Abstract. To prevent concurrency errors, programmers need to obey a locking discipline. Annotations that specify that discipline, such as Java’s @GuardedBy, are already widely used. Unfortunately, their semantics is expressed informally and is consequently ambiguous. This article highlights such ambiguities and formalizes the semantics of @GuardedBy in two alternative ways, building on an operational semantics for a small concurrent fragment of a Java-like language. It also identifies when such annotations are actual guarantees against data races. Our work aids in understanding the annotations and supports the development of sound formal tools that verify or infer such annotations.

1 Introduction

Concurrency allows computations to occur inside autonomous threads, which are distinct processes that share the same heap memory. Threads can increase program performance by scheduling parallel independent tasks on multicore hardware, and can enable responsive user interfaces [10]. However, concurrency might induce problems such as data races, concurrent access to shared data, with consequent unpredictable or erroneous software behavior. Such errors are difficult to understand, diagnose, and reproduce at runtime. They are also difficult to prevent: testing tends to be incomplete due to nondeterministic scheduling choices made by the runtime, and model-checking scales poorly to real-world code.

The best approach to prevent data races is to follow a locking discipline while accessing shared data: always hold a given lock when accessing a datum. It is all too easy to violate the locking discipline, so tools that verify the adherence to the discipline are desirable. They require a specification language to express the intended locking discipline. The focus of this paper is on this specification language and its formal semantics.

In Java, the most popular specification language for the locking discipline is the @GuardedBy annotation. When the programmer declares a variable \( v \) as @GuardedBy \( E \), then a thread may access \( v \) only while holding the lock \( E \) (the actual syntax provides \( E \) as a string, but we drop the double quotes for notational simplicity). The @GuardedBy annotation was proposed by Goetz [8] as a documentation convention only, without tool support. Tool support now exists in Java PathFinder [15], in the Checker Framework [6], and in IntelliJ [16]. The annotation has been adopted by practitioners, for instance in the Google Guava project [9].
public final class ExecutionList {
  @GuardedBy(this) private RunnableExecutorPair runnables;
  
  public void execute() {
    RunnableExecutorPair list;
    synchronized (this) {
      list = runnables;
      runnables = null;
    }
    RunnableExecutorPair reversedList = null;
    while (list != null) {
      RunnableExecutorPair tmp = list;
      list = list.next;
      tmp.next = reversedList;
      reversedList = tmp;
    }
  }
}

All of these sources use a similar, informal definition of @GuardedBy(E) [13]: accesses to the annotated variable or field must occur when the current thread holds the lock on the guard expression E. But informality leads to ambiguity. (1) These definitions do not clarify how an occurrence of this in E is interpreted in client code. If a field f is @GuardedBy(this), then this must be contextualized into the container of f wherever f is accessed. (2) They do not clarify what an access is. (3) They do not clarify whether the lock statement must use the guard expression E as written (after being contextualized), or whether a different expression that evaluates to the same value is permitted. (4) They do not indicate whether the lock that must be taken is that of E at the time of synchronization or at the time of variable or field access: side-effects on E might make a difference here. (5) They do not clarify whether the name or the value of the variable is guarded. This choice leads to very different semantics with very different guarantees against data races. These ambiguities are serious problems since programmers may expect different semantics and guarantees from what tools check and entail.

The contribution of this article is the formalization of both the by-name and by-value semantics of @GuardedBy, hence clarifying the above ambiguities, so that programmers and tools know what those annotations mean and which guarantees they entail. Such a formalization is a prerequisite for the construction of any tool that infers or checks such annotations. This article shows that the by-value semantics can give a guarantee against data races. Nevertheless, the by-name semantics is implicitly assumed in the tools.

We build on a structural operational semantics in the style of Plotkin [17] for a fragment of Java that focuses on the relevant aspects of Java concurrency. Based on that semantics, we have extended the Julia static analyzer [18] with an analysis that both checks and infers @GuardedBy annotations in arbitrary Java code. Julia allows the user to select either by-name or by-value semantics. Although this implementation is not a contribution of this article, the analyzer reveals that most @GuardedBy annotations in Guava (release 18) are sound by-name, but none is sound by-value, and this is a
Fig. 2 A simple use of @GuardedBy. This code exhibits a potential data race.

```java
public class Observable {
    @GuardedBy(this) private List<Listener> listeners = new ArrayList<>();
    public Observable() {}
    public Observable(Observable original) { // copy constructor
        synchronized (original) {
            listeners.addAll(original.listeners);
        }
        public void register(Listener listener) {
            synchronized (this) {
                listeners.add(listener);
            }
        }
        public List<Listener> getListeners() {
            synchronized (this) {
                return listeners;
            }
        }
    }
}
```

3 programmer’s design choice, so that it seems difficult to rewrite the code with new annotations for the by-value semantics. Fig. 1 shows an example.

The rest of this paper is organized as follows. Section 2 shows examples of the use of @GuardedBy and discusses its informal semantics. Section 3 defines the syntax and semantics of the fragment of Java that we use. Section 4 gives formal definitions for both the by-name and by-value semantics. Section 5 discusses related work and concludes.

2 Informal Semantics of @GuardedBy

This section illustrates the use of @GuardedBy by way of examples. Fig. 2 defines an observable object, which allows many clients to concurrently register listeners. Registration must be synchronized to avoid data races: simultaneous modifications of the ArrayList might result in a corrupted list or lost registrations. Synchronization is needed in the getListeners() method as well, or otherwise the Java memory model would not guarantee the inter-thread visibility of the registrations.

The @GuardedBy(this) annotation on field listeners is very informal because (ambiguity (1) from Sec. 1) it actually expresses the intent that every use of listeners is enclosed within synchronized (container) { ... }, where container refers to the container of listeners at the context of the access (ambiguity (2)). For instance, the access original.listeners in the copy constructor is guarded by synchronized (original) { ... }. This contextualization, similar to viewpoint adaptation [7], is not clarified in any informal definitions of @GuardedBy. It is not clear if a definite alias of original can be used as synchronization guard at line 5 (ambiguity (3)). It is not clear if original can be reassigned between lines 5 and 6 and yet satisfy the @GuardedBy annotation (ambiguity (4)). As a special case, the copy constructor does not synchronize on this even though it accesses this.listeners. This is safe so long as the constructor does not leak this. This paper assumes that an escape analysis [3] has established that constructors do not leak this.
Fig. 3 With the by-value semantics and no side-effects on the guard, this `@GuardedBy` annotation bans data races since it restricts the operations that clients may perform.

```java
public class Observable {
    @GuardedBy(itself) private List<Listener> listeners = new ArrayList<>();
    public Observable() {} public Observable(Observable original) { // copy constructor
        synchronized (original.listeners) {
            listeners.addAll(original.listeners);
        }
    }
    public void register(Listener listener) {
        synchronized (listeners) {
            listeners.add(listener);
        }
    }
    public @GuardedBy(itself) List<Listener> getListeners() {
        synchronized (listeners) {
            return listeners;
        }
    }
}
```

A different interpretation of `@GuardedBy` (ambiguity (5)) refers to the value rather than to the name of the variable or field. Whenever that value is accessed, the given lock must be held. The variable is only allowed to hold such values. Another way of stating this is that `@GuardedBy` is interpreted in this case as a type annotation (a restriction on uses of value) rather than as a declaration annotation (a restriction on uses of a variable). In the by-value semantics, the notion of container has no meaning. For instance, a client of our program might execute:

```
List<Listener> l = new Observable(original).getListeners(); ... uses of l follow
```

where the container of the listeners (the brand new Observable) cannot be accessed when the list is later used through l: there is no syntactical handle and it might even have been garbage-collected. Hence the syntax of `@GuardedBy` for the by-value semantics does not allow this in the guard but it will be shown that it does allow a special variable itself, as in Fig. 3 where the `@GuardedBy` requires all accesses to the value of listeners to occur where the current thread locks that value, also outside class Observable, in a client that operates on the value returned by `getListeners()`.

