A JMM-Faithful Non-Interference Calculus for Java

Vladimir Klebanov

University of Koblenz-Landau
Institute of Computer Science
vladimir@uni-koblenz.de

Abstract. We present a calculus for establishing non-interference of several Java threads running in parallel. The proof system is built atop an implemented sequential Java Dynamic Logic calculus with 100% Java Card coverage. We present two semantic and one syntactic type of non-interference conditions to make reasoning efficient. In contrast to previous works in this direction, our method takes into full account the weak guarantees of the Java Memory Model concerning visibility and ordering of memory updates between threads.

1 Introduction

Concurrent programming in Java, as in other languages supporting concurrency and shared memory, exposes the phenomenon of *interference*. Sequential programs proven correct may go awry when composed as threads in a concurrent setting. The problem results from concurrent modification of shared datastructures, and its control has been long of interest in software verification (albeit mostly for simple programming languages).

We present a proof system for establishing non-interference of Java threads, i.e., we specify the conditions $\Phi_1 \dots \Phi_n$ such that the following parallel composition rule (stated using dynamic logic) is sound.

$$\frac{\langle p_1 \rangle \phi_1 \qquad \langle p_2 \rangle \phi_2 \qquad \Phi_1 \dots \Phi_n}{\langle p_1 \parallel p_2 \rangle \phi_1 \wedge \phi_2} \text{ PAR\_COMP}$$

In contrast to previous works in this field [1], our proof system takes into full account the weak guarantees of the Java Memory Model (JMM) concerning visibility and ordering of memory updates between threads. Software verified with our method will thus always work as expected when executed on a real Java Virtual Machine.

The calculus we present follows the style of Owicki and Gries [9]. While the Owicki-Gries method is not compositional, we have chosen this fundamental approach for our work before working on compositionality. Also, only the mutual exclusion primitives of Java (the `synchronized` keyword) are considered. Primitives for condition synchronization (`wait()` and `notify()`) are not.

Our work is based on an implemented and complete sequential proof system — the KeY system [7, 2] — which we will introduce briefly.

2 Foundations

2.1 Java Dynamic Logic

Introduced in [3], Java Dynamic Logic (Java DL) is a modal logic with the modalities $\langle p \rangle$ ("diamond") and $[p]$ ("box") for every program p. The modality $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state, in which ϕ holds. In contrast, the formula $[p]\phi$ asserts that, if the program p terminates, then in a state satisfying ϕ.

A formula $\phi \rightarrow \langle p \rangle \psi$ is valid if, for every state s satisfying precondition ϕ, a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds. Thus, the formula $\phi \rightarrow [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$, while $\phi \rightarrow \langle p \rangle \psi$ implies the total correctness of p.

2.2 The KeY Calculus

As usual for deductive program verification, we use a sequent-style calculus. A *sequent* is of the form $\Gamma \vdash \Delta$, where Γ, Δ are duplicate-free lists of formulas. Intuitively, its semantics is the same as that of the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$.

A *proof* for a goal (a sequent) S is an upside-down tree with root S. In practice, rules are applied from bottom to top. That is, proof construction starts with the initial proof obligation at the bottom and ends with axioms (rules with an empty premiss tuple).

Besides the standard first-order and rewriting rules, the KeY calculus contains rules for symbolic execution of Java programs and induction. Most rules (rule instances) have a *focus*, i.e., a single formula, term, or program part (in the conclusion of the rule) that is modified or deleted by applying the rule.

Furthermore, symbolic execution rules operate only on the first *active* statement p of a program $\pi p \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "{", labels, beginnings "`try{`" of `try-catch-finally` blocks, etc. The postfix ω denotes the "rest" of the program. For example, if a rule is applied to the following Java block, the active statement is `i=0;`: $\underbrace{\texttt{l:\{try\{}}_{\pi}$ `i=0;` $\underbrace{\texttt{j=0; \}finally\{ k=0; \}\}}}_{\omega}$.

