race on a field $o.f$. Of all the data races in all the executions of π, pick one where the number of steps that intervene between the steps that constitute the data race is minimal. Let those steps be S_i and S_{i+1+n}.*

Assume first that $n = 0$. In state σ_{i-1} (the pre-state of step S_i) o is in the access set of the thread tid_i that performs S_i. Since a field access does not modify any access sets, this is still true in state σ_i. However, in this state, it is also true that o is in the access set of thread tid_{i+1} that performs S_{i+1}. This contradicts Theorem 2.

Assume now that $n > 0$. Call S_j the last step preceding S_{i+1+n} that does not happen before S_{i+1+n}. Since S_i precedes S_{i+1+n} and does not happen before S_{i+1+n}, such a step exists, and either $i = j$ or $i < j$. We consider two cases:

—***Case*** *$i = j$. By repeated application of Lemma 1, step S_i can be moved directly before S_{i+1+n}, which results in a data race without intervening steps, contradicting the minimality assumption.*

—***Case*** *$i < j$. By repeated application of Lemma 1, step S_j can be moved after S_{i+1+n}, which results in a data race with $n - 1$ intervening steps, contradicting the minimality assumption.*

□

2.1.5 *Non-interference.* In this subsection, we introduce the notion of a *non-interfering state change* and we prove that in executions of legal programs, with respect to the access set of one thread, steps of other threads are non-interfering state changes.

DEFINITION 15. *Two states are related by a* non-interfering state change *with respect to a given access set if*

—*all objects that are allocated in the first state are also allocated in the second state,*
—*all objects that are shared in the first state are also shared in the second state, and*
—*if an object is in the access set, then its state in the heap is unchanged and if additionally the object is unshared in the first state, then it is unshared in the second state.*

Formally:

$$(\mathsf{H},\mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}',\mathsf{S}')$$
$$\Updownarrow$$
$$\mathrm{dom}(\mathsf{H}) \subseteq \mathrm{dom}(\mathsf{H}') \wedge \mathsf{S} \subseteq \mathsf{S}' \wedge \mathsf{H}'|_{\mathsf{A}} = \mathsf{H}|_{\mathsf{A}} \wedge \mathsf{S}' \cap \mathsf{A} = \mathsf{S} \cap \mathsf{A}$$

Note: the non-interfering state change relation $\overset{\mathsf{A}}{\rightsquigarrow}$ for a given access set A relates two heap-shared set pairs rather than full program states. This allows us to re-use this relation in the context of thread-relevant states (see Section 2.2).

The following theorem states a key property of the programming model. Note: we write $\sigma \overset{\mathsf{tid}}{\rightarrow} \sigma'$ to denote that program states σ and σ' are related by an execution step performed by thread tid.

THEOREM 4 THREAD ISOLATION. *In an execution of a legal program, with respect to the access set of one thread, a sequence of steps of other threads constitutes a non-interfering state change.*

$$\mathsf{program\_legal} \wedge \mathsf{initial}(\sigma^0) \wedge \sigma^0 \rightarrow^* \sigma_0 \overset{\mathsf{tid}_1}{\rightarrow} \sigma_1 \cdots \overset{\mathsf{tid}_n}{\rightarrow} \sigma_n$$
$$\wedge$$
$$(\forall i \in \{0,\ldots,n\} \bullet \sigma_i = (\mathsf{H}_i, \mathsf{L}_i, \mathsf{S}_i, \mathsf{T}_i \triangleleft (\mathsf{tid}, \mathsf{A}, \mathsf{F}))) \wedge \mathsf{tid} \notin \{\mathsf{tid}_1,\ldots,\mathsf{tid}_n\}$$
$$\Downarrow$$
$$(\mathsf{H}_0,\mathsf{S}_0) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}_n,\mathsf{S}_n)$$

PROOF. *Since the program is legal, by Theorem 2 we have that all program states reached are legal and well-formed. We prove the theorem by induction on n. The base case $n = 0$ is trivial. Assume $n > 0$. By induction we have*

$$(\mathsf{H}_0,\mathsf{S}_0) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}_{n-1},\mathsf{S}_{n-1})$$

We perform case analysis on the rule used to derive the last step. We consider case WRITE.

Suppose the step's pre-state is well-formed and that the step writes value v to field f of object o. No new objects are allocated when assigning to fields, and therefore the domain of the old and the new heap are equal. The old and the new heap differ only at a single location, namely (o, f). From legality it follows that o is an element of thread tid_n's access set, and by well-formedness it is not an element of A. Finally, updating a field does not modify the shared set. □

2.2 Static verification

The previous section proved that legal programs are data-race-free. However, legality of a program is not a modular notion. Furthermore, the issue remains of how to verify legality of a given program. In this section, we define the notion

of *valid* programs, which is a condition that is modular and that is suitable for submission to an automatic theorem prover, and we show that valid programs are legal. It follows that valid programs are data-race-free and that they perform no null dereferences or ill-typed operations.

The validity notion is based on provability of *verification conditions* in the *verification logic*, i.e., the logic used to interpret the verification conditions.

Before we define and prove the validity notion, we establish the verification logic and we discuss the modular verification approach.

2.2.1 *Verification logic.* We target multi-sorted first-order predicate logic with equality. That is, a *term* t is a *logical variable* $y \in Y$ or a *function application* $f(t_1, \ldots, t_n)$ where f is a *function symbol* from the *signature* with *arity* n. A *formula* φ is an *equality* $t_1 = t_2$, an *inequality* $t_1 \neq t_2$, a *literal* **true** or **false**, an *atom* $P(t_1, \ldots, t_n)$ where P is a predicate symbol from the signature with arity n, a *propositional formula* using the connectives $\wedge$, $\vee$, $\Rightarrow$, and $\neg$, or a *quantification* $(\forall y \bullet \varphi)$.

We use the signature shown in Figure 6, for a given program. Note:

—All sorts are countably infinite.

—We leave the sorts of quantifications, function symbols, and predicate symbols implicit when they are clear from the context.

The widening and narrowing conversions are inserted implicitly to convert between program values and object references. That is, where below we write a term t of sort ref in a location where a term of sort value is expected, t should be read as $\mathsf{asvalue}(t)$, and vice versa. Also, some of the function symbols are overloaded. Specifically, implicitly there is a separate emptyset symbol for each set or function sort, and there are separate apply, update, and dom symbols for each function sort.

Additionally, the signature contains a nullary function symbol τ for each class or interface τ declared by the program, and a nullary function symbol f for each field f declared by the program.

Note: we also use the following abbreviations:

Abbreviation	Meaning
$t_1(t_2, t_3)$	$t_1(t_2)(t_3)$
$t_1[(t_2, t_3) \mapsto t_4]$	$t_1[t_2 \mapsto (t_1(t_2)[t_3 \mapsto t_4])]$
$t_1 \subseteq t_2$	$(\forall y \bullet y \in t_1 \Rightarrow y \in t_2)$
$t_2\|_{t_1} = t_3\|_{t_1}$	$(\forall y \bullet y \in t_1 \Rightarrow t_2(y) = t_3(y))$
$t_1 \preceq t_2$	$t_1 \prec t_2 \vee t_1 = t_2$

where y is a fresh logical variable.

Throughout, we interpret the logic according to an interpretation $\mathfrak{I}$ of the signature, which is as expected. Corner cases are as follows. $\mathfrak{I}(\mathsf{asobjref})(\mathfrak{I}(\mathsf{null}))$ yields some (fixed) object reference. If e_2 is not in the domain of e_1, then $\mathfrak{I}(\mathsf{apply})(e_1, e_2)$ yields some (fixed) element of the range sort. $\mathfrak{I}(\mathsf{subtype})(C, I)$ holds if the program declares a class C that mentions an interface I in its **implements** clause. (Remember that the language does not allow subclassing or interface extension.) Function $\mathfrak{I}(\mathsf{insertnonnull})(S, v)$ returns S if v is null, and $S \cup \{v\}$ otherwise.

Sort	Notes
ref	Object references
value	Program values (object reference or null)
ref set	Finite sets of object references
field	Field names
class	Class names
interface	Interface names
objstate = (field, value) func	Object states, i.e. finite partial functions from field names to program values
heap = (ref, objstate) func	Heaps, i.e. finite partial functions from object references to object states

Predicate symbol	Sorts	Syntax	Notes
$\mathsf{in}(t_1, t_2)$	$\alpha \times \alpha$ set	$t_1 \in t_2$	
$\mathsf{subtype}(t_1, t_2)$	interface × class	$t_1 \prec t_2$	subtype relation

Function symbol	Sorts	Syntax	Notes
emptyset	α set or (α, β) func	$\emptyset$	empty set, empty function
$\mathsf{insert}(t_1, t_2)$	α set × $\alpha \to \alpha$ set	$t_1 \cup \{t_2\}$	
$\mathsf{insertnonnull}(t_1, t_2)$	ref set × value → ref set	$t_1 \cup \{t_2\}_{-\mathsf{null}}$	
$\mathsf{remove}(t_1, t_2)$	α set × $\alpha \to \alpha$ set	$t_1 \setminus \{t_2\}$	
$\mathsf{intersect}(t_1, t_2)$	α set × α set → α set	$t_1 \cap t_2$	
$\mathsf{setminus}(t_1, t_2)$	α set × α set → α set	$t_1 - t_2$ or $t_1 \setminus t_2$	
$\mathsf{apply}(t_1, t_2)$	(α, β) func × $\alpha \to \beta$	$t_1(t_2)$	
$\mathsf{update}(t_1, t_2, t_3)$	(α, β) func × $\alpha \times \beta$ → (α, β) func	$t_1[t_2 \mapsto t_3]$	function update
$\mathsf{dom}(t)$	(α, β) func → α set	$\mathrm{dom}(t)$	function domain
null	value	null	
$\mathsf{classof}(t)$	ref → class	$\mathsf{classof}(t)$	class of an object
$\mathsf{asvalue}(t)$	ref → value	t	implicit widening
$\mathsf{asobjref}(t)$	value → ref	t	implicit narrowing

Fig. 6. Signature of the verification logic

Let Σ be an (incomplete) axiomatization of $\mathfrak{I}$. By the soundness of first-order logic, it follows that if a formula φ is provable from Σ, then φ is true under $\mathfrak{I}$:

$$\text{if } \Sigma \vdash \varphi \text{ then } \mathfrak{I} \models \varphi$$

Note:

—A multi-sorted first-order logic where all sorts are countably infinite can be reduced to a one-sorted first-order logic by having a universe of natural numbers and mapping it to the various sorts as appropriate in the interpretation. This allows us to use theorem provers for one-sorted logic, such as Simplify.

—We will sometimes be loose with the separation between object logic and metalogic. I.e., we will sometimes write just φ instead of $\mathfrak{I}, V \models \varphi$, where φ is a formula of the verification logic, if it is clear what the intended valuation V of the free variables of φ is. Also, we will sometimes write $\varphi[v/x]$ when we mean $\mathfrak{I}, V[x \mapsto v] \models \varphi$.

2.2.2 *Thread-relevant states and state predicates.* The verification conditions used by our modular static verification approach are stated not in terms of entire program states, but in terms of just the state variables that are relevant to the

current thread, i.e. the *thread-relevant state*.

DEFINITION 16. *In a given program state, the* thread-relevant state $(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})$ *of a certain thread consists of the heap, the objects locked by the thread, the shared set and the thread's access set.*

In order to enable modular verification, the verification approach imposes restrictions on program states beyond those imposed by program legality. Specifically, only references to shared objects may be stored in fields. This is discussed further in Section 2.2.3.

DEFINITION 17. *A thread-relevant state* $(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})$ *is* well-formed*, written* $\vdash (\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}) : \mathsf{ok}$*, if the heap is well-formed, the shared set is well-formed with respect to the heap, the access set is a subset of the heap's domain, the lock set is a subset of the shared set, and non-null field values are shared.*

$$\begin{array}{c} \vdash (\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}) : \mathsf{ok} \\ \Updownarrow \\ \vdash \mathsf{H} : \mathsf{ok} \ \wedge \ \mathsf{H} \vdash \mathsf{S} : \mathsf{ok} \ \wedge \ \mathsf{L_t} \subseteq \mathsf{S} \wedge \ \mathsf{A} \subseteq \mathrm{dom}(\mathsf{H}) \\ \wedge \\ (\forall o \in \mathrm{dom}(\mathsf{H}), f \in \mathrm{dom}(\mathsf{H}(o)) \bullet \mathsf{H}(o, f) = \mathsf{null} \vee \mathsf{H}(o, f) \in \mathsf{S}) \end{array}$$

A continuation, as it appears in a program text, may contain free program variables. Consequently, the corresponding continuation verification condition contains these program variables as free logical variables.