It is interesting to consider if these two possible interpretations of `@GuardedBy` actually protect against data races on the annotated variable or field. The implementation in Fig. 2 satisfies the simple by-name locking discipline expressed by the `@GuardedBy`, for field `listeners`. Every use of `listeners` occurs in a program point where the current thread locks its container, and we conclude that `@GuardedBy(this)` holds by-name. Nevertheless, a data race is possible, since two threads could call `getListeners()` and later access the returned value concurrently. But data races might occur with the by-value semantics as well, if the guard expression is side-effected during the execution of the program, so that two different threads that contain syntactically-identical `@GuardedBy(E)` statements might acquire different locks, as in Fig. 4. In that figure, the value used for synchronization is distinct at each execution of method `add`, so that no mutual exclusion guarantee exists. Only if the guard expression is not side-effected
can one conclude that the by-value semantics protects against data races, as it will be shown later. An example is Fig. 3.

The two semantics for @GuardedBy are not comparable: neither entails the other. In Fig. 4 field x is @GuardedBy(itself) by-value, since its value is only accessed at line 8 inside a synchronization on itself, but not by-name: x is used at line 5. Field y is @GuardedBy(this.x) by-name but not by-value: its value is accessed at line 11 via w. In some cases the semantics do coincide. Variable z is @GuardedBy(itself) according to both semantics: its name and value are only accessed at line 8, where they are locked, since, as it will be later formalized, line 5 (the definition of z) and line 7 (a synchronization) are not relevant. Variable w is not @GuardedBy according to any semantics: its name and value are accessed at line 11.

### 3 A Core Calculus for Concurrent Java

Some preliminary mathematical notions are needed for the formal semantics. A partial function \( f \) from \( A \) to \( B \) is denoted by \( f : A \to B \) and its domain is \( \text{dom}(f) \). We write \( f(v) \downarrow \) if \( v \in \text{dom}(f) \) and \( f(v) \uparrow \) otherwise. The symbol \( \phi \) denotes the empty function, such that \( \text{dom}(\phi) = \emptyset \); \( \{ v_1 \mapsto t_1, \ldots, v_n \mapsto t_n \} \) denotes the function \( f \) with \( \text{dom}(f) = \{ v_1, \ldots, v_n \} \) and \( f(v_i) = t_i \) for \( i = 1, \ldots, n \); \( f[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n] \) denotes the update of \( f \), where \( \text{dom}(f) \) is enlarged for every \( i \) such that \( v_i \notin \text{dom}(f) \). A tuple is denoted as \( \langle v_0, \ldots, v_n \rangle \). A poset is a structure \( \langle A, \leq \rangle \) where \( A \) is a set with a reflexive, transitive and antisymmetric relation \( \leq \). Given \( a \in A \), we define \( \uparrow a \equiv \{ a' : a \leq a' \} \). A chain is a totally ordered poset.

#### 3.1 Syntax

Symbols \( f, g, x, y, \ldots \) range over a set of variables \( \text{Var} \) that includes \( \text{this} \). Variables identify either local variables in methods or instance variables (fields) of objects. Symbols \( m, p, \ldots \) range over a set \( \text{MethodName} \) of method names. There is a set \( \text{Loc} \) of
public class K {
  private K1 x = new K1();
  private K2 y = new K2();
  public void m() {
    K1 z = x;
    K2 w = new Object();
    synchronized (z) {
      y = z.f;
      w = y;
    }
    w.g = new Object();
  }
}

class K1 {
  K2 f = new K2();
}
class K2 {
  Object g = new Object();
}

memory locations, ranged over by l. Symbols κ₁, κ₀, κ₂, … range over a set of classes (or types) Class, ordered by a subclass relation ≤; ⟨Class, ≤⟩ is a poset such that for all κ ∈ Class the set ↑κ is a finite chain. Intuitively, κ₁ ≤ κ₂ means that κ₁ is a subclass (or subtype) of κ₂.

The partial function lookup(): Class × MethodName → Class formalizes Java’s dynamic method lookup, namely the runtime process of determining the class containing the implementation of a method on the basis of the class of the receiver object:

\[ \text{lookup}(\kappa, m) = \begin{cases} 
  \text{min}(\uparrow\kappa.m) & \text{if } \uparrow\kappa.m \neq \emptyset \\
  \text{undefined} & \text{otherwise}
\end{cases} \]

where \(\uparrow\kappa.m \equiv \{ \kappa' \in \kappa | m \text{ is implemented in } \kappa' \}\) is a finite chain since \(\uparrow\kappa.m \subseteq \uparrow\kappa\).

The set of expressions Exp, ranged over by E, and the set of commands Com, ranged over by C, are defined as follows. The set of method bodies, ranged over by B, contains skip-terminated commands.

\[
\begin{align*}
\text{Exp} & : \quad x \mid E.f \mid \kappa(f_1 = E_1, \ldots, f_n = E_n) \\
\text{Com} & : \quad \text{decl } x = E \mid x := E \mid x.f := E \mid C; C \mid \text{skip} \\
& \quad \mid E.m() \mid \text{spawn } E.m() \mid \text{sync}(E)(C) \mid \text{unlock}(l) \\
\text{Body} & : \quad \text{skip} \mid C; \text{skip}
\end{align*}
\]

Expressions and commands of our language are simplified versions of those of Java. Loops, for instance, must be implemented through recursion. We assume that the compiler insures some standard syntactical properties of the code: for instance, an already declared variable cannot be redeclared in a method; the only free variable in a method’s body is this. These simplifying assumptions can be relaxed without affecting our results.
Expressions are variables, field accesses, or object creations \( \kappa(f_1 = E_1, \ldots, f_n = E_n) \), that create an object of class \( \kappa \) and initialize each field \( f_i \) to the value of \( E_i \). Command \texttt{decl} declares a local variable. The declaration of a local variable in the body \( B \) of a method \( m \) must introduce a fresh variable never declared before in \( B \), whose lifespan starts from there and reaches the end of \( B \). The empty command is \texttt{skip}. Method call \( E.m() \) looks up and runs method \( m \) on the runtime value of \( E \). Command \texttt{spawn} \( E.m() \) does the same asynchronously, on a new thread. Command \texttt{sync}(\( E \{ C \} \)) is like Java’s \texttt{synchronized}: the command(s) \( C \) can be executed only once the current thread holds the lock on the value of \( E \). Command \texttt{unlock}(\( l \)) cannot be used by the programmer: our semantics will introduce it to represent the release of the lock of the object stored at location \( l \).

