A Dynamic Logic for Deductive Verification of Concurrent Programs

Bernhard Beckert and Vladimir Klebanov
Institute for Computer Science
University of Koblenz-Landau
www.key-project.org

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, and we demonstrate it by verifying a central method of the `StringBuffer` *class from the Java standard library.*

1. Introduction

Motivation and Goals Verification of programs with concurrency has traditionally been—with a few exceptions—the domain of model checking tools. This holds also for Java program verification, where several very successful model checking frameworks have been established [13, 7]. Nonetheless, for verification problems that are data-centric or that involve an unbounded number of threads, deductive verification offers advantages. In general, the properties we deal with in this paper can neither be expressed in temporal logic nor verified with a model checker.

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. An application of our method to verify one of the most common pieces of production Java code in presence of unbounded concurrency is described towards the end.

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.

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 [4]. 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 [5, 15]. 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

Design of the Program Logic The logic we present in this paper is an instance of Dynamic Logic (DL) [6], 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. A program is a passive template "without life" until 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. In this paper we only feature programs with a single code template or thread class. Our implementation supports multiple thread classes as this requires simply an additional case distinction.

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. $(\{3, 17, 5\}, \{\}, \{2\})$ is a configuration of size three. 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. 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 ${}^{\{5\}}$`v=(x<10); if (v) {`${}^{\{3,4\}}$`a=x;`${}^{\{1\}}$`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. 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, then $\mathit{sched}(r, c) = 0$. 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".

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.

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.

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 seed, ρ is a (partial) function: $\rho(r, c, p) : S \rightarrow S$.

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 attached 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.

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.

The INVARIANT rule in Section 5 has to be applied exactly as shown. From all other rules 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. A Calculus For 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 [14] (the Bogor tool) and [16] (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 $p_1, \ldots, p_k$ 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 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 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 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;` (with `v` a fresh local variable). Java's `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

Configuration Skolemization The rule given now 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 concrete 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.

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. P is axiomatized as follows.

The axiom $0 \le P(r,c) < \mathit{size}(c)$ effectively amounts to a disjunction over the positions of c, which during the proof gives rise to a case distinction. Furthermore, the values of P are restricted to the enabled positions in the configuration: $P(r,c) \neq 0 \rightarrow \mathit{enabled}(c, P(r,c))$. P may only return 0 if no position is enabled, which is expressed by: $P(r,c) = 0 \rightarrow \forall \mathit{pos}.(1 \le \mathit{pos} < \mathit{size}(c) \rightarrow \neg\mathit{enabled}(c,\mathit{pos}))$. Remember that for skolemized configurations $\mathit{enabled}(c,\mathit{pos}) \equiv (c(\mathit{pos}) > 0) \wedge (\mathit{pos} < \mathit{size}(c))$.

The Rule for Concurrent Execution

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

Listed above is 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. $\mathit{path}(\mathit{pos},p)$ 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` $= \mathit{FALSE} \wedge$ `a` $= \mathit{TRUE}$.

Furthermore, $\{\mathit{lhs}^{*(\mathit{pos})} := \mathit{rhs}^{*(\mathit{pos})}\}$ is a state update built by replacing every occurrence of a local variable v in lhs and rhs, by $v(p_{\mathit{pos}}(\mathit{Post}(\mathit{pos})+1))$ using the configuration of p (cf. definition of $\mathit{Post}(\cdot)$ 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 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.

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:

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

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 $i+1$.

Treating Concurrency 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 wrapped 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

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

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, $P(r,c) = pos \rightarrow enabled(c, pos)$).

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

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

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.

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.

Another important concurrency feature of Java is condition variables. It allows threads to suspend execution until an external signal is received. Condition variables can be modeled in a manner similar to locks, since their usage does not involve thread identities. On the other hand, it requires a (special kind of) non-atomic loop for correctness. We can verify programs where the condition is atomic, but do not show the rules here.

4. A Simple Example

Consider a financial transaction system that processes concurrent incoming payments for an account. We wish to establish that all payments end up deposited, regardless of their number and the order in which the threads are scheduled. This can be expressed by the following proof obligation, where `sum` is a shared variable and `e` is a local variable whose thread-local copies contain the payments. Which scheduling seed r is chosen is irrelevant, and p is an arbitrary permutation of $\{1,\dots,n\}$:

$$\{\text{sum}:=0\}\langle {}^{\{p:n\}}\texttt{sum = sum+e;}\ {}^{\{\}}\rangle(\texttt{sum} = \textstyle\sum_{i=1}^{n}\texttt{e}(i))$$

Note that for the sake of the example we have abused the programming language, writing an atomic assignment with two occurrences of a shared variable. In reality, it would unfold to `v=sum+e;sum=v;` (with `v` a fresh local variable) and locking would be necessary to avoid a race condition.