DEFINITION 18. *A* program variable $z \in \mathit{Var}$ *is either* **this**, **result**, *or a method parameter or local variable* $x \in \mathcal{X}$.

$$\mathit{Var} = \{\mathbf{this}, \mathbf{result}\} \cup \mathcal{X}$$

A continuation verification condition is a *state predicate*. A state predicate may be a *two-state predicate* or a *one-state predicate*. A two-state predicate may refer to the current thread-relevant state (using free variables H, $\mathsf{L_t}$, S, and A) and to the *old state* (using free variables $\mathsf{H}^{\mathrm{old}}$, $\mathsf{L}_\mathsf{t}^{\mathrm{old}}$, $\mathsf{S}^{\mathrm{old}}$, and $\mathsf{A}^{\mathrm{old}}$). A one-state predicate may refer only to the current state (using free variables H, $\mathsf{L_t}$, S, and A). Since method postconditions are two-state predicates, so are continuation verification conditions.

DEFINITION 19. *A* two-state predicate *(or* state predicate *for short)* Q *is a formula of the verification logic, whose free logical variables are the current and old thread-relevant state variables and program variables:*

$$\mathsf{free}(Q) \subseteq \{\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}, \mathsf{H}^{\mathrm{old}}, \mathsf{L}_\mathsf{t}^{\mathrm{old}}, \mathsf{S}^{\mathrm{old}}, \mathsf{A}^{\mathrm{old}}\} \cup \mathit{Var}$$

A state predicate Q *is called a* one-state predicate *if the old thread-relevant state variables do not appear free in it.*

$$\mathsf{free}(Q) \subseteq \{\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}\} \cup \mathit{Var}$$

A state predicate is program-closed *if no program variables appear free in it.*

We will use the notation $\mathfrak{I}, \mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}, V \vDash Q$ to denote the truth of a state predicate in a particular thread-relevant state and under a particular valuation V that maps program variables to program values.

The verification conditions used by our approach are *local*.

DEFINITION 20. *A state predicate Q is* local *if it is preserved by a non-interfering state change.*

$$\begin{array}{c}\mathsf{local}(Q) \equiv \\ (\forall \overline{v}, \mathsf{H}^{\mathrm{old}}, \mathsf{L}_{\mathrm{t}}^{\mathrm{old}}, \mathsf{S}^{\mathrm{old}}, \mathsf{A}^{\mathrm{old}}, \mathsf{H}, \mathsf{L}_{\mathrm{t}}, \mathsf{S}, \mathsf{A}, \mathsf{H}', \mathsf{S}' \bullet \\ Q[\overline{v}/\overline{g}] \wedge (\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}') \Rightarrow Q[\overline{v}/\overline{g}, \mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}])\end{array}$$

where $\overline{g}$ are the program variables that appear free in Q.

2.2.3 *Modular verification.* To determine validity of a method, one could take the entire program into account at once. However, this approach is not modular, and hence does not scale to large programs. Instead, we propose a modular approach where the validity of a method does not depend on the entire program, but only on its body, its *method contract*, and the contracts of its callees.

A method contract is of the form

$$\textbf{requires } P; \ \textbf{ensures } Q;$$

where P is a state predicate and Q is a two-state predicate. Specifically, the free variables allowed in P are H,L_{t},S, and A, as well as **this** and the method's parameters. The free variables allowed in Q are H,L_{t},S,A, $\mathsf{H}^{\mathrm{old}}$, $\mathsf{L}_{\mathrm{t}}^{\mathrm{old}}$, $\mathsf{S}^{\mathrm{old}}$, and $\mathsf{A}^{\mathrm{old}}$, as well as **this**, the method's parameters, and **result**. P is called the method's precondition and Q is called the method's postcondition.

To verify whether a method complies with the model and respects its method contract and those of its callees, one may only assume that the method's precondition holds and that the program state is well-formed when the method is called. Furthermore, note that methods cannot query the access set in the sense that branching on whether an object is accessible or not is impossible. From this we can derive a framing property: a method call does not modify or share an object o that is allocated and accessible before the call, if the precondition does not require o to be accessible. Thanks to this framing property, we do not need explicit modifies clauses in method contracts.

Consider a formula φ in the verification logic such that $\mathsf{free}(\varphi) \subseteq \{\mathsf{A}\}$. The *required access set* $\mathcal{A}(\varphi)$ of φ is defined as

$$\{o \in \mathcal{O} \mid \forall \mathsf{A} \subseteq \mathcal{O} \bullet \varphi \Rightarrow o \in \mathsf{A}\}$$

That is, when interpreting φ as a set of access sets, we have

$$\mathcal{A}(\varphi) = \bigcap \varphi$$

In our verification approach, for a given method precondition P, we compute a first-order expression for $\mathcal{A}(P)$ syntactically. This is possible if the syntax of preconditions is restricted sufficiently.

We require method preconditions and postconditions to be local.

In this section, we impose the additional restriction that fields can only hold shared objects. We verify that any field write stores a shared object; as a result, we may assume that the objects read from fields are shared. The example of Figure 3 shows a typical scenario that exploits this property. Method *run* attempts to acquire the lock of the object stored in the *counter* field. Therefore, this field must hold a shared object. However, this cannot be stated as part of the method's

precondition, since method *run* is mentioned in a **start** statement as the main method for a new thread, and therefore its precondition is fixed. To solve this problem, we build the restriction that fields must hold shared objects into the verification approach of this section.

This restriction is lifted in Section 3, where invariants offer a more flexible way to specify that a given field holds a shared object. Another way to lift the restriction is to introduce a **shared** modifier, which indicates that a certain field can only hold shared objects. In the latter approach, it is verified that at every field write $o.f := v$; where f is marked **shared**, v is either null or a shared object. In return, it may be assumed that values read from fields marked **shared** are either null or shared objects.

2.2.4 *Valid programs.* In this subsection, we define which *verification conditions* are generated for a given program. A verification condition is a closed formula in the verification logic. A program is *valid* if its verification conditions are provable from the verification logic's theory Σ.

We define program verification conditions in terms of *continuation verification conditions*. A continuation verification condition $\mathsf{vc}(\overline{s}, Q)$ is a formula of the verification logic that expresses conditions under which a continuation $\overline{s}$ (i.e., a list of statements to be executed next) executes correctly and satisfies postcondition Q when started in a given program state. A continuation verification condition depends only on the state variables relevant to the thread that executes the continuation. These state variables appear as free logical variables in the formula.

Figure 7 defines the predicate transformer vc, which maps a continuation $\overline{s}$ and a state predicate Q to the continuation verification condition for $\overline{s}$ and Q. As noted above, $\mathsf{vc}(\overline{s}, Q)$ is a sufficient condition such that any thread in a state satisfying $\mathsf{vc}(\overline{s}, Q)$ ends up in a state satisfying Q after executing the continuation $\overline{s}$. Note: substitutions apply to predicates of the object logic; they do not apply directly to predicate transformer applications of the metalogic. Specifically, it is generally not the case that $\mathsf{vc}(\overline{s}, Q)[t/x] \equiv \mathsf{vc}(\overline{s}[t/x], Q[t/x])$. In other words, to compute a verification condition, first apply all verification condition rules exhaustively and then apply the substitutions. $\mathsf{pre}(o.m(\overline{v}))$ and $\mathsf{post}(o.m(\overline{v}))$ denote the precondition and postcondition, respectively, declared by method m, with o substituted for **this** and $\overline{v}$ for the method's parameters.

The intuition underlying the verification condition rules is as follows. The verification condition for VC-If is a standard weakest precondition. Since an **assert** statement blocks forever if its condition is false, the statement's continuation is verified under the assumption that the condition is true. For a field access (VC-Read or VC-Write), the target of the access must be non-null and accessible. A newly created object (VC-New) was previously unallocated and is of the correct class. In order to share (VC-Share) a value, the value should be non-null, accessible and unshared. Locking (VC-Synchronized) is disallowed if the object is unshared or already locked by the current thread. Moreover, since between synchronized statements other threads may modify shared objects, we should assume nothing about the fields of the newly locked object. The only property we may assume is that the old state and the new state are related by a non-interfering state change with respect to the thread's access set. Unlocking (VC-Unlock) is allowed

only for locked and accessible objects. When invoking a method (VC-Call), the target should not be null, the callee's precondition should hold and when returning (VC-Receive) the postcondition should hold. When a method call returns, we make some assumptions about the post-state. First of all, as per the method framing approach, we may assume that the post-state is related to the pre-state by a non-interfering state change with respect to the pre-state access set minus the callee's required access set. Secondly, we may assume the post-state is well-formed and satisfies the callee's postcondition. Finally, we may assume the return value is allocated. Starting a new thread via **start** (VC-NewThread) requires the target object to be accessible and non-null. Accessibility of the target object is transferred from the current thread to the new thread.

An important property of continuation verification conditions is that they are *local*.

Theorem 5. *If a state predicate Q is local, then the verification condition of a continuation $\overline{s}$ with respect to Q is local as well.*

$$\mathsf{local}(Q) \Rightarrow \mathsf{local}(\mathsf{vc}(\overline{s}, Q))$$

Proof. *By induction on $\overline{s}$.*

—***Case*** $\overline{s} =$ **synchronized** (v) { $\overline{s}''$ } $\overline{s}'$.
  (*1*) *We may assume that $\overline{s}$ is valid (i.e., $\mathsf{vc}(\overline{s}, Q)$ holds) in a state $(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})$.*
  (*2*) *It follows by* VC-Synchronized *that the continuation $\overline{s}''$* **unlock** *v; $\overline{s}'$ of $\overline{s}$ is valid in any state $(\mathsf{H}', \mathsf{L_t} \cup \{v\}, \mathsf{S}', \mathsf{A} \cup \{v\})$ where $(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}')$.*
  (*3*) *We need to prove that $\overline{s}$ is valid in any state $(\mathsf{H}'', \mathsf{L_t}, \mathsf{S}'', \mathsf{A})$ where $(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}'', \mathsf{S}'')$.*
  (*4*) *This requires that we prove that the continuation $\overline{s}''$* **unlock** *v; $\overline{s}'$ of $\overline{s}$ is valid in any state $(\mathsf{H}''', \mathsf{L_t} \cup \{v\}, \mathsf{S}''', \mathsf{A} \cup \{v\})$ where $(\mathsf{H}'', \mathsf{S}'') \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}''', \mathsf{S}''')$.*
  (*5*) *It is easy to see that the non-interfering state change relation is transitive; therefore, from the assumptions in points 3 and 4 we have $(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}''', \mathsf{S}''')$. By instantiating the rule in point 2, we obtain the goal in point 4.*

—***Case*** **receive** *is analogous to case* **synchronized**.

—*The other cases are easy.*

□

Notice that in the verification conditions, inter-thread interference only surfaces in rules VC-Synchronized and VC-Receive, even though operations from other threads may be interleaved between any two operations of a given thread. Informally speaking, this is sound because on the one hand each thread's verification condition is local and therefore is preserved by non-interfering state changes, and on the other hand, each thread's actions constitute non-interfering state changes with respect to other threads. For example, a write by a thread t_1 cannot invalidate the continuation of another thread t_2, since t_1 writes only fields of objects in its own access set A_1, and validity of t_2 depends only on fields of objects in its access set A_2, and both are disjoint. See cases Write and Share in the proof of Theorem 7 on page 28.