Since there is (at least) one rule schema in the Java DL calculus for each Java programming construct, we cannot present all of them in this paper. Instead, we give a simple but typical example, the rule IF_ELSE_SPLIT for the `if` statement:

$$\frac{\Gamma,\ b = \text{TRUE} \ \vdash\ \langle \pi\ \ p\ \ \omega \rangle \phi \qquad \Gamma,\ b = \text{FALSE} \ \vdash\ \langle \pi\ \ q\ \ \omega \rangle \phi}{\Gamma \ \vdash\ \langle \pi\ \texttt{if(}b\texttt{)}\ \ p\ \texttt{else}\ \ q\ \ \omega \rangle \phi}\ \text{IF\_ELSE\_SPLIT}$$

The rule has two premisses, which correspond to the two cases of the `if` statement. The semantics of this rule is that, if the two premisses hold in a state, then the conclusion is true in that state. In particular, if the two premisses are

valid, then the conclusion is valid. Note, that this rule is only applicable if the condition b is known (syntactically) to be free of side-effect. Otherwise, if b is a complex expression, other rules have to be applied first to evaluate b.

2.3 Symbolic Execution

We will call an application of a rule that has a program as its focus (e.g., IF_ELSE_SPLIT) a *symbolic execution step*. Each symbolic execution step is thus inherently related to (1) a sequent (matching the conclusion of the rule), (2) a modal formula in this sequent, (3) the program in this formula, and (4) the active (first) statement in this program.

To unify the presentation, we assume that the focus sequent of every symbolic execution step is of the form

$$\Gamma \vdash \mathcal{U}\langle p\rangle\phi \quad \text{or} \quad \Gamma \vdash \mathcal{U}[p]\phi$$

where $\mathcal{U}$ is a (possibly empty) list of updates, which are described below. This requirement does not destroy completeness and can be easily achieved by inserting first-order normalization steps into any given proof. In the following we will use the sequent form with a diamond; the results are valid for the box form as well though.

2.4 Updates

A special significance comes to the assignment rules(s) when handling program state. The Java Dynamic Logic does not work with states as first-class citizens. Assignment cannot be treated by syntactic substitution either because of aliasing (the possibility that different syntactical entities reference the same storage location). The solution Java DL employs is called *updates*.

These (state) updates are of the form $\langle loc := se\rangle$ and can be put in front of any formula or term. This expression then has to be evaluated in the state where loc has the value se. The expressions loc and se must be simple in the following sense: loc is (a) a local variable `var`, or (b) a field access `obj.attr`, or (c) an array access `arr[i]`; and se is free of side effects. More complex expressions are not allowed in updates. Other rules have to be applied first to break these down.

The assignment rule takes the following form ($\mathcal{U}$ stands for an arbitrary sequence of updates):

$$\frac{\Gamma \vdash \mathcal{U}\langle loc := se\rangle\langle \pi\ \ \omega\rangle\phi}{\Gamma \vdash \mathcal{U}\langle \pi\ \ loc\ \texttt{=}\ se\texttt{;}\ \ \omega\rangle\phi}\ \text{ASSIGNMENT}$$

That is, it just adds the assignment to the list of updates $\mathcal{U}$. The KeY system uses special simplification rules to compute the result of applying an update to logical terms and formulas not containing programs. This delayed evaluation has the advantage that a maximal amount of information is available for efficient simplification after the program has been symbolically executed to completion.

3 Characterizing Program State With Formulas

To reason about (non-)interference of symbolic execution steps we need to make tangible the notion of program state, which the KeY calculus never handles explicitly. All sequents are evaluated in the same (start) state; evaluation of individual formulas can be performed in a changed state by attachment of updates.

Definition 1 (Sequent state formula *state*(S)) By restricting the focus sequent form as described above, we define a single formula characterizing the program state in which a given symbolic execution step originates. For a sequent S of the form $\Gamma \vdash \langle x := y\rangle\langle p\rangle\phi$

$$state(S) := \exists v\, \langle x := v\rangle \Gamma \wedge x \doteq \langle x := v\rangle y$$

where Γ in this formula is a conjunction of all formulas in the antecedent of the sequent S.

Note. The above definition is simplified for the assumption that there is only one update and x is unqualified. If x is of the form $o.a$ then the second conjunct must read $(\langle x := v\rangle o).a \doteq \langle x := v\rangle y$. If x is of the form $a[i]$ then the second conjunct must read $(\langle x := v\rangle a)[\langle x := v\rangle i] \doteq \langle x := v\rangle y$. The extension for several updates is straightforward. ◁

The definition given above encodes information about the state contained in Γ and the updates attached to $\langle p\rangle\phi$ as a single formula of our logic. The fresh variable v is used to capture the value of x prior to performing the update. An example is presented in Table 1.