The proof of the property boils down to a simple induction argument. Let n be arbitrary but fixed, then the induction hypothesis is that $n-k$ transactions have been completed correctly, while k remain:

$$\begin{array}{l} \{\text{sum}:=\textstyle\sum_{i=1}^{n-k}\texttt{e}(p(i))\} \\ \quad \langle {}^{\{p:k\}}\texttt{sum = sum+e;}\ {}^{\{n-k\}}\rangle(\texttt{sum} = \textstyle\sum_{i=1}^{n}\texttt{e}(p(i))) \end{array}$$

Now we have to prove that the above holds for $k+1$ transactions, i.e.:

$$\{\texttt{sum}{:=}\textstyle\sum_{i=1}^{n-k-1} \texttt{e}(p(i))\}$$
$$\langle^{\{p:k+1\}}\texttt{sum = sum+e;}^{\{n-k-1\}}\rangle(\texttt{sum} = \textstyle\sum_{i=1}^{n} \texttt{e}(p(i)))$$

Applying the STEP rule to the above formula once, we obtain (there is only one position, and thus one permutation function, namely p):

$$\{\texttt{sum}{:=}\textstyle\sum_{i=1}^{n-k-1} \texttt{e}(p(i))\}\{\texttt{sum}{:=}\texttt{sum}+\texttt{e}(p(n-k))\}$$
$$\langle^{\{p:k\}}\texttt{sum = sum+e;}^{\{n-k\}}\rangle(\texttt{sum} = \textstyle\sum_{i=1}^{n} \texttt{e}(p(i)))$$

Where the updates can be combined to:

$$\{\texttt{sum}{:=}\textstyle\sum_{i=1}^{n-k} \texttt{e}(p(i))\}$$
$$\langle^{\{p:k\}}\texttt{sum = sum+e;}^{\{n-k\}}\rangle(\texttt{sum} = \textstyle\sum_{i=1}^{n} \texttt{e}(p(i)))$$

Now, the induction hypothesis for k applies, and the step case of the induction is closed. The base case $k=0$ is trivially valid. Thus, we have established the hypothesis for any $k \leq n$. Instantiating k with n yields:

$$\{\texttt{sum}{:=}\textstyle\sum_{i=1}^{0} \texttt{e}(p(i))\}$$
$$\langle^{\{p:n\}}\texttt{sum = sum+e;}^{\{0\}}\rangle(\texttt{sum} = \textstyle\sum_{i=1}^{n} \texttt{e}(p(i)))$$

The sum in the update is empty, so rewriting the postcondition with the lemma $\sum_{i=1}^{n} \texttt{e}(p(i)) = \sum_{i=1}^{n} \texttt{e}(i)$ we can derive the original conjecture. The lemma does not depend on the particular program but expresses properties of permutations (p is a permutation from $1 \ldots n$) and the commutativity of integer addition.

We have now verified the transaction mechanism for an arbitrary number of threads. This is important, since it is very easy to devise code that works for n but not for $n+1$ threads. The state explosion caused by the potentially different ordering of transactions is efficiently controlled, even without further knowledge of concrete data.

5. An Invariant Rule

For systems with a high number of potentially simultaneously enabled positions, applying induction may be unwieldy. In the following we present an additional invariant rule, which allows tackling each potentially enabled statement separately. Instead of an induction hypothesis, the user has to state (and then prove) a suitable invariant INV of the system. The rule is:

INVARIANT

$$\frac{\begin{array}{rl} & \vdash \mathcal{U} INV(r, c_0) \\ INV(r, c_\downarrow) & \vdash \phi \\ INV(r, c),\ path(1, p),\ enabled(c, 1) & \vdash \{lhs_1^{*(1)} {:=} rhs_1^{*(1)}\} INV(r, c_1) \\ & \vdots \\ INV(r, c),\ path(q, p),\ enabled(c, q) & \vdash \{lhs_q^{*(q)} {:=} rhs_q^{*(q)}\} INV(r, c_q) \end{array}}{\vdash \mathcal{U}\langle\!\langle r | c_0 | p \rangle\!\rangle \phi}$$

We assume that the program p has $q+1$ positions. c_0 is here the initial configuration $(n, 0, 0, \ldots, 0)$. From c_0 the final configuration $c_\downarrow = (0, 0, \ldots, 0, n)$ is eventually reached, where all n threads have run to completion. The generic configuration c is a tuple of variables $(t_1, \ldots, t_{q+1})$. The configurations c_i are the same as c except that $c_i(i) = c(i) - 1$ and $c_i(i+1) = c(i+1) + 1$ (i.e., c_i is c with one thread having moved from position i to $i+1$).