$$\mathsf{vc}(\textbf{if } (v_1 = v_2) \ \{\ \overline{s}_1\ \} \textbf{ else } \{\ \overline{s}_2\ \}\ \overline{s}, Q) \equiv \quad \text{[VC-If]}$$
$$(v_1 = v_2 \Rightarrow \mathsf{vc}(\overline{s}_1\ \overline{s}, Q)) \wedge (v_1 \neq v_2 \Rightarrow \mathsf{vc}(\overline{s}_2\ \overline{s}, Q))$$

$$\mathsf{vc}(\textbf{assert } v \textbf{ instanceof } \tau;\ \overline{s}, Q) \equiv \quad \text{[VC-Assert]}$$
$$(v \neq \mathsf{null} \wedge \mathsf{classof}(v) \preceq \tau) \Rightarrow \mathsf{vc}(\overline{s}, Q)$$

$$\mathsf{vc}(x := v.f;\ \overline{s}, Q) \equiv \quad \text{[VC-Read]}$$
$$v \neq \mathsf{null} \wedge \mathsf{classof}(v) = \mathsf{declaringClass}(f) \wedge v \in \mathsf{A} \wedge \mathsf{vc}(\overline{s}, Q)[\mathsf{H}(v, f)/x]$$

$$\mathsf{vc}(v_1.f := v_2;\ \overline{s}, Q) \equiv \quad \text{[VC-Write]}$$
$$v_1 \neq \mathsf{null} \wedge \mathsf{classof}(v_1) = \mathsf{declaringClass}(f) \wedge v_1 \in \mathsf{A} \wedge (v_2 = \mathsf{null} \vee v_2 \in \mathsf{S})$$
$$\wedge\ \mathsf{vc}(\overline{s}, Q)[\mathsf{H}[(v_1, f) \mapsto v_2]/\mathsf{H}]$$

$$\mathsf{vc}(x := \textbf{new } C;\ \overline{s}, Q) \equiv \quad \text{[VC-New]}$$
$$\forall o \bullet o \notin \mathrm{dom}(\mathsf{H}) \wedge \mathsf{classof}(o) = C \Rightarrow$$
$$\mathsf{vc}(\overline{s}, Q)[o/x, \mathsf{H}[o \mapsto \emptyset[f_1 \mapsto \mathsf{null}] \cdots [f_n \mapsto \mathsf{null}]]/\mathsf{H}, (\mathsf{A} \cup \{o\})/\mathsf{A}]$$
$$\text{where } \mathsf{fields}(C) = \{f_1, ..., f_n\}$$

$$\mathsf{vc}(\textbf{share } v;\ \overline{s}, Q) \equiv \quad \text{[VC-Share]}$$
$$v \neq \mathsf{null} \wedge v \in \mathsf{A} \wedge v \notin \mathsf{S} \wedge \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{v\})/\mathsf{A}, (\mathsf{S} \cup \{v\})/\mathsf{S}]$$

$$\mathsf{vc}(\textbf{synchronized } (v)\ \{\ \overline{s}'\ \}\ \overline{s}, Q) \equiv \quad \text{[VC-Synchronized]}$$
$$v \neq \mathsf{null} \wedge v \in \mathsf{S} \wedge v \notin \mathsf{L_t}$$
$$\wedge\ (\forall \mathsf{H}', \mathsf{S}' \bullet$$
$$((\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}') \wedge \vdash (\mathsf{H}', \mathsf{L_t}, \mathsf{S}', \mathsf{A}) : \mathsf{ok})$$
$$\Rightarrow$$
$$\mathsf{vc}(\overline{s}'\ \textbf{unlock } v;\ \overline{s}, Q)[\mathsf{H}'/\mathsf{H}, (\mathsf{A} \cup \{v\})/\mathsf{A}, (\mathsf{L_t} \cup \{v\})/\mathsf{L_t}, \mathsf{S}'/\mathsf{S}])$$

$$\mathsf{vc}(\textbf{unlock } o;\ \overline{s}, Q) \equiv \quad \text{[VC-Unlock]}$$
$$o \in \mathsf{A} \wedge o \in \mathsf{L_t} \wedge \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{o\})/\mathsf{A}, (\mathsf{L_t} \setminus \{o\})/\mathsf{L_t}]$$

$$\mathsf{vc}(x := v.m(\overline{v});\ \overline{s}, Q) \equiv \quad \text{[VC-Call]}$$
$$v \neq \mathsf{null} \wedge \mathsf{classof}(v) \preceq \mathsf{declaringType}(m) \wedge \mathsf{pre}(v.m(\overline{v}))$$
$$\wedge\ \mathsf{vc}(x := \textbf{receive } [(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})]\ v.m(\overline{v});\ \overline{s}, Q)$$

$$\mathsf{vc}(\textbf{start } v.m();\ \overline{s}, Q) \equiv \quad \text{[VC-NewThread]}$$
$$v \neq \mathsf{null} \wedge \mathsf{classof}(v) \preceq \mathsf{declaringType}(m) \wedge v \in \mathsf{A} \wedge \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{v\})/\mathsf{A}]$$

$$\mathsf{vc}(\textbf{return } v;, Q) \equiv \quad \text{[VC-Return]}$$
$$Q[v/\textbf{result}]$$

Fig. 7. Continuation verification conditions *(Part 1 of 2)*

$$
\begin{array}{l}
\mathsf{vc}(x := \textbf{receive}\ [(\mathsf{H}^{\mathrm{pre}}, \mathsf{L}_{\mathrm{t}}^{\mathrm{pre}}, \mathsf{S}^{\mathrm{pre}}, \mathsf{A}^{\mathrm{pre}})]\ o.m(\overline{v});\ \overline{s}, Q) \equiv \qquad \text{[VC-Receive]} \\
\quad \forall \mathsf{H}', \mathsf{S}', \mathsf{A}', v_{\mathrm{r}} \bullet \\
\qquad ((\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}_{\mathrm{caller}}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}') \wedge (\vdash (\mathsf{H}', \mathsf{L}_{\mathrm{t}}, \mathsf{S}', \mathsf{A}') : \mathsf{ok}) \\
\qquad \wedge\ \mathsf{A}' \cap \mathsf{A}_{\mathrm{caller}} = \mathsf{A} \cap \mathsf{A}_{\mathrm{caller}} \\
\qquad \wedge\ Q'[\mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}, \mathsf{A}'/\mathsf{A}, \mathsf{L}_{\mathrm{t}}^{pre}/\mathsf{L}_{\mathrm{t}}, v_{\mathrm{r}}/\textbf{result}, \\
\qquad\qquad \mathsf{H}^{\mathrm{pre}}/\mathsf{H}^{\mathrm{old}}, \mathsf{L}_{\mathrm{t}}^{\mathrm{pre}}/\mathsf{L}_{\mathrm{t}}^{\mathrm{old}}, \mathsf{S}^{\mathrm{pre}}/\mathsf{S}^{\mathrm{old}}, \mathsf{A}^{\mathrm{pre}}/\mathsf{A}^{\mathrm{old}}] \\
\qquad \wedge\ (v_{\mathrm{r}} = \mathsf{null} \vee v_{\mathrm{r}} \in \mathrm{dom}(\mathsf{H}')) \\
\qquad \Rightarrow \\
\qquad \mathsf{vc}(\overline{s}, Q)[\mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}, \mathsf{A}'/\mathsf{A}, \mathsf{L}_{\mathrm{t}}^{old}/\mathsf{L}_{\mathrm{t}}, v_{\mathrm{r}}/x] \\
\quad \text{where } P' = \mathsf{pre}(o.m(\overline{v})) \text{ and } Q' = \mathsf{post}(o.m(\overline{v})) \\
\quad \text{and } \mathsf{A}_{\mathrm{caller}} = \mathsf{A}^{\mathrm{pre}} - \mathcal{A}(P'[\mathsf{H}^{\mathrm{pre}}/\mathsf{H}, \mathsf{L}_{\mathrm{t}}^{\mathrm{pre}}/\mathsf{L}_{\mathrm{t}}, \mathsf{S}^{\mathrm{pre}}/\mathsf{S}])
\end{array}
$$

Figure 7: Continuation verification conditions *(Part 2 of 2)*

We are now ready to define *program validity.*

Definition 21. *A well-formed program is* valid *if all of the following hold:*

—*each method precondition or postcondition φ is local*

$$\Sigma \vdash \mathsf{local}(\varphi)$$

—*each method $m(\overline{x})$* **requires** P; **ensures** Q; $\{\ \overline{s}\ \}$ *in each class C is valid, i.e., it is provable from the verification logic that the precondition implies validity of the method's body with respect to the postcondition.*

$$
\begin{array}{c}
\Sigma \vdash \\
\forall o, \overline{v}, \mathsf{H}, \mathsf{L}_{\mathrm{t}}, \mathsf{S}, \mathsf{A} \bullet \\
(\vdash (\mathsf{H}, \mathsf{L}_{\mathrm{t}}, \mathsf{S}, \mathsf{A}) : \mathsf{ok} \wedge \mathsf{classof}(o) = C \wedge \{\overline{v}\} \subseteq \{\mathsf{null}\} \cup \mathrm{dom}(\mathsf{H}) \wedge P[o/\textbf{this}, \overline{v}/\overline{x}]) \\
\Downarrow \\
\mathsf{vc}(\overline{s}, Q)[o/\textbf{this}, \overline{v}/\overline{x}, \mathsf{H}/\mathsf{H}^{\mathrm{old}}, \mathsf{L}_{\mathrm{t}}/\mathsf{L}_{\mathrm{t}}^{\mathrm{old}}, \mathsf{S}/\mathsf{S}^{\mathrm{old}}, \mathsf{A}/\mathsf{A}^{\mathrm{old}}]
\end{array}
$$

—*the main routine $\overline{s}$ is valid:*

$$\Sigma \vdash \mathsf{vc}(\overline{s}, \textbf{true})[\emptyset/\mathsf{H}, \emptyset/\mathsf{L}_{\mathrm{t}}, \emptyset/\mathsf{S}, \emptyset/\mathsf{A}]$$

The example of Figure 3 is a valid program.

2.2.5 *Soundness.* In this subsection we define a notion of *valid program state* and we prove that valid states are legal states. We then prove that program states reached by valid programs are valid, by proving that the initial state is valid and that execution steps preserve validity. It follows that valid programs are legal programs and therefore they are data-race-free.

A program state is *valid* if each thread state is *consistent* and each activation record is *valid*. The latter means that the continuation verification condition of the activation record's continuation holds with respect to the activation record's postcondition. An activation record's validity is independent of actions performed by other activation records. We prove this using the notion of *activation record access sets*. We prove that an activation record's validity depends only on the state of the objects in its access set, and actions of other activation records are non-interfering with respect to this access set.

We use the following shorthands. We denote the components of a thread-relevant state θ as H_θ, L_θ, S_θ, and A_θ. That is, we have

$$\theta = (\mathsf{H}_\theta, \mathsf{L}_\theta, \mathsf{S}_\theta, \mathsf{A}_\theta)$$

Furthermore, we use $\mathsf{A}_{\mathsf{req}}(\mathit{call}, \theta)$ to denote the required access set of call *call* in pre-state θ:

$$\mathsf{A}_{\mathsf{req}}(o.m(\overline{v}), \theta) = \mathcal{A}(\mathsf{pre}(o.m(\overline{v}))[\mathsf{H}_\theta/\mathsf{H}, \mathsf{L}_\theta/\mathsf{L}_\mathrm{t}, \mathsf{S}_\theta/\mathsf{S}])$$

We use $\mathsf{post}(\mathit{call}, \theta)$ to denote the postcondition of call *call* with respect to pre-state θ:

$$\mathsf{post}(o.m(\overline{v}), \theta) = \mathsf{post}(o.m(\overline{v}))[\mathsf{H}_\theta/\mathsf{H}^{\mathrm{old}}, \mathsf{L}_\theta/\mathsf{L}_\mathrm{t}^{\mathrm{old}}, \mathsf{S}_\theta/\mathsf{S}^{\mathrm{old}}]$$

DEFINITION 22. *An* activation record's access set *is recursively defined as follows:*

—*The top (i.e., active) activation record's access set is the thread's access set minus the other activation records' access sets.*

—*The access set of an activation record that is suspended while waiting for a call to return, is the pre-state thread access set, minus the access sets of transitive caller activation records, minus the call's required access set.*

Formally:

$$\begin{array}{c} t = (\mathsf{tid}, \mathsf{A}, (\overline{s}_1) \cdot (x_2 := \textbf{receive}\ [\theta_2]\ \mathit{call}_2;\ \overline{s}_2) \\ \qquad\qquad \cdot \ldots \cdot (x_n := \textbf{receive}\ [\theta_n]\ \mathit{call}_n;\ \overline{s}_n) \cdot \epsilon) \\ \Downarrow \\ \forall i \in \{1, \ldots, n\} \bullet \mathsf{A}_{\mathsf{ar}(i)}(t) = \begin{cases} \mathsf{A} - \bigcup_{1<j\leq n} \mathsf{A}_{\mathsf{ar}(j)}(t) & \text{if } i = 1 \\ \mathsf{A}_{\theta_i} - \bigcup_{i<j\leq n} \mathsf{A}_{\mathsf{ar}(j)}(t) - \mathsf{A}_{\mathsf{req}}(\mathit{call}_i, \theta_i) & \text{if } i > 1 \end{cases} \end{array}$$

DEFINITION 23. *A thread state t is* consistent *with respect to a given lockset L_t, written $\mathsf{L}_\mathrm{t} \vdash \mathsf{consistent}(t)$, if t is of the form*

$$\begin{array}{c} (\mathsf{tid}, \mathsf{A}, (\overline{s}_1) \cdot (x_2 := \textbf{receive}\ [\theta_2]\ \mathit{call}_2;\ \overline{s}_2) \\ \qquad\qquad \cdot \ldots \cdot (x_n := \textbf{receive}\ [\theta_n]\ \mathit{call}_n;\ \overline{s}_n) \cdot \epsilon) \end{array}$$

where all of the following hold:

—*No* unlock *statement appears inside another statement in the thread's activation records' continuations*

$$\begin{array}{c} (\forall i \in \{1, \ldots, n\} \bullet \overline{s}_i \neq \cdots\ \textbf{if}\ (\cdots)\ \{\ \cdots\ \textbf{unlock}\ o;\ \cdots\ \}\ \cdots) \\ (\forall i \in \{1, \ldots, n\} \bullet \overline{s}_i \neq \cdots\ \textbf{if}\ (\cdots)\ \{\ \cdots\ \}\ \textbf{else}\ \{\ \cdots \textbf{unlock}\ o;\ \cdots\ \}\ \cdots) \\ (\forall i \in \{1, \ldots, n\} \bullet \overline{s}_i \neq \cdots\ \textbf{synchronized}\ (\cdots)\ \{\ \cdots\ \textbf{unlock}\ o;\ \cdots\ \}\ \cdots) \end{array}$$