A class specifies its methods: \texttt{Class} = \{ \( \kappa : \text{MethodNames} \mapsto B \mid \text{dom}(\kappa) \text{ is finite} \} \). The binding of fields to their defining class is not relevant in our formalization. Given a class \( \kappa \) and a method name \( m \), if \( \kappa(m) = B \) then \( \kappa \) implements \( m \) with body \( B \) and \( \kappa.m \) denotes that implementation. For simplicity, this is the only free variable in \( B \) and methods have no return value. A program is a finite set of classes and includes a distinguished class \texttt{Main} that only defines a method \texttt{main} where the program starts: \texttt{Main} = \{ \texttt{main} \mapsto B_{\text{main}} \}.

Example 1. The body of method \texttt{m} in Fig. 5 is translated into \( B_m = \texttt{decl } z = \texttt{this.x}; \texttt{decl } w = \texttt{Object} ; \texttt{sync} (z) \{ \texttt{this.y} = z.f; w = \texttt{this.y} \}; w.g = \texttt{Object} ; \texttt{skip} \) and classes \texttt{K} = \{ \( m \mapsto B_m \} \), \texttt{K1} = \( \phi \), \texttt{K2} = \( \phi \), and \( \texttt{Object} \equiv \phi \).

### 3.2 Semantic Domains

A running program has a pool of threads that share a memory. Initially, a single thread runs the main method. The \texttt{spawn} \( E.m() \) commands create new threads that run in parallel with the already existing ones. Each thread has an activation stack \( S \) and a set \( \mathcal{L} \) of locations that it currently locks. The activation stack \( S \) is a stack of activation records \( R \) of methods. Each \( R \) consists of the identifier \( \kappa.m \) of the method, the command \( C \) to be executed when \( R \) will be on top of the stack (continuation), and the environment or binding \( \sigma \) that provides values to the variables in scope in \( R \). For simplicity, we only have classes and no primitive types, so the only possible values are locations. Formally, \( \text{Env} \equiv \{ \sigma : \text{Var} \mapsto \text{Loc} \mid \text{dom}(\sigma) \text{ is finite} \} \).

**Definition 1.** The set of activation records ranged over by \( R \), the set of activation stacks ranged over by \( S \) and the set of thread pools ranged over by \( T \) are

\[
R ::= \kappa.m[C]_{\sigma} \quad \text{(activation record for } \kappa.m) \\
S ::= \varepsilon \mid R :: S \quad \text{(activation stack, possibly empty)} \\
T ::= [S]\mathcal{L} \mid T || T \quad \text{(thread pool)}
\]

\( R :: \varepsilon \) is often abbreviated into \( R \). The number of threads in \( T \) is written as \( \# T \).

A configuration is a pair \( \langle T, \mu \rangle \) where \( T \) is a pool of threads and \( \mu \) is a memory that models the heap of the system and hence maps a finite set of already allocated memory
locations to objects (values). An object has a class, values bound to its fields, and a lock, i.e., an integer counter incremented every time a thread locks the object.

**Definition 2.** Objects and memories are defined as:

\[ \text{Object} \triangleq \text{Class} \times \text{Env} \times \mathbb{N} \quad \text{Memory} \triangleq \{ \mu : \text{Loc} \rightarrow \text{Object} \mid \text{dom(\(\mu\)) is finite} \} \]

with selectors class(\(o\)) \(\triangleq\) \(\kappa\), env(\(o\)) \(\triangleq\) \(\sigma\) and lock\(^{\#}(o) \triangleq n\) for every \(o = \langle \kappa, \sigma, n \rangle \in \text{Object}\). We also define \(o[\(f \mapsto l\)] \triangleq \langle \kappa, \sigma[\(f \mapsto l\)], n \rangle\) and functions to increase/decrease the lock counter: \(\text{lock}^+(o) \triangleq \langle \kappa, \sigma, n + 1 \rangle\) and \(\text{lock}^-(o) \triangleq \langle \kappa, \sigma, \text{max}(0,n-1) \rangle\).

For simplicity, we do not model delayed publication of field updates, which the Java memory model allows but that is not relevant for our semantics and results. Hence, the memory is a deterministic map shared by all threads. Moreover, each execution step of our semantics is atomic.

The evaluation of an expression \(E\) in an environment \(\sigma\) and in a memory \(\mu\) yields a pair \(\langle l, \mu'\rangle\), where \(l\) is a location (the run-time value of \(E\)) and \(\mu'\) is the memory resulting after the evaluation of \(E\). This allows \(E\) to have side-effects, such as a new object allocation. Given a pair \(\langle l, \mu\rangle\) we use selectors \(\text{loc}(\langle l, \mu\rangle) = l\) and \(\text{mem}(\langle l, \mu\rangle) = \mu\).

**Definition 3 (Evaluation of Expressions).** The evaluation function for expressions is \([\ ] : (\text{Exp} \times \text{Env} \times \text{Memory}) \rightarrow (\text{Loc} \times \text{Memory})\) defined as:

\[
\begin{align*}
[x]_\sigma^\mu & \triangleq \langle \sigma(x), \mu \rangle \\
E.f]_\sigma^\mu & \triangleq \langle \text{env}(\mu'(l))(f), \mu' \rangle, \text{ where } [E]_\sigma^\mu = \langle l, \mu' \rangle \\
[k\langle f_1 = E_1, \ldots, f_n = E_n \rangle]_\sigma^\mu & \triangleq \langle l, \mu_n[l \mapsto \langle \kappa, \sigma', 0 \rangle] \rangle, \text{ where }
\end{align*}
\]

(1) \(\mu_0 = \mu\) and \(\langle l_i, \mu_i \rangle = [E_i]_\sigma^{|i-1}\) for \(i \in [1..n]\)

(2) \(l\) is fresh in \(\mu_n\), that is \(\mu_n(l) \uparrow\)

(3) \(\sigma' \in \text{Env}\) is such that \(\sigma'(f_i) = l_i\) for \(i \in [1..n]\), while \(\sigma'(y)\uparrow\) elsewhere

assuming that \([\ ]\) is undefined if any of the function applications is undefined.

In the evaluation of the object creation expression, a fresh location \(l\) is allocated and bound to an unlocked object whose environment \(\sigma'\) binds its fields to the values of the corresponding initialization expressions.