The first premiss of the rule states that the systems satisfies the invariant in its initial configuration. The second premiss states that the invariant implies the desired property, once all threads have completed their work. What follows are q premisses—one for each position in the program but the last—stating that the execution of the statement at this position preserves the invariant. More precisely, we show INV to be invariant under updates originating from every position i ($1 \leq i \leq q$) in the program. For each such position we can assume its enabledness and the corresponding path condition.

An example for using the invariant rule is given in the next section.

6. A Real-World Example

We have applied our calculus to verify the full functional correctness of a method of the `StringBuffer` class in presence of unbounded concurrency. The class `java.lang.StringBuffer` is a key class of the standard Java library that represents a mutable character sequence. Its central method is `append(char c)`, which appends the character `c` to the end of the sequence.

We have used the original source code shipped by SUN with the JDK 1.4.2 (shown in Figure 1). The `StringBuffer` implementation is backed by a `char` array, which is initially 16 elements long. Should the array become full, a new, longer array is allocated and the contents copied. This happens transparently for the user.

```
private char value[];
private int count;

public synchronized StringBuffer append(char c) {
    int newcount = count + 1;
    if (newcount > value.length)
        expandCapacity(newcount);
    value[count++] = c;
    return this;
}

private void expandCapacity(int minimumCapacity) {
    int newCapacity = (value.length + 1) * 2;
    if (newCapacity < 0) {
        newCapacity = Integer.MAX_VALUE;
    } else if (minimumCapacity > newCapacity) {
        newCapacity = minimumCapacity;
    }

    char newValue[] = new char[newCapacity];
    System.arraycopy(value, 0, newValue, 0, count);
    value = newValue;
    shared = false;
}
```

Figure 1. `StringBuffer` **source code (excerpt)**

A functional specification of the append method can be given as:

$$\begin{aligned}&\texttt{strb.<lockcount>} = 0 \wedge \neg \texttt{strb} = \texttt{null} \wedge \\ &\qquad \texttt{strb.count} = 0 \rightarrow \forall n.\ n > 0 \rightarrow \\ &\langle^{\{p_1:n\}}\texttt{strb.append(c);}^{\{0\}}\rangle \texttt{strb.count} = n \wedge \\ &\forall k.\ 0 \leq k < n \rightarrow \texttt{strb.value}[k] = \texttt{c}(p_1(k+1))\end{aligned}$$

where `strb` is a *shared* reference of type `StringBuffer`.

Plainly speaking: if n threads are concurrently performing an append on a shared (and initially empty) `StringBuffer` object, then all threads will eventually run to completion and the StringBuffer will contain exactly the characters deposited by the threads. Furthermore, the characters will fill the backing array in the "natural" order, i.e., the order induced by the thread scheduling.

We now describe the verification process, which has three major parts.

Unfolding First, we have "unfolded" the implementation. The `expandCapacity()` method has been inlined, and fresh local variables have been introduced to eliminate side effects and make explicit the atomicity granularity of the code. The result is shown in Figure 2, though exceptions and array creation are still in their folded state for brevity.

The code also shows a call to `System.arraycopy()`, which cannot be unfolded. This native method call can be seen as one big parallel assignment, which is sound under the atomicity proviso proven below. During symbolic execution, the KeY system translates a call like

```
strb.<lock>();
newcount=strb.count+1;
j_1=strb.value.length;
b=newcount>j_1;
if (b) {
    j_2=strb.value.length;
    j_3=j_2+1;
    newCapacity=j_3*2;
    b_1=newCapacity_<0;
    if (b_1) {
        newCapacity=Integer.MAX_VALUE;
    } else {
        b_2=newcount>newCapacity;
        if (b_2) {
            newCapacity=newcount;
        }
    }
    b_3=newCapacity<0;
    if (b_3) throw new NegativeArraySizeException();
    newObject=new char[newCapacity];
    src_1=strb.value;
    len_2=strb.count;
    System.arraycopy(src_1,0,newObject,0,len_2);
    strb.value=newObject;
}
val_1=strb.value;
j_4=strb.count;
strb.count=j_4+1;
val_1[j_4]=c;
strb.<unlock>();
```