—*The thread's lockset is equal to the set of objects that appear in* **unlock** *statements in the thread's activation records' continuations*

$$\mathsf{L}_\mathrm{t} = \{o \mid (\exists i \in \{1, \ldots, n\} \bullet \overline{s}_i = \cdots\ \textbf{unlock}\ o;\ \cdots)\}$$

—*No object appears more than once in an* **unlock** *statement*

$$\neg(\overline{s}_1 \cdots \overline{s}_n = \cdots\ \textbf{unlock}\ o;\ \cdots\ \textbf{unlock}\ o;\ \cdots)$$

—*A caller's pre-state lockset is equal to the set of* **unlock** *statements in the call's continuation plus transitive caller continuations*

$$(\forall i \in \{2,\ldots,n\} \bullet L_{\theta_i} = \{o \mid (\exists j \in \{i,\ldots,n\} \bullet \overline{s}_j = \cdots \ \mathbf{unlock}\ o;\ \cdots)\})$$

—*Each non-top activation record's required access set is included in its pre-state activation record access set*

—*Each non-top activation record's pre-state access set includes the access sets of transitive caller activation records*

—*The thread's access set includes the non-top activation records' access sets*

$$\begin{array}{l} n = 1 \\ \vee\ (\mathsf{A}_{\mathsf{req}}(call_n, \theta_n) \subseteq \mathsf{A}_{\theta_n} \\ \quad \wedge\ (\forall i \in \{2,\ldots,n-1\} \bullet \\ \qquad \mathsf{A}_{\theta_{i+1}} - \mathsf{A}_{\mathsf{req}}(call_{i+1}, \theta_{i+1}) \subseteq \mathsf{A}_{\theta_i} \\ \qquad \wedge\ \mathsf{A}_{\mathsf{req}}(call_i, \theta_i) \subseteq \mathsf{A}_{\theta_i} - (\mathsf{A}_{\theta_{i+1}} - \mathsf{A}_{\mathsf{req}}(call_{i+1}, \theta_{i+1}))) \\ \quad \wedge\ \mathsf{A}_{\theta_2} - \mathsf{A}_{\mathsf{req}}(call_2, \theta_2) \subseteq \mathsf{A}) \end{array}$$

Notice that if a thread state is consistent, then its activation records' access sets partition the thread's access set.

DEFINITION 24. *An* activation record's postcondition $Q_{\mathsf{ar}(i)}(t)$ *is the postcondition of the call stored in the* **receive** *statement in the caller's continuation, or* **true** *if the activation record has no caller.*
Formally:

$$\begin{array}{c} t = (\mathsf{tid}, \mathsf{A}, (\overline{s}_1) \cdot (x_2 := \mathbf{receive}\ [\theta_2]\ call_2;\ \overline{s}_2) \\ \qquad\qquad \cdot \ldots \cdot (x_n := \mathbf{receive}\ [\theta_n]\ call_n;\ \overline{s}_n) \cdot \epsilon) \\ \Downarrow \\ \forall i \in \{1,\ldots,n\} \bullet Q_{\mathsf{ar}(i)}(t) = \begin{cases} \mathsf{post}(call_{i+1}, \theta_{i+1}) & \text{if } i < n \\ \mathbf{true} & \text{if } i = n \end{cases} \end{array}$$

DEFINITION 25. *An activation record is* valid *if the verification condition of its continuation with respect to its postcondition holds under the current heap, lock set, and shared set, and under the activation record's access set. Formally:*

$$\begin{array}{l} t = (\mathsf{tid}, \mathsf{A}, (\overline{s}_1) \cdot \ldots \cdot (\overline{s}_n) \cdot \epsilon) \\ \Rightarrow\ (\forall i \in \{1,\ldots,n\} \bullet \\ \qquad \mathsf{H}, \mathsf{L}, \mathsf{S} \vdash \mathsf{valid}_{\mathsf{ar}(i)}(t) \\ \qquad \Leftrightarrow \mathfrak{I}, \mathsf{H}, \mathsf{L}^{-1}(\mathsf{tid}), \mathsf{S}, \mathsf{A}_{\mathsf{ar}(i)}(t) \vDash \mathsf{vc}(\overline{s}_i, Q_{\mathsf{ar}(i)}(t))) \end{array}$$

DEFINITION 26. *A program state is* valid*, written* $\mathsf{valid}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$*, if all of the following hold:*

—*it is well-formed*

$$\mathsf{wf}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$$

—*for each thread state* $t \in \mathsf{T}$ *where* $t = (\mathsf{tid}, \mathsf{A}, r_1 \cdot \cdots \cdot r_n)$*, all of the following hold:*
  —*it is well-formed*

$$\vdash (\mathsf{H}, \mathsf{L}^{-1}(\mathsf{tid}), \mathsf{S}, \mathsf{A}) : \mathsf{ok}$$

—*it is consistent*

$$\mathsf{L}^{-1}(\mathsf{tid}) \vdash \mathsf{consistent}(t)$$

—*each activation record is valid*

$$\forall i \in \{1, \ldots, n\} \bullet \mathsf{H}, \mathsf{L}, \mathsf{S} \vdash \mathsf{valid}_{\mathsf{ar}(i)}(t)$$

THEOREM 6. *A valid program state is a legal program state.*

PROOF. *One can easily prove that each thread is in a legal state by performing a case analysis on the first statement of the continuation of the top activation record and for each case using the validity of the top activation record.* □

THEOREM 7. *In a valid program π, the small step relation $\rightarrow$ preserves validity.*

$$\forall \sigma_1, \sigma_2 \bullet (\mathsf{valid}(\sigma_1) \wedge \sigma_1 \rightarrow \sigma_2) \Rightarrow \mathsf{valid}(\sigma_2)$$

PROOF. *By case analysis on the step. Note that preservation of well-formedness is given by Theorem 1. We refer to the thread that performs the step as the current thread and the top activation record of the current thread as the current activation record. For each step rule, we have to prove that in σ_2, thread states are well-formed and consistent and activation records are valid.*

Each step changes the current activation record's continuation. We only note other changes. Also, we only detail the argument for preservation of validity of the current activation record if it does not follow easily from the verification condition.

—***Case*** IF, ASSERT. *The step changes only the continuation of the current activation record. Therefore, thread state consistency and validity of the other activation records is preserved trivially.*

—***Case*** WRITE. *The step changes the heap at some location $o.f$, and the current activation record's continuation. Thread state consistency depends on neither so it is preserved trivially. By* VC-WRITE, *we have that o is in the current activation record's access set. Well-formedness of the thread-relevant state is preserved since the value written into the field is either* **null** *or a shared object. Since activation record access sets are disjoint and, as a result of the locality of continuation verification conditions (Theorem 5), activation record validity depends only on heap locations in the activation record's access set, validity of other activation records is preserved trivially.*

—***Case*** SHARE. *The step changes the shared set and the current thread's access set. Since the object being shared was in the current activation record's access set, it is not in the access set of any other activation record of the current thread so the access set still contains those. This establishes thread state consistency. The validity of an activation record is preserved by sharing an object that is not in its access set, and by removing from the thread's access set an object that is not in the activation record's access set, so the validity of non-current activation records is preserved.*

—***Case*** CALL. *The step changes the current activation record's continuation and adds a new activation record. Since the precondition holds, its required access set is contained in the caller's pre-state access set. Therefore, thread state consistency is preserved. Validity of existing non-current activation records is preserved trivially. Since the program is valid, the method being called is valid, and*

the method's body is valid in any state that satisfies the precondition. Note that the new activation record's access set is equal to the precondition's required access set.

—***Case*** RETURN. *The step replaces the caller and callee activation records with an activation record containing the continuation of the call. Validity of the caller activation record implies that the call's continuation is valid in any thread state that a) differs from the current state only as allowed by the method's frame condition and b) satisfies the postcondition. Validity of the callee implies that the postcondition holds in state σ_1. Therefore the call's continuation is valid in state σ_1 and, therefore, in state σ_2.*

—***Case*** NEWTHREAD. *The step adds a new thread with a single activation record, and removes the target object from the creating thread's access set. Thread state consistency of the new thread is trivial. The new activation record is valid since the method it executes is valid and its precondition, which by well-formedness of the program must be $\mathsf{L}_\mathrm{t} = \emptyset \wedge \mathbf{this} \in \mathsf{A}$, is satisfied.*

—***Case*** READ. *The step changes only the current activation record's continuation. Validity follows trivially from* VC-READ.

—***Case*** NEW. *The step adds an object to the heap domain and the thread's access set. Since by well-formedness access sets contain only allocated objects, access sets remain disjoint.*

—***Case*** SYNCHRONIZED. *The step adds an object to the lock set and the access set. Since by well-formedness the free set is disjoint from access sets and the object was in the free set, access sets remain disjoint.*

—***Case*** UNLOCK. *The step removes an object from the thread's access set. Since it was in the current activation record's access set, no other activation records are affected.*

This concludes the proof. □

THEOREM 8. *A valid program is a legal program.*

PROOF. *The initial state of a valid program is a valid state. It follows, by Theorem 7, that all reachable states are valid. Therefore, by Theorem 6, all reachable states are legal. Therefore, the program is legal.* □

THEOREM 9 MAIN THEOREM. *Valid programs are data-race-free.*

PROOF. *By combining Theorem 8 and Theorem 3.* □

3. INVARIANTS AND OWNERSHIP

The approach as described in the preceding section ensures absence of low-level data races. However, it does not prevent higher-level race conditions, where the programmer protects individual field accesses, but not updates involving accesses of multiple fields or objects that are part of the same data structure. As a result, accesses may be interleaved in such a way that the data structure's consistency is not maintained.

3.1 Programming model

To prevent race conditions that break the consistency of multi-object data structures, we integrate the Spec# methodology's object invariant and ownership system [Barnett et al. 2004] into our approach. The model supports objects that use other objects to help represent their state, and object invariants that express consistency constraints on such multi-object structures.

The programming model requires the programmer to designate a subset of each class's fields as the class's *rep fields*. The objects pointed to by an object o's non-null *rep* fields in a given program state are called o's *rep objects*. An object's *rep* objects may have *rep* objects themselves, and so on; we refer to all of these as the object's transitive *rep* objects. The fields of an object, along with those of its transitive *rep* objects, are considered in our approach to constitute the entire representation of the state of the object; hence the name. As will be explained later, a shared object o's lock protects both o and its transitive *rep* objects. (Formally, we have $o\ \mathrm{rep}_{\mathsf{H}}\ p$ if a field f exists such that $(o, f) \in \mathrm{dom}(\mathsf{H})$ and $\mathsf{H}(o, f) \neq \mathsf{null}$ and $\mathsf{H}(o, f) = p$ and f is a *rep* field. We use $\mathrm{rep}^*_{\mathsf{H}}$ to denote the reflexive-transitive closure of $\mathrm{rep}_{\mathsf{H}}$. If R is a relation on elements x, then we use $R(x)$ to denote $\{y \mid x\ R\ y\}$.)

In addition to a set of *rep* fields, the programming model requires the programmer to designate, for each class C, an *object invariant*, denoted $\mathsf{inv}(C)$. $\mathsf{inv}(C)$ is a predicate whose free variables are the heap H, the shared set S, and the target object **this**. $\mathsf{inv}(C)$ must depend only on the state of **this**, that is, the fields of **this** and of the transitive *rep* objects of **this**. Also, it must be preserved by growth of the shared set. Formally:

$$\forall \mathsf{H}, \mathsf{S}, \mathbf{this}, \mathsf{H}', \mathsf{S}' \bullet \mathsf{inv}(C) \wedge \mathsf{H}'|_{\mathrm{rep}^*_{\mathsf{H}}(\mathbf{this})} = \mathsf{H}|_{\mathrm{rep}^*_{\mathsf{H}}(\mathbf{this})} \wedge \mathsf{S} \subseteq \mathsf{S}' \Rightarrow \mathsf{inv}(C)[\mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}]$$

The object invariant for an object o need not hold in each program state. Rather, the programming model introduces a new global state variable P, called the *packed set*, which denotes a set of objects. The object invariant for an object o needs to hold only in a state where $o \in \mathsf{P}$.

The programming model requires an object to be in the packed set when a thread shares the object or unlocks it, i.e. when the object becomes free. It follows that each free object is in the packed set and its object invariant holds. As a result, when a thread locks an object, it may assume that the object is in the packed set and its object invariant holds.

