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

1.0.1 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. First, we do not consider thread identities in programs. Second, we do not handle dynamic thread creation (but systems with an unbounded number of threads). Third, 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.

Our calculus has been implemented in the KeY system [1,2], 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 (with static initialization), efficient aliasing treatment, full handling of exceptions and method calls, Java-faithful arithmetics, etc.

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

This paper extends [3] with rules to verify programs with condition variables. Conversely, the former paper includes a survey of related work, extensive introduction and motivation, an additional invariant rule, explanatory examples and a description of the application of our method to verify a piece of code from the Java standard library.

2 A Logic for Concurrent Java

2.1 Design of the Program Logic

The logic we present in this paper is an instance of Dynamic Logic (DL) [4], 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; in DL for sequential Java, programs are deterministic, and there is exactly one such world (if p terminates) or there is no such world (if p does not terminate). Facing the choice of semantics for our logic for concurrent programs, we wish to argue (surprisingly maybe) for a deterministic semantics.

2.1.1 Concurrent Programs

The programs we consider are Java programs with the inherent restrictions posed in Section 1.0.1.

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.

2.1.2 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.

2.1.3 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.

2.1.4 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. It does not depend on the 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*: $\mathit{sched}(r, c)$ is the id of the thread scheduled to run next in configuration c given the seed r. If no position is enabled in c, $\mathit{sched}(r, c) = 0$. 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".

2.1.5 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. Every thread has its own private copy of each local variable, such that assignments to these are not visible in other threads. We give non-rigid functions used to model thread-local variables another argument, which is the thread id, such that the local copies can be distinguished. For example, $l(k)$ denotes the copy of variable l used by the thread with id k. This distinction, though, is unavailable *within* concurrent programs, as one thread is unaware of other threads' copies of the same local variable. As a peculiar consequence a 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.

2.1.6 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, $\{lhs{:=}rhs\}\phi$ is a formula. The expression $\{lhs{:=}rhs\}$ 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.

2.1.7 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 (to be more precise: 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 $s \in S$ provide interpretations of functions (including program variables) via first-order structures for the signature Σ. We 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 $\mathcal{K}$. 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 assume that the set S of states of any Kripke structure consists of *all* first-order structures with signature Σ over some fixed universe and for some fixed interpretation of the rigid symbols.

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, $val_{s,\beta}(\{lhs{:=}rhs\}\phi) = \mathit{true}$ iff $val_{s',\beta}(\phi) = \mathit{true}$ 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.

2.1.8 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 $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. Its informal semantics is the same as that of the formula $\bigwedge_{\phi\in\Gamma}\phi \;\rightarrow\; \bigvee_{\psi\in\Delta}\psi$. 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

To make verification of unbounded systems feasible we extend and apply the technique of symmetry reduction. First, we identify threads with different local data as long as they follow the same path through the code. Furthermore, we can achieve even stronger reduction by forcing a separation of thread scheduling and control

flow. To obtain symmetry between threads with different paths through the program, we assume 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 we have 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. Then, the exact configuration is fixed as well. That allows us to write configurations with r positions in the form $(p_1 : k_1, \ldots, p_r : k_r)$, where $p_1, \ldots, p_k$ are terms representing the permutations and $k_1, \ldots, k_r$ are terms representing the number of threads at the positions. 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 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 four 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 it 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 also 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 arithmetics, which contain permutation functions encapsulating the scheduler decisions. In the desirable case that the program is scheduling-independent the permutation functions 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 Pre-Processing Complex Sequential Program Parts

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 [2]), 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 concurrency 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 local variables not occurring in the body of the statement. The latter property greatly simplifies the rules of our calculus. The fact that these variables—once set—cannot change their value eliminates technical difficulties when specifying execution path conditions.

During the simplification process, the KeY calculus introduces fresh local variables. For instance, KeY unfolds Java's `o.a=u.a++;` into `v=u.a+1; u.a=v; o.a=v;` (`v` is a fresh local variable). Java's conditional `if (o.a>1) {`α`} else {`β`}` is unfolded to `v=o.a>1; if (v) {`α'`} else {`β'`}` and, slightly more involved, the Java program fragment `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 The 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.

$$\frac{\vdash \langle\!\langle r|c_p|p\rangle\!\rangle \phi}{\vdash \langle\!\langle r|c|p\rangle\!\rangle \phi}\ \text{CONF}$$

where c is a thread configuration of the form $(\{i_1^1,\ldots,i_{l_1}^1\},\ldots,\{i_1^r,\ldots,i_{l_r}^r\})$; 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.

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

Fig. 1. The concurrent symbolic execution rule

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, $enabled(c,pos) = (c(pos) > 0) \wedge (pos < size(c))$.

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. π and ω denote unchanged program parts. 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 `b` $= FALSE \wedge$ `a` $= TRUE$.

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 template for the thread given by the appropriate permutation and the current configuration.

For example, if we consider the assignment `v=o.a;` at position one in some program, and the configuration before execution is $(p_1 : 2,\ p_2 : 5,\ p_3 : 7)$, then the generated update is $\{\mathtt{v}(p_1(13)) := \mathtt{o}(p_1(13))\mathtt{.a}\}$. The update will be tackled by the

update simplification rules, after the program has been completely executed. This will happen at some point, since the rule reduces the general measure of enabledness in the system.

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:

$$\frac{\vdash P(r,c) = 0 \qquad \vdash \phi}{\vdash \langle\!\langle r|c|p \rangle\!\rangle \phi} \text{ EMPTY-PROGRAM}$$

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 expresses the fact that exactly the threads can be scheduled at a given position that have already passed the previous position, but not yet the next.

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 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:

LOCK

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

The structure of this rule is similar to the STEP rule for handling normal assignments. Execution is successful if the path condition is satisfied and the statement is enabled (remember, $P(r,c) = pos$ implies $enabled(c, pos)$).

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

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

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. All other rules can remain the same.

UNLOCK

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

For efficiency reasons we never clear the `<lockedby>` flag, since it does not prevent the acquisition of the lock when the `<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 (indeterministically 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, under certain conditions outlined below, we can consider the wait-in-loop idiom as one atomic statement. These conditions are indeed satisfied in great many cases. 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 function 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 while loop), 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 2 and 3.

Finally, we need some means to differentiate between threads that are ready to

```
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. 2. Blocking queue source code

be scheduled at the position of `<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 allowed 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 the scheduler). This can be written as:

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

```
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. 3. Blocking queue source code (unfolded)

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 $Post(pos)$ 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 2, while Figure 3 shows the result of a partial unfolding (list operations and exceptions are not unfolded). An optimization is possible by leaving out the commented part of the code in the `get()` method, since it serves little function in this setting. 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()`. 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.

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

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 straight-

forward, and we leave it out here.

Now we present the rule for symbolic execution of `<waitUntil>()`:

waitUntil

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

The first premiss requires that the position in question is enabled: $enabled(c, pos)$. 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>()` operation at position pos, we update the predicate again, to:

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

This means that at least one thread at poshas 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 $\langle$`boolean x =`$b^{*(pos)}$`;`$\rangle$`x` $= TRUE$ 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 $p_{pos}(Post(pos) + 1)$). In the case of the blocking queue, Φ is simply `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] 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.

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

[3] 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.

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