### 3.3 Structural Operational Semantics

Our operational semantics is given in terms of a reduction relation \(\langle T, \mu \rangle \xrightarrow{n} \langle T', \mu' \rangle\) on configurations, where \(n \geq 1\) is the number of the thread in \(T\) that performs the transition, starting from the leftmost in the pool \(T\) (thread number 1). We will often write \(\rightarrow\) instead of \(\xrightarrow{n}\) when we wish to abstract on the running thread; \(\rightarrow^*\) denotes the reflexive and transitive closure of \(\rightarrow\). We give reduction rules by cases, on the basis of the structure of the command at the top of the activation stack of the \(n\)th thread. Transition rules where the activation stack consists of a single activation record are first introduced and then lifted to the general case.
**Declaration**

\[
\begin{align*}
\text{executed in a} & \\
\text{which must be an unlocked object with lock counter} & \\
\text{will release the same} & \\
\end{align*}
\]

\[
\begin{align*}
\text{The execution of a fork is similar to that of a method call, but the body of the method runs in its own new thread with an initially empty set of locked locations:} & \\
\text{from an initial state where only variable this is in scope, bound to the receiver:} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Synchronization.} & \\
\text{The first rule models a thread that locks the value } \mu'(l) \text{ of a guard } E, & \\
\text{which must be an unlocked object with lock counter 0. The command } C \text{ that must be executed in a critical section is put on top of the activation stack. At its end, an unlock} & \\
\text{will release the same } \mu'(l) \text{, since its location is recorded. This is important when } E & \\
\text{might be side-effected during the critical section and matches the behavior of Java’s synchronized statement:} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Assignments.} & \\
\text{Method Invocation.} & \\
\text{Spawn.} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Method Invocation.} & \\
\text{The receiver } E \text{ is evaluated and the method implementation is looked up from the dynamic class of the receiver. The body of the method is executed from an initial state where only variable this is in scope, bound to the receiver:} & \\
\text{Synchronization.} & \\
\text{The first rule models a thread that locks the value } \mu'(l) \text{ of a guard } E, & \\
\text{which must be an unlocked object with lock counter 0. The command } C \text{ that must be executed in a critical section is put on top of the activation stack. At its end, an unlock} & \\
\text{will release the same } \mu'(l) \text{, since its location is recorded. This is important when } E & \\
\text{might be side-effected during the critical section and matches the behavior of Java’s synchronized statement:} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Spawn.} & \\
\text{The execution of a fork is similar to that of a method call, but the body of the method runs in its own new thread with an initially empty set of locked locations:} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Synchronization.} & \\
\text{The first rule models a thread that locks the value } \mu'(l) \text{ of a guard } E, & \\
\text{which must be an unlocked object with lock counter 0. The command } C \text{ that must be executed in a critical section is put on top of the activation stack. At its end, an unlock} & \\
\text{will release the same } \mu'(l) \text{, since its location is recorded. This is important when } E & \\
\text{might be side-effected during the critical section and matches the behavior of Java’s synchronized statement:} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Method Invocation.} & \\
\text{The receiver } E \text{ is evaluated and the method implementation is looked up from the dynamic class of the receiver. The body of the method is executed from an initial state where only variable this is in scope, bound to the receiver:} & \\
\text{Synchronization.} & \\
\text{The first rule models a thread that locks the value } \mu'(l) \text{ of a guard } E, & \\
\text{which must be an unlocked object with lock counter 0. The command } C \text{ that must be executed in a critical section is put on top of the activation stack. At its end, an unlock} & \\
\text{will release the same } \mu'(l) \text{, since its location is recorded. This is important when } E & \\
\text{might be side-effected during the critical section and matches the behavior of Java’s synchronized statement:} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Method Invocation.} & \\
\text{The receiver } E \text{ is evaluated and the method implementation is looked up from the dynamic class of the receiver. The body of the method is executed from an initial state where only variable this is in scope, bound to the receiver:} & \\
\text{Synchronization.} & \\
\text{The first rule models a thread that locks the value } \mu'(l) \text{ of a guard } E, & \\
\text{which must be an unlocked object with lock counter 0. The command } C \text{ that must be executed in a critical section is put on top of the activation stack. At its end, an unlock} & \\
\text{will release the same } \mu'(l) \text{, since its location is recorded. This is important when } E & \\
\text{might be side-effected during the critical section and matches the behavior of Java’s synchronized statement:} & \\
\end{align*}
\]
The next rule models the lock of an already locked object. Lock reentrancy, as in Java, is only possible for the thread that already owns the lock:

\[
\begin{align*}
[E]_\sigma^k &= \langle l, \mu' \rangle \quad l \in \mathcal{L} \quad \mu'' \equiv \mu'[l \mapsto \text{lock}^+(\mu(l))] \\
\langle [\kappa.m[\text{sync}(E)(C_1)]\sigma] \mathcal{L}, \mu \rangle & \xrightarrow{1} \langle [\kappa.m[C; \text{unlock}(l)]\sigma] \mathcal{L}, \mu'' \rangle
\end{align*}
\]

[lock-many]

The next rule models the unlock of an object that still remains locked, since it was locked more than once:

\[
\begin{align*}
\text{lock}^\#(\mu(l)) > 1 \quad &\quad \mu'' \equiv \mu[l \mapsto \text{lock}^-(\mu(l))] \\
\langle [\kappa.m[\text{unlock}(l)]\sigma] \mathcal{L}, \mu \rangle & \xrightarrow{1} \langle [\kappa.m[\text{skip}]\sigma] \mathcal{L}, \mu' \rangle
\end{align*}
\]

[unlock-many]

The last rule models the unlock of an object that becomes unlocked (its lock counter reaches 0): the lock on that object is removed from those held by the current thread:

\[
\begin{align*}
\text{lock}^\#(\mu(l)) = 1 \quad &\quad \mathcal{L}' \equiv \mathcal{L} \setminus \{l\} \quad \mu'' \equiv \mu[l \mapsto \text{lock}^-(\mu(l))] \\
\langle [\kappa.m[\text{unlock}(l)]\sigma] \mathcal{L}, \mu \rangle & \xrightarrow{1} \langle [\kappa.m[\text{skip}]\sigma] \mathcal{L}', \mu' \rangle
\end{align*}
\]

[unlock-last]

**Garbage and Structural Rules.** These rules lift the execution of an activation record to that of a stack of activation records and lift the execution of a thread to the non-deterministic execution of a pool of threads:

\[
\begin{align*}
\langle [R']\mathcal{L}, \mu \rangle & \xrightarrow{1} \langle [R']'\mathcal{L}', \mu' \rangle \quad \text{[push]} \\
\langle [R::S]\mathcal{L}, \mu \rangle & \xrightarrow{1} \langle [R::S]'\mathcal{L}', \mu' \rangle \quad \text{[pop]} \\
\langle T_1, \mu \rangle & \xrightarrow{\alpha} \langle T_1', \mu' \rangle \quad \text{[par-l]} \\
\langle T_1 || T_2, \mu \rangle & \xrightarrow{\alpha} \langle T_1', || T_2', \mu' \rangle \quad \text{[par-r]} \\
\langle \varepsilon \mathcal{L}, \mu \rangle & \xrightarrow{1} \langle T, \mu \rangle \quad \text{[end-l]} \\
\langle T || \varepsilon \mathcal{L}, \mu \rangle & \xrightarrow{\#T_1 + \alpha} \langle T || T_2', \mu' \rangle \quad \text{[end-r]}
\end{align*}
\]