In the program's initial state, the packed set is empty, and newly created objects are not in the packed set. The programmer may move an object into or out of the packed set using new **pack** o; and **unpack** o; annotations.

To ensure that whenever an object is in the packed set, its object invariant holds, the programming model imposes the following restrictions:

—A thread may assign to an object's fields only when the object is in the thread's access set *and* the object is not in the packed set. Furthermore, the remaining restrictions ensure that whenever an object is in the packed set, then so are its transitive *rep* objects. As a result, an object's state (defined as the values of its fields and those of its transitive *rep* objects) does not change while it is in the packed set.

—A thread is allowed to perform a **pack** o; operation only when o is in the thread's

access set, its object invariant holds, it is not yet in the packed set, and its *rep* objects are in the thread's access set and in the packed set. Furthermore, besides inserting the object into the packed set, the operation removes o's *rep* objects from the thread's access set.

—A thread is allowed to perform an **unpack** o; operation only when o is in the thread's access set and in the packed set. The operation removes o from the packed set and adds o's *rep* objects to the thread's access set.

We say that an object *owns* its *rep* objects whenever it is in the packed set. It follows from the above restrictions that an object has at most one owner.

Note that our approach supports ownership transfer; a *rep* object can be moved from one owner to another by first unpacking both owners and then simply updating the relevant *rep* fields.

The updated syntax rules are as follows:

$$\begin{array}{rcl} \mathit{class} & ::= & \textbf{class}\ C\ \textbf{implements}\ I^*\ \{\ \mathit{field}^*\ \textbf{invariant}\ \varphi;\ \mathit{meth}^*\ \} \\ \mathit{field} & ::= & \textbf{rep}^?\ f; \\ s & ::= & \cdots \mid \textbf{pack}_C\ g; \mid \textbf{unpack}_C\ g; \end{array}$$

Note: $\textbf{pack}_C\ g;$ and $\textbf{unpack}_C\ g;$ require g to be of class C. By fixing the class of g in the syntax, we can look up g's **rep** fields and invariant at verification condition generation time. This simplifies the verification logic and the verification conditions.

The new step rules and the new and updated legality rules are as follows:

$$[\textsc{Legal-Pack}]\ \frac{v \neq \mathsf{null} \qquad v \in \mathsf{A} \qquad \mathrm{rep}_{\mathsf{H}}(v) \subseteq \mathsf{A}}{\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{P} \vdash (\mathsf{tid},\mathsf{A},(\textbf{pack}_C\ v;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Pack}]\ \ (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{P},\mathsf{T} \lhd (\mathsf{tid},\mathsf{A},(\textbf{pack}_C\ v;\ \overline{s})\cdot\mathsf{F})) \rightarrow (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{P}\cup\{v\},\mathsf{T} \lhd (\mathsf{tid},\mathsf{A}\setminus \mathrm{rep}_{\mathsf{H}}(v),(\overline{s})\cdot\mathsf{F}))$$

$$[\textsc{Legal-Unpack}]\ \frac{v \neq \mathsf{null} \qquad v \in \mathsf{A}\cap\mathsf{P}}{\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{P} \vdash (\mathsf{tid},\mathsf{A},(\textbf{unpack}_C\ v;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Unpack}]\ \ (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{P},\mathsf{T} \lhd (\mathsf{tid},\mathsf{A},(\textbf{unpack}_C\ v;\ \overline{s})\cdot\mathsf{F})) \rightarrow (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{P}\setminus\{v\},\mathsf{T} \lhd (\mathsf{tid},\mathsf{A}\cup \mathrm{rep}_{\mathsf{H}}(v),(\overline{s})\cdot\mathsf{F}))$$

$$[\textsc{Legal-Write}]\ \frac{v_1 \neq \mathsf{null} \qquad v_1 \in \mathsf{A}\setminus\mathsf{P} \qquad \mathsf{classof}(v_1) = \mathsf{declaringClass}(f)}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(v_1.f := v_2;\ \overline{s})\cdot\mathsf{F}) : \mathsf{legal}}$$

More programs are legal in the programming model of this section (that is, more programs can be augmented with **share**, **rep**, **pack**, and **unpack** annotations so that the annotated program is legal under the programming model) since objects may be protected against data races by the lock of a transitive owner.

The definition of well-formedness of program states is updated as follows. The rule saying that the access sets and the free set partition the heap is replaced by a rule saying that the access sets, the free set, and the rep sets of packed objects (that is, the sets $\mathrm{rep}_{\mathsf{H}}(o)$ of objects $o \in \mathsf{P}$) partition the heap.

The definition of non-interfering state change is updated as follows:

Definition 27. *Two states are related by a* non-interfering state change *with respect to a given access set if*

—*all objects that are allocated in the first state are also allocated in the second state,*

—*all objects that are shared in the first state are also shared in the second state, and*

—*if an object is in the access set, then*
  —*the values of its fields are unchanged, and*
  —*if it is unshared in the pre-state, it is unshared in the post-state, and*
  —*if it is packed in the pre-state, it is packed in the post-state.*

Formally:

$$(\mathsf{H},\mathsf{S},\mathsf{P}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}',\mathsf{S}',\mathsf{P}')$$
$$\Updownarrow$$
$$\mathrm{dom}(\mathsf{H}) \subseteq \mathrm{dom}(\mathsf{H}') \wedge \mathsf{S} \subseteq \mathsf{S}' \wedge \mathsf{H}'|_{\mathsf{A}} = \mathsf{H}|_{\mathsf{A}} \wedge \mathsf{S}' \cap \mathsf{A} = \mathsf{S} \cap \mathsf{A} \wedge \mathsf{P}' \cap \mathsf{A} = \mathsf{P} \cap \mathsf{A}$$

Note: the object invariants and the monitor invariant (i.e. the invariant that says that free objects are packed) are not part of the programming model; they are part of the modular verification approach.

3.2 Program annotations

The example in Figure 8 shows the annotations required by the approach. A *Rectangle* object is used to store the bounds of an application's window. The *Rectangle*'s state is represented internally using two *Point* objects, that represent the location of the upper-left and lower-right corner, respectively. If the user drags the window's title bar, the window manager moves the window, even if the application is painting the window contents. Our methodology ensures that the application sees only valid states of the *Rectangle* object.

Developers designate a class's *rep* fields using the **rep** modifier, they define a class's object invariant using **invariant** declarations, and they insert **pack** and **unpack** commands in method bodies. Additionally, the new global state variable P may appear free in method preconditions and postconditions.

Another example is shown in Figure 9. It shows how an object may own an unbounded number of objects. If a class declares regular **rep** fields only, the number of owned objects is bounded by the number of **rep** fields. To remove this restriction, the approach allows *set-valued ghost rep fields*. Specifically, the approach allows the programmer to declare *ghost fields*, which are fields required for static verification but not for execution. The approach additionally allows ghost field assignments. Their right-hand side may be an arbitrary term in the verification logic. A ghost field may hold either an object reference, a null reference, or a set of object references. If a **ghost rep** field holds a set of object references, all objects referenced by the field are considered to be *rep* objects. This allows a *LinkedList* object to own all of its nodes, even though at run time it holds direct references to the sentinel nodes only.

$$\mathrm{objrefs}(v) \equiv \begin{cases} \emptyset & \text{if } v = \mathsf{null} \\ \{v\} & \text{if } v \in \mathcal{O} \\ v & \text{if } v \in \mathcal{P}(\mathcal{O}) \end{cases}$$

$$\mathrm{rep}_{\mathsf{H}}(o) \equiv \bigcup_{f \in \mathsf{repfields}(\mathsf{classof}(o))} \mathrm{objrefs}(\mathsf{H}(o,f))$$

```
class Point {
  int x, y;
  void move(int dx, int dy)
    requires this ∈ A ∩ P;
    ensures this ∈ A ∩ P;
  {
    unpack_Point this; x := x + dx;
    y := y + dy; pack_Point this;
  }
}
class Rectangle {
  rep Point ul, lr;
  invariant ul.x ≤ lr.x ∧ ul.y ≤ lr.y;
  void move(int dx, int dy)
    requires this ∈ A ∩ P;
    ensures this ∈ A ∩ P;
  {
    unpack_Rectangle this; ul.move(dx, dy);
    lr.move(dx, dy); pack_Rectangle this;
  }
  int getHeight()
    requires this ∈ A ∩ P;
    ensures this ∈ A ∩ P;
    ensures 0 ≤ result;
  {
    unpack_Rectangle this;
    int h := lr.y − ul.y;
    pack_Rectangle this; return h;
  }
}
```

```
class Application {
  Rectangle windowBounds;
  invariant windowBounds ∈ S;
  void paint()
    requires L_t = ∅;
    requires this ∈ A ∩ P;
    ensures this ∈ A ∩ P;
  {
    int height;
    synchronized (windowBounds) {
      height := windowBounds.getHeight();
    }
    ...
  }
}
class WindowManager {
  Rectangle windowBounds;
  invariant windowBounds ∈ S;
  void mouseDragged(int dx, int dy)
    requires L_t = ∅;
    requires this ∈ A ∩ P;
    ensures this ∈ A ∩ P;
  {
    synchronized (windowBounds) {
      windowBounds.move(dx, dy);
    }
  }
}
```

Fig. 8. An example illustrating the data race prevention approach, combined with object invariants and ownership. (In predicates, we abbreviate $\mathsf{H}(o, f)$ as $o.f$. Also, we abbreviate **this**.f as f, and we show variable types and method return types as an aid to the reader. In the example, we use integer values and operations in the program and in the verification logic. These are not in the formal development as they pose no difficulties.)

The example also shows how the approach supports ownership transfer. Method *append* transfers ownership of the non-sentinel nodes of *other* to **this**.

Note: An alternative to using set-valued ghost rep fields is to exploit the transitive nature of ownership. In the example, this would mean marking fields *first* and *next* as **rep**. A difficulty with this approach, however, is that modifying the i'th node requires $i+1$ **unpack** and **pack** operations, to unpack the node's transitive owners and the node itself before the modification, and then pack all of these objects in the reverse order afterwards. This practically imposes the use of recursion, which may be undesirable, especially for constant-time operations such as method *append* in the example.

```
class Node { Node prev, next; int value; }

class LinkedList {
  Node first, last;
  ghost rep Set⟨Node⟩ nodes;
  invariant first ≠ last;
  invariant first ∈ nodes ∧ last ∈ nodes;
  invariant (∀ n ∈ nodes • n = first ∨ n.prev ≠ null);
  invariant (∀ n ∈ nodes • n = last ∨ n.next ≠ null);
  invariant first.prev = null ∧ last.next = null;
  invariant (∀ n ∈ nodes • n.next ≠ null ⇒ n.next ∈ nodes);
  invariant (∀ n ∈ nodes • n.prev ≠ null ⇒ n.prev ∈ nodes);
  invariant (∀ n ∈ nodes • n.next = null ∨ n.next.prev = n);
  invariant (∀ n ∈ nodes • n.prev = null ∨ n.prev.next = n);

  void append(LinkedList other)
    requires other ≠ this;
    requires this ∈ A ∩ P ∧ other ∈ A ∩ P;
    ensures this ∈ A ∩ P;
  {
    Node otherFirst := other.first;
    Node otherLast := other.last;
    unpack this;
    unpack other;
    if (otherFirst.next ≠ otherLast) {
      nodes := nodes ∪ other.nodes \ {otherFirst, otherLast};
      Node left := last.prev;
      left.next := otherFirst.next;
      last.prev := otherLast.prev;
      left.next.prev := left;
      last.prev.next := last;
    }
    pack this;
  }

  void appendSync(LinkedList other)
    requires other ≠ this;
    requires this ∉ Lt ∧ other ∈ A ∩ P;
  { synchronized (this) { this.append(other); } }
}
```

Fig. 9. An example illustrating ownership transfer of unbounded numbers of aliased objects

```
class Rational {
  int p, q;
  invariant q ≠ 0;
  void multiply(int z)
    requires this ∈ A ∩ P;
    ensures this ∈ A ∩ P;
  { unpack this; p := p × z; pack this; }
}
class RationalBuffer {
  rep Rational element;
  void put(Rational x)
    requires this ∈ S \ Lt ∧ x ≠ null ∧ x ∈ A ∩ P;
    ensures true;
  {
    synchronized (this) {
      if (this.element = null) {
        done := true;
        unpack this;
        this.element := x;
        pack this;
      } else {
        done := false;
      }
    }
    if (¬done) { put(x); }
  }
  Rational get()
    requires this ∈ S \ Lt;
    ensures result ≠ null ∧ result ∈ A ∩ P;
  {
    synchronized (this) {
      unpack this;
      x := this.element;
      this.element := null;
      pack this;
    }
    if (x = null) { x := get(); }
    return x;
  }
}
```

```
class Producer {
  RationalBuffer buffer;
  invariant buffer ≠ null ∧ buffer ∈ S;
  void run()
    requires Lt = ∅ ∧ this ∈ A ∩ P;
    ensures true;
  {
    r := new Rational;
    r.p := 1;
    r.q := 1;
    pack r;
    buffer.put(r);
    this.run();
  }
}
class Processor {
  RationalBuffer buffer1, buffer2;
  invariant buffer1 ≠ null ∧ buffer1 ∈ S;
  invariant buffer2 ≠ null ∧ buffer2 ∈ S;
  void run()
    requires Lt = ∅ ∧ this ∈ A ∩ P;
    ensures true;
  {
    r := buffer1.get();
    r.multiply(7);
    buffer2.put(r);
    this.run();
  }
}
class Consumer {
  RationalBuffer buffer;
  invariant buffer ≠ null ∧ buffer ∈ S;
  void run()
    requires Lt = ∅ ∧ this ∈ A ∩ P;
    ensures true;
  {
    r := buffer1.get();
    // Print r.p and r.q (not shown)
    this.run();
  }
}
```

```
b1 := new RationalBuffer; pack b1; share b1;
b2 := new RationalBuffer; pack b2; share b2;
prod := new Producer; prod.buffer := b1; pack prod;
proc := new Processor; proc.buffer1 := b1; proc.buffer2 := b2; pack proc;
cons := new Consumer; cons.buffer := b2; pack cons;
start prod.run(); start proc.run(); start cons.run();
```

Fig. 10. An example showing how the ownership system can be used to transfer unshared objects between threads. In methods *get* and *put*, tail-recursive busy waiting is used instead of the conventional *Object.wait* loop because the latter are not part of the formal language of this article.