Sequent S	$state(S)$	Validity Eqv.
$x \doteq 0 \vdash \langle x := x + 1\rangle\langle p\rangle\phi$	$\exists v\, \langle x := v\rangle x \doteq 0 \wedge x \doteq \langle x := v\rangle x + 1$	$x \doteq 1$
$x \doteq 0 \vdash \langle x := 2\rangle\langle p\rangle\phi$	$\exists v\, \langle x := v\rangle x \doteq 0 \wedge x \doteq \langle x := v\rangle 2$	$x \doteq 2$

Table 1. Example for state characterization.

Theorem 1 (State characterization is adequate) The sequent S of the form $\Gamma \vdash \langle x := y\rangle\langle p\rangle\phi$ and the sequent $\vdash state(S) \rightarrow \langle p\rangle\phi$ are validity equivalent. ◁

4 Semantic Non-Interference Conditions

A proof tree for a property of a single sequential program represents all possible paths of program execution steps. We wish to ascertain that each of these steps can be performed correctly (w.r.t. our desired property) even if the scheduler

chooses to interleave steps from other threads that are running in parallel. In other words, we verify that the assumptions required for the correctness proof of one thread are not damaged by the updates that other threads might carry out on the common state.

Definition 2 (Proof robustness) A proof P_1 is *robust* under parallel composition with proof P_2 if for every symbolic execution step S_1 in P_1 and every symbolic execution step S_2 in P_2 that performs a state update the condition $\Phi(S_1, S_2)$ holds. Different kinds of the condition Φ are presented below. $\triangleleft$

Now we employ the notion of proof robustness to state the main result of symmetric non-interference.

Theorem 2 (Non-Interference) Two proofs P_1 and P_2 are *non-interfering* if P_1 is robust under parallel composition with P_2, and P_2 is robust under parallel composition with P_1. The parallel composition rule PAR_COMP is correct with these premisses. A justification is presented in Section 4.3. $\triangleleft$

Thus, to establish non-interference involving two threads with m and n statements we have to verify $O(m \times n)$ conditions. For this reason it is desirable that the conditions are as simple as possible. The majority of these conditions can, in fact, be discharged automatically. In the following we present and discuss two semantic and one syntactic condition.

Note on inter- vs. intra-object interference The number of noninterference conditions can be reduced dramatically up-front if we prohibit qualified access to fields in programs (as in [1]). Under this (sensible) restriction, expressions like $o.a$ are not allowed, and methods can only refer to fields of the local object, like $this.a$. Interference is thus confined within object boundaries, which allows us to drop all conditions that involve code from classes not in a direct line of inheritance.

Note on double and long variables The semantical conditions rely on the atomicity of a simple assignment in the sense of Section 2.4. This atomicity is not given for variables declared as `double` or `long` [8, §8.4].

4.1 Preservation of Pre-State

A naive version We will start with a simplified version of $\Phi(S_1, S_2)$, which is analogous to previous formulations of the Owicki-Gries method. This simplification assumes the existence of a consistent global state for both threads. It is adequate for theoretical programming languages or a Java VM with much stronger memory model guarantees than the ones actually given by the current official specification. We will weaken these assumptions later on.

The now following condition ensures that the execution of an (atomic) assignment loc=se; in the sequent of S_2 does not falsify assumptions appearing in the symbolic execution step with sequent S_1. The condition $\Phi(S_1, S_2)$ is expressed in this case by a logical formula, which has to be proved in our calculus. We define

$$\Phi_{\text{pre}}(S_1, S_2) := state(S_1) \wedge state(S_2) \rightarrow [loc\texttt{=}se\texttt{;}]state(S_1)$$

Note the use of the box modality here, since *loc*=*se*; could terminate abruptly due to a `NullPointerException` (if *loc* is a field or array access) or an `ArrayIndexOutOfBoundsException` (if *loc* is an array access). In case of abrupt termination no state update is performed, and there is no danger of interference, as the thrown exception is not visible to other threads. A diamond modality would, in contrast, always require normal termination.