**Definition 4 (Operational Semantics of a Program).** The initial configuration of a program is \(\langle T_0, \mu_0 \rangle\) where \(T_0 \equiv [\text{Main}.\text{main}[B_{\text{main}}]\{z \mapsto \text{this} \rightarrow \text{main} \rightarrow \text{init} \mapsto \langle \text{Main}, \phi, 0 \rangle\} \) and \(\mu_0 \equiv \{l_{\text{init}} \mapsto \langle \text{Main}, \phi, 0 \rangle\} \). The operational semantics of a program is the set of traces of the form \(\langle T_0, \mu_0 \rangle \xrightarrow{*} \langle T, \mu \rangle\).

This semantics enables us to prove that two threads never lock the same object; a lock can only be reentered by the same thread that already holds it; a thread only releases locks that already holds; and other properties.

**Example 2.** The implementation in Ex. 1 at the end of Sec. 3.1 becomes a program by defining \(B_{\text{main}}\) as: \(\text{x} = \text{x1} \{\text{f} = \text{x2} \{\text{g} = \text{Object}()\}\}, \text{y} = \text{x2} \{\text{g} = \text{Object}()\}; \text{skip} \).
The operational semantics builds the following maximal trace from $T_0$, $\mu_0$:

1. $\frac{1}{\top} \left( [\text{decl } z = \text{this.x}; \ldots]_{\sigma_1} :: \text{Main.main}[\text{skip}]_{(\text{this-stm}_{\text{init}})} ] \emptyset, \mu_1 \right)$ with $\mu_1 \models \mu_0 l \mapsto o, l_1 \mapsto o_1, l_2 \mapsto o_2, l_3 \mapsto o_3, l_4 \mapsto o_4, l_5 \mapsto o_5$

2. $\frac{1}{\top} \left( [\text{decl } w = \text{Object}(); \ldots]_{\sigma_2} :: \ldots | \emptyset, \mu_1 \right)$ with $\sigma_2 \models \sigma_1 [z \mapsto l_1]$

3. $\frac{1}{\top} \left( [\text{sync}(x); \ldots]_{\sigma_3} :: \ldots | \emptyset, \mu_2 \right)$ with $\mu_2 \models \mu_1 [l_6 \mapsto o_4]; \sigma_3 \models \sigma_2 [w \mapsto l_6]$

4. $\frac{1}{\top} \left( [\text{this.y} := \text{z.f}; \ldots; \text{unlock}(l_1); \ldots]_{\sigma_3} :: \ldots | \{l_1\}, \mu_3 \right)$

5. $\frac{1}{\top} \left( [\text{this.y} := \text{z.f}; \ldots; \text{unlock}(l_1); \ldots]_{\sigma_3} :: \ldots | \{l_1\}, \mu_4 \right)$ with $\mu_4 \models \mu_3 [l \mapsto o']; \sigma' \models \sigma [x \mapsto l_1, y \mapsto l_3], 0$

6. $\frac{1}{\top} \left( [\text{unlock}(l_1); w.g = \text{Object}(); \text{skip}]_{\sigma_4} :: \ldots | \{l_1\}, \mu_4 \right)$ with $\sigma_4 \models \sigma_3 [w \mapsto l_3]$

7. $\frac{1}{\top} \left( [\text{w.g} = \text{Object}(); \text{skip}]_{\sigma_4} :: \ldots | \emptyset, \mu_5 \right)$ with $\mu_5 \models \mu_4 [l_1 \mapsto o_1]$

8. $\frac{1}{\top} \left( [\text{Main.main}[\text{skip}]_{(\text{this-stm}_{\text{init}})} ] \emptyset, \mu_6 \right)$ with $\mu_6 \models \mu_5 [l_5 \mapsto o_4]$

4 Two Semantics for \texttt{@GuardedBy} Annotations

Sec.\textsuperscript{2} highlights that there are two possible semantics for the declaration \texttt{@GuardedBy(E)}

Type \texttt{x.j}. In the \texttt{by-name} interpretation, a thread must hold the lock on \texttt{E} whenever it accesses (reads or writes) the variable named \texttt{x}. In the \texttt{by-value} interpretation, a thread must hold the lock on \texttt{E} whenever it dereferences a location (objects at locations are our values) that was ever bound to \texttt{x}. We formalize both interpretations now.

4.1 By-name Semantics

Def.\textsuperscript{5} formalizes the notion of accessing an expression in a single reduction step of our semantics, when a given command is executed. Since a single step is considered, the accesses in $C_1; C_2$ are only those in $C_1$. Since only the current thread can access a newly created object, the creation expression does not access the fields of the latter. The access refers to the value of the expression, not to its lock counter, hence sync(E) | (C) does not access E. For accesses to fields, Def.\textsuperscript{5} keeps the exact expression used for the container of the field, that will be used in Def.\textsuperscript{6} for the contextualization of this.

**Definition 5 (Expressions Accessed in a Single Reduction Step).** The set of expressions accessed for the evaluation of an expression E is defined as

$$acc(x) = \{x\} \quad acc(E.f) = acc(E) \cup \{E.f\}$$

$$acc(n\{f_1 = E_1, \ldots, f_n = E_n\}) = \cup \{acc(E_i) | 1 \leq i \leq n\}$$

The expressions accessed in a single execution step of a command are:

- $acc(\text{decl } x = E) = acc(E)$
- $acc(x.f := E) = acc(x.f) \cup acc(E)$
- $acc(E.m()) = acc(E)$
- $acc(\text{sync}(x) | C) = \emptyset$
- $acc(\text{sync}(E.f) | C) = acc(E)$
- $acc(\text{unlock } \ell) = \emptyset$
acc(sync (κ(f₁ = E₁, . . . , fₙ = Eₙ)) (C)) = acc(κ(f₁ = E₁, . . . , fₙ = Eₙ))

We say that a single step of a command C’s execution accesses a variable x if and only if x ∈ acc(C); we say that it accesses a field f if and only if E.f ∈ acc(C), for some expression E.

We now define GuardedBy for local variables (Def. 6) and for fields (Def. 7).

Definition 6 (GuardedBy for Local Variables). Local variable x of a method κ.m in a program is @GuardedBy(E) by-name if and only if for every derivation ⟨T₀, µ₀⟩ →* ⟨T, µ⟩ → · · · , where the n-th thread in the thread pool T is [κ.m|C]₀ :: S|L and C accesses x, we have [E]₁₇ = (l, µ′) with l ∈ L.

Def. 6 formalizes the fact that the synchronization guard E is evaluated and its lock must be held whenever x is accessed. Hence E can only refer to variables that are in scope at those program points.