In our approach, once an object is shared, it never reverts to the unshared state. This has the advantage that, when verifying a method, it may be assumed that if an object is shared at a given point in time, then it will be shared in all subsequent program states, regardless of the actions of other threads. Note also that this does not prevent scenarios where an object is passed from one thread to another and then accessed without locking. This is illustrated by the example in Figure 10. In the example, an unshared *Rational* object is passed from a producer thread to a processor thread and then on to a consumer thread. Each thread accesses the object without locking. Each transfer proceeds via a shared *RationalBuffer* object: first, the *RationalBuffer* object is locked and unpacked. Then, a reference to the *Rational* object is stored in the *RationalBuffer* object's *element* field, which is a **rep** field. Then, the *RationalBuffer* object is packed, which causes the *Rational* object to be removed from the thread's access set and to become owned by the *RationalBuffer* object. Then, the *RationalBuffer* object is unlocked. When the receiver thread subsequently locks the *RationalBuffer* object, it unpacks it, which causes it to relinquish ownership of the *Rational* object and causes the latter to be added to the receiver thread's access set. Finally, clearing the *element* field prevents the *Rational* object from again becoming owned by the *RationalBuffer* object when the latter is packed again.

Note: The example of Figure 10 shows how invariants can be used to state that certain fields hold shared objects. This is why the built-in invariant of the previous section that all fields hold shared objects, is no longer needed.

3.3 Static verification

The definition of program well-formedness is updated as follows. The rule concerning the method contract of a method used in a **start** statement is replaced with the following: If a method is used in a **start** statement, then its precondition must be exactly $\mathsf{L}_\mathrm{t} = \emptyset \wedge \mathbf{this} \in \mathsf{A} \cap \mathsf{P}$.

The definition of thread-relevant state is updated to include the packed set.

$$\theta = (\mathsf{H}, \mathsf{L}_\mathrm{t}, \mathsf{S}, \mathsf{P}, \mathsf{A})$$

The definition of well-formedness of a thread-relevant state is updated as follows. The following conjunct is added: if an object is in the packed set, then its object invariant holds and its rep objects are also in the packed set.

$$\forall o \in \mathsf{P} \bullet \mathsf{inv}(C)[o/\mathbf{this}] \wedge \mathrm{rep}_\mathsf{H}(o) \subseteq \mathsf{P}$$

Also, the conjunct that says that objects pointed to by fields are shared is dropped, since now object invariants can be used to express this.

New and updated continuation verification condition rules are shown in Figure 11.

The definition of validity of a program state is updated by adding a conjunct saying that free objects are packed.

$$S - \mathrm{dom}(\mathsf{L}) \subseteq \mathsf{P}$$

THEOREM 10. *In the approach of this section, legal programs are data-race-free, and valid programs are legal.*

$\mathsf{vc}(v_1.f := v_2;\ \overline{s}, Q) \equiv$ [VC-Write]
$v_1 \neq \mathsf{null} \wedge \mathsf{classof}(v_1) = \mathsf{declaringClass}(f) \wedge v_1 \in \mathsf{A} \setminus \mathsf{P} \wedge \mathsf{vc}(\overline{s}, Q)[\mathsf{H}[(v_1, f) \mapsto v_2]/\mathsf{H}]$

$\mathsf{vc}(\mathbf{share}\ v;\ \overline{s}, Q) \equiv$ [VC-Share]
$v \neq \mathsf{null}\ \wedge\ v \in \mathsf{A} \cap \mathsf{P}\ \wedge\ v \notin \mathsf{S} \wedge\ \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{v\})/\mathsf{A}, (\mathsf{S} \cup \{v\})/\mathsf{S}]$

$\mathsf{vc}(\mathbf{synchronized}\ (v)\ \{\ \overline{s}'\ \}\ \overline{s}, Q) \equiv$ [VC-Synchronized]
$v \neq \mathsf{null}\ \wedge\ v \in \mathsf{S}\ \wedge\ v \notin \mathsf{L_t}$
$\wedge\ (\forall \mathsf{H}', \mathsf{S}', \mathsf{P}' \bullet$
$\quad ((\mathsf{H}, \mathsf{S}, \mathsf{P}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}', \mathsf{P}') \wedge\ \vdash (\mathsf{H}', \mathsf{L_t}, \mathsf{S}', \mathsf{P}', \mathsf{A}) : \mathsf{ok} \wedge v \in \mathsf{P}')$
$\quad \Rightarrow$
$\quad \mathsf{vc}(\overline{s}'\ \mathbf{unlock}\ v;\ \overline{s}, Q)[\mathsf{H}'/\mathsf{H}, (\mathsf{A} \cup \{v\})/\mathsf{A}, (\mathsf{L_t} \cup \{v\})/\mathsf{L_t}, \mathsf{S}'/\mathsf{S}, \mathsf{P}'/\mathsf{P}])$

$\mathsf{vc}(\mathbf{unlock}\ o;\ \overline{s}, Q) \equiv$ [VC-Unlock]
$o \in \mathsf{A} \cap \mathsf{P}\ \wedge\ o \in \mathsf{L_t}\ \wedge\ \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{o\})/\mathsf{A}, (\mathsf{L_t} \setminus \{o\})/\mathsf{L_t}]$

$\mathsf{vc}(x := v.m(\overline{v});\ \overline{s}, Q) \equiv$ [VC-Call]
$v \neq \mathsf{null}\ \wedge\ \mathsf{classof}(v) \preceq \mathsf{declaringType}(m) \wedge \mathsf{pre}(v.m(\overline{v}))$
$\wedge\ \mathsf{vc}(x := \mathbf{receive}\ [(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{P}, \mathsf{A})]\ v.m(\overline{v});\ \overline{s}, Q)$

$\mathsf{vc}(\mathbf{start}\ v.m();\ \overline{s}, Q) \equiv$ [VC-NewThread]
$v \neq \mathsf{null}\ \wedge \mathsf{classof}(v) \preceq \mathsf{declaringType}(m) \wedge\ v \in \mathsf{A} \cap \mathsf{P}\ \wedge\ \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{v\})/\mathsf{A}]$

$\mathsf{vc}(x := \mathbf{receive}\ [(\mathsf{H}^{\mathrm{old}}, \mathsf{L_t^{old}}, \mathsf{S}^{\mathrm{old}}, \mathsf{P}^{\mathrm{old}}, \mathsf{A}^{\mathrm{old}})]\ o.m(\overline{v});\ \overline{s}, Q) \equiv$ [VC-Receive]
$\forall \mathsf{H}', \mathsf{S}', \mathsf{P}', \mathsf{A}', v_{\mathrm{r}} \bullet$
$\quad ((\mathsf{H}, \mathsf{S}, \mathsf{P}) \overset{\mathsf{A}^{\mathrm{old}} - \mathcal{A}(P')}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}', \mathsf{P}') \wedge\ \vdash (\mathsf{H}', \mathsf{L_t}, \mathsf{S}', \mathsf{P}', \mathsf{A}') : \mathsf{ok}$
$\quad \wedge\ \mathsf{A}' \cap (\mathsf{A}^{\mathrm{old}} - \mathcal{A}(P')) = \mathsf{A} \cap (\mathsf{A}^{\mathrm{old}} - \mathcal{A}(P'))$
$\quad \wedge\ Q'[\mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}, \mathsf{P}'/\mathsf{P}, \mathsf{A}'/\mathsf{A}, \mathsf{L}_t^{old}/\mathsf{L_t}, v_r/\mathbf{result}]$
$\quad \wedge\ (v_{\mathrm{r}} = \mathsf{null} \vee v_{\mathrm{r}} \in \mathrm{dom}(\mathsf{H}'))$
$\quad \Rightarrow$
$\quad \mathsf{vc}(\overline{s}, Q)[\mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}, \mathsf{P}'/\mathsf{P}, \mathsf{A}'/\mathsf{A}, \mathsf{L}_t^{old}/\mathsf{L_t}, v_r/x]$
where $P' = \mathsf{pre}(o.m(\overline{v}))$ and $Q' = \mathsf{post}(o.m(\overline{v}))$

$\mathsf{vc}(\mathbf{pack}_C\ v;\ \overline{s}, Q) \equiv$ [VC-Pack]
$v \neq \mathsf{null} \wedge \mathsf{classof}(v) = C \wedge v \in \mathsf{A} \setminus \mathsf{P} \wedge (\forall\ p \in \mathrm{rep}_{\mathsf{H}}(v) \bullet p \in \mathsf{A} \wedge p \in \mathsf{P})$
$\wedge\ \mathsf{inv}(C)[v/\mathbf{this}]\ \wedge\ \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \mathrm{rep}_{\mathsf{H}}(v))/\mathsf{A}, (\mathsf{P} \cup \{v\})/\mathsf{P}]$

$\mathsf{vc}(\mathbf{unpack}_C\ v;\ \overline{s}, Q) \equiv$ [VC-Unpack]
$v \neq \mathsf{null} \wedge \mathsf{classof}(v) = C \wedge v \in \mathsf{A} \cap \mathsf{P} \wedge \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \cup \mathrm{rep}_{\mathsf{H}}(v))/\mathsf{A}, (\mathsf{P} \setminus \{v\})/\mathsf{P}]$

Fig. 11. VC generation for Section 3

4. LOCK RE-ENTRY

In the programming model preceding sections, lock re-entry is ruled out. That is, a program that re-enters a lock is considered illegal and invalid. However, it is not difficult to add support for lock re-entry to the programming model of the preceding section. We show an approach where a lock re-entry is treated like a no-op. In this approach, the only modification required to the programming model is to relax the legality rule and to add a second step rule for **synchronized** blocks, and the only modification required to the static verification approach is to add a case split to the verification condition rule for **synchronized** blocks. The new or updated definitions are shown in Figure 12.

$$\text{[Legal-Synchronized]}\ \frac{v \neq \mathsf{null} \qquad v \in \mathsf{S}}{\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash (\mathsf{tid}, \mathsf{A}, (\textbf{synchronized}\ (v)\ \{\ \overline{s}'\ \}\ \overline{s}) \cdot \mathsf{F}) : \mathsf{legal}}$$

$$\text{[Synchronized-Reentrant]}\ \frac{v \in \mathsf{L}^{-1}(\mathsf{tid})}{(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}, \mathsf{A}, (\textbf{synchronized}\ (v)\ \{\ \overline{s}'\ \}\ \overline{s}) \cdot \mathsf{F})) \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}, \mathsf{A}, (\overline{s}'\ \overline{s}) \cdot \mathsf{F}))}$$

$$\begin{array}{l}
\mathsf{vc}(\textbf{synchronized}\ (v)\ \{\ \overline{s}'\ \}\ \overline{s}, Q) \equiv \qquad \text{[VC-Synchronized]} \\
\quad v \neq \mathsf{null}\ \wedge\ v \in \mathsf{S} \\
\quad \wedge\ (v \notin \mathsf{L_t} \Rightarrow \\
\qquad (\forall \mathsf{H}', \mathsf{S}', \mathsf{P}' \bullet \\
\qquad\quad ((\mathsf{H}, \mathsf{S}, \mathsf{P}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}', \mathsf{P}')\ \wedge\ \vdash (\mathsf{H}', \mathsf{L_t}, \mathsf{S}', \mathsf{P}', \mathsf{A}) : \mathsf{ok} \wedge v \in \mathsf{P}') \\
\qquad\quad \Rightarrow \\
\qquad\quad \mathsf{vc}(\overline{s}'\ \textbf{unlock}\ v;\ \overline{s}, Q)[\mathsf{H}'/\mathsf{H}, (\mathsf{A} \cup \{v\})/\mathsf{A}, (\mathsf{L_t} \cup \{v\})/\mathsf{L_t}, \mathsf{S}'/\mathsf{S}, \mathsf{P}'/\mathsf{P}])) \\
\quad \wedge\ (v \in \mathsf{L_t} \Rightarrow \mathsf{vc}(\overline{s}'\ \overline{s}, Q))
\end{array}$$

Fig. 12. New or updated legality rule, step rule, and verification condition rule for lock re-entry

Program	Lines of Code	Lines Changed or Added	Overhead	After Defaults and Inference	Net Overhead
chat	344	117	34%	41	13%
phone	222	50	23%	14	6%
prod-cons	84	24	29%	0	0%
philosophers	64	21	33%	5	8%

Table I. Annotation overhead

5. EXPERIENCE

To verify the applicability of our approach to realistic, useful programs, we implemented it in a custom build of the Spec# program verifier [Barnett et al. 2006] and used it to verify four programs written in C# with annotations inserted in the form of specially marked comments. The approach that we implemented includes elements omitted from this article, including deadlock prevention and immutable objects [Jacobs 2007]. Each program verifies successfully; this guarantees the following:

—The program is free from data races and deadlocks.