A JMM-faithful version For a single thread the JMM provides strong guarantees about the visibility and ordering of memory updates, which are consistent with our intuition and reflected by the KeY calculus [8, § 8.1]. There is, however, no guarantee that memory updates performed by one thread will be visible (in any particular order, or even at all) by other threads in absence of proper synchronization [8, §§8.1, 8.3, 8.11].

Example 1 Let x, y be object fields. With naive semantics in mind, one could believe the proof for $\langle$`y=2;x=2;`$\rangle x \doteq 2$ to be robust under execution of the assignment `x=y;` in a second thread. The crucial condition required to prove this is (we simplify the state characterizations)

$$x \doteq 2 \land y \doteq 2 \rightarrow [\texttt{x=y;}]x \doteq 2$$

which obviously holds. In the JMM-faithful semantics this robustness, however, cannot be expected. The effect of the assignment `y=2;` may be not visible for the thread number two, and the assignment `x=y;` (scheduled after `x=2;`) would operate with a stale value of y, which is not necessarily 2.

To reflect the fact that we cannot rely on updates performed by other threads, we have to establish variable disjointness by renaming variables in one of the threads. This turns the condition above into

$$x \doteq 2 \land y \doteq 2 \rightarrow [\texttt{x=y';}]x \doteq 2$$

which (correctly) cannot be proved. ◁

The condition Φ_{pre} has thus to be recast as Φ'_{pre}:

$$\Phi'_{\text{pre}}(S_1, S_2) := \mathit{state}(S_1) \land \mathit{state}'(S_2) \rightarrow [\mathit{loc}\texttt{=}\mathit{se}'\texttt{;}]\mathit{state}(S_1)$$

where $\mathit{state}'(S_2)$ differs from $\mathit{state}(S_2)$ in that all appearing object fields have been renamed (accented by a dash). se' appears in that manner in the place of *se* in the assignment in the box. This version of the condition is significantly stronger, as no information flow is assumed from the first thread to the second. Since the JMM does not guarantee memory update visibility between threads (in absence of proper synchronization) such an assumption would be indeed false.

On the other hand, should the update `loc=se';` not become visible to the first thread, we would have solely proved one condition too many, thus erring on the safe side.

Note on volatile variables A relaxation of the above condition can be achieved for variables declared as `volatile`. For volatile variables the JMM

enforces state coherence, i.e., the value of a volatile variable is always visible correctly across all threads. Volatile variables thus need not be accented with a dash in Φ'_{pre}.

4.2 Assertion Insensitivity

Failing to prove a Φ_{pre} condition does not necessarily mean interference. Non-interference could still be established by considering a more general condition at this point. This time the assignment $loc{=}se;$ in the sequent S_2 is allowed to falsify assumptions made in S_1, but only if this does not affect the provability of the main assertion of S_1. We define

$$\Phi_{\text{post}}(S_1, S_2) := state(S_1) \wedge state'(S_2) \rightarrow [loc{=}se';]\langle p\rangle\phi$$

Such a criterion is more powerful but less practical, as it involves proving a version of the (complicated) assertion $\langle p\rangle\phi$. Furthermore, this proof has to be checked (with the usual criteria) for non-interference too.

4.3 Correctness of the Parallel Composition Rule

We give a general proof-theoretical argument for the correctness of the composition rule from Section 1 under the specified non-interference conditions, concentrating on the Φ_{pre} case. Details for other condition types can be found in the corresponding sections. An important prerequisite for the argument is the completeness of the KeY calculus w.r.t. the Java Dynamic Logic[1]: we assume that there is a proof for every true assertion in the sequential fragment.

In the following, we will present and justify a transformation that allows (under conditions specified above) to derive a proof for $\langle p_1 \parallel p_2\rangle\phi_1$ from the proof for $\langle p_1\rangle\phi_1$ alone. Since the situation is symmetrical, we can derive that $\langle p_1 \parallel p_2\rangle\phi_2$ holds whenever $\langle p_2\rangle\phi_2$ holds, and thus establish $\langle p_1 \parallel p_2\rangle\phi_1 \wedge \phi_2$.