Example 3. In Ex. 2 at the end of Sec. 3.3, variable z of method K.mis @GuardedBy(this.x) by-name since the name z is accessed at reduction 5 only, where [this.x]₁₃ = (l₁, µ₃) and, during that reduction, the current thread holds the lock on the object bound to l₁.

According to Def. 5, reduction 2 is not an access since it is a declaration; reduction 4 is not an access since it is a synchronization.

Definition 7 (GuardedBy for Fields). A field f in a program is @GuardedBy(E) by-name if and only if for every derivation ⟨T₀, µ₀⟩ →* ⟨T, µ⟩ → · · · , where the n-th thread in the thread pool T is [κ.m|C]₀ :: S|L, and for every E′ such that E′.f ∈ acc(C), we have [E′]₁₇ = (l′, µ′) and [E]₁₇ = (l″, µ″) with l″ ∈ L.

Example 4. In Ex. 2 field y is @GuardedBy(this.x) by-name since y is accessed at reductions 5 and 6 only, where [this.x]₁₃ = (l₁, µ₃) and [this.x]₁₄ = (l₁, µ₄), and the current thread holds the lock on the object bound to l₁.

4.2 By-value Semantics

An alternative semantics for @GuardedBy refers to the values held in variables or fields, rather than to their names. In this new semantics, a variable x is @GuardedBy(E) if whenever a thread, at some program points P, dereferences a location l eventually bound to x, it holds the lock on the object derived by the evaluation of E at p ∈ P. In object-oriented parlance, dereferencing a location l means accessing the object stored at l in order to read or write its fields or its class tag. Accesses to the class tag happen, for instance, at method lookup. In Java, accesses to the lock counter are synchronized at a low level, so they are not relevant here. Dereferences (Def. 8) are very different from variable accesses (Def. 5). For instance, statement v.f := w.g.h accesses expressions v, v.f, w, w.g and w.g.h but dereferences only the locations held in v, w and w.g. Since the program points P are arbitrary, possibly occurring in distinct methods of distinct classes, the guard expression E must be such that it can be evaluated at all of them. As a consequence, variables are not allowed in E.
Definition 8 (Dereferenced Locations). Given a memory $\mu$ and an environment $\sigma$, the locations dereferenced in a single evaluation step of an expression are defined as
\[
\text{deref}(x)_{\mu}^\sigma = \emptyset \quad \text{deref}(E.f)_{\mu}^\sigma = \{ \text{loc}(\llbracket E \rrbracket_{\mu}^\sigma) \} \cup \text{deref}(E)_{\mu}^\sigma \\
\text{deref}(\kappa(f_1 = E_1, \ldots, f_n = E_n))_{\mu}^\sigma = \cup\{\text{deref}(E_i)_{\mu}^\sigma \mid 1 \leq i \leq n\}
\]
The locations dereferenced during a single execution step of a command are defined as
\[
\text{deref}(\text{decl}\ x = E)_{\mu}^\sigma = \text{deref}(E)_{\mu}^\sigma \\
\text{deref}(x.f := E)_{\mu}^\sigma = \{\sigma(x)\} \cup \text{deref}(E)_{\mu}^\sigma \\
\text{deref}(\text{skip})_{\mu}^\sigma = \emptyset \\
\text{deref}(\text{sync}(E) \ (C))_{\mu}^\sigma = \text{deref}(E)_{\mu}^\sigma \\
\text{deref}(E.m())_{\mu}^\sigma = \text{deref}(\text{spawn } E.m())_{\mu}^\sigma = \{\text{loc}(\llbracket E \rrbracket_{\mu}^\sigma)\} \cup \text{deref}(E)_{\mu}^\sigma
\]

Def. 9 and 10 formalize the by-value semantics of $\text{GuardedBy}(E)$. No variable is allowed in $E$, but we do allow a special token $\text{itself}$ in $E$, that programmers cannot use and that refers to the same value that gets dereferenced. The by-name semantics does not need $\text{itself}$, since it would only be syntactic sugar there, for the same name that is annotated. That is, in the by-name semantics, stating that $x$ is $\text{GuardedBy}(\text{itself})$ just means that $x$ is $\text{GuardedBy}(x)$.

Definition 9 (GuardedBy for Local Variables). Let $x$ be a local variable of a method $\kappa.m$ and $E$ an expression that can only contain the special variable $\text{itself}$. Variable $x$ is $\text{GuardedBy}(E)$ by-value if and only if for any derivation $\langle T_0, \mu_0 \rangle \xrightarrow{n_1} \cdots \xrightarrow{n_i} \langle T_i, \mu_i \rangle \cdots$, letting $T^n_i = [k^n_i.m^n_i[C^n_i]_{\mu^n_i}]_{\sigma^n_i} \vdash S^n_i \mathcal{L}^n_i$ be the $n$-th thread of $T_i$, for $i \geq 1$, $\mathcal{L} = \cup\{\sigma^j_1(x) \mid k^j_i.m^j_i = \kappa.m \text{ and } \sigma^j_1(x) \wr j \geq 1\}$ and $\mathcal{X} = \text{deref}(C_{i-1}^{\mu_{i-1}})_{\sigma_{i-1}^j} \cap \mathcal{L}$, it follows that for every $l' \in \mathcal{X}$ we have $[E]_{\sigma_{i-1}^j,[\text{itself} \rightarrow l']}^{\mu_{i-1}} = \{l, \mu\}$ with $l \in \mathcal{L}_{i-1}^{n_i}$.

Def. 9 collects the set $\mathcal{L}$ of locations that have ever been bound to $x$ in some execution trace and requires that, whenever a thread dereferences one of them, that thread must hold the lock on the object obtained by evaluating $E$.

Example 5. In Ex. 2 variable $z$ is $\text{GuardedBy}(\text{itself})$ by-value. Namely, the set $\mathcal{X}$ of Def. 9 computed for $z$ is $\{l_1\}$ and $l_1$ is only dereferenced at reduction 5, when it is locked by the current thread.

Definition 10 (GuardedBy for Fields). Let $f$ be a field and $E$ an expression that can only contain the special variable $\text{itself}$. Field $f$ is $\text{GuardedBy}(E)$ by-value if and only if for any derivation $\langle T_0, \mu_0 \rangle \xrightarrow{n_1} \cdots \xrightarrow{n_i} \langle T_i, \mu_i \rangle \cdots$, letting $T^n_i = [k^n_i.m^n_i[C^n_i]_{\mu^n_i}]_{\sigma^n_i} \vdash S^n_i \mathcal{L}^n_i$ be the $n$-th thread of $T_i$, for $i \geq 1$, and $\mathcal{L} = \cup\{\text{env}(\mu_j(l))(f) \mid l \in \text{dom}(\mu_j) \text{ and } \text{env}(\mu_j(l))(f) \wr j \geq 1\}$ and $\mathcal{X} = \text{deref}(C_{i-1}^{\mu_{i-1}})_{\sigma_{i-1}^j} \cap \mathcal{L}$ it follows that for every $l' \in \mathcal{X}$ we have $[E]_{\sigma_{i-1}^j,[\text{itself} \rightarrow l']}^{\mu_{i-1}} = \{l, \mu\}$ with $l \in \mathcal{L}_{i-1}^{n_i}$.