—Object invariants, loop invariants, method preconditions and postconditions, and assert statements declared by the program hold.

—The program is free from null dereferences, array index out of bounds errors, and typecasting errors.

—The program is free from races on platform resources such as network sockets. This is achieved by enforcing concurrency contracts on the relevant API methods.

Table I shows the annotation overhead of the four programs which we annotated and verified. Programs chat and phone were derived from the ones used in [Boyapati et al. 2002].

We assessed which lines containing annotations could be eliminated by adopting common annotations implicitly by default and by inferring annotations that can be inferred easily. The measures considered include inferring that objects of a given class are always shared after construction (eliminates 38 lines across all four programs), adding unpack and pack statements at the start and end of methods automatically (17 lines), inferring simple loop invariants stating that certain objects are in the access set and the packed set (25 lines), and a default method contract for a thread's *Run* method (8 lines).

The estimated number of lines containing annotations remaining after these and other simple measures are applied are shown in Column 5 in Table I. The remaining annotations deal mainly with the lock order for deadlock prevention (not discussed in this article) and the need to track the read and write channel of network sockets separately, even though in the .NET Framework they are represented by a single object. A more elaborate inference scheme might be able to infer these annotations as well. This preliminary assessment indicates that the annotation overhead of our approach can probably be brought to an acceptable level; however, this needs to be confirmed through experiments on larger codebases.

Our experiments gave us some insight into the consequences of our decision to use a automatic theorem prover (specifically, the Simplify theorem prover). On the one hand, the theorem prover successfully proved the verification conditions generated from the abovementioned programs. On the other hand, performance was less than stellar: verification of the chat server takes half an hour. Furthermore, we experienced the theorem prover's incompleteness: some methods verified only after we manually inserted a number of well-chosen intermediate proof obligations (using special annotations in the source code), which the theorem prover, after proving them, could use as lemmas to prove subsequent proof obligations. It took some trial and error for us to find the right intermediate proof obligations. Nine of them were needed in program chat, and none in the other programs. An encouraging observation in this respect is that ongoing theorem proving research continues to yield more powerful and more efficient theorem provers.

The prototype verifier and the sample programs are available at the first author's home page at http://www.cs.kuleuven.be/~bartj/specleuven/.

6. DISCUSSION

In this article, we propose a programming model and verification approach for multithreaded object-oriented programs. We focused on designing a simple approach to multithreading that integrated well with the Boogie approach [Barnett et al. 2004] for object invariants and dynamic ownership, yielding what we believe to be the first sound program verification approach that supports both multithreading, dynamic ownership, and object invariants over an object and its transitively owned objects.

Our approach is not complete. That is, not every Java program that is data-race-free can be annotated such that verification using our approach succeeds. The incompleteness exists at three levels: the programming model, the verification approach, and the theorem prover.

6.1 Programming model

Not every program that is data-race-free complies with the programming model, or can be made to do so by inserting **share**, **rep**, **pack**, and **unpack** annotations.

A basic limitation is that objects, not fields, are in access sets; therefore, two threads can never access distinct fields of a given object concurrently. We inherited this limitation from the Boogie approach. One way to lift it would be to drop the Boogie approach in favor of an approach based on *dynamic frames* [Kassios 2006], which subsume the Boogie approach's support for dynamic ownership and object invariants. However, the suitability of dynamic frames for automatic verification has not been shown.

Our approach, as described in this article, does not distinguish between read and write access. However, it is fairly easy to replace access sets with read sets and write sets, and this has been implemented in the prototype verifier. Based on read sets and write sets, it is easy to add support for unrestricted sharing of immutable objects, and for reader-writer locks.

Programs that use synchronization constructs other than Java's **synchronized** blocks may or may not be supported. For example, in Figure 10, a *RationalBuffer* object can be used as a binary semaphore, where *put* and *get* calls correspond to V and P operations, respectively. However, programs where volatile fields are used to protect data structures, for example, are not supported. Still, in some cases it might be possible to encapsulate an unsupported construct within a class and then enforce the correct use of the class by annotating the class's methods with appropriate method contracts and verifying client code using our approach.

6.2 Static verification

Even if a program complies with our programming model, it might not be a valid program; i.e., it might not be possible to annotate each method with a method contract and each class with an invariant, so that in the resulting program, each method's verification condition holds. The main source of incompleteness on this level is the imprecise modeling of inter-thread interference.

In our verification approach, when verifying a thread, the interference of other threads is taken into account by generating verification conditions as if on entry to a **synchronized** block, an arbitrary new value is assigned to each field of each object that is not in the access set, with the only restriction being that if an object is in the packed set, its object invariant holds. This means that any monotonicity properties preserved by the program are not taken into account. For example, if an integer field of a shared object is only ever incremented, and never decremented, by threads that lock the object, it is still not possible to prove, in our approach, that a value read by a thread from the field is greater than or equal to a value read by that thread in an earlier **synchronized** block. It seems possible to lift this source of incompleteness by extending our approach with support for rely-guarantee conditions. In this extended approach, a non-interfering state change would be defined as one that, in addition to the current requirements, satisfies the rely condition. The locality requirement on method preconditions and postconditions would become weaker accordingly: a contract may mention state outside the access set, provided it is preserved by state changes that satisfy the rely condition.

Note that the ability to insert ghost field declarations and ghost field updates into the program is also essential for completeness. Indeed, a method's correctness may depend on the local states of other threads. For example, suppose one thread initially only increments a shared counter, and then, after it receives a message through some shared message queue, it only decrements the counter. A method executing in another thread may depend on the first thread being in its initial state. However, since neither method contracts nor rely conditions may mention threads' local state (i.e. their call stacks), there is no way to express this, except by mirroring a thread's local state in the global state using ghost fields which are kept up to date by the thread.

Given rely conditions and ghost fields, we can now show completeness. Take an arbitrary program that complies with the programming model. Insert a single static integer-valued ghost field. It will at all times contain a Gödel-like encoding of the entire program state. Initialize it with an encoding of the initial program state. After each statement in each method, insert a ghost field update that updates the ghost field to reflect the new program state. As the rely condition, take the condition that says that the old and new global state corresponds to the program states encoded in the old and new values of the ghost field, and that the program state encoded in the new value of the ghost field is reachable from the program state encoded in the old value of the ghost field through steps by threads other than the current thread. Remember that reachability is definable in arithmetic. As a method's precondition, take the condition that says that the pre-state corresponds to the ghost field value and that it is reachable from the initial state. As a method's postcondition, take the condition that says that the state corresponds to the ghost field value and that the post-state is reachable from the pre-state.

6.3 Theorem prover

The third source of incompleteness is the proof step. There are two aspects to this: there is a theoretical limitation, and additionally, there is a practical limitation. The theoretical limitation is that not all true statements in arithmetic, and therefore in our verification logic, are provable from any given axiomatization. Indeed, for any theory Σ, there is a program that is data-race-free and yet its data-race-freedom is not provable from Σ. (Consider the program that enumerates all proofs, and, if it finds a proof of its own data-race-freedom, performs a data race.)

The practical limitation is that even provable statements are often not proved within a reasonable time bound by automatic theorem provers. After all, even propositional satisfiability is NP-complete. This is a serious concern for our approach. In Section 5, we report on our initial experience in this respect.

7. RELATED WORK

The present approach evolved from [Jacobs et al. 2005a], [Jacobs et al. 2006], and [Jacobs 2007]. It improves upon this prior work by adding a formalization of the approach with invariants and ownership. (A soundness proof [Jacobs et al. 2005b] accompanies [Jacobs et al. 2005a] but it does not formalize verification condition generation, and it does not formalize or prove the method effect framing approach.) As did the prior work, the present approach builds on and extends the Spec# programming methodology [Barnett et al. 2004] that enables sound reasoning about

object invariants in sequential programs. For brevity, we omitted the description of the deadlock prevention approach and the approach for verification of immutable objects, static fields, and lazy class initialization [Jacobs 2007] from this article.

The Extended Static Checkers for Modula-3 [Detlefs et al. 1998] and Java [Flanagan et al. 2002] attempt to statically find errors in object-oriented programs. These tools include support for the prevention of data races and deadlocks. For each field, a programmer can designate which lock protects it. However, these two tools trade soundness for ease of use; for example, they do not take into consideration the effects of other threads between regions of exclusion. Moreover, various engineering trade-offs in the tools notwithstanding, the methodology used by the tools was never formalized enough to allow a soundness proof.

Calvin-R [Freund and Qadeer 2004; Flanagan et al. 2005] is a static checker for multithreaded Java programs. To use the tool, the developer first annotates the program: for each field, they specify an *access predicate*, which may mention the current thread **tid**. The tool checks through verification condition generation that whenever a thread reads or writes a field, the field's access predicate holds for that thread. The environment's interference is taken into account by assuming that between any two operations of the method being checked, the environment performs an arbitrary set of field updates, constrained only by the access predicates. Besides the access predicates, the developer may specify for each method a *performs annotation*, which specifies a *guarantee program*, which is a set of atomic action specifications composed using sequential composition, choice, and iteration operators. An atomic action specification is a two-state predicate. The tool checks, again through verification condition generation, that each method's performs annotation *simulates* the method's body. In summary, Calvin-R verifies the program's compliance with the synchronization policy specified in the access predicates, as well as the functional properties specified in the performs annotations.

Like our approach, Calvin-R is thread-modular and method-modular and based on verification condition generation. However, Calvin-R is strictly more expressive: on the one hand, its access predicates enable verifying strictly more synchronization patterns, such as patterns where different fields are protected by different locks, or where semaphores, barriers, or wait-free constructs are used instead of mutexes, or where hand-over-hand locking or other fine-grained synchronization approaches are used. On the other hand, Calvin-R's performs annotations enable the specification and verification of functional properties which cannot be expressed using our approach's method contracts based on pre- and postconditions. Indeed, if a program can be annotated and verified using our approach, the annotations can probably be translated fairly straightforwardly into Calvin-R's syntax and verified using Calvin-R, but not the other way around.

The contribution of our work, then, is to propose a particular *specification approach*, which ensures that, if a given class is specified using our approach, an object of that class may initially be used by the creating thread without synchronization, then shared and accessed through synchronization, or, alternatively, not shared but used as a *rep* object of some owner object. Furthermore, our specification approach ensures that an object's client code is independent of the object's internal structure, e.g., whether or not it owns any *rep* objects. In other words, our work

addresses the issue of data structure abstraction and data structure composition, which is not addressed by the Calvin-R work. (They do address a different type of abstraction: an atomic action specification in a performs annotation abstracts over the particular operations performed to implement the atomic action.)

A number of type systems have been proposed that prevent data races in object-oriented programs. For example, Boyapati *et al.* [Boyapati et al. 2002] parameterize classes by the protection mechanism that will protect their objects against data races. The type system supports thread-local objects, objects protected by a lock (its own lock or its root owner's lock), read-only objects, and unique pointers. The system does not support forms of ownership transfer other than transfer of unique pointers. For example, it cannot type the program of Figure 9. Also, the type system does not support object invariants.

