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

Bernhard Beckert and Vladimir Klebanov

Abstract

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

1 Introduction

1.1 Motivation and Goals

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

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

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

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

full handling of exceptions and method calls, Java-faithful arithmetics, etc.

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

1.2 Achieved Java Coverage

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

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

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

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

1.3 Related Work

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

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

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

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

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

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

2 A Logic for Concurrent Java

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

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

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

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

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

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

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

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

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

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

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

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

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

The Scheduler

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

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

Signatures and Variables

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

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

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

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

Formulas

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

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

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

Semantics of Terms, Programs, and Formulas

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

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

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

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

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

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

A Deductive Calculus

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

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

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

3 Symbolic Execution of Concurrent Programs

3.1 Extending Symmetry Reduction

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

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

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

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

3.2 Expressing Unbounded Concurrency

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

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

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

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

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

3.3 Program Unfolding

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

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

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

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

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

3.4 Concurrency-Related Rules

3.4.1 Configuration Skolemization

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

$$\textsf{conf}\ \frac{\Longrightarrow \langle\!| r | c_p | p |\!\rangle \phi}{\Longrightarrow \langle\!| r | c | p |\!\rangle \phi}$$

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

3.4.2 Position Choice

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

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

Fig. 1. The concurrent symbolic execution rule

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

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

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

3.4.3 The Rule for Concurrent Execution

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

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

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

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

Fig. 2. The rule for lock acquisition

3.4.4 The Rule for Empty Programs

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

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

3.4.5 Reasoning About Permutations

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

Together with the following permutation interplay axiom

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

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

4 Treating Concurrency Primitives

4.1 Treating Locking Primitives

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

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

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

Fig. 3. The rule for lock release

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

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

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

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

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

4.2 Treating Condition Variables

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

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

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

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

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

4.2.1 Additional Means of Expression

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

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

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

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

4.2.2 Restrictions Posed on Programs

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

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

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

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

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

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

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

Fig. 4. Blocking queue source code

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

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

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

Fig. 5. Blocking queue source code (unfolded)

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

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

4.2.3 An Example Application

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

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

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

Fig. 6. The rule for notification

the scheduler). This can be written as:

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

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

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

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

4.2.4 The Rules for Symbolic Execution

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

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

waitUntil

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

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

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

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

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

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

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

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

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

5 Conclusion

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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