Every proof step in a proof falls into one of the following categories:

- **Proof steps without modality in focus** These rules are in the "propositional" fragment. It is thus safe to replace every modality of the form $\langle p\rangle$ in the conclusion and premisses of this step with $\langle p \parallel p_2\rangle$.
- **Update simplification steps** Update simplification rules have no dependency on the modality they are attached to. The same replacement can be performed.
- **Symbolic execution steps** Every symbolic execution step establishes the validity of the sequent $S_1 : \Gamma \vdash U\langle p\rangle\phi_1$. A single non-interference condition $\Phi(S_1, S_2)$ states that starting in any state that satisfies $state(S_1)$ and executing an assignment from the program p_2 running in parallel, we arrive in a state that still satisfies $state(S_1)$ (it need not be the same state). The same is true of any sequence of such assignments, or finally the whole program p_2. Thus we can replace every occurrence of the formula $\langle p\rangle\phi$ in the focus of a symbolic execution step with the formula $\langle p \| p_2\rangle\phi$ without sacrificing the correctness of the proof.

[1] Efforts are currently underway to provide a formal completeness proof [10]

5 Syntactic Non-Interference Condition

This type of condition is less powerful than the two discussed above, but allows in many cases to dismiss the possibility of interference immediately and without interaction. Informally, syntactical non-interference is given if programs deal with disjoint memory locations if write access is involved. We will identify a read set and a write set for every rule of our calculus denoting symbolic memory locations read resp. written by the corresponding symbolic execution step.

Definition 3 (Write Set $W(r)$) The *write set* contains symbolic locations whose content is changed by a symbolic execution rule. The only rule with a non-empty write set is the ASSIGNMENT rule:

$$\frac{\Gamma \vdash \mathcal{U}\langle loc := se\rangle\langle \pi\ \omega\rangle\phi}{\Gamma \vdash \mathcal{U}\langle \pi\ loc\ \texttt{=}\ se\texttt{;}\ \omega\rangle\phi}\ \text{ASSIGNMENT}$$

We define $W = loc$, if loc is not a local variable. ◁

Definition 4 (Read Set $R(r)$) An expression e contained in the read set of a rule r is characterized by the following conditions:

1. e appears inside the diamond in the conclusion of r (i.e., the focus of r)
2. e appears outside a diamond in the premisses of r (including updates, though not on the left-hand side)
3. e is not a local variable ◁

Example 2 – Assuming se is not a local variable, $R(\text{ASSIGNMENT}) = \{se\}$ as se appears in the focus diamond of the conclusion and outside of it in the premiss. $loc \notin R(\text{ASSIGNMENT})$ since it appears on the left-hand side of an update. On the other hand $loc \in W(\text{ASSIGNMENT})$.

– $R(\text{IF\_ELSE\_SPLIT}) = \{b\}$, again, assuming b is not a local variable.

– Consider the rule IF_ELSE_EVAL

$$\frac{\Gamma \vdash \langle \pi\ \texttt{boolean}\ b\texttt{;}\ b\texttt{=}nse\texttt{;}\ \texttt{if(}nse\texttt{)}\ p\ \texttt{else}\ q\ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \texttt{if(}nse\texttt{)}\ p\ \texttt{else}\ q\ \omega\rangle\phi}\ \text{IF\_ELSE\_EVAL}$$

$R(\text{IF\_ELSE\_EVAL}) = \emptyset$ as this rule replaces one diamond through another with the same transition relation. There is no state change and no change in observable conditions. Note that there is a subtle issue at stake here in that the symbolic execution rules do not "swallow" any state transitions. Replacing `i++;i--;` with an empty statement is not allowed. ◁

Definition 5 (Syntactic Non-Interference) Within the framework of Theorem 2 we define a syntactical non-interference condition $\Phi_{\text{synt}}(S_1, S_2)$. For this we need to consider the rules r_1 and r_2 involved in the steps S_1 and S_2. We define that $\Phi_{\text{synt}}(S_1, S_2)$ holds if and only if

$$\begin{aligned} R_{S_1}(r_1) \sqcap W_{S_2}(r_2) &= \emptyset \\ W_{S_1}(r_1) \sqcap R_{S_2}(r_2) &= \emptyset \\ W_{S_1}(r_1) \sqcap W_{S_2}(r_2) &= \emptyset \end{aligned}$$

where $\sqcap$ denotes intersection under aliasing. ◁

Intersection under aliasing treats two symbolic locations as the same if — regardless of their syntactical form — there is a possibility that they are referring to the same effective location due to aliasing. Since we have excluded local variables up front, we can safely assume two symbolic locations as distinct under aliasing only if they are of non-compatible types.