Example 6. In Ex. 2 field $x$ is $\text{GuardedBy}(\text{itself})$ by-value. Namely, the set $\mathcal{X}$ of Def. 10 computed for $x$ is $\{l_1\}$ and $l_1$ is only dereferenced at reduction 5, when it is locked by the current thread.
4.3 Protection against Data Races

A variable or field that is \texttt{@GuardedBy} by-name is not necessarily protected against data races. For instance, field \texttt{listeners} in Fig. 2 is \texttt{@GuardedBy(this)} by-name and its values can still be involved in a data race since only the field name is guarded, but its values are not guarded and escape the class: once they are outside the class, the programmer can use those values however she wants, without any form of synchronization.

A variable or field that is \texttt{@GuardedBy} by-value is not necessarily protected against data races either. For instance, field \texttt{set} in Fig. 4 is \texttt{@GuardedBy(this.set.lock)} by-value and its values can still be involved in a data race since distinct locks are taken by distinct threads that synchronize at line 6. Note that the guard \texttt{this.set.lock} is allowed in the by-value semantics of Def. 10 since \texttt{@GuardedBy(this.set.lock)} can be expressed as \texttt{@GuardedBy(itself.lock)} in Fig. 4. If the guard expression is never side-effected in the program (an object creation expression is assumed to be side-effected by itself), data races can still occur in the by-name semantics, as Fig. 2 shows, again. But it turns out that, in the by-value semantics, that condition protects against data races (a simple application of this result is in Fig. 3).

\textbf{Proposition 1.} If a variable or field \( x \) is \texttt{@GuardedBy(\( E \))} by-value and \( E \) is never side-effected by the program, then no data race can occur on the values bound to \( x \).

\textit{Proof.} If a data race occurred on a value \( v \) ever bound to \( x \) during the execution of the program, then there was an instant in time \( t \) when two threads had access to the shared value \( v \). Since \( x \) is \texttt{@GuardedBy(\( E \))} by-value, both threads synchronized on the value of \( E \) at some times \( t_1 < t \) and \( t_2 < t \). We can assume, without loss of generality, that they never released that lock between \( t_1 \) and \( t \) and between \( t_2 \) and \( t \), respectively. Since \( E \) is not side-effected by the program and since it can only contain the special variable \( \texttt{itself} \) that is bound to \( v \) in both threads (Def. 9 and 10), it follows that both threads held the lock on the same value (that of \( E \)) at time \( t \), which is impossible. \( \square \)

5 Conclusions and Related Work

We have formalized two possible semantics for Java’s \texttt{@GuardedBy} annotations. Coming back to the ambiguities sketched in Sec. 1 we have clarified that: (1) \texttt{this} in the guard expression must be interpreted as the container of the guarded field and consistently contextualized (Def. 7). (2) An access is a variable use for the by-name semantics (Def. 5, 6 and 7). A value access is a dereferences (field get/set or method call) for the by-value semantics; copying a value is not an access in this case (Def. 8, 9 and 10). (3) It is the value of the guard expression that must actually be locked when a name or value is accessed, regardless of how it is accessed for locking (Def. 6, 7, 9 and 10). (4) The lock is taken on the value of the guard expression as evaluated at the access to the guarded variable or field (Def. 6, 7, 9 and 10). (5) Either the name or the value of a variable can be guarded, but this choice leads to very different semantics. Namely, in the by-name semantics, the lock must be held whenever the named variable is accessed (Def. 5, 6 and 7). In the by-value semantics, the lock must be held whenever the variable’s value is accessed (Def. 8, 9 and 10), regardless of what expression is used to
access the value. The by-value semantics yields a guarantee against data races, under suitable conditions (Prop. 1).

There are many other formalizations of the syntax and semantics of sequential Java, such as [14]. Our goal here was to keep it to a minimum core needed for the formalization of the semantics of @GuardedBy. Another well-known formalization is featherweight Java [12], that however has no assignment, which results in a functional language, while assignments to shared data are necessary for our purposes. The need of a formal specification for reasoning about Java’s concurrency and for building verification tools is recognized [5,14] but we are not aware of any formalization of the semantics of Java’s concurrency annotations. Our formalization will support tools based on model-checking such as Java PathFinder [15] and Bandera [112], on type-checking such as the Checker Framework [6], or on abstract interpretation such as Julia [18].

References

1. E. Ábrahám-Mumm, F. S. de Boer, W. P. de Roever, and M. Steffen. Verification for Java’s Reentrant Multithreading Concept. In FOSSACS, pages 5–20, Grenoble, France, 2002.
2. Bandera. About Bandera. http://bandera.projects.cis.ksu.edu
3. B. Blanchet. Escape Analysis for Java™: Theory and Practice. TOPLAS, 25(6):713–775, 2003.
4. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In AMAST, pages 75–90, Sydney, Australia, December 1997.
5. J. C. Corbett, M. B. Dwyer, J. Hatchfield, and Robby. Expressing Checkable Properties of Dynamic Systems: the Bandera Specification Language. STTT, 4(1):34–56, 2002.
6. W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and Using Pluggable Type-Checkers. In ICSE, pages 681–690, Waikiki, Honolulu, HI, USA, May 2011.
7. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In ECOOP, pages 28–53, Berlin, Germany, August 2007.
8. B. Goetz, T. Peierls, J. Bloch, and J. Bowbeer. Java Concurrency in Practice. Addison Wesley, May 2006.
9. Google. Guava: Google Core Libraries for Java 1.6+. https://code.google.com/p/guava-libraries
10. A. Göransson. Efficient Android Threading. O’Reilly Media, June 2014.
11. J. Hatchfield and M. B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In CONCUR, pages 39–58, Aalborg, Denmark, August 2001.
12. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. TOPLAS, 23(3):396–450, 2001.
13. Javadoc for @GuardedBy. https://jsr-305.googlecode.com/svn/trunk/javadoc/javassist/annotation/concurrency
14. B. Long and B. W. Long. Formal Specification of Java Concurrency to Assist Software Verification. In IPDPS, page 136, Nice, France, April 2003. IEEE Computer Society.
15. NASA. Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf
16. V. Pech. Concurrency is Hot, Try the JCIP Annotations. http://jetbrains.dzone.com/tips/concurrency-hot-try-jcip
February 2010.
17. G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.
18. Julia Srl. The Julia Static Analyzer. http://www.juliasoft.com/julia
A Properties of the Operational Semantics

Let us provide a few properties showing the soundness of both the locking and unlocking mechanisms of our operational semantics.