We enable sequential reasoning and ensure consistency of aggregate objects by preventing data races. Some authors propose pursuing a different property, called *atomicity*, either through dynamic checking [Flanagan and Freund 2004], by way of a type system [Flanagan and Qadeer 2003], or using a theorem prover [Rodríguez et al. 2005]. An atomic method can be reasoned about sequentially. However, we enable sequential reasoning even for non-atomic methods, by assuming only the object invariant for a newly acquired object (see Figure 7). Also, in [Flanagan and Qadeer 2003] the authors claim that data-race-freedom is unnecessary for sequential reasoning. It is true that some data races are benign, even in the Java and C# memory models; however, the data races allowed in [Flanagan and Qadeer 2003] are generally not benign in these memory models; indeed, the authors prove soundness only for sequentially consistent systems, whereas we prove soundness for the Java memory model, which is considerably weaker.

Ábrahám-Mumm *et al.* [Ábrahám-Mumm et al. 2002] propose an assertional proof system for Java's reentrant monitors. It supports object invariants, but these can depend only on the fields of **this**. No claim of modular verification is made.

The rules in our methodology that an object must be valid when it is released, and that it can be assumed to be valid when it is acquired, are taken from Hoare's work on monitors and monitor invariants [Hoare 1974].

There are also tools that try dynamically to detect violations of safe concurrency. A notable example is Eraser [Savage et al. 1997]. It finds data races by looking for locking-discipline violations. The tool has been effective in practice, but does not come with guarantees about the completeness nor the soundness of the method.

This article focuses on programs that use **synchronized** blocks for synchronization. Significant research has been done on improving the implementation of **synchronized** blocks in virtual machines and/or compilers, so that, while preserving Java semantics, opportunities for parallelism are increased. For example, some proposals infer, fully automatically or aided by annotations, fine-grained locking schemes. Others propose applying a form of optimistic concurrency, such as transactional monitors [Welc et al. 2004]. Typically, these schemes require the input program to be data-race-free; therefore, our approach is equally applicable in these settings.

8. CONCLUSION

We propose a programming model for concurrent programming in Java-like languages, and the design of a set of program annotations that make the use of the programming model explicit and that enable automated verification of compliance. Our programming model ensures absence of data races, and provides a sound approach for local reasoning about program behavior. We have prototyped the verifier as a custom build of the Spec# programming system. Through a case study we show that the model supports non-trivial, useful programs, and we assess the annotation overhead.

Future work includes extending the programming model to encompass read-write locks, reducing the annotation overhead, and obtaining further experience.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

Ábrahám-Mumm, E., de Boer, F. S., de Roever, W.-P., and Steffen, M. 2002. Verification for Java's reentrant multithreading concept. In *Proc. Foundations of Software Science and Computation Structures (FoSSaCS)*, M. Nielsen and U. Engberg, Eds. Lecture Notes in Computer Science, vol. 2303. Springer, 5–20.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. Formal Methods for Components and Objects (FMCO)*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Lecture Notes in Computer Science, vol. 4111. Springer, 364–387.

Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. 2004. Verification of object-oriented programs with invariants. *Journal of Object Technology 3,* 6, 27–56.

Barnett, M., Leino, K. R. M., and Schulte, W. 2004. The Spec# programming system: An overview. In *Proc. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds. Lecture Notes in Computer Science, vol. 3362. Springer, 49–69.

Boyapati, C., Lee, R., and Rinard, M. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA), S. Matsuoka, Ed. *SIGPLAN Notices 37,* 11, 211–230.

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. 1998. Extended static checking. Research Report 159, Compaq Systems Research Center.

Flanagan, C. and Freund, S. N. 2004. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. Principles of Programming Languages (POPL)*, X. Leroy, Ed. ACM, 256–267.

Flanagan, C., Freund, S. N., Qadeer, S., and Seshia, S. A. 2005. Modular verification of multithreaded programs. *Theoretical Computer Science 338,* 1-3, 153–183.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for Java. In Proc. Programming Language Design and Implementation (PLDI), L. J. Hendren, Ed. *SIGPLAN Notices 37,* 5, 234–245.

Flanagan, C. and Qadeer, S. 2003. A type and effect system for atomicity. In *Proc. Programming Language Design and Implementation (PLDI)*, S. Amarasinghe, Ed. ACM, 338–349.

Freund, S. N. and Qadeer, S. 2004. Checking concise specifications for multithreaded software. *Journal of Object Technology 3,* 6, 81–101.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2005. *The Java Language Specification (3rd Edition).* Prentice Hall.

Notation	Page	Meaning
o	4	an object reference (Note: null is not an object reference)
tid	4	a thread identifier
$\mathcal{C}$	6	the set of class names
$\mathcal{I}$	6	the set of interface names
$\mathcal{M}$	6	the set of method names
$\mathcal{F}$	6	the set of field names
$\mathcal{X}$	6	the set of variable names
Φ	6	the set of logical formulae
$\mathcal{O}$	6	the set of object references
Θ	6	the set of thread-relevant states
C	6	a class name
I	6	an interface name
m	6	a method name
f	6	a field name
x	6	a variable name
s	6	a statement
φ	6	a logical formula
θ	6	a thread-relevant state
π	6	a program
g	6	an expression (i.e., **this** or a variable name or or a literal)
v	6	a value (i.e., null or an object reference)
L_t	7	a lockset (i.e., the set of objects whose lock is held by the current thread)
A	7	an access set
$\emptyset$	7	the empty set or the empty function
$\mathcal{T}$	9	the set of thread identifiers
$\mathsf{fields}(C)$	10	the set of fields declared by class C
$C \prec I$	10	class C mentions interface I in its **implements** clause
$\overline{x}, \overline{v}, \overline{s}$	10	a sequence of variables, values, statements
$\mathsf{classof}(o)$	10	the name of the class of an object o
$\mathsf{mbody}(o.m(\overline{v}))$	10	the body of method m declared by $\mathsf{classof}(o)$, with o substituted for **this** and $\overline{v}$ for the method's parameters
$\mathsf{declaringClass}(f)$	10	the name of the class that declares field f
$\mathsf{declaringType}(m)$	10	the name of the interface that declares method m, or the name of the class that declares method m if no interface declares a method m
$\mathsf{objectRefs}(\phi)$	10	the set of object references in syntactic entity ϕ
$\mathsf{free}(\phi)$	10	the set of free variables in syntactic entity ϕ
$f[a \mapsto b]$	10	function update
$\overline{s}[v/x]$	10	substitution of a value for a variable in a statement list
$\varphi[t/x]$	10	substitution of a term for a variable in a formula
ϵ	10	the empty sequence
$h \cdot t$	10	the sequence with head h and tail t
σ	10	a program state
$S \hookrightarrow S'$	10	the set of finite partial functions from set S to set S'
H	10	a heap
L	10	a lock map
S	10	a shared set
T	10	a thread set
F	10	a call stack (i.e., a sequence of activation records)
$\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash t : \mathsf{legal}$	10	thread state t is legal in the given context
$\sigma_1 \rightarrow \sigma_2$	10	an execution step is possible from program state σ_1 to program state σ_2
τ	11	a class or interface name
$\tau_1 \preceq \tau_2$	11	a shorthand for $\tau_1 \prec \tau_2 \vee \tau_1 = \tau_2$
$\overline{s}_1\ \overline{s}_2$	11	the concatenation (i.e., sequential composition) of statement lists $\overline{s}_1$ and $\overline{s}_2$
$\mathsf{T} \lhd t$	12	a shorthand for $\mathsf{T} \cup \{t\}$
$\mathsf{initial}(\sigma)$	13	program state σ is an initial program state
$\mathsf{legal}(\sigma)$	13	program state σ is a legal program state
$\mathsf{program\_wf}$	13	the program is well-formed (Note: Here and throughout, the program is implicit)
$\mathsf{program\_legal}$	13	the program is legal
$\forall x \bullet \varphi(x)$	13	statement $\varphi(x)$ holds for all x
R^*	13	the reflexive and transitive closure of relation R
$S \setminus S'$	13	set difference
$S - S'$	13	multiset difference

Table II. Overview of notations used in the article

Notation	Page	Meaning
$S \ll T$	13	the multiset of sets S partitions set T
$\mathsf{H} \vdash \mathsf{S} : \mathsf{ok}$	13	shared set S is well-formed in the given context
$\vdash \mathsf{H} : \mathsf{ok}$	14	heap H is well-formed
$\mathsf{H}, \mathsf{S} \vdash \mathsf{L} : \mathsf{ok}$	14	lock map L is well-formed in the given context
$\mathsf{wf}(\sigma)$	14	program state σ is well-formed
$S \uplus S'$	14	multiset union
$(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}')$	17	a state with heap H and shared set S is related to a state with heap H' and shared set S' by a non-interfering state change with respect to access set A
$f\|_S$	17	function restriction
t	18	logical term
$\mathcal{I}$	18	the intended interpretation of the verification logic
Σ	19	an axiomatization of $\mathcal{I}$
$\vdash (\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}) : \mathsf{ok}$	20	thread-relevant state $(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})$ is well-formed
$\mathcal{I}, \mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}, V \vDash Q$	20	the truth of a state predicate Q under interpretation $\mathcal{I}$, in thread-relevant state $\mathsf{L_t}, \mathsf{S}, \mathsf{A}$, and under program variable valuation V
$\mathsf{local}(Q)$	21	state predicate Q is local
$\mathcal{A}(\varphi)$	21	the required access set of a formula φ
$\mathsf{vc}(\overline{s}, Q)$	22	the continuation verification condition of statement list $\overline{s}$ under post-condition Q
$\mathsf{pre}(o.m(\overline{v}))$	25	the precondition of call $o.m(\overline{v})$
$\mathsf{post}(o.m(\overline{v}))$	25	the postcondition of call $o.m(\overline{v})$ (with unbound pre-state)
$\Sigma \vdash \varphi$	25	formula φ is provable from theory Σ
$\mathsf{post}(o.m(\overline{v}), \theta)$	26	the postcondition of call $o.m(\overline{v})$ with respect to pre-state θ
$\mathsf{H}_\theta, \mathsf{L}_\theta, \mathsf{S}_\theta, \mathsf{A}_\theta$	26	the heap, lockset, shared set, resp. access set of thread-relevant state θ
$call$	26	a method call (of the form $o.m(\overline{v})$)
$\mathsf{A}_{\mathsf{req}}(call, \theta)$	26	the required access set of call $call$ in pre-state θ
$\mathsf{A}_{\mathsf{ar}(i)}(t)$	26	the activation record access set of activation record i of thread state t
$\mathsf{L_t} \vdash \mathsf{consistent}(t)$	26	thread state t is consistent in the given context
$Q_{\mathsf{ar}(i)}(t)$	27	activation record postcondition of activation record i of thread state t
$\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash \mathsf{valid}_{\mathsf{ar}(i)}(t)$	27	activation record i of thread state t is valid in the given context
$\mathsf{valid}(\sigma)$	27	program state σ is valid
$o\ \mathrm{rep}_{\mathsf{H}}\ p$	30	p is a *rep* object of o in heap H
$\mathsf{inv}(C)$	30	the object invariant declared by class C
P	30	a packed set
$\mathrm{objrefs}(v)$	32	the object references in a value v
$\mathcal{P}(S)$	32	the power set of set S

Table II. Overview of notations used in the article (*continued*)

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM 17,* 10, 549–557.

Jacobs, B. 2007. A statically verifiable programming model for concurrent object-oriented programs. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven.

Jacobs, B., Leino, K. R. M., Piessens, F., and Schulte, W. 2005a. Safe concurrency for aggregate objects with invariants. In *Proc. Software Engineering and Formal Methods (SEFM)*, B. K. Aichernig and B. Beckert, Eds. IEEE Computer Society, 137–147.

Jacobs, B., Leino, K. R. M., Piessens, F., and Schulte, W. 2005b. Safe concurrency for aggregate objects with invariants: Soundness proof. Tech. Rep. MSR-TR-2005-85, Microsoft Research.

Jacobs, B., Smans, J., Piessens, F., and Schulte, W. 2006. A statically verifiable programming model for concurrent object-oriented programs. In *Proc. International Conference on Formal Engineering Methods (ICFEM)*, Z. Liu and J. He, Eds. Lecture Notes in Computer Science, vol. 4260. Springer, 420–439.

Kassios, I. T. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proc. Formal Methods (FM)*, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Lecture Notes in Computer Science, vol. 4085. Springer, 268–283.

Qadeer, S., Rajamani, S. K., and Rehof, J. 2004. Summarizing procedures in concurrent programs. In *Proc. Principles of Programming Languages (POPL)*, X. Leroy, Ed. ACM, 245–255.

Rodríguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G. T., and Robby. 2005. Extending sequential specification techniques for modular specification and verification of multi-threaded programs. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, A. P. Black, Ed. Lecture Notes in Computer Science, vol. 3586. Springer, 551–576.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. 1997. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems 15,* 4, 391–411.

Welc, A., Jagannathan, S., and Hosking, A. L. 2004. Transactional monitors for concurrent objects. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, M. Odersky, Ed. Lecture Notes in Computer Science, vol. 3086. Springer, 519–542.

Received May 2007; revised November 2007; accepted February 2008