Ultimately, $\Phi_{\text{synt}}(S_1, S_2)$ ensures that program behavior does not change in presence of other programs. It can be used in place of $\Phi'_{\text{pre}}(S_1, S_2)$ until the "last" symbolic execution is reached (i.e., S_1 removes an empty modality). At this point, the intactness of the first-order assertions must be checked with $\Phi'_{\text{pre}}(S_1, S_2)$.

6 Synchronization

A special concurrency primitive of Java is the `synchronized` keyword. This primitive can be used to achieve mutual exclusion of critical code sections. Blocks of code or whole methods can be declared as synchronized. Synchronization happens with respect to an object being locked (for synchronized methods it is always the current object).

The semantics of synchronized is that no two distinct threads can concurrently execute code marked as synchronized w.r.t. the same lockable object reference. One of the threads would block at the attempt to acquire the lock already held by the other thread. Note that mutual exclusion does not take place if only one code section is declared as synchronized (synchronization is not atomicity), or if the lock object references of two synchronized code sections are different.

This leads us to the following refinement of our semantic conditions. If the statement in focus of the symbolic execution step S_1 as well as the assignment in focus of S_2 both lie within a code section marked as synchronized, and the statement of S_1 is not the first statement in the section, we can relax the non-interference condition Φ'_{pre} to:

$$\Phi'_{\text{pre(s)}}(S_1, S_2) := \mathit{state}(S_1) \wedge \mathit{state}'(S_2) \wedge \neg \mathit{syncref}(S_1) \doteq \mathit{syncref}(S_2) \rightarrow [\mathit{loc}\texttt{=}\mathit{se}'\texttt{;}]\mathit{state}(S_1)$$

where $\mathit{syncref}(S_1)$ and $\mathit{syncref}(S_2)$ are the lock references of the two sections. If these references are the same, the non-interference condition holds automatically since the JVM guarantees mutual exclusion.

Synchronization and memory update visibility The `synchronized` keyword has also a second meaning in Java. Beside mutual exclusion functions, it has a signaling function to the JVM memory subsystem. When a thread exits a synchronized code section, the content of its local store is flushed into the main memory. An entry into a synchronized code section, on the other hand, effects a reloading of the main memory content into the thread's working store [8, §8.6].

An immediate succession of these two events is thus a means for one thread to obtain a complete and consistent visibility of memory updates performed by another thread. Unfortunately, tracking such *rendez-vous* requires considering complete concurrency histories, which is is outside of the scope of a non-interference calculus.

7 Conclusion, Comparison and Future Work

We have presented (to our knowledge) the first non-interference proof system for the Java language, which reflects the actual execution semantics as stated by the Java Virtual Machine specification. Among related works, [1] does not take the JMM into account, while [4] gives a JMM-faithful operational semantics but does not provide a proof system. Furthermore, our proof system is built atop an implemented, complete calculus for sequential Java. A prototype implementation of it is available with the latest version of the KeY system.

It remains to be seen how the compositional extensions to the Owicki-Gries method, e.g., [5, 11], can be made to work in the JMM-constrained situation. Changes would also probably be necessary if the Java Memory Model revision effort [6] is successful. Development of more powerful parallel composition rules is furthermore of interest.

Acknowledgment I wish to thank Bernhard Beckert for fruitful discussions, and Christoph Gladisch and Anne Tretow for their help with the implementation.

References

1. Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Inductive proof-outlines for monitors in Java. In *International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, 2003. A longer version appeared as Software Technologie technical report TR-ST-03-1, April 2003.
2. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling (SoSysM)*, pages 1–42, 2004. Available at `http://www.springerlink.com`.
3. Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
4. Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. In *Formal Syntax and Semantics of Java*, pages 157–200. Springer-Verlag, 1999.
5. Cliff B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.
6. Java memory model and thread specification revision. Website at `http://jcp.org/en/jsr/detail?id=133`.
7. KeY Project. Website at `www.key-project.org`.
8. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
9. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.
10. André Platzer. An object-oriented dynamic logic with updates. Master's thesis, Universität Karlsruhe, 2004.
11. C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.