Two different threads never lock the same location:

**Proposition 2 (Locking vs. multithreading).** Given an arbitrary run

\[
\langle T_0, \mu_0 \rangle \rightarrow^* \langle \big[ S_1 \big] \mathcal{L}_1 \parallel \ldots \parallel \big[ S_n \big] \mathcal{L}_n , \mu \rangle
\]

then for every \( i, j \in \{1 \ldots n\} \) we have \( i \neq j \) implies \( \mathcal{L}_i \cap \mathcal{L}_j = \emptyset \).

When a thread starts it does not hold any lock:

**Proposition 3 (Thread initialization vs. locking ).** Let

\[
\langle T_0, \mu_0 \rangle \rightarrow^* \langle \big[ S_1 \big] \mathcal{L}_1 \parallel \ldots \parallel \big[ S_n \big] \mathcal{L}_n , \mu \rangle \xrightarrow{i} \langle \big[ \hat{S}_1 \big] \hat{\mathcal{L}}_1 \parallel \ldots \parallel \big[ \hat{S}_m \big] \hat{\mathcal{L}}_m , \hat{\mu} \rangle
\]

be an arbitrary run where \( S_i = \kappa.m[\text{spawn}\,E.p();\,C]\sigma :: S, \) for some \( \kappa, m, E, p, C, \sigma \) and \( S \), then

- \( \hat{\sigma}_i = \kappa', p[B]\sigma', \) for appropriate \( \kappa', B \) and \( \sigma' \)
- \( \hat{\sigma}_{i+1} = \kappa.m[C]\sigma :: S \)
- \( \hat{\mathcal{L}}_i = 0 \)
- \( \hat{\mathcal{L}}_{i+1} = \mathcal{L}_i \).

When a thread terminates it does not keep locks on locations:

**Proposition 4 (Thread termination vs. locking ).** Let

\[
\langle T_0, \mu_0 \rangle \rightarrow^* \langle \big[ S_1 \big] \mathcal{L}_1 \parallel \ldots \parallel \big[ S_n \big] \mathcal{L}_n , \mu \rangle
\]

be an arbitrary run where \( S_i = \epsilon, \) , then \( \mathcal{L}_i = 0 \).

A thread may not lock a location by mistake:

**Proposition 5 (Locking).** Let

\[
\langle T_0, \mu_0 \rangle \rightarrow^* \langle \big[ S_1 \big] \mathcal{L}_1 \parallel \ldots \parallel \big[ S_n \big] \mathcal{L}_n , \mu \rangle \xrightarrow{i} \langle \big[ \hat{S}_1 \big] \hat{\mathcal{L}}_1 \parallel \ldots \parallel \big[ \hat{S}_m \big] \hat{\mathcal{L}}_m , \hat{\mu} \rangle
\]

be an arbitrary run. Then \( \bigcup_{j=1}^n \mathcal{L}_j \subseteq \bigcup_{j=1}^m \hat{\mathcal{L}}_j \), if and only if

- \( S_i = \kappa.m[\text{sync}\,(E)\,(C);\,C']\sigma :: S, \) for some \( \kappa, m, C, E, C', \sigma \) and \( S \)
- \( [E]_\sigma = (l, \hat{\mu}), \) for some \( l \) and \( \hat{\mu} \)
- lock\#(\mu(l)) = 0 and lock\#(\hat{\mu}(l)) = 1
- \( \hat{\mathcal{L}}_i = \mathcal{L}_i \cup \{l\} \)
- \( m = n \) and \( \hat{\mathcal{L}}_j = \mathcal{L}_j \) for every \( j \in \{1 \ldots n\} \setminus \{i\} \).

Reentrant locks are allowed: only threads that already own the lock on an object can synchronize again on that object.
Proposition 6 (Reentrant locking). Given an arbitrary run
\[
\langle T_0, \mu_0 \rangle \rightarrow^* \langle [S_1]L_1 \parallel \ldots \parallel [S_n]L_n, \mu \rangle
\]
where \( l \in \bigcup^n_{j=1} L_j \) for some \( l \), and \( S_i = \kappa.m[[\text{sync}(E)(C); C']_\sigma : S \), for some \( i \in \{1..n\}, \kappa.m, C, E, C', \sigma \) and \( S \), with \( \|E\|_\sigma = (l, \hat{\mu}) \). Then
\[
\langle [S_1]L_1 \parallel \ldots \parallel [S_n]L_n, \mu \rangle \rightarrow^i \langle [\hat{S}_1]\hat{L}_1 \parallel \ldots \parallel [\hat{S}_n]\hat{L}_n, \hat{\mu} \rangle
\]
if and only if
- \( l \in L_i \)
- \( \text{lock}^\#(\hat{\mu}(l)) = \text{lock}^\#(\mu(l)) + 1 \)
- \( m = n \) and \( \hat{L}_j = L_j \) for every \( j \in \{1..n\} \).

Locks on locations are never released by mistake:

Proposition 7 (Lock releasing). Let
\[
\langle T_0, \mu_0 \rangle \rightarrow^* \langle [S_1]L_1 \parallel \ldots \parallel [S_n]L_n, \mu \rangle \rightarrow^i \langle [\hat{S}_1]\hat{L}_1 \parallel \ldots \parallel [\hat{S}_n]\hat{L}_n, \hat{\mu} \rangle
\]
be an arbitrary run. Then \( \bigcup^n_{j=1} L_j \supset \bigcup^n_{j=1} \hat{L}_j \), if and only if
- \( S_i = \kappa.m[[\text{unlock}(l); C]_\sigma : S \), for some \( \kappa.m, l, C, \sigma \) and \( S \)
- \( L_i = \hat{L}_i \cup \{l\} \)
- \( \text{lock}^\#(\mu(l)) = 1 \) and \( \text{lock}^\#(\hat{\mu}(l)) = 0 \)
- \( m = n \) and \( \hat{L}_j = L_j \) for every \( j \in \{1..n\} \) \setminus \{i\}.

Unlocking always happens after some locking: it may release the lock or not, depending on the number of previous lockings.

Proposition 8 (Unlocking). Let
\[
\langle T_0, \mu_0 \rangle \rightarrow^* \langle [S_1]L_1 \parallel \ldots \parallel [S_n]L_n, \mu \rangle \rightarrow^i \langle [\hat{S}_1]\hat{L}_1 \parallel \ldots \parallel [\hat{S}_n]\hat{L}_n, \hat{\mu} \rangle
\]
be an arbitrary run where \( S_i = \kappa.m[[\text{unlock}(l); C]_\sigma : S \), for some \( \kappa.m, C, \sigma \) and \( S \), then
- \( l \in L_i \)
- if \( \text{lock}^\#(\mu(l)) \geq 1 \) then \( \hat{L}_i = L_i \) else \( \hat{L}_i = \hat{L}_i \cup \{l\} \)
- \( \text{lock}^\#(\hat{\mu}(l)) = \text{lock}^\#(\mu(l)) - 1 \)
- \( m = n \) and \( \hat{L}_j = L_j \) for every \( j \in \{1..n\} \) \setminus \{i\}.