Figure 2. Source code after unfolding

`arraycopy`$(src, srcPos, dest, destPos, len)$ into a so called *quantified update*

$$\begin{aligned}&\{\text{for } 0 \leq l < len;\\ &\qquad dest[srcDest + l] := src[srcPos + l]\}\end{aligned}$$

which is a concise way to express a number of updates at once. We also use quantified updates to state the induction hypothesis later on.

Establishing Atomicity Subsequently, we used the invariant rule to establish atomicity of the method. This greatly simplified further proof. We have to show that the method can only be executed by one thread at a time (on the same object). This property can be stated as $N \leq 1$, with $N \equiv \sum_{i=2}^{q} c_i$, so the configuration never has more than one thread between its second and the last but one position. Before the proof can proceed, the above has to be strengthend to $INV(r, c) \equiv N \leq 1 \wedge (N > 0 \leftrightarrow \texttt{<lockcount>} > 0)$.

This invariant clearly holds in the initial state, since both N and `<lockcount>` are zero. Statements at positions $2 \ldots q$ preserve the invariant, since they cannot increase the value of N, as only the statement at position 1 can. Finally, the locking statement at position 1 also preserves the invariant. If the lock is available then $N = 0$ before the locking per the second conjunct of the invariant. After the execution, both N and `<lockcount>` are equal to 1. If the lock is not available then the locking statement is disabled altogether.

Thus, the method is atomic, and we can assume the simplification lemma $p_i(k) = p_j(k)$ for $1 \leq i, j \leq q$ in the proof. Once a thread has entered the method it will run to completion without interference.

Establishing Functional Correctness So far, we know that the method is correctly synchronized, but is it also functionally correct? Using the Java-faithful bounded integer semantics of KeY, we have, of course, discovered that the specification shown above is not quite right, as it holds true only for $n \leq 2^{32} - 1$. Trying to insert more characters into a `StringBuffer` results in an `ArrayIndexOutOfBoundsException`. This bound may seem of little practical importance, but it is an instance of a general problem. Concurrent access to bounded data structures is likely to result in subtle bugs, even in presence of proper synchronization.

Since there is no way to fix the method, we amended the conjecture with a pre-condition limiting the initial value of n. After that it was easy to establish by induction. The induction hypothesis $I(k)$ we used is:

$$
\begin{array}{l}
k \leq n_0 \rightarrow \{\texttt{count}{:=}n_0 - k\} \\
\{\text{for } 0 \leq l \leq n_0 - k; \texttt{strb.value[}l\texttt{]} := \texttt{c}(p_1(l+1))\} \\
\langle^{\{p_1:k\}}\texttt{strb.append(c);}^{\{n_0-k\}}\rangle\texttt{strb.count} = n_0 \wedge \\
\quad \forall k.\ 0 \leq k < n_0 \rightarrow \texttt{strb.value[}k\texttt{]} = \texttt{c}(p_1(k+1))
\end{array}
$$

It summarizes the state of the system after $n_0 - k$ threads have run to completion. n_0 is the Skolem symbol introduced after eliminating the quantifier in the conjecture.

The induction base case $k = 0$ is trivial, since it corresponds to a system with no enabled threads. The step case $I(k_0) \rightarrow I(k_0 + 1)$ requires performing an "iteration" of the system with $k_0 + 1$ threads waiting, i.e., a symbolic execution of the append by the $(n_0 - k_0)$th scheduled thread. The execution has a choice between three relevant program paths: (1) `b` is false (2) `b` is true and `b_2` is false or (3) `b` is true and `b_2` is true. All of these can be discharged without complications. At last, instantiating k in the hypothesis with n_0 yields the conjecture.

In total the proof comprises 14622 proof steps in 238 branches. User interaction was required in two steps: specifying an induction hypothesis and instantiating the resulting quantifier. Proof search took about one minute on an average desktop computer.

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] 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.

[5] 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.

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

[7] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 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, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *ESEC/FSE-11*, pages 267–276, 2003.

[14] 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.

[15] 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.

[